BETTER FEEDBACK, SMALLER AGENTS: RETHINKING AGENTIC CODE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern coding agents are often made more capable by adding specialized agents, tools, memory, and additional reasoning stages. But does adding more machinery actually produce better optimized code? We study this question systematically in code optimization, where an agent must discover faster implementations while keeping the optimization process affordable. We develop a controlled framework with 24 workflows spanning four design choices: direct versus delegated coding, raw versus analyzed execution feedback, optional tools and skills, and how earlier attempts are stored. We evaluate these workflows with GPT-OSS-20B, GPT-OSS-120B, and Qwen3.8-27B on 54 NPBench CPU kernels. The trade-offs are hard to anticipate: more elaborate workflows do not consistently perform better, and most components help in one setting while costing tokens or speed in another. The clearest gain comes from what the main agent receives between attempts: an analysis agent turns execution results into concise guidance, and each turn starts from a compact context. Across both coding modes, analyzed feedback makes programs 1.15–1.46× faster for all three models; with direct coding, the GPT-OSS models gain 1.47–1.51× and use 25–55% fewer tokens, although searches take longer. With analyzed direct coding, tools and skills raise token use by 36–134% without a significant speed-up, and history helps GPT-OSS direct coding more than delegated coding. The configuration with every component enabled uses 54–146% more tokens than direct coding with analysis and history, again without a significant speed-up. On three further models, analyzed feedback helps both Qwen3.5 models and Kimi-K2.7-Code's direct coding, while the value of the other components changes: Qwen3.5-397B, unlike GPT-OSS, gains substantially from delegation. Our results show that code-optimization performance does not grow monotonically with workflow complexity: simpler combinations can produce faster code with fewer tokens.

1 INTRODUCTION

Coding agents increasingly combine specialized agents, tools, memory, and additional reasoning stages. Each addition expands what an agent can do, but does it help the agent produce faster code? Code optimization makes this question concrete. Passing tests is only the first requirement: the agent must also find implementations that run faster, and the search for them consumes inference and execution resources (Deng et al., 2026; Gai et al., 2026). A workflow that buys a modest runtime improvement with a much larger search may be a poor choice, so a component has to be judged by the speed of the programs it helps produce and by the cost of finding them (Kapoor et al., 2025).

Recent systems guide this search in several ways. *Feedback analysis* turns execution measurements into guidance for the next edit; PerfAgent uses profiling evidence to locate bottlenecks and direct iterative refinement (Deng et al., 2026). *Delegation* assigns code generation to a separate agent while a lead agent decides what to optimize, and *history* makes earlier attempts available to later decisions; STARK combines the two, separating planning, coding, and debugging roles and rebuilding agent contexts from candidates and outcomes stored in a search tree (Dong et al., 2026a). *Tools and skills* supply measurements and reusable optimization methods; KernelSkill retrieves expert techniques to guide its planner (Sun et al., 2026).

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.

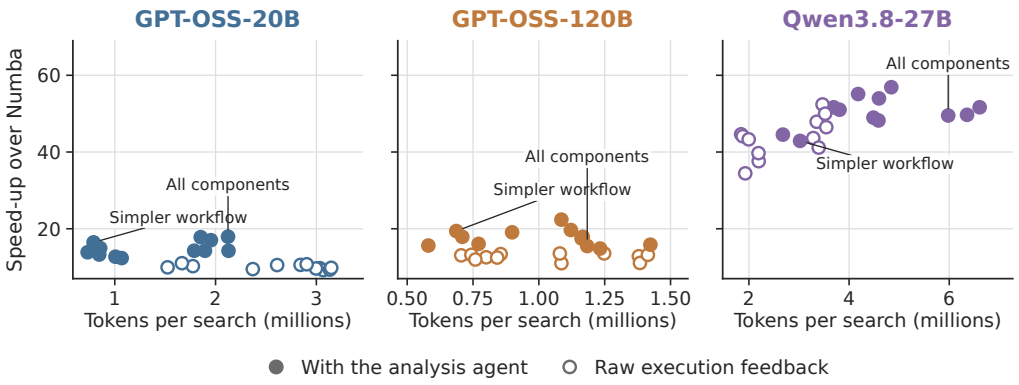

Figure 1: No single choice of components wins on both speed and cost. Each point is one of the 24 workflows on the full suite: geometric-mean speed-up over Numba of its selected programs against the mean tokens spent per search. Filled points use an analysis agent to interpret execution results; open points pass raw results to the main agent. *All components* enable every option; the *simpler workflow* uses direct coding, analysis, and history, without tools and skills (Section 2).

These benefits need not add up. An analysis report can repeat what history retrieval already supplies. Delegation adds a handoff: the lead must decide what to pass to the coding agent and how to use its response. Tools expose measurements that later calls must interpret. Studies of coding harnesses already show that the value of context management, planning, and tool selection depends on the model and task (Fan et al., 2026). In code optimization, the speed of the generated program and the cost of the search must be counted as well.

A component, therefore, cannot be judged in isolation: its value depends on which other components are present and on the model that drives them, so they must be varied together, under one evaluator and one budget. We ask three questions about individual components in the presence of the others, and one about complete workflows. How does analyzed feedback, which also rebuilds the lead's context at every turn, affect program speed and token cost, and does delegation change that trade-off (**RQ1**)? Once feedback is analyzed, what do tools and skills add in speed and in tokens (**RQ2**)? Does history help equally when the lead writes the code and when it delegates (**RQ3**)? How do complete workflows trade program speed against token cost, and does enabling every component improve that trade-off (**RQ4**)?

To answer them, we build a framework that combines four design choices into 24 workflows under one task contract, evaluator, and turn budget. The same controller generates, validates, and benchmarks every candidate while varying who writes the code and what information reaches the next attempt, and it records token cost by role, lead occupancy, and search time. We run all 24 workflows on the 54 CPU kernels of NPBench (Ziogas et al., 2021) with three models, GPT-OSS-20B, GPT-OSS-120B (OpenAI, 2025), and Qwen3.8-27B, which we abbreviate as 20B, 120B, and Qwen3.8. Smaller fixed panels test transfer to Qwen3.5-27B, Qwen3.5-397B, and Kimi-K2.7-Code (Kimi).

The results show why these trade-offs are hard to anticipate. The 24 workflows spread widely in both program speed and token cost (Figure 1), and adding components does not reliably move a workflow toward faster code or lower cost. Three findings summarize the study.

First, what the lead receives between attempts matters for every model. Across both coding modes, analyzed feedback produces faster programs for all three full-suite models: 1.46× for 20B, 1.38× for 120B, and 1.15× for Qwen3.8; on the smaller transfer panels, it also helps both Qwen3.5 models and Kimi's direct coding. Its token cost depends on the model. For GPT-OSS direct coding, analysis also lowers token use: the lead's context stays small, whereas with raw feedback it fills up and has to be truncated. Searches still take 30–62% longer on the 28-kernel panel because analysis adds output tokens and model-call time. Qwen3.8 gains more from analysis under delegation than under direct coding, but under delegation its token use triples.

Second, enabling every component is not the most efficient choice. On the 28-kernel panel, direct coding with analyzed feedback and history, without tools and skills, uses 59%, 35%, and 52%

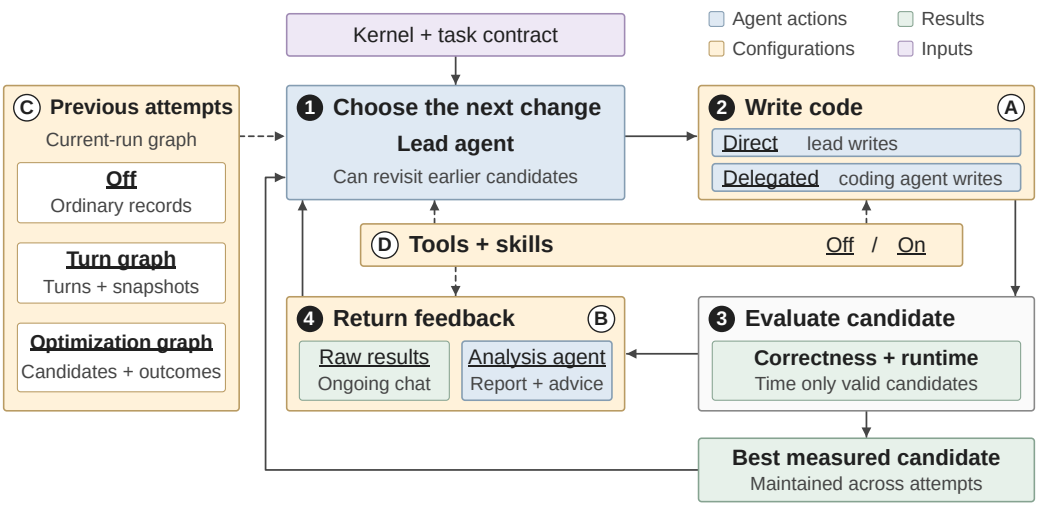

Figure 2: The optimization loop of our framework. Filled circles identify workflow steps; outlined circles identify configuration choices. Blue marks agents, green results, purple inputs, and yellow configuration groups. Underlined labels identify selectable settings. Solid arrows carry candidates and execution evidence; dashed arrows show optional history recall and access to tools and skills.

fewer tokens than the complete stack for 20B, 120B, and Qwen3.8, and the complete stack is not significantly faster for any of them. Tools and skills add tokens without a significant direct-coding speed-up on this panel.

Third, the remaining choices depend on the model. History helps GPT-OSS direct coding more than delegated coding, but Qwen3.8 does not show this interaction. Delegation does not make GPT-OSS programs significantly faster, yet it makes Qwen3.5-397B programs 2.29× faster than direct coding, at more than three times the tokens, and Kimi programs 1.13× faster. A workflow chosen for one model therefore has to be tested again when the model changes.

2 Design

Our framework runs an iterative optimization loop around a lead agent. Figure 2 follows one turn through steps ❶–❹; choices (A)–(D) set who writes the code, how evaluation results reach the lead, how earlier attempts are revisited, and whether tools and skills are available.

The lead chooses an optimization and an implementation to build on ❶, using the task requirements, its plan and notes, and the evidence from earlier attempts. Choice (A) decides who writes the candidate. In *direct coding*, the lead writes it; under *delegation*, it hands the change to a coding agent, which can return up to three alternative implementations ❷. The coding agent receives a focused request, a starting implementation, and the relevant task and plan information, so implementation happens in its own context.

Every candidate reaches the same evaluator ❸, which checks compilation and correctness before timing. A rejected candidate may enter a bounded repair loop; each submission or delegated request allows up to five evaluations, repairs included. The harness keeps the fastest validated candidate as the result, but the lead may continue from any earlier implementation.

Choice (B) decides how evaluation results return to the lead ❹. With *raw feedback*, results and diagnostics are appended to the lead's ongoing conversation. With *analyzed feedback*, an analysis agent reads the code and execution evidence and writes a report with suggested next steps, and the harness rebuilds the lead's next prompt from the task, plan, notes, accumulated reports, and history. Candidate source, raw timing arrays, and evaluator logs are not copied into the rebuilt prompt; the lead can open saved candidates upon request. The analysis report carries forward what the lead would otherwise read in raw logs, and the plan and notes carry the lead's own reasoning. Together,

they let each turn start from a compact prompt. The two parts are designed as one mechanism: a compact prompt without the report would drop the execution evidence, and a report without the rebuild would be appended to a conversation that keeps growing.

Choice (C) sets how the lead can revisit earlier work when the loop returns to (1). The plan, notes, and evaluation records persist in every setting; with history *off*, nothing else is added. A *turn graph* adds recall of completed turns and their saved workspace contents; an *optimization graph* organizes the evidence around evaluated implementations, their outcomes, and analyzes. Both graphs support retrieval and inspection within the current search.

Choice (D) enables *tools and skills* for planning, coding, and interpretation. Tools supply information such as web results and performance profiles; skills provide role-specific instructions and optimization guidance. The two are enabled together because skills describe how to use the tools: skills without their tools would give the agent instructions it cannot act on. With both disabled, the agents retain the operations needed to plan, submit, and evaluate.

These choices give $2 \times 2 \times 2 \times 3 = 24$ configurations. We name them by code: `L-A1-T0-opt`, for example, is direct coding (`L`; `M` is delegated) with analyzed feedback (`A1`; `A0` is raw), tools and skills off (`T0`; `T1` is on), and the optimization graph (`opt`; also `turn` on and `off`; Appendix A). Appendix K details the implementation.

3 Experimental Setup

Models, workloads, and validation. We evaluate the 24 configurations on the 54 NPBench CPU kernels with GPT-OSS-20B, GPT-OSS-120B, and Qwen3.8-27B, the three models with a full configuration sweep. Fixed subsets additionally compare Qwen3.5-27B, Qwen3.5-397B, and Kimi-K2.7-Code (Section 4.5). Each model–kernel–configuration combination receives one search, with a configured budget of ten lead turns and up to five evaluations per submission or delegated request, including repairs, in both coding modes. The agents write C++20 implementations, which must match NPBench's NumPy reference (Harris et al., 2020) on returned values and modified arrays within task-specific elementwise or relative-norm tolerances; kernels that modify their inputs also receive a deterministic perturbed-input check. An AddressSanitizer audit of every model's selected programs found two GPT-OSS-120B programs that violate the memory contract; their timings are excluded (2 of its 1,296 searches; Appendix B.3).

Baseline and timing. We measure speed-up relative to NPBench's Numba implementation (Lam et al., 2015), using its parallel variant where available and its nopython fallback otherwise. Numba compiles the baseline through LLVM; candidates are compiled with GCC 16.2.0, and both run on the same eight physical CPU cores. Each timed call receives fresh inputs whose cache lines are flushed first, and runtime is measured over 10–100 calls after adaptive warmup, excluding compilation and validation. During a search, the harness keeps the validated candidate with the highest ratio of median baseline to median candidate runtime, and we report that candidate. The baseline runtime varies slightly across searches, so for each kernel we pool the compatible Numba measurements into one mean baseline and divide it by the candidate's mean runtime; every configuration of a kernel thus has the same reference. Selection uses medians, which are robust to occasional slow calls, and reporting uses means, which reflect the total time of repeated calls; recomputing the full-suite and 28-kernel analysis comparisons with the selection medians preserves their interval classifications (Table 6). Appendix B specifies warmup, timeouts, and pooling.

Hardware and serving. Searches are run on nodes with four AMD Instinct MI300A devices and 96 physical CPU cores. Agent workers evaluate candidates on disjoint eight-core allocations, while model requests go to separate inference servers. Serving topology and concurrent load differ by model (Appendix A.1), so we compare search times only within a model.

Comparisons and metrics. A component comparison matches searches with the same model, kernel, and other choices, changing only that component. For each pair in which both selected candidates have usable timings, we divide one speed-up by the other; a ratio of 2 means that the first setting's program runs in half the time. Ratios are averaged geometrically within each kernel and then across kernels, so that every kernel receives equal weight. Failed or unmeasured searches

remain in the coverage counts but receive no speed-up. Token cost counts the input and output tokens of all agents, including repeated prompts and failed attempts. Search time covers model calls, evaluation, and waiting; for searches that ran in several segments, we sum the segments and exclude the gaps between them. Token and time ratios divide totals over the matched pairs that have both timings and the metric. Lead occupancy is the fraction of the context window filled by the lead's prompt, averaged over the lead's requests in a search and then over searches and kernels.

Each speed-up ratio has a 95% interval, obtained by resampling the kernels with replacement 5,000 times. The interval shows how much the result depends on which kernels are included; it does not capture variation between repeated runs of the same search (Appendix E.4). We call an effect *significant* when this interval excludes 1, so that its direction does not depend on the kernel sample; otherwise, we report the estimate without claiming an effect.

4 Evaluation

4.1 Coverage and workload selection

Across all 24 configurations, 89.9% of 20B searches, 90.7% of 120B searches, and 93.9% of Qwen3.8 searches produce an implementation that passes the correctness checks; on the 28-kernel panel defined below, the rates are 99.0%, 99.7%, and 99.9% (Appendix C).

Not every kernel distinguishes the configurations. For each GPT-OSS model, we keep the kernels that have usable timings for at least 20 of the 24 configurations and on which the configuration speed-ups differ by more than 25% on average (Appendix C). This selects 28 kernels for 20B and 23 for 120B; their union, our fixed 28-kernel panel, equals the 20B selection. The panel is chosen from the GPT-OSS results only and is held fixed for Qwen3.8, which therefore tests it on held-out outcomes. Applied to Qwen3.8 alone, the same rule would select 25 kernels, 19 of them on the panel; the other six are `adist`, `cavtflow`, `crc16`, `mlp`, `nussinov`, and `sselfeng`.

The cutoff is sign-neutral: it keeps kernels on which the configurations differ, whichever configuration is faster. It does favor large effects, and analysis is the largest factor for GPT-OSS, so the panel inflates the GPT-OSS analysis effect: under direct coding, from 1.51× on the full suite to 1.79× for 20B and from 1.47× to 1.71× for 120B (Section 4.2). In exchange, the panel resolves the smaller components better, since kernels on which every configuration reaches nearly the same speed-up would pull every ratio toward 1. We report the analysis comparison on both the full suite and the panel, as well as the other comparisons on the panel. Appendix D gives the full-suite results.

4.2 Analyzed feedback produces faster code, with different token costs (RQ1)

Across both coding modes, analyzed feedback makes full-suite programs faster for all three models: 1.46 for 20B, 1.38 for 120B, and 1.15 for Qwen3.8, over 497, 501, and 555 pairs across 42, 43, and 48 kernels. Its token cost differs: total tokens change by factors of 0.54, 1.03, and 1.73.

On the full suite, every configuration with analyzed feedback reaches a higher geometric-mean speed-up than every configuration without it, for both GPT-OSS models (Figure 1; Table 5): the lowest `A1` values are 12.4 and 14.8, and the highest `A0` values are 11.0 and 13.6. The ordering also holds on the 37 and 38 kernels timed under all 24 configurations. Under direct coding, with the other choices matched, analyzed feedback makes programs 1.51× times faster for 20B and 1.47× times faster for 120B over the 42 and 43 kernels with paired timings, improves 36 and 38 of them, and uses 55% and 25% fewer tokens (Table 6). On the 23 panel kernels selected for both GPT-OSS models, the effect is 1.86× and 1.89×, against 1.79× and 1.71× on the whole panel (Figure 3), so taking the union does not inflate it.

The selected programs differ in substance. On `conv2d` with 120B, direct coding with raw feedback and the optimization graph ends with a program that revisits the input patch for each output channel (16×); with analyzed feedback, the program accumulates channels together and reuses each weight vector across a 4×4 tile of outputs (570×; Appendix J).

The lead's context explains the GPT-OSS token savings. Under direct coding on the panel, mean lead prompt size falls from about 75K and 70K tokens to 11K and 12K (Figure 3). With raw feedback and tools off, the harness had to shorten a conversation to fit the window in 67% and 79% of direct-

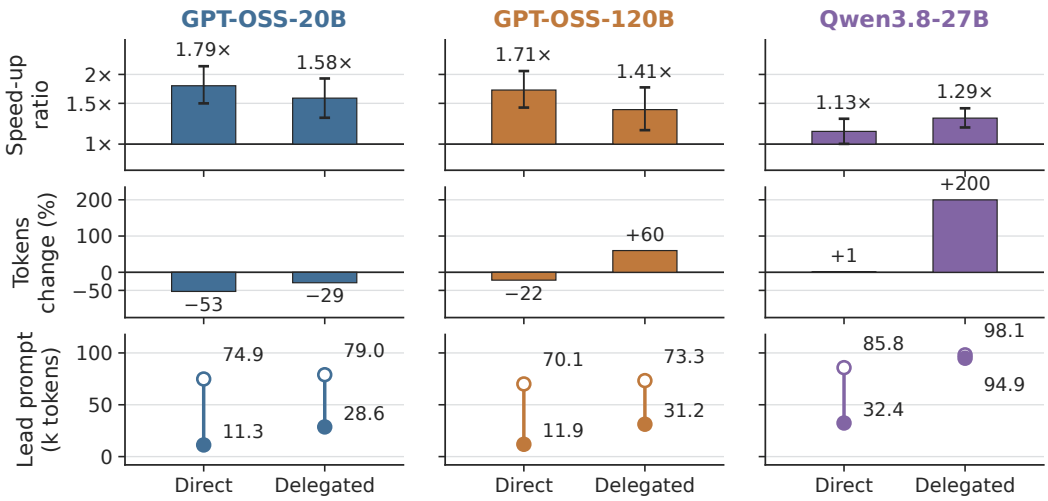

Figure 3: Effect of analyzed feedback on the 28-kernel panel, with other choices matched. Speed-up ratios aggregate kernels equally; token changes count all roles. Whiskers show 95% kernel-resampling intervals. Lead prompt size is in thousands of tokens, allowing comparison across different context windows; open circles show raw feedback and filled circles analyzed feedback.

coding searches with recorded context events, against one of 168 searches with analyzed feedback (Appendix G.2). With output tokens weighted four times the input, the direct-coding saving is about 29% for 20B and 3% for 120B (Appendix I). Searches take 30% and 62% longer. On matched timing records, model-call time accounts for over 90% of the increase, while evaluator time rises by only 4% and 1%: analysis adds 60% and 132% more output tokens, which are generated sequentially.

Qwen3.8 draws its gain mainly from delegated coding. Under direct coding, its ratio is 1.05 on the full suite, with 5% more tokens and faster programs on 30 of 48 kernels, and 1.13 on the panel, with nearly unchanged token use; its analyzed and raw configurations overlap in Figure 1. Under delegation, analysis makes its programs 1.29× faster on both the full suite and the panel, at about three times the tokens. The mechanism is the same as for GPT-OSS, with a different balance: under direct coding, the mean lead prompt falls from 86K to 32K tokens and, with tools off, context cleanups fall from 75 of 84 raw-feedback searches to 17 of 84 analyzed-feedback searches, but its output tokens rise by 92% and offset the input savings.

Under delegated coding on the panel, analyzed feedback still yields faster programs for GPT-OSS (1.58× and 1.41×), but total tokens fall by 29% for 20B and rise by 60% for 120B: the lead's prompt shrinks while the analysis, coding, and repair roles consume more. Delegation alone does not make programs faster: with analyzed feedback, the direct/delegated speed ratios are 1.01, 1.19, and 1.04 for 20B, 120B, and Qwen3.8, and none is significant. With raw feedback, Qwen3.8 favors direct coding, 1.19.

4.3 Tools, skills, and history (RQ2, RQ3)

RQ2: Tools and skills add tokens without a significant gain on the panel. With analyzed feedback and direct coding, enabling tools and skills raises token use by 134%, 80%, and 36% for 20B, 120B, and Qwen3.8, and none of the speed-up ratios, 1.03, 1.07, and 1.08, is significant (Figure 4a). Under delegation, token use rises by 108%, 24%, and 40%; the 20B ratio of 1.23 is significant, while the 120B ratio of 0.79 and the Qwen3.8 ratio of 0.98 are not. The conclusion is specific to the panel: on the full suite, tools do make Qwen3.8 direct coding faster, 1.13.

RQ3: History depends on both coding mode and model. With analyzed feedback and tools off, the optimization graph raises the observed GPT-OSS direct-coding speed-up (1.32× and 1.16×) and lowers the delegated one (0.81× and 0.69×); of these four effects, only the 120B drop is significant. RQ3 concerns the difference, which the interaction measures directly. On kernels with timings for

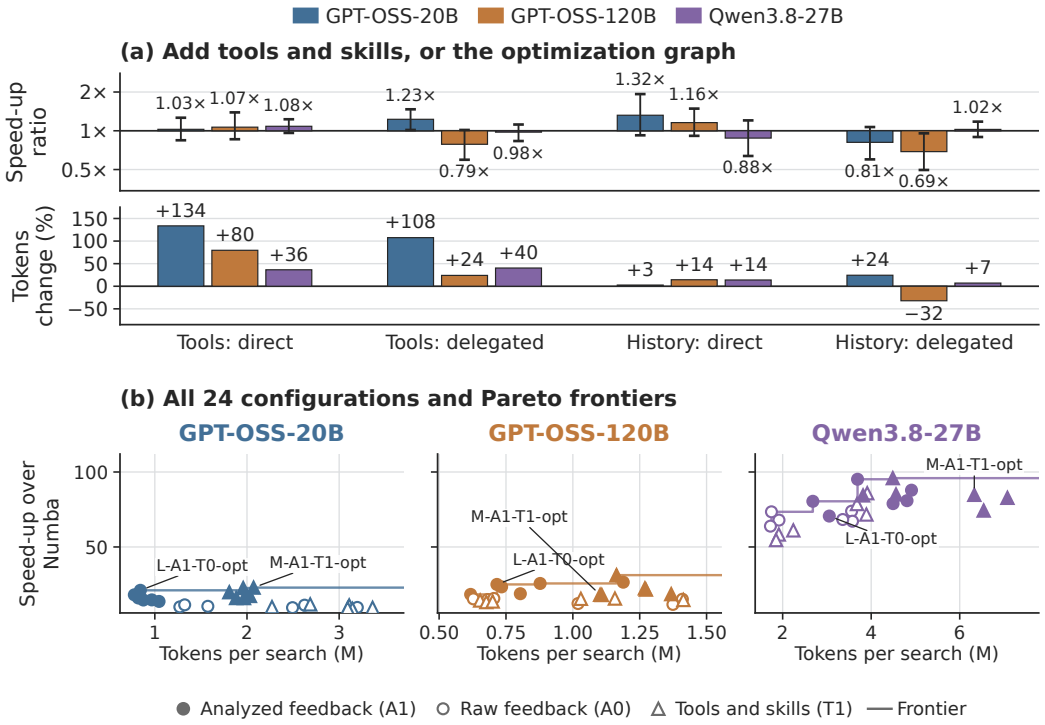

Figure 4: (a) Tools and skills on/off with analyzed feedback, and optimization-graph history/off with analyzed feedback and tools off, on the 28-kernel panel. Whiskers show 95% kernel-resampling intervals. (b) All 24 configurations: geometric-mean speed-up over timed kernels versus mean tokens per search. Filled markers use analyzed feedback, triangles tools and skills; lines mark the observed frontiers. Twelve of 2,016 cells lack a timing. Frontiers are descriptive because timed populations can differ; Appendix D gives matched-population checks.

all four configurations, the direct-coding history ratio divided by the delegated one is 1.66 for 120B on 26 kernels, a clear difference, and 1.62 for 20B on 28 kernels, a suggestive one. Qwen3.8 does not show this interaction: its history ratios are 0.88 for direct and 1.02 for delegated coding, and their ratio is 0.86. Appendix F compares the turn and optimization graphs.

4.4 The cost of the complete stack (RQ4)

Figure 4b places all 24 workflows by token cost and speed-up on the 28-kernel panel; the line marks the frontier, the cheapest workflow that reaches each speed-up. For both GPT-OSS models, every workflow on the frontier uses the analysis agent, and workflows with tools and skills appear only at its expensive end. Among the cheapest frontier workflows is `L-A1-T0-opt`: the lead writes the code itself, an analysis agent interprets the results, the optimization graph stores earlier attempts, and tools and skills are off. Qwen3.8's frontier is more mixed. Its cheap end consists of delegated workflows that pass raw results to the lead, and its fast end of direct-coding workflows with the analysis agent, the fastest of which also uses tools.

We call `L-A1-T0-opt` the simpler workflow and compare it with the complete stack, `M-A1-T1-opt`, which adds the two remaining components: delegated coding and tools and skills. Over the 28 searches of each workflow, the complete stack uses 146%, 54%, and 107% more tokens for 20B, 120B, and Qwen3.8. The extra tokens do not buy significantly faster programs. On the 28-kernel panel, the simpler workflow's programs run 1.06, 1.38, and 0.83 times as fast as the complete stack's, and every interval includes 1. Only for Qwen3.8 does the point estimate favor the complete stack, which produces 1.20× faster programs at about twice the tokens.

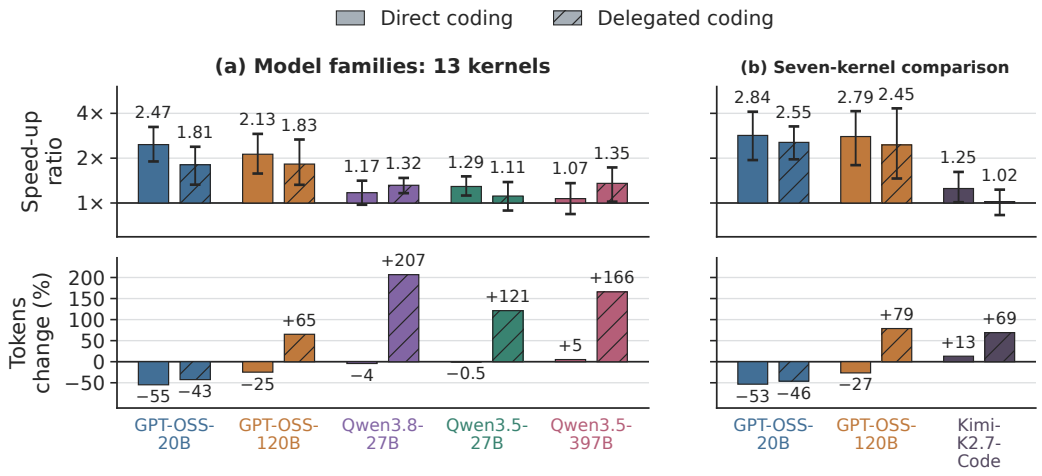

Figure 5: Analyzed versus raw feedback for direct and delegated coding: speed-up ratio and change in total tokens. (a) Fixed 13-kernel model comparison. (b) Seven kernels selected for the analysis effect. Each model uses the same kernels as its GPT-OSS references; other choices are matched. Labels give the estimates; negative token changes indicate savings. Whiskers show 95% kernel-resampling intervals.

Model size does not predict cost either. Across the full suite, 20B spends twice the tokens of 120B per search (2.00M against 1.01M), mostly because it generates three times as many output tokens (0.29M against 0.09M).

4.5 Transfer across model families and scales

To test whether the effects transfer, we run Qwen3.5-27B, Qwen3.5-397B, and Kimi-K2.7-Code on smaller fixed panels (Figure 5). The 13-kernel panel is the union of three seven-kernel selections, targeting the analysis effect, the tools effect with analysis on, and the interaction between history and coding mode; Kimi runs only the seven analysis-selected kernels. All panels are selected from GPT-OSS outcomes and held fixed across models (Appendix C).

On the 13-kernel panel, analyzed feedback again makes GPT-OSS direct coding faster while saving tokens, with larger gains on these effect-selected kernels (Figure 5a). Across coding modes, both Qwen3.5 models also benefit.

The coding mode that carries the gain differs. Qwen3.5-27B benefits under direct coding with nearly unchanged tokens, whereas Qwen3.5-397B benefits under delegation at a substantial token increase. Qwen3.8 and Qwen3.5-397B show no significant direct-coding gain.

On the seven analysis-selected kernels, Kimi benefits from analysis under direct coding with a modest token increase (Figure 5b). Under delegation, analysis adds substantially more tokens without a significant speed-up.

Delegation pays off for some models. With analyzed feedback, delegation substantially helps Qwen3.5-397B, while Qwen3.5-27B shows no significant benefit. The difference remains significant on matched kernels. Kimi also benefits from delegation; both models pay a large token increase. Across matched settings, Qwen3.5-397B produces faster programs than Qwen3.5-27B while using fewer tokens.

Other components also vary (Appendix E.3). With analyzed direct coding and tools off, optimization-graph history helps Qwen3.5-27B at little extra token cost, but adds cost without a significant gain for Qwen3.5-397B. Tools and skills make Qwen3.5-397B's direct-coding programs slower while costing more tokens. Its delegation benefit therefore does not justify enabling every component.

5 Related Work

Code optimization and feedback. PIE learns performance-improving C++ edits (Shypula et al., 2024); PerfCoder trains optimization strategies for direct generation or planner–optimizer workflows (Yang et al., 2025). AlgoTune studies budgeted numerical CPU optimization (Press et al., 2025); GSO, SWE-Perf, and SWE-fficiency evaluate repository optimization (Shetty et al., 2025; He et al., 2025; Ma et al., 2025), and KernelBench evaluates GPU kernels (Ouyang et al., 2025). AlphaEvolve and its open counterpart, OpenEvolve, evolve programs through LLM-proposed edits scored by automated evaluators (Novikov et al., 2025; Sharma, 2025). Reflexion and Self-Refine feed critiques of earlier outputs into later attempts (Shinn et al., 2023; Madaan et al., 2023). For optimization, AutoGuide translates execution evidence into parallel-program mapping guidance (Wei et al., 2025), and KernelPro converts profiling into advice; its ablation finds raw counters worse than no profiling feedback (Gai et al., 2026). PerfAgent combines profiler summaries, refinement, and selective verification, outperforming an oracle best-of-five baseline at lower cost (Deng et al., 2026). An expert-assisted workflow guides GPU optimization through skills, workload context, and history (Shui et al., 2026). Rather than proposing another feedback mechanism, we study how the benefits and costs of feedback analysis change with coding mode, tools and skills, and history.

Coordination and history. KernelBlaster transfers optimization knowledge across CUDA tasks (Dong et al., 2026b); KernelArc shares conclusions among parallel agents and compares coordination and memory under a fixed candidate budget (Kundu et al., 2026). ContextEvolve assigns summarization, optimization direction, and example selection to specialist agents, evaluating quality and token use (Su et al., 2026). Our graphs retrieve turns or candidate implementations of the current search, ordinary records remain when the graphs are disabled, and our delegation is serial, which separates our setting from cross-task transfer and parallel sharing. Where ContextEvolve ablates individual agents from its full system, we cross history with feedback, coding mode, tools, and skills, and can therefore test whether history that helps direct coding remains useful under delegation.

Harnesses, context, and budgets. SWE-agent shows that the agent–computer interface shapes software-engineering performance (Yang et al., 2024), and Agentless obtains competitive repairs with a fixed workflow (Xia et al., 2025). Fan et al. cross context strategies and window budgets across four models and two coding benchmarks, but they evaluate planning and tools in one context setting (Fan et al., 2026). Language models use information in the middle of long inputs poorly (Liu et al., 2024), and the Complexity Trap finds that observation masking can match generated summaries for agent context management (Lindenbauer et al., 2025); other studies examine harness-level token differences and capability-dependent coordination (Vats & Golev, 2026; Kim et al., 2025). ACCLAIM studies compiler–agent cooperation and compute allocation (Mikek et al., 2026); ARTEMIS evolves agent configurations, including on SWE-Perf (Brookes et al., 2025), and GEPA explores reflective prompt and kernel optimization (Agrawal et al., 2026). These studies establish that harness effects depend on context. We examine those dependencies for program optimization through all 24 combinations of four workflow choices, measuring achieved speed-up together with token cost by role.

6 Conclusion

We asked whether more machinery makes a code-optimization agent write faster code, testing 24 workflows on six models under one protocol, and found that it does not systematically do so: the most consistent gains come from better feedback, not a larger agent stack. The one choice that helps every full-suite model and both Qwen3.5 models is an analysis agent that turns execution results into concise guidance for the next attempt; for GPT-OSS direct coding, it also cuts token use by 25–55%. This makes what the agent learns from one attempt and carries into the next the most consistent design principle we observe.

The other choices depend on the model and often cost more. Delegation helped only Qwen3.5-397B (2.29×) and Kimi (1.13×), at roughly three times the tokens. Tools and skills raise direct-coding token use by 36–134% for speed-up ratios of 1.03–1.08, none significant, and enabling every component costs 54–146% more tokens than the simpler workflow without being significantly faster. We therefore recommend starting from analyzed feedback, choosing the coding mode for the model, and adding tools last. The broader lesson is that, for code optimization, better feedback matters more than more machinery: an agent that learns better from each attempt can write faster code without needing the largest possible agent stack.

AI USE STATEMENT

We used generative AI tools to assist with writing, editing, and code implementation under the author's direction and with the author's review of every output. AI tools drafted code from author-defined specifications, including the harness, the analysis scripts, and plotting scripts. They also helped improve the grammar and flow of author-written text. We did not use generative AI to originate research ideas or determine the interpretation of the results. The authors take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We will provide the harness code as a zipped artifact with this ICLR submission. The artifact contains the runnable harness and the configurations of all 24 workflows and model serving, supporting reruns in a compatible environment. The paper and appendices document the experimental setup: Sections 2–3 and Appendices A–C specify the configurations, serving, evaluation protocol, implementation audit, and kernel selection; Appendix E.4 defines the estimators; and Appendix K documents the harness.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *International Conference on Learning Representations*, 2026:8479–8565, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0e9e708b6f48e14fd0ac29e167413f76-Abstract-Conference.html.

Paul Brookes, Vardan Voskanyan, Rafail Giavrimis, Matthew Truscott, Mina Ilieva, Chrystalla Pavlou, Alexandru Staicu, Manal Adham, Will Evers Hood, Jingzhi Gong, Kejia Zhang, Matvey Fedoseev, Vishal Sharma, Roman Bauer, Zheng Wang, Hema Nair, Wei Jie, Tianhua Xu, Aurora Constantin, Leslie Kanthan, and Michail Basios. Evolving Excellence: Automated Optimization of LLM-based Agents, December 2025. URL http://arxiv.org/abs/2512.09108.

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, July 2026. URL http://arxiv.org/abs/2607.19653.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. STARK: Strategic Team of Agents for Refining Kernels. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=nWaZTH1JMx.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. KernelBlaster: Continual Cross-Task CUDA Optimization via Memory-Augmented In-Context Reinforcement Learning, February 2026b. URL http://arxiv.org/abs/2602.14293.

Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, and Xiaoyang Wang. An Empirical Study of Harness Design for Coding Agents, September 2026. URL http://arxiv.org/abs/2609.20804.

Jiading Gai, Shuai Zhang, Kaj Bostrom, Jin Huang, Vihang Patil, Haoyang Fang, Bernie Wang, Huzefa Rangwala, and George Karypis. Optimizing CUDA like a Human: Micro-Profiling Tools as Expert Surrogates for LLM-Based GPU Kernel Optimization, June 2026. URL http://arxiv.org/abs/2606.26453.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane,

Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-2649-2. URL https://www.nature.com/articles/s41586-020-2649-2.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories?, July 2025. URL http://arxiv.org/abs/2507.12415.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI Agents That Matter. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=Zy4uFzMviZ.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a Science of Scaling Agent Systems, December 2025. URL http://arxiv.org/abs/2512.08296.

Joyjit Kundu, Ben Stoffelen, Kaili Wang, Peter Vrancx, and Ludovic Denoyer. KernelArc: A Multi-Agent Framework for GPU Kernel Optimization, August 2026. URL http://arxiv.org/abs/2608.17071.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pp. 1–6, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-4005-2. doi: 10.1145/2833157.2833162. URL https://dl.acm.org/doi/10.1145/2833157.2833162.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management, August 2025. URL http://arxiv.org/abs/2508.21433.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

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?, November 2025. URL http://arxiv.org/abs/2511.06090.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Benjamin Mikek, Danylo Vashchilenko, Bryan Lu, and Panpan Xu. Agentic Code Optimization via Compiler-LLM Cooperation, April 2026. URL http://arxiv.org/abs/2604.04238.

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, June 2025. URL http://arxiv.org/abs/2506.13131.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, August 2025. URL https://openai.com/index/gpt-oss-model-card/.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs Write Efficient GPU Kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, Nathanael Bosch, Eli Meril, Albert Steppi, Arman Zharmagambetov, Fangzhao Zhang, David Pérez-Piñeiro, Alberto Mercurio, Ni Zhan, Talor Abramovich, Kilian Lieret, Hanlin Zhang, Shirley Huang, Matthias Bethge, and Ofir Press. AlgoTune: Can Language Models Speed Up General-Purpose Numerical Programs? *Advances in Neural Information Processing Systems*, 38, 2025. doi: 10.52202/085713-3585. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/9a648e8e1014c2427156dcb5465cd488-Abstract-Datasets_and_Benchmarks_Track.html.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. https://github.com/codelion/openevolve, May 2025. URL https://github.com/codelion/openevolve.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging Software Optimization Tasks for Evaluating SWE-Agents. *Advances in Neural Information Processing Systems*, 38, 2025. doi: 10.52202/085713-5054. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/ddd98389004161943e646322a16348c3-Abstract-Datasets_and_Benchmarks_Track.html.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Yue Shui, Chenyu Ma, Hangfei Xu, Shengzhao Wen, and Yanpeng Wang. Harness Engineering for LLM-Driven GPU Kernel Generation, July 2026. URL http://arxiv.org/abs/2607.17979.

Alexander G. Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

Hongyuan Su, Yu Zheng, and Yong Li. ContextEvolve: Multi-Agent Context Compression for Systems Code Optimization, February 2026. URL http://arxiv.org/abs/2602.02597.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. KernelSkill: A Multi-Agent Framework for GPU Kernel Optimization, March 2026. URL http://arxiv.org/abs/2603.10085.

Naman Vats and Oleg Golev. The Scaffold Effect in Coding Agents: Harness Choice as a Hidden Variable in Coding-Agent Evaluation, July 2026. URL http://arxiv.org/abs/2607.22585.

Anjiang Wei, Allen Nie, Thiago S. F. X. Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. Improving Parallel Program Performance with LLM Optimizers via Agent-System Interfaces. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 66155–66177. PMLR, 2025. URL https://proceedings.mlr.press/v267/wei25j.html.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-Based Software Engineering Agents. *Proceedings of the ACM on Software Engineering*, 2(FSE): FSE037:801–FSE037:824, 2025. doi: 10.1145/3715754. URL https://dl.acm.org/doi/10.1145/3715754.

Jiuding Yang, Shengyao Lu, Hongxuan Liu, Shayan Shirahmad Gale Bagi, Zahra Fazel, Tomasz Czajkowski, and Di Niu. PerfCoder: Large Language Models for Interpretable Code Performance Optimization, December 2025. URL http://arxiv.org/abs/2512.14018.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. NPBench: A benchmarking suite for high-performance NumPy. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, pp. 63–74, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8335-6. doi: 10.1145/3447818.3460360. URL https://dl.acm.org/doi/10.1145/3447818.3460360.

A CONFIGURATION NOTATION AND EXPERIMENTAL SETTINGS

Configurations use the code `coding-analysis-tools-history` introduced in Section 2. The first field is `L` for lead-only (direct) coding or `M` for delegated coding. The binary fields `A0/A1` and `T0/T1` select raw or analyzed feedback and disable or enable tools and skills, respectively. The final field selects history `off`, the `turn` graph, or the `opt`imization graph, as defined in Table 1. These four fields correspond to the choices in Section 2 and give $2 \times 2 \times 2 \times 3 = 24$ configurations.

Table 1: Configuration-code key, in field order. A binary value of 1 enables the component and 0 disables it. Tools and skills change together. The history field has three values.

Field	Code	Meaning
Coding	`L`	Lead-only (direct) coding
	`M`	Delegated coding
Analysis	`A0` / `A1`	Raw feedback / analyzed feedback
Tools and skills	`T0` / `T1`	Disabled / enabled
History	`off`	Ordinary current-run records; no graph
	`turn`	Turn graph
	`opt`	Optimization graph

For example, `L-A1-T0-off` uses direct coding with analyzed feedback, tools and skills disabled, and history off. `M-A0-T1-turn` uses delegated coding with raw feedback, tools and skills enabled, and a turn graph.

A.1 MODEL SERVING AND EXECUTION SETTINGS

All roles within a search use the same model. GPT-OSS-20B and GPT-OSS-120B are served with vLLM on separate inference nodes with four MI300A GPUs each, using tensor parallelism across the four GPUs. Each server supports up to 128 concurrent sequences and a context window of 131,072 tokens. The Qwen and Kimi models use SGLang with a context window of 262,144 tokens; Table 2 gives their inference topology.

For GPT-OSS, each of the 24 configurations runs on its own agent node. An agent node runs up to 15 searches at a time and evaluates candidates in nine slots of eight physical cores each; an evaluation that finds no free slot waits. Model requests go to separate inference servers: four 20B agent nodes share one server, and two 120B agent nodes share one. Qwen3.8 uses up to 30 direct-coding or 25 delegated searches per configuration. The transfer campaigns group four configurations on an agent node, each with up to seven searches, and share an inference allocation. The serving load differs between models; we therefore compare search times only within a model.

Searches allow up to ten lead turns and up to five evaluations per submission or delegated request, including repairs, in both coding modes, with no early stopping based on a target speed-up. Each model request permits at most 10,000 output tokens for the lead, 4,000 for analysis, and 6,000 for coding or repair. These are per-request limits; a role can make several requests during a search. All models use a temperature of 0.7.

Table 2: Model-serving settings. GPUs counts inference GPUs per server allocation; TP/PP denotes tensor/pipeline parallelism. Context is in tokens. The Qwen models are served from FP8 checkpoints.

Model	Engine	GPUs	TP/PP	Context
GPT-OSS-20B	vLLM	4	4/1	131,072
GPT-OSS-120B	vLLM	4	4/1	131,072
Qwen3.8-27B	SGLang	4	4/1	262,144
Qwen3.5-27B	SGLang	4	4/1	262,144
Qwen3.5-397B	SGLang	16	16/1	262,144
Kimi-K2.7-Code	SGLang	16	4/4	262,144

B NATIVE EVALUATION AND IMPLEMENTATION AUDIT

B.1 INPUTS, COMPILATION, AND VALIDATION

The experiment uses NPBench's `paper` preset. Its initializer supplies each kernel's arguments, and its NumPy implementation defines the expected returned values and modified arrays. Candidates implement the same task in C++20 and are compiled with GCC 16.2.0 using `-O3 -march=native -fPIC -shared -std=c++20 -fno-fast-math -fopenmp`. Baseline and candidate run sequentially on the same eight physical CPU cores: the baseline measurement batch finishes before the candidate batch begins. Concurrent searches share an agent node but receive disjoint core allocations. The repeated-call dispersion is reported below; disjoint core allocations prevent direct core sharing, but do not isolate shared memory or system-level contention.

Validation compares every observable output with the reference. A numerical output is accepted when either the elementwise comparison passes with $\mathrm{rtol} = 10^{-5}$ and $\mathrm{atol} = 10^{-8}$, or its relative norm error is below the task-specific threshold. Kernels that modify their inputs also receive a deterministic perturbed-input check, which verifies the same mutation contract. Compilation, loading, and validation must succeed before candidate timing is accepted.

The Numba baseline is checked against NumPy as well. Consequently, a correct candidate may still lack a speed-up if the baseline fails validation. This occurs in the GPT-OSS measurements for `adi`, `durbin`, `mlp`, and `seidel2d`. For Qwen3.8, the parallel Numba baseline fails in every configuration for `adi`, `durbin`, and `seidel2d`, and intermittently for `mlp` (14 of 24 configurations) and `correlat` (6 of 24); all 652 failures occur after the candidate itself has passed validation. These candidates contribute to the correctness counts but cannot enter runtime comparisons.

B.2 WARMUP, TIMING, AND CANDIDATE SELECTION

We use NPBench's parallel Numba implementation where available and its nopython implementation otherwise. Each implementation is warmed immediately before its own measurement batch. Warmup runs for at least three calls and stops after reaching 200 ms or 20 calls. The estimated call duration then determines the number of timed repetitions, targeting one second of execution within a minimum of ten and a maximum of 100 calls. Baseline and candidate are calibrated separately. Each call receives fresh inputs whose cache lines are flushed before the host timer starts; compilation, validation, copying, and flushing are outside the measured interval. A native evaluation has a 600-second task timeout, with a further 120 seconds allowed for process management.

During optimization, the harness selects the validated candidate with the highest ratio of median baseline to median candidate runtime. For reporting, we keep that program and compute its arithmetic-mean runtime. The baseline runtime varies slightly between searches, so we pool baseline repetitions for the same kernel when the input preset, Numba implementation, thread count, cache policy, and timing protocol agree, counting duplicate measurement batches only once. The mean of this pool supplies one baseline time for every configuration of the kernel. Selection uses medians because they are robust to occasional slow calls; reporting uses means because they reflect the total time of repeated calls. Appendix E checks that the analysis comparison is unchanged when speed-ups instead use the per-evaluation medians used in selection.

Table 3 reports the coefficient of variation of a candidate's repeated calls for every model. Over the 1,333 selected GPT-OSS evaluations on the 28-kernel panel it averages 3.4% (median 1.3%); the other models range from 2.2% to 4.3%, with medians below 1.7%. Pooling reduces the relative standard error of the shared baseline to 0.46%, 0.41%, and 0.45% on average on the 28-, 13-, and seven-kernel panels. The reported candidate is the best of many noisy evaluations, and under direct coding analysis raises the number of evaluations by 23% and 12% (Table 12); repeated calls measure timing dispersion, but do not quantify the bias from selecting the best of multiple candidates. Independent repeated searches would be required to estimate that contribution.

Table 3: Repeated-call runtime variation of the selected candidates. CV is the standard deviation divided by the mean runtime over a candidate's repeated calls; each selected evaluation receives equal weight. Qwen3.8 has usable timings on 48 of the 54 kernels. These values describe repeated calls, not variation between independent searches.

Model	Panel	Kernels	Evaluations	Mean CV (%)	Median CV (%)
GPT-OSS (both)	28	28	1,333	3.43	1.34
Qwen3.8-27B	28	28	671	4.31	1.44
Qwen3.8-27B	Full suite	48	1,129	4.16	1.24
Qwen3.5-27B	13	13	308	2.18	0.76
Qwen3.5-397B	13	13	312	2.35	0.86
Kimi-K2.7-Code	7	7	168	4.33	1.68

Using a common baseline also makes the component comparisons easier to interpret. If B_k is the shared baseline time and T_{k1}, T_{k0} are two candidate times, then their speed-up ratio is $(B_k/T_{k1})/(B_k/T_{k0}) = T_{k0}/T_{k1}$. Thus a ratio above one means that the first configuration produced a faster candidate, independent of the baseline's absolute runtime.

B.3 Audited exclusions

We ran AddressSanitizer on the selected implementations of every model, including the transfer panels. Two selected 120B implementations fail this audit and are excluded, because their memory-management operations violate the evaluation contract. For `chanflow`, `M-A1-T0-opt` swaps pointers and deletes caller-owned storage. For `azimnaiv`, `L-A1-T0-off` releases storage from `aligned_alloc` with `delete[]`. AddressSanitizer reports allocation/deallocation mismatches in both cases.

The audit finds no other violations, so these are the only exclusions: two of the 2,592 GPT-OSS searches (0.077%), or two of 2,012 selected timings before exclusion (0.099%). Their speed-ups are omitted without substituting an earlier candidate. The correctness counts retain the original outcomes of the numerical checks, which did not detect these defects.

C Kernel selection and complete workload lists

C.1 Kernels that distinguish configurations

The 28-kernel panel focuses the comparison on workloads for which agent configuration changes the resulting program's performance. Kernels with nearly identical outcomes under every configuration contribute little information about those choices and dilute their aggregate effects. We therefore combine a minimum-coverage rule with a measure of variation across configurations. The full-suite results remain available in Appendix D, and Appendix E checks the analysis effect without this selection.

For each model, we first require usable, correctness-validated timings for at least 80% of the 24 configurations, hence at least 20 of 24. Coverage is counted after audited exclusions; a correctness pass without a usable timing does not count. For the n_k usable timings of kernel k, we compute the average relative difference between every pair of configuration speed-ups:

$$D_k = \frac{2}{n_k(n_k-1)} \sum_{i<j} \frac{|s_{ki} - s_{kj}|}{\max(s_{ki}, s_{kj})}. \tag{1}$$

The score is zero when all configurations yield the same speed-up and grows as their outcomes differ, whichever configuration is faster. For example, speed-ups of $2\times$ and $4\times$ have a relative difference of 0.5. Normalizing by the larger value makes the score comparable across kernels with different absolute speed-ups. We retain scores above 0.25.

These rules select 28 kernels for 20B and 23 for 120B. The 120B selection is a subset of the 20B selection: only `3mm`, `doitgen`, `gemm`, `gemver`, and `resnet` qualify for 20B alone. The 28-kernel panel is the union of the two selections and therefore equals the 20B selection; no kernels were added by hand, and both models are evaluated on the same 28 kernels. Taking the union keeps one panel for both models, so that per-model results compare the same workloads, and it does not

inflate the analysis effect: on the 23 kernels selected for both models, the direct-coding analysis effect is 1.86× [1.51, 2.32] for 20B and 1.89× [1.55, 2.34] for 120B, compared with 1.79× and 1.71× on the union. Table 4 identifies every retained kernel. This panel emphasizes configuration-sensitive workloads, so its aggregate effects answer a different question from an average over the entire suite.

The panel is defined from the GPT-OSS results only and held fixed for Qwen3.8, so that Qwen3.8 tests the selection on outcomes that did not help choose it. Applying the same coverage and dispersion rule independently to Qwen3.8 selects 25 kernels, of which 19 overlap the 28-kernel panel. It would add `adist`, `cavtflow`, `crc16`, `mlp`, `nussinov`, and `sselfeng`, and remove nine fixed-panel kernels. The different membership shows that configuration sensitivity depends on the model. Table 6 reports its analysis effects both on the fixed panel and across the full suite.

C.2 Smaller panels for additional models

The CPU and GPU cost of evaluating all 24 configurations limits the number of workloads that can be repeated across larger models. We therefore use smaller panels to examine whether the component effects transfer, prioritizing kernels with the strongest differences in the GPT-OSS experiments. We select seven kernels for each of three comparisons: enabling analysis, enabling tools and skills with analysis already on, and changing how the optimization graph affects direct versus delegated coding. Their union contains 13 kernels. The seven analysis-selected kernels alone form the panel for Kimi-K2.7-Code.

Each ranking requires at least 75% of its relevant matched comparisons to be available. Analysis selection favors large improvements; tools selection retains large changes in either direction; history selection favors a larger benefit under direct coding than under delegation. The selected kernels are then held fixed across models, so changes in the extended comparison cannot be caused by changing its workload set. Table 4 gives the complete memberships.

The 13-kernel comparison evaluates all 24 configurations for Qwen3.5-27B and Qwen3.5-397B (312 cells per model); the seven-kernel Kimi comparison has 168 cells. GPT-OSS and Qwen3.8 contribute the same panel subsets from their full-suite campaigns. Table 8 gives matched kernel and configuration-pair counts for every analysis contrast, retaining the same estimator used in the main comparison.

Table 4: Complete workload list and panel membership. A check marks inclusion in a panel; a blank means that the kernel is not selected for that panel. Every row belongs to the full 54-kernel suite. Timed counts usable speed-ups out of 24 configurations; zero means that no usable speed-up is available. Reference LoC is defined in Appendix H.

Kernel	LoC	28	13	7	20B	120B	Q3.8-27B
`2mm`	3	✓	✓		24/24	24/24	24/24
`3mm`	3	✓			24/24	24/24	24/24
`adi`	44				0/24	0/24	0/24
`adist`	10				24/24	24/24	24/24
`atax`	3				24/24	24/24	24/24
`azimhist`	5	✓			24/24	23/24	24/24
`azimnaiv`	11	✓			24/24	23/24	24/24
`bicg`	3				24/24	24/24	24/24
`cavtflow`	59				0/24	0/24	21/24
`chanflow`	109	✓			21/24	23/24	24/24
`cholesky`	9				0/24	0/24	0/24
`cholesky2`	3				0/24	0/24	0/24
`clipping`	3				24/24	24/24	24/24
`coninteg`	18	✓			24/24	24/24	24/24
`conv2d`	18	✓	✓	✓	24/24	23/24	24/24
`correlat`	11				0/24	0/24	18/24
`covarian`	8				0/24	0/24	18/24
`covarian2`	3	✓			23/24	24/24	24/24

Continued on next page

Kernel	LoC	28	13	7	20B	120B	Q3.8-27B
crc16	17				24/24	24/24	24/24
deriche	36	✓			24/24	24/24	24/24
doitgen	3	✓	✓	✓	23/24	24/24	24/24
durbin	12				0/24	0/24	0/24
fdtd_2d	8				24/24	24/24	24/24
floydwar	4				24/24	24/24	24/24
gemm	3	✓			24/24	24/24	24/24
gemver	5	✓			24/24	24/24	24/24
gesummv	3				24/24	24/24	24/24
gramschm	12	✓	✓	✓	24/24	24/24	24/24
hdiff	23	✓			24/24	24/24	24/24
heat3d	17				24/24	24/24	24/24
jacobi1d	5	✓			24/24	24/24	24/24
jacobi2d	7				24/24	24/24	24/24
lenet	38	✓			23/24	24/24	24/24
lu	8	✓	✓		23/24	24/24	24/24
ludcmp	15	✓	✓		24/24	24/24	24/24
mandel1	13	✓	✓	✓	24/24	24/24	24/24
mandel2	23	✓	✓		24/24	24/24	24/24
mlp	13				0/24	0/24	23/24
mvt	4				24/24	24/24	24/24
nbody	65				0/24	0/24	0/24
npgofast	6				24/24	24/24	24/24
nussinov	24				24/24	24/24	24/24
resnet	34	✓	✓		24/24	24/24	24/24
seidel2d	10				0/24	0/24	0/24
softmax	6	✓			24/24	24/24	23/24
spmv	8				24/24	24/24	24/24
sselfeng	15				0/24	5/24	24/24
sthamfft	17	✓			24/24	24/24	24/24
symm	9	✓	✓		24/24	24/24	24/24
syr2k	7	✓	✓	✓	24/24	24/24	24/24
syrk	6	✓	✓	✓	24/24	24/24	24/24
trisolv	4				0/24	0/24	18/24
trmm	6	✓			24/24	24/24	24/24
vadv	53	✓	✓	✓	24/24	24/24	24/24

D FULL-SUITE OUTCOMES AND CONFIGURATION OVERVIEW

The full-suite view shows which kernels benefit across many configurations and which produce isolated gains, losses, or missing measurements. Figures 6–8 place all 54 kernels and 24 configurations on the same absolute speed-up scale. Rows with similar colors indicate consistent outcomes across workflows; variation within a row identifies kernels where the workflow matters. Missing timings have explicit symbols and are never assigned zero or baseline-equivalent performance.

Table 5 adds correctness and timing counts, token cost, and search time for each configuration. Its speed-up means describe the timed kernels available to each row, which can differ; Figure 1 plots these values. The following sections compare matched kernels and settings: Appendix E studies analysis, tools and skills, and coding mode; Appendix F examines the history settings; Appendix G separates search time and costs across agent roles. Appendix J connects selected outcomes to the generated programs.

As a sensitivity check, we restrict the 28-kernel panel to kernels timed by all 24 configurations of a model: 23 for 20B, 24 for 120B, and 27 for Qwen3.8. Every configuration is then compared on the same kernels, including its token mean. The nondominated GPT-OSS configurations all use analyzed feedback. Qwen3.8 retains both analyzed direct-coding configurations and raw-feedback delegated coding (`M-A0-T0-opt`) on its speed–token frontier. Matching the complete configuration population therefore supports model-specific workflow choices rather than a universal ordering of analyzed and raw feedback.

Table 5: All configurations on the 54-kernel suite, using the notation in Table 1. V/T gives native-valid and usable-timing counts, each out of 54. S is the geometric-mean speed-up over timed kernels. Tokens are mean millions of tokens per search; minutes are mean search time, including failures. Missing values are excluded, with counts in the companion data.

	GPT-OSS-20B				GPT-OSS-120B				Qwen3.8-27B			
Configuration	V/T	S	Tokens	Minutes	V/T	S	Tokens	Minutes	V/T	S	Tokens	Minutes
`L-A0-T0-off`	49/42	9.5	2.37	60.5	48/41	11.0	1.09	36.9	51/47	43.6	3.28	998.9
`L-A0-T0-turn`	49/42	9.7	3.03	59.3	49/42	12.8	1.38	38.2	51/48	41.2	3.39	974.8
`L-A0-T0-opt`	48/42	10.5	2.61	58.8	49/42	11.1	1.38	33.9	51/48	47.9	3.35	1002.6
`L-A1-T0-off`	49/42	13.9	0.73	69.4	49/41	15.6	0.58	48.8	51/48	44.5	2.68	1424.1
`L-A1-T0-turn`	49/42	13.3	0.85	66.5	49/42	17.9	0.71	46.1	51/48	51.7	3.69	1477.9
`L-A1-T0-opt`	49/42	16.5	0.79	70.4	49/42	19.4	0.69	46.1	51/48	42.9	3.02	1404.9
`L-A0-T1-off`	48/42	10.6	2.84	85.9	49/42	13.6	1.08	37.9	51/48	52.4	3.47	850.4
`L-A0-T1-turn`	49/42	9.2	3.07	94.0	50/43	13.2	1.41	37.9	50/45	46.5	3.54	851.8
`L-A0-T1-opt`	48/41	9.7	3.00	83.6	49/42	13.6	1.25	41.6	50/47	50.0	3.52	964.1
`L-A1-T1-off`	48/41	14.3	1.79	109.7	49/42	17.5	1.16	63.7	51/48	51.0	3.81	1716.2
`L-A1-T1-turn`	48/41	17.8	1.85	109.2	50/43	17.9	1.17	66.3	50/47	54.0	4.60	1681.4
`L-A1-T1-opt`	49/42	14.3	1.89	116.6	49/42	22.4	1.09	59.4	51/48	55.1	4.18	1714.6
`M-A0-T0-off`	49/42	10.0	1.52	34.6	49/42	13.1	0.71	29.5	50/45	44.6	1.85	676.1
`M-A0-T0-turn`	49/42	10.3	1.77	36.3	48/42	13.4	0.86	34.2	50/45	44.1	1.88	688.5
`M-A0-T0-opt`	49/42	11.0	1.67	39.8	49/42	13.2	0.74	30.9	51/44	43.3	2.00	682.1
`M-A1-T0-off`	49/42	15.0	0.86	53.6	49/42	19.7	1.12	52.6	51/48	48.2	4.59	1472.1
`M-A1-T0-turn`	49/42	12.4	1.07	58.6	49/42	19.1	0.90	46.6	50/47	56.9	4.84	1647.5
`M-A1-T0-opt`	49/42	12.7	1.01	60.3	49/41	16.1	0.77	40.5	51/47	49.0	4.49	1513.0
`M-A0-T1-off`	48/41	10.7	2.90	57.1	48/42	12.6	0.80	37.6	51/48	37.6	2.20	758.2
`M-A0-T1-turn`	47/41	9.4	3.13	60.8	50/43	12.0	0.76	38.7	50/45	39.7	2.19	736.8
`M-A0-T1-opt`	47/41	9.8	3.15	61.8	50/43	12.5	0.84	36.8	51/46	34.4	1.93	640.7
`M-A1-T1-off`	49/42	17.0	1.95	87.4	49/42	15.8	1.42	63.8	51/48	51.7	6.61	2085.4
`M-A1-T1-turn`	49/42	14.3	2.13	91.1	48/42	14.8	1.23	61.9	51/48	49.7	6.36	2010.9
`M-A1-T1-opt`	48/41	17.9	2.12	105.6	48/42	15.5	1.18	55.9	51/48	49.5	5.98	2001.4

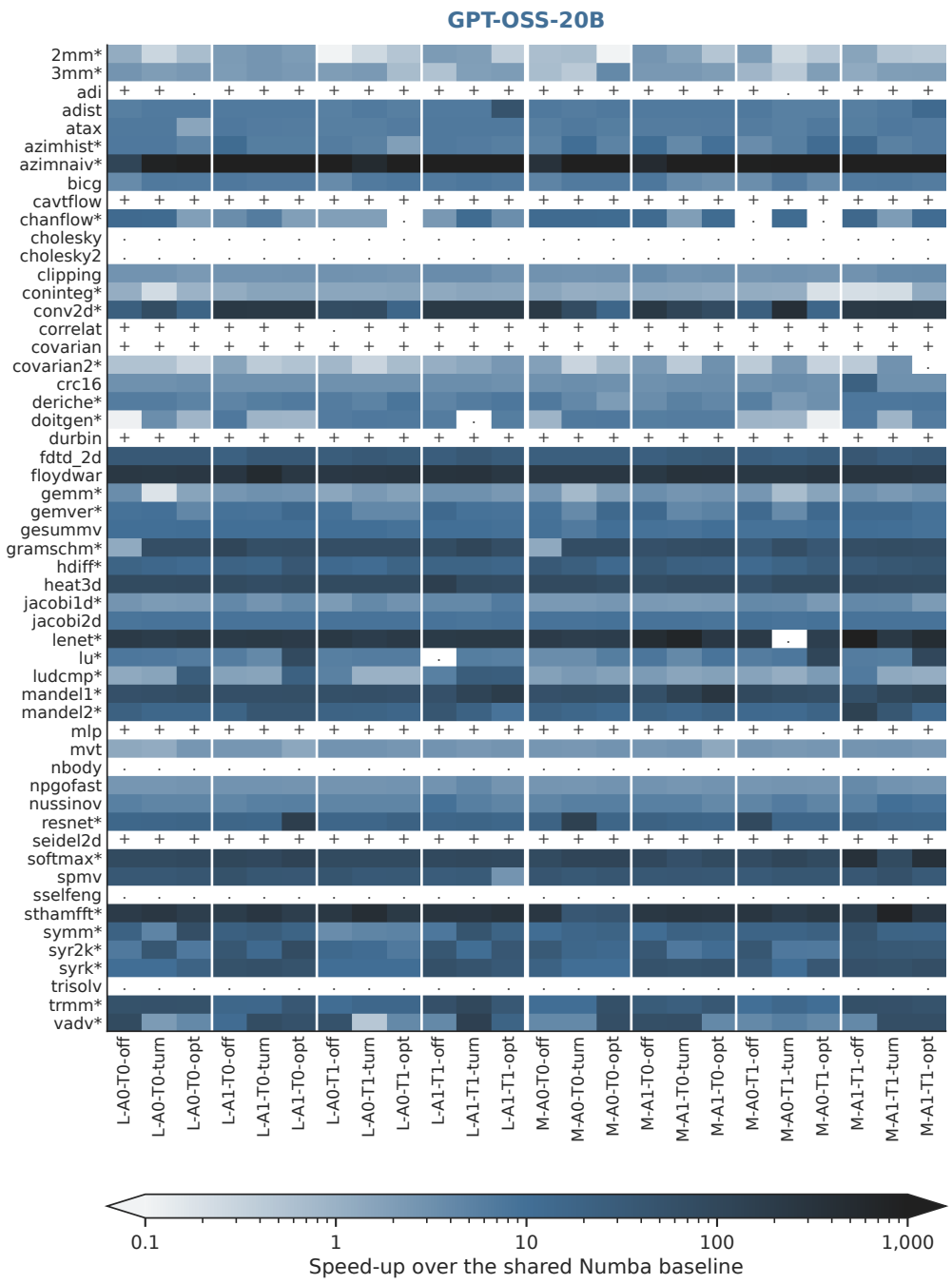

Figure 6: GPT-OSS-20B: all 1,296 kernel–configuration cells. A star beside a kernel marks membership in the 28-kernel panel. White cells contain + for native-valid but unmeasured, · for no recorded native-valid candidate, or × for an audited exclusion. Colors use the same logarithmic range in all three model matrices; values outside 0.1–1,000 saturate at the endpoints. Configuration codes follow Table 1.

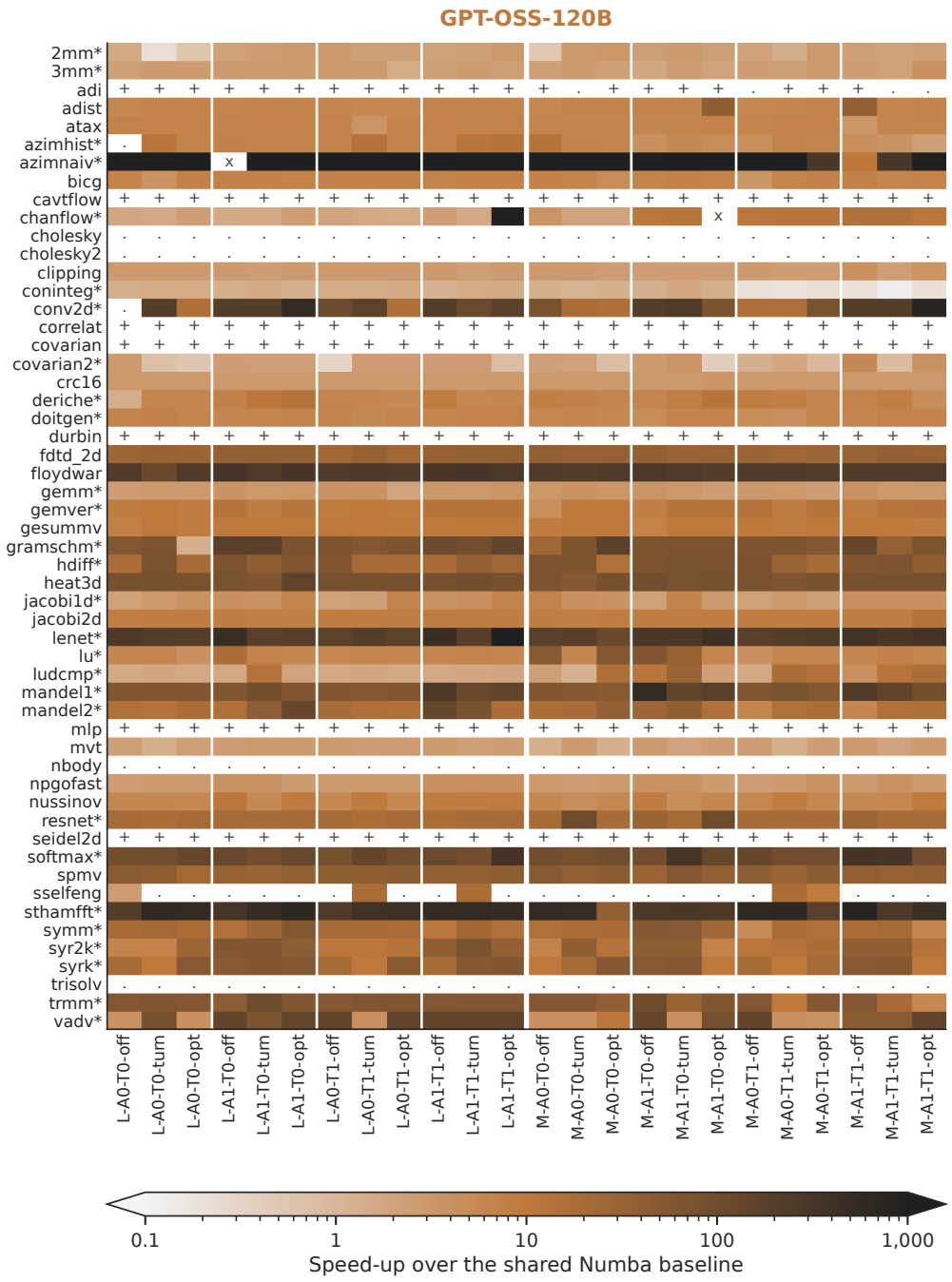

Figure 7: GPT-OSS-120B: all 1,296 kernel–configuration cells, with the same ordering, scale, and missing-outcome symbols as Figure 6. Configuration codes follow Table 1. The audited exclusions are `L-A1-T0-off azimnaiv` and `M-A1-T0-opt chanflow`. An empty timing cell remains part of the planned search population.

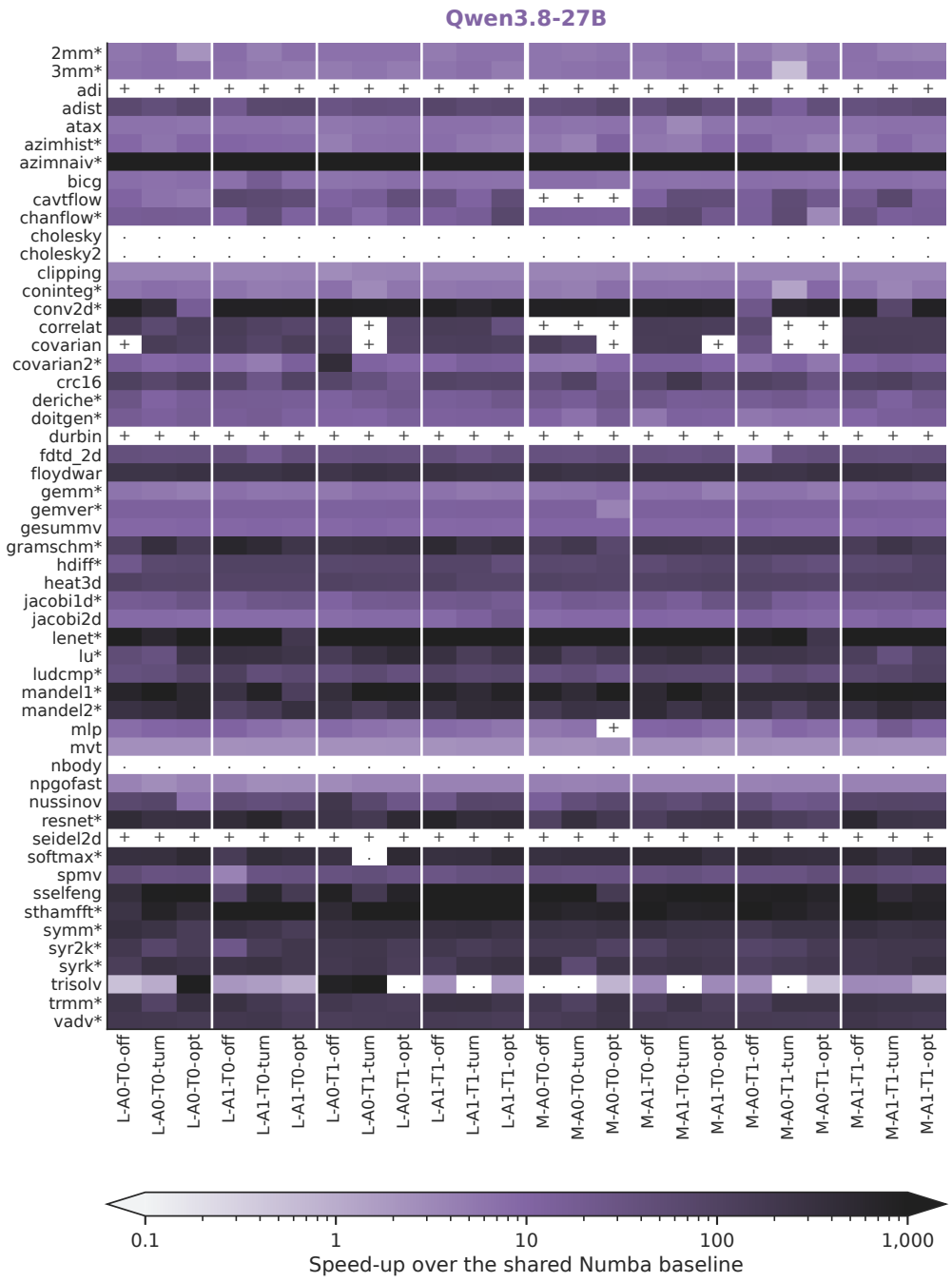

Figure 8: Qwen3.8-27B: all 1,296 kernel–configuration cells, with the same ordering, scale, and outcome symbols as Figure 6. The panel stars remain fixed from GPT-OSS, so Qwen3.8 does not participate in choosing the comparison workloads.

E MATCHED EFFECTS AND SENSITIVITY ANALYSES

The main analysis result raises two questions: whether the improvement depends on selecting configuration-sensitive kernels, and whether it depends on the runtime summary used to compute speed-up. We first compare the effect across the full suite and the smaller panels, then examine individual kernels and the other configurable components. The estimator at the end of this section specifies how these comparisons give each kernel equal weight.

E.1 ANALYZED FEEDBACK ACROSS WORKLOAD PANELS

For GPT-OSS, Table 6 shows that the direction of the analysis effect persists as the workload set changes. On the full suite, direct-coding speed-up ratios are 1.51 for 20B and 1.47 for 120B, over 42 and 43 kernels with matched timings. They rise to 1.79 and 1.71 on the 28-kernel panel. The 13- and seven-kernel panels show even larger effects because they emphasize workloads with strong component differences. Selection changes the magnitude of the reported benefit, and the positive aggregate effect is present without the variation cutoff.

Under delegation, analyzed feedback still produces faster programs, but its token cost differs by model: across all four panels, it lowers total tokens for 20B and raises them for 120B. Faster programs and cheaper searches need not move together.

The Median column recomputes the effect with the per-evaluation median speed-ups used during selection for the same selected programs. For the GPT-OSS models, every row keeps a ratio above one; for direct coding on the full suite, the ratios change from 1.51 to 1.56 for 20B and from 1.47 to 1.48 for 120B.

Table 6: Analysis on/off across workload panels. Matched kernels and settings define each row; Median uses the original selection speed-ups.

Panel	Model	Mode	K	Pairs	Speed ratio [95% interval]	Tokens	Median
54	20B	L	42	249	1.51 [1.31, 1.76]	0.45	1.56
54	20B	M	42	248	1.40 [1.22, 1.62]	0.68	1.42
54	120B	L	43	250	1.47 [1.29, 1.71]	0.75	1.48
54	120B	M	42	251	1.30 [1.13, 1.53]	1.52	1.32
54	Q3.8-27B	L	48	282	1.05 [0.89, 1.19]	1.05	1.09
54	Q3.8-27B	M	48	273	1.29 [1.20, 1.40]	3.04	1.30
28	20B	L	28	165	1.79 [1.50, 2.17]	0.47	1.85
28	20B	M	28	164	1.58 [1.30, 1.92]	0.71	1.60
28	120B	L	28	165	1.71 [1.44, 2.07]	0.78	1.73
28	120B	M	28	167	1.41 [1.15, 1.76]	1.60	1.42
28	Q3.8-27B	L	28	167	1.13 [1.00, 1.29]	1.01	1.17
28	Q3.8-27B	M	28	168	1.29 [1.18, 1.43]	3.00	1.29
13	20B	L	13	76	2.47 [1.90, 3.24]	0.45	2.64
13	20B	M	13	78	1.81 [1.33, 2.39]	0.57	1.82
13	120B	L	13	77	2.13 [1.58, 2.91]	0.75	2.10
13	120B	M	13	78	1.83 [1.32, 2.67]	1.65	1.85
13	Q3.8-27B	L	13	78	1.17 [0.97, 1.41]	0.96	1.25
13	Q3.8-27B	M	13	78	1.32 [1.16, 1.47]	3.07	1.31
7	20B	L	7	41	2.84 [1.94, 4.10]	0.47	2.86
7	20B	M	7	42	2.55 [1.96, 3.27]	0.54	2.52
7	120B	L	7	41	2.79 [1.80, 4.14]	0.73	2.79
7	120B	M	7	42	2.45 [1.46, 4.32]	1.79	2.49
7	Q3.8-27B	L	7	42	1.14 [0.85, 1.55]	0.98	1.24
7	Q3.8-27B	M	7	42	1.31 [1.22, 1.39]	3.54	1.31

Figure 9 shows how broadly the effect is distributed. Direct coding improves on 36 of 42 matched kernels for 20B and 38 of 43 for 120B. Under delegation, the corresponding counts are 36 of 42 and 34 of 42. Gains therefore extend beyond a few high-speed-up kernels, although individual regressions remain. For example, the direct-coding ratio on `spmv` is 0.63 for 20B, whereas `conv2d` reaches 5.79. The open points retain kernels outside the 28-kernel panel, making the relationship between the selected and full-suite results visible.

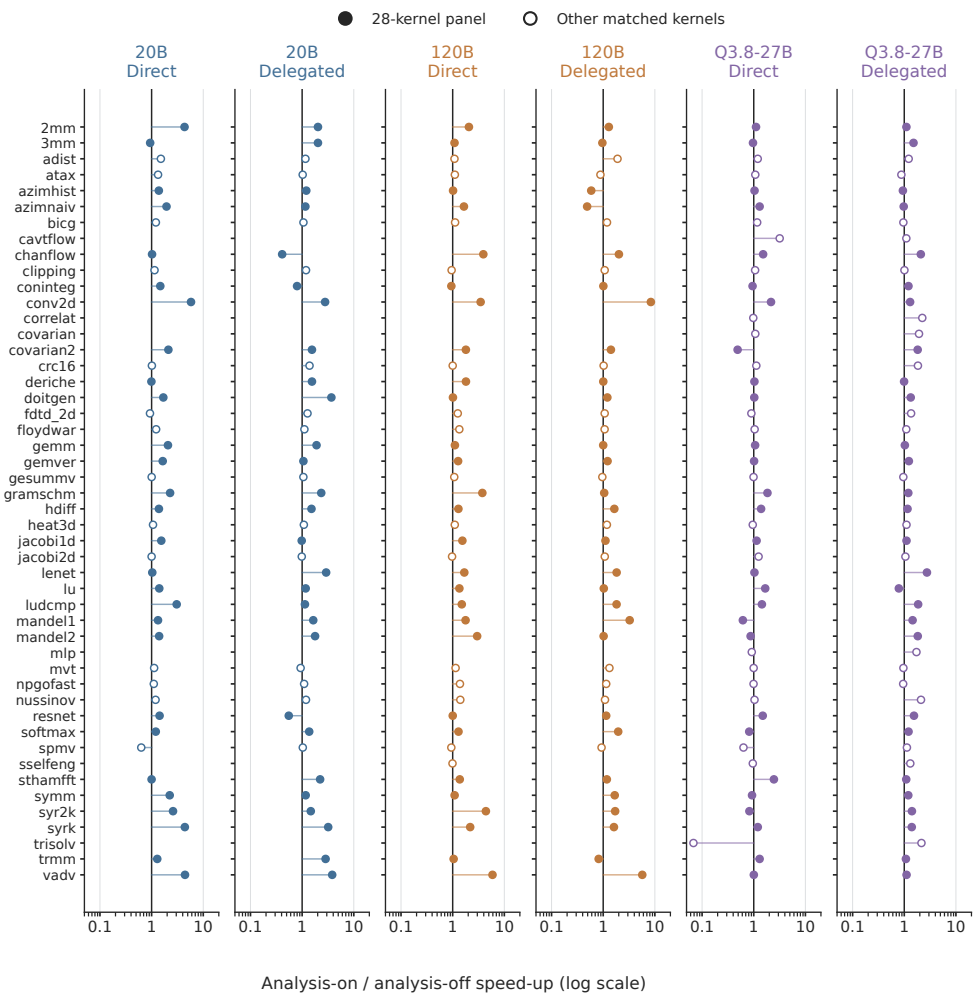

Figure 9: Per-kernel analysis effects across all matched workloads. Each point geometrically averages matched configuration ratios within one kernel. Filled points belong to the 28-kernel panel; open points are outside it. Values above one favor analysis, and the logarithmic scale shows gains and losses symmetrically as ratios. A missing point indicates no eligible pair for that model and coding mode.

For Qwen3.8, analysis produces fixed-panel speed ratios of 1.13 [1.00, 1.29] with direct coding and 1.29 [1.18, 1.43] with delegation. Its total-token ratios are 1.01 and 3.00, respectively. Across the full suite, its direct-coding ratio is 1.05 [0.89, 1.19], while delegation yields 1.29 [1.20, 1.40]. The direct coding effect is therefore sensitive to workload selection, and the joint improvement in speed and token cost observed for GPT-OSS direct coding does not transfer.

Pooling both coding modes while matching the remaining choices gives full-suite speed ratios of 1.46 [1.30, 1.66] for 20B, 1.38 [1.22, 1.60] for 120B, and 1.15 [1.06, 1.24] for Qwen3.8. All three intervals exclude one. Their total-token ratios are 0.54, 1.03, and 1.73, respectively: faster programs are the common benefit, while token savings depend on the model and coding mode.

E.2 Tools and skills, and coding mode

Table 7 places speed-up and cost side by side. For GPT-OSS direct coding with analyzed feedback, tools and skills give speed-up ratios of 1.03 and 1.07, neither significant, while token use grows by 134% and 80%. Under delegation, the 20B ratio of 1.23 barely excludes one and the 120B ratio of 0.79 does not, with higher token use in each case.

For GPT-OSS, coding mode also changes the cost comparison. With raw feedback, direct coding uses 49% and 83% more tokens than delegation, with speed-up ratios of 0.94 and 0.99. With analyzed feedback, it uses 1% and 10% fewer tokens, with ratios of 1.01 and 1.19. None of the four direct/delegated speed-up ratios is significant.

Table 7: Matched component effects on the 28-kernel panel. `A0/A1` denotes raw/analyzed feedback (Table 1). A speed ratio above one favors the numerator; a cost ratio below one means that it costs less. Eval. is evaluator wall time. Token and time ratios divide totals. Table 12 gives the timing denominators.

Contrast	Model	K	Pairs	Speed ratio [95% interval]	Tokens	Elapsed	Eval.
Analysis: direct	20B	28	165	1.79 [1.50, 2.17]	0.47	1.30	1.04
Analysis: direct	120B	28	165	1.71 [1.44, 2.07]	0.78	1.62	1.01
Analysis: direct	Q3.8-27B	28	167	1.13 [1.00, 1.29]	1.01	1.69	1.16
Analysis: delegated	20B	28	164	1.58 [1.30, 1.92]	0.71	1.67	1.66
Analysis: delegated	120B	28	167	1.41 [1.15, 1.76]	1.60	1.73	1.28
Analysis: delegated	Q3.8-27B	28	168	1.29 [1.18, 1.43]	3.00	2.73	1.48
Tools: direct, A1	20B	28	82	1.03 [0.84, 1.26]	2.34	1.69	0.97
Tools: direct, A1	120B	28	83	1.07 [0.86, 1.39]	1.80	1.38	0.95
Tools: direct, A1	Q3.8-27B	28	84	1.08 [0.96, 1.23]	1.36	1.21	0.63
Tools: delegated, A1	20B	28	83	1.23 [1.01, 1.47]	2.08	1.62	1.16
Tools: delegated, A1	120B	28	83	0.79 [0.60, 1.01]	1.24	1.14	0.96
Tools: delegated, A1	Q3.8-27B	28	84	0.98 [0.83, 1.12]	1.40	1.32	1.42
Direct/delegated, A1	20B	28	165	1.01 [0.86, 1.19]	0.99	1.34	1.23
Direct/delegated, A1	120B	28	166	1.19 [0.95, 1.50]	0.90	1.09	1.14
Direct/delegated, A1	Q3.8-27B	28	168	1.04 [0.92, 1.18]	0.65	0.89	0.79
Direct/delegated, A0	20B	28	165	0.94 [0.81, 1.09]	1.49	1.70	1.95
Direct/delegated, A0	120B	28	166	0.99 [0.81, 1.21]	1.83	1.16	1.42
Direct/delegated, A0	Q3.8-27B	28	167	1.19 [1.08, 1.33]	1.94	1.43	0.81

Qwen3.8 tools-and-skills ratios are 1.08 [0.96, 1.23] for direct coding and 0.98 [0.83, 1.12] for delegation, with token ratios of 1.36 and 1.40. Neither speed interval excludes one. This independently reproduces the weak aggregate speed benefit and increased token cost of tools seen in GPT-OSS direct coding.

The per-kernel effects in Figure 10 further separate these outcomes. Tools and skills improve delegated speed-up on 21 of 28 kernels for 20B, compared with 14 of 28 for 120B. The optimization graph also produces gains and losses within each model: for 20B direct coding, its ratio ranges from 0.12 on `doitgen` to 13.34 on `ludcmp`. Such variation motivates examining history together with coding mode and tools and skills in Appendix F.

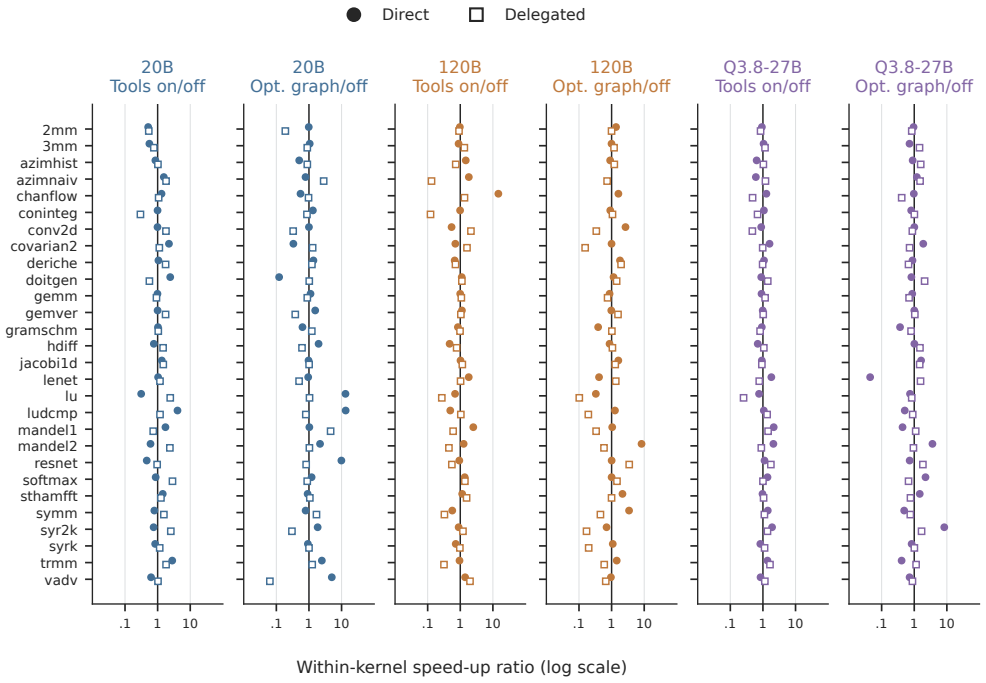

Figure 10: Effects of tools and skills and of the optimization graph on the 28-kernel panel. Analysis is enabled throughout. Tools compares on/off with history matched; history compares `opt/off` with tools and skills disabled. Solid circles indicate direct coding and open squares delegation. Each point summarizes available pairs for one kernel; missing points are unmeasured contrasts.

E.3 Transfer and model scale

Table 8 reports matched analysis effects on the fixed 13-kernel and seven-kernel panels; the latter includes Kimi.

Table 8: Analysis effects on the fixed transfer panels. Speed ratios compare matched timings; token ratios divide usage totals over the same eligible pairs. Workload intervals resample kernels.

Panel	Model	Mode	K	Pairs	Speed ratio [95% interval]	Tokens
13	GPT-OSS-20B	L	13	76	2.47 [1.90, 3.24]	0.45
13	GPT-OSS-20B	M	13	78	1.81 [1.33, 2.39]	0.57
13	GPT-OSS-120B	L	13	77	2.13 [1.58, 2.91]	0.75
13	GPT-OSS-120B	M	13	78	1.83 [1.32, 2.67]	1.65
13	Qwen3.8-27B	L	13	78	1.17 [0.97, 1.41]	0.96
13	Qwen3.8-27B	M	13	78	1.32 [1.16, 1.47]	3.07
13	Qwen3.5-27B	L	13	78	1.29 [1.12, 1.51]	1.00
13	Qwen3.5-27B	M	13	76	1.11 [0.89, 1.38]	2.21
13	Qwen3.5-397B	L	13	78	1.07 [0.84, 1.36]	1.05
13	Qwen3.5-397B	M	13	78	1.35 [1.02, 1.73]	2.66
7	GPT-OSS-20B	L	7	41	2.84 [1.94, 4.10]	0.47
7	GPT-OSS-20B	M	7	42	2.55 [1.96, 3.27]	0.54
7	GPT-OSS-120B	L	7	41	2.79 [1.80, 4.14]	0.73
7	GPT-OSS-120B	M	7	42	2.45 [1.46, 4.32]	1.79
7	Qwen3.8-27B	L	7	42	1.14 [0.85, 1.55]	0.98
7	Qwen3.8-27B	M	7	42	1.31 [1.22, 1.39]	3.54
7	Qwen3.5-27B	L	7	42	1.27 [1.01, 1.67]	0.98
7	Qwen3.5-27B	M	7	42	1.12 [0.77, 1.61]	2.16
7	Qwen3.5-397B	L	7	42	1.28 [0.97, 1.73]	1.08
7	Qwen3.5-397B	M	7	42	1.14 [0.75, 1.69]	2.55
7	Kimi-K2.7-Code	L	7	42	1.25 [1.01, 1.62]	1.13
7	Kimi-K2.7-Code	M	7	42	1.02 [0.83, 1.23]	1.69

The Qwen3.5 size comparison reveals a coding-mode interaction. With analysis enabled, the direct/delegated speed ratio is 0.91 for Qwen3.5-27B and 0.44 for Qwen3.5-397B, over 76 and 78

matched pairs on all 13 kernels. Restricting to the same 76 four-configuration blocks, the delegated/direct advantage is 2.07 times larger for Qwen3.5-397B [1.51, 2.78]. The larger model benefits much more from delegation. These are properties of the measured models and workloads: parameter count alone does not determine the best workflow, and a rule inferred from the two GPT-OSS models need not transfer to another family.

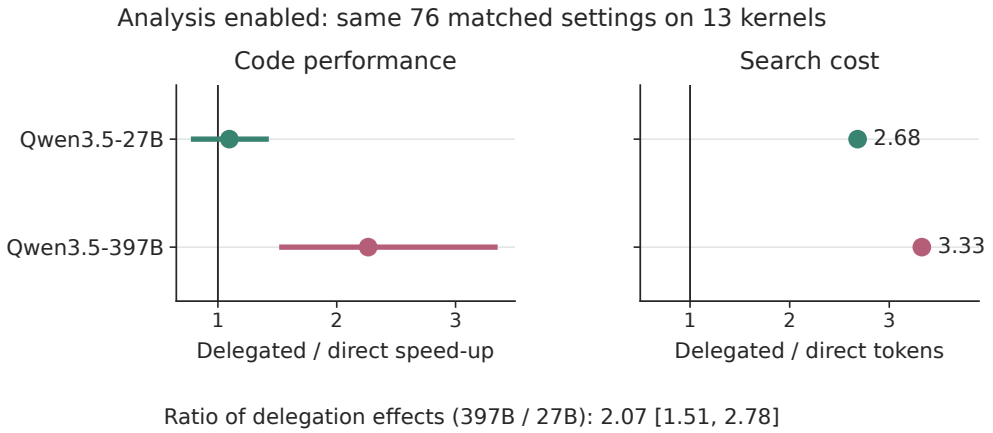

Figure 11: Coding mode and model scale within the Qwen3.5 family. Both models use analyzed feedback and exactly the same 76 kernel–tools–history combinations on 13 kernels, with timings for both coding modes. Speed ratios geometrically average within kernels and then across kernels; intervals resample kernels. Token ratios divide summed recorded usage on the same matched combinations. A speed ratio above one favors delegation; a token ratio above one means that delegation costs more.

Other components also respond differently to model scale. With direct coding and analysis, adding the optimization graph without tools gives a speed ratio of 1.97 [1.05, 3.60] for Qwen3.5-27B at 5% more tokens, versus 0.83 [0.43, 1.40] for Qwen3.5-397B at 37% more tokens (13 matched kernels each). Enabling tools and skills for Qwen3.5-397B direct coding instead gives 0.71 [0.51, 0.98] at 23% more tokens, across 39 matched pairs on those 13 kernels. The larger model's benefit from delegation therefore does not justify enabling every additional component.

Table 9 gives the search cost of each transfer kernel, averaged over the 24 configurations. Qwen3.5-397B uses fewer tokens than Qwen3.5-27B on 10 of the 13 kernels, consistent with its lower repair cost (Table 15). Search times reflect different serving loads and are comparable only within a model.

Table 9: Search cost per transfer kernel: mean millions of tokens / mean minutes per search, averaged over the 24 configurations. Times sum recorded search work and exclude gaps between continuations; serving loads differ, so they are not controlled cross-model comparisons. Dashes mark kernels outside Kimi's seven-kernel panel.

Kernel	Qwen3.5-27B	Qwen3.5-397B	Kimi-K2.7-Code
`2mm`	3.01 / 1,170	2.50 / 787	–
`conv2d`	3.32 / 1,491	3.44 / 1,608	2.38 / 1,259
`doitgen`	2.97 / 1,316	2.13 / 610	2.61 / 1,449
`gramschm`	2.72 / 1,077	1.63 / 256	2.55 / 1,594
`lu`	2.55 / 912	2.25 / 606	–
`ludcmp`	2.85 / 1,188	2.20 / 547	–
`mandel1`	2.76 / 1,126	2.04 / 510	2.51 / 1,116
`mandel2`	2.72 / 1,006	1.97 / 546	–
`resnet`	4.05 / 2,107	3.25 / 1,244	–
`symm`	2.58 / 821	2.24 / 534	–
`syr2k`	2.27 / 828	2.07 / 320	2.49 / 1,589
`syrk`	2.10 / 570	2.12 / 247	2.34 / 1,414
`vadv`	2.52 / 689	2.57 / 566	2.86 / 1,851

E.4 Effect estimator and uncertainty

A component comparison changes one configuration field while holding the model, kernel, and other fields fixed. Let P_k contain the available matched pairs for kernel k, and let K contain kernels with at least one pair. For speed-ups s_{kc_1} and s_{kc_0}, we report

$$R_s = \exp\left[\frac{1}{|K|}\sum_{k\in K}\frac{1}{|P_k|}\sum_{(c_1,c_0)\in P_k}\log\frac{s_{kc_1}}{s_{kc_0}}\right]. \tag{2}$$

Averaging within kernels before averaging across them prevents kernels with more measured pairs from receiving greater weight. A ratio of two means that the numerator's candidates run twice as fast in this geometric-mean comparison. Cost ratios instead divide summed tokens or time over matched pairs for which the corresponding quantity is recorded on each side.

To measure sensitivity to the workload composition, we draw $|K|$ kernels with replacement 5,000 times and recompute the aggregate. The interval spans the 2.5th and 97.5th percentiles of these estimates. For example, a draw from five kernels A, B, C, D, E might contain A, A, C, E, E: it counts the measured effects of A and E twice and omits B and D. All matched settings for a sampled kernel stay together. Wider intervals indicate greater dependence on which kernels contribute; they do not measure variability between repeated optimization searches. We call an effect significant when its interval excludes one.

F History settings and search effort

History can preserve either completed turns (turn graph) or individual optimization attempts (optimization graph). Table 10 compares each graph with history off and then compares the two graphs directly. Analysis is enabled throughout; separate rows show how the result changes with coding mode and with tools and skills. The `turn/off` and `opt/off` ratios measure the benefit of adding each graph, while `opt/turn` measures which graph produces faster code.

Table 10: History effects on the fixed 28-kernel panel. Speed ratios above one favor the numerator; costs divide matched totals.

Model	Mode	Tools	Contrast	K	Speed ratio [95% interval]	Tokens	Evaluations	Time
20B	L	T0	turn/off	28	0.92 [0.70, 1.18]	1.07	0.88	0.90
20B	L	T0	opt/off	28	1.32 [0.92, 1.93]	1.03	0.88	0.93
20B	L	T0	opt/turn	28	1.44 [1.04, 2.11]	0.96	1.00	1.03
20B	L	T1	turn/off	26	1.44 [1.03, 2.10]	1.03	0.99	0.98
20B	L	T1	opt/off	27	1.13 [0.84, 1.51]	1.06	0.93	1.02
20B	L	T1	opt/turn	27	0.80 [0.61, 1.02]	1.03	0.97	1.03
20B	M	T0	turn/off	28	0.76 [0.58, 0.97]	1.34	1.15	1.05
20B	M	T0	opt/off	28	0.81 [0.60, 1.07]	1.24	1.01	1.24
20B	M	T0	opt/turn	28	1.08 [0.77, 1.46]	0.93	0.88	1.19
20B	M	T1	turn/off	28	0.80 [0.54, 1.21]	1.09	1.14	1.00
20B	M	T1	opt/off	27	1.00 [0.66, 1.53]	1.15	1.20	1.19
20B	M	T1	opt/turn	27	1.35 [0.98, 1.92]	1.05	1.04	1.18
120B	L	T0	turn/off	27	1.08 [0.87, 1.37]	1.18	0.91	0.93
120B	L	T0	opt/off	27	1.16 [0.91, 1.49]	1.14	0.94	0.93
120B	L	T0	opt/turn	28	1.07 [0.86, 1.32]	0.98	1.04	1.00
120B	L	T1	turn/off	28	1.04 [0.88, 1.23]	1.00	1.00	1.06
120B	L	T1	opt/off	28	1.45 [0.91, 2.71]	0.91	1.14	0.92
120B	L	T1	opt/turn	28	1.39 [0.88, 2.72]	0.92	1.15	0.87
120B	M	T0	turn/off	28	0.97 [0.68, 1.28]	0.74	0.77	0.87
120B	M	T0	opt/off	27	0.69 [0.50, 0.96]	0.68	0.82	0.84
120B	M	T0	opt/turn	27	0.71 [0.47, 1.12]	0.92	1.07	0.95
120B	M	T1	turn/off	28	0.97 [0.71, 1.37]	0.81	0.87	0.91
120B	M	T1	opt/off	28	0.98 [0.62, 1.70]	0.81	1.00	0.82
120B	M	T1	opt/turn	28	1.01 [0.74, 1.41]	0.99	1.14	0.90
Q3.8-27B	L	T0	turn/off	28	1.18 [0.95, 1.49]	1.38	0.97	1.15
Q3.8-27B	L	T0	opt/off	28	0.88 [0.64, 1.20]	1.14	0.93	1.01
Q3.8-27B	L	T0	opt/turn	28	0.74 [0.54, 0.97]	0.83	0.96	0.87
Q3.8-27B	L	T1	turn/off	28	1.01 [0.87, 1.17]	1.20	0.93	1.02
Q3.8-27B	L	T1	opt/off	28	1.14 [0.95, 1.39]	1.18	1.06	1.13
Q3.8-27B	L	T1	opt/turn	28	1.13 [0.99, 1.31]	0.98	1.13	1.11
Q3.8-27B	M	T0	turn/off	28	1.11 [0.98, 1.27]	1.09	1.08	1.17
Q3.8-27B	M	T0	opt/off	28	1.02 [0.89, 1.18]	1.07	1.09	1.10

Continued on next page

Model	Mode	Tools	Contrast	K	Speed ratio [95% interval]	Tokens	Evaluations	Time
Q3.8-27B	M	T0	opt/turn	28	0.92 [0.80, 1.07]	0.98	1.02	0.94
Q3.8-27B	M	T1	turn/off	28	0.90 [0.69, 1.12]	0.92	0.97	0.92
Q3.8-27B	M	T1	opt/off	28	1.02 [0.90, 1.18]	0.89	0.95	0.95
Q3.8-27B	M	T1	opt/turn	28	1.14 [0.95, 1.43]	0.97	0.98	1.04

The preferred graph depends on the workflow. For 20B direct coding without tools and skills, the optimization graph produces a speed ratio of 1.44 [1.04, 2.11] relative to the turn graph. With tools and skills enabled, this ratio falls to 0.80 [0.61, 1.02], so the turn graph now produces faster programs, although that ratio is not significant. These comparisons use 28 and 27 kernels, respectively. For 120B, the ratios are 1.07 and 1.39, and neither is significant.

For Qwen3.8 without tools, the optimization graph gives speed ratios of 0.88 [0.64, 1.20] under direct coding and 1.02 [0.89, 1.18] under delegation. Neither interval excludes one. History therefore requires validation for the model and coding workflow; the positive GPT-OSS direct-coding result is not a model-independent rule.

Interaction between history and coding mode. Section 4.3 reports this interaction. To compute it on the same workloads, we retain kernels with timings for all four configurations `L-A1-T0-off`, `L-A1-T0-opt`, `M-A1-T0-off`, and `M-A1-T0-opt`. For each kernel, we divide the direct-coding history ratio by the delegated-coding history ratio, then aggregate kernels equally. A value above one means that adding the optimization graph is more favorable under direct coding; it does not mean that direct coding is faster in absolute terms. Requiring the same four configurations for every kernel keeps differences between kernel populations out of the comparison.

History also changes how much the agents search. The Evaluations column counts evaluated candidates, including unsuccessful ones. Under delegation without tools and skills, the optimization graph changes that count by factors of 1.01 for 20B and 0.82 for 120B: about 1% more evaluations in the first case and 18% fewer in the second. The common turn limits bound the workflow, but do not force equal evaluation counts or token cost.

G Search time, lead occupancy, and token cost by role

G.1 Search time and evaluator time

Faster generated code can require a longer optimization process. Table 11 characterizes that process over the full 54-kernel suite, including unsuccessful searches. Two 120B searches, `L-A0-T0-off` on `azimhist` and `conv2d`, failed at agent start-up with an import error in under a second and recorded no model usage; their token and evaluation counts are unknown, which leaves 1,294 values for those measures. Elapsed time uses the recorded search duration or, for continuations, the union of logged active intervals, excluding gaps and counting copied records once. Evaluator time measures the work of compiling, validating, and benchmarking candidates; cumulative model-call time sums the durations of requests across roles. These quantities describe different parts of the workflow and should not be added or subtracted to infer an exclusive breakdown of elapsed time.

The distributions have long upper tails: the 90th percentiles of elapsed time are about three times the medians for both GPT-OSS models. Because the models ran under different serving loads, these times are not comparable across models. Token counts measure request volume under each model's tokenizer. Figure 12 separates the distributions by configuration: each dot shows a typical search, while the thick and thin segments expose variation across kernels.

Table 12 compares components on matched kernels and settings within each model. For GPT-OSS direct coding, analysis makes searches 30% and 62% longer, while evaluator time rises by only 4% and 1%. Cumulative model-call time rises by 34% and 97%, while output tokens grow by 60% and 132% (Table 13). Under delegation, evaluator time rises as well.

For Qwen3.8, analyzed feedback raises recorded elapsed time by 69% under direct coding and 173% under delegation, over 167 and 168 timing-eligible matched pairs, respectively. The larger delegated time increase accompanies its threefold token cost: the stronger program-speed gain also requires more search work.

Table 11: Full-suite search-cost distributions. n gives the number of searches with each measure recorded; the two 120B start-up failures have no token or evaluation counts. M denotes millions of tokens. Elapsed time includes inference, evaluation, and waiting; evaluator time covers compilation, validation, and benchmarking. Model-call time sums request durations. Elapsed and model-call times depend on the serving load, which differs between models (Appendix A.1).

Measure	Model	n	Mean	Median	90th pct.	Maximum
Elapsed (min)	20B	1296	72.11	52.09	162.67	358.72
Elapsed (min)	120B	1296	45.24	33.83	94.87	330.91
Elapsed (min)	Q3.8-27B	1296	1248.92	1133.26	2351.15	5698.29
Evaluator (min)	20B	1288	9.73	3.57	25.66	125.64
Evaluator (min)	120B	1287	10.95	3.55	27.92	122.61
Evaluator (min)	Q3.8-27B	679	13.21	3.74	33.02	306.48
Model calls (min)	20B	1288	59.18	39.57	142.35	345.72
Model calls (min)	120B	1287	30.61	23.50	69.08	193.43
Model calls (min)	Q3.8-27B	679	787.06	708.22	1407.58	2426.50
Evaluations	20B	1296	18.63	16.00	38.00	55.00
Evaluations	120B	1294	17.23	13.00	41.00	55.00
Evaluations	Q3.8-27B	1296	18.23	15.00	30.00	127.00
All tokens (M)	20B	1296	2.00	1.65	3.72	17.88
All tokens (M)	120B	1294	1.01	0.95	1.75	5.18
All tokens (M)	Q3.8-27B	1296	3.64	3.35	6.36	17.10
Input tokens (M)	20B	1296	1.71	1.31	3.32	17.69
Input tokens (M)	120B	1294	0.92	0.87	1.60	5.14
Input tokens (M)	Q3.8-27B	1296	2.28	2.15	4.08	8.69
Output tokens (M)	20B	1296	0.29	0.17	0.70	2.73
Output tokens (M)	120B	1294	0.09	0.07	0.21	0.56
Output tokens (M)	Q3.8-27B	1296	1.36	1.08	2.67	9.80

Table 12: Matched cost ratios on the 28-kernel panel. n_t, n_e, and n_v count pairs with elapsed durations, evaluator durations, and evaluation counts on each side. All pairs have usable candidate timings. Ratios divide totals; values below one mean less in the numerator setting.

Contrast	Model	n_t	Elapsed	n_e	Evaluator	n_v	Evaluations
Analysis: direct	20B	165	1.30	165	1.04	165	1.23
Analysis: direct	120B	165	1.62	163	1.01	165	1.12
Analysis: direct	Q3.8-27B	167	1.69	29	1.16	167	0.88
Analysis: delegated	20B	164	1.67	161	1.66	164	1.79
Analysis: delegated	120B	167	1.73	164	1.28	167	1.26
Analysis: delegated	Q3.8-27B	168	2.73	24	1.48	168	1.68
Tools: direct, A1	20B	82	1.69	82	0.97	82	0.98
Tools: direct, A1	120B	83	1.38	81	0.95	83	1.00
Tools: direct, A1	Q3.8-27B	84	1.21	7	0.63	84	0.90
Tools: delegated, A1	20B	83	1.62	81	1.16	83	1.03
Tools: delegated, A1	120B	83	1.14	80	0.96	83	0.92
Tools: delegated, A1	Q3.8-27B	84	1.32	4	1.42	84	1.02

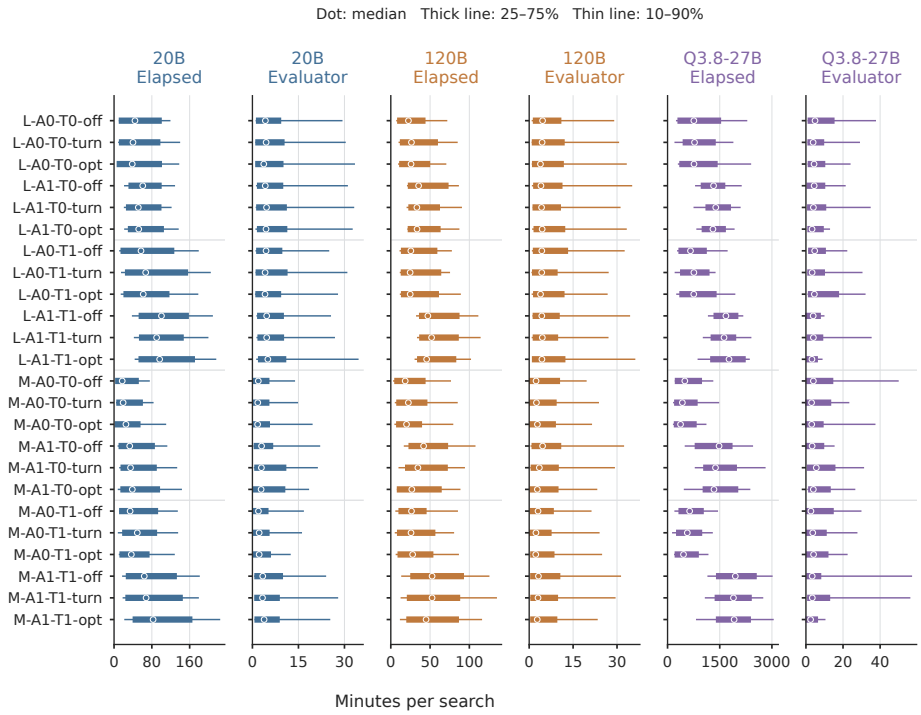

Figure 12: Search-time distributions for every configuration on the full suite. Dots mark medians, thick segments the interquartile range, and thin segments the 10th–90th percentiles across kernels. Configuration codes follow Table 1. Elapsed time includes inference, evaluation, and waiting; evaluator time covers native-evaluation execution. Horizontal ranges follow each model's recorded durations; serving loads differ across models.

G.2 Lead occupancy, context cleanups, and total tokens

Lead occupancy measures how much of the context window a lead request uses: 131,072 tokens for GPT-OSS and 262,144 for Qwen3.8. We first average occupancy within each search, then across matched settings and kernels. Total token cost answers a different question: it adds all input and output tokens across every role and request, including repeated prompts.

Table 13 makes this distinction explicit. For GPT-OSS direct coding, analysis reduces input tokens to 0.36 and 0.71 of the raw-feedback totals while increasing output tokens to 1.60 and 2.32. The larger reduction in input yields total-token ratios of 0.47 and 0.78. Under delegation, lead occupancy also falls, but the 120B input and output totals both increase. The context reduction is a property of the lead's requests; it does not guarantee a lower cost for the complete search.

Table 13: Analysis on/off on the 28-kernel panel. n_c counts pairs with lead occupancy measurements and n_t pairs with token usage. Raw and Analyzed are mean lead occupancy percentages. Input, Output, and Total are token ratios across all roles.

Model	Coding	n_c	Raw (%)	Analyzed (%)	n_t	Input	Output	Total
20B	Direct	165	57.1	8.6	165	0.36	1.60	0.47
20B	Delegated	161	60.3	21.8	164	0.61	2.20	0.71
120B	Direct	165	53.5	9.1	165	0.71	2.32	0.78
120B	Delegated	164	55.9	23.8	167	1.54	2.40	1.60
Q3.8-27B	Direct	167	32.7	12.4	167	0.77	1.92	1.01
Q3.8-27B	Delegated	168	37.4	36.2	168	3.12	2.84	3.00

For Qwen3.8 direct coding, analysis lowers mean lead occupancy from 32.7% to 12.4% and input tokens to 0.77 of the raw-feedback total, while output tokens rise to 1.92. The resulting total-token ratio is 1.01: less lead context does not deliver the GPT-OSS token saving. With delegation, the

lead occupancy changes only from 37.4% to 36.2%, and total tokens increase threefold. The same controller mechanism can therefore have different cost consequences across models and coding modes.

The harness shortens a role's conversation when a request would not fit the context window (Appendix K.8). Table 14 reports every configuration on the 28-kernel panel, using searches with recorded context events. With raw feedback, at least one cleanup occurs in 195 of 336 searches for 20B, 249 of 334 for 120B, and 291 of 336 for Qwen3.8. With analyzed feedback, the corresponding counts are two of 336, zero of 336, and 185 of 336. The reduction is substantial for every model, but analysis does not eliminate context pressure in Qwen3.8. These counts cover all roles, including analysis itself, and therefore complement lead occupancy rather than measure the same quantity.

Table 14: Context cleanups for all 24 configurations on the fixed 28-kernel panel. Searches counts searches with at least one cleanup; Per search is the mean number of cleanups across all roles.

	GPT-OSS-20B		GPT-OSS-120B		Qwen3.8-27B	
Configuration	Searches	Per search	Searches	Per search	Searches	Per search
L-A0-T0-off	3/28	0.11	9/26	0.35	22/28	0.89
L-A0-T0-turn	26/28	1.25	28/28	1.96	26/28	1.82
L-A0-T0-opt	27/28	1.43	28/28	1.36	27/28	1.68
L-A1-T0-off	0/28	0.00	0/28	0.00	0/28	0.00
L-A1-T0-turn	1/28	0.14	0/28	0.00	17/28	0.82
L-A1-T0-opt	0/28	0.00	0/28	0.00	0/28	0.00
L-A0-T1-off	11/28	0.43	10/28	0.36	25/28	1.00
L-A0-T1-turn	21/28	0.75	28/28	1.21	27/28	2.04
L-A0-T1-opt	22/28	0.89	28/28	1.07	28/28	1.96
L-A1-T1-off	0/28	0.00	0/28	0.00	1/28	0.04
L-A1-T1-turn	1/28	0.04	0/28	0.00	12/28	0.61
L-A1-T1-opt	0/28	0.00	0/28	0.00	1/28	0.04
M-A0-T0-off	10/28	0.43	20/28	1.61	19/28	2.54
M-A0-T0-turn	12/28	0.61	22/28	2.07	22/28	2.25
M-A0-T0-opt	12/28	0.75	19/28	1.71	26/28	2.82
M-A1-T0-off	0/28	0.00	0/28	0.00	22/28	1.89
M-A1-T0-turn	0/28	0.00	0/28	0.00	26/28	2.57
M-A1-T0-opt	0/28	0.00	0/28	0.00	26/28	2.00
M-A0-T1-off	19/28	1.36	22/28	1.86	20/28	2.11
M-A0-T1-turn	18/28	1.25	16/28	1.64	24/28	2.75
M-A0-T1-opt	14/28	0.82	19/28	1.79	25/28	2.36
M-A1-T1-off	0/28	0.00	0/28	0.00	27/28	2.39
M-A1-T1-turn	0/28	0.00	0/28	0.00	28/28	2.82
M-A1-T1-opt	0/28	0.00	0/28	0.00	25/28	2.18

G.3 Token cost by role

Figure 13 locates the token cost in the lead, coding, analysis, and repair roles. Each bar averages the three history settings over the full suite. An inactive role contributes zero to a search's total; searches without token measurements are omitted. The stacked totals include unsuccessful attempts, so they represent the cost of the optimization process and not only of its final candidate.

For direct coding without tools and skills, enabling analysis lowers mean lead tokens from 2.41 to 0.22 million per search for 20B and from 1.20 to 0.18 million for 120B. The analysis role itself uses 0.23 and 0.38 million tokens, less than the reduction in lead tokens in these groups. With tools and skills enabled, the analysis role uses 0.95 and 0.79 million tokens. Adding a role can reduce work in another role, while tools and skills can increase the material that those roles process.

Table 15 gives the same breakdown for the other models, averaged over all 24 configurations. The lead role accounts for 39–61% of their tokens. Repair varies most: it takes 39% of Qwen3.5-27B's tokens but 19% of Qwen3.5-397B's, so the smaller model spends much of its budget fixing rejected candidates.

Table 15: Mean millions of tokens per search by role, input and output combined, averaged over all 24 configurations on each model's kernels. Inactive roles contribute zero.

Model	Kernels	Lead	Coding	Analysis	Repair	Total
Qwen3.8-27B	54	1.96	0.34	0.80	0.54	3.64
Qwen3.5-27B	13	1.10	0.19	0.42	1.09	2.80
Qwen3.5-397B	13	1.16	0.16	0.58	0.45	2.34
Kimi-K2.7-Code	7	1.56	0.19	0.45	0.34	2.53

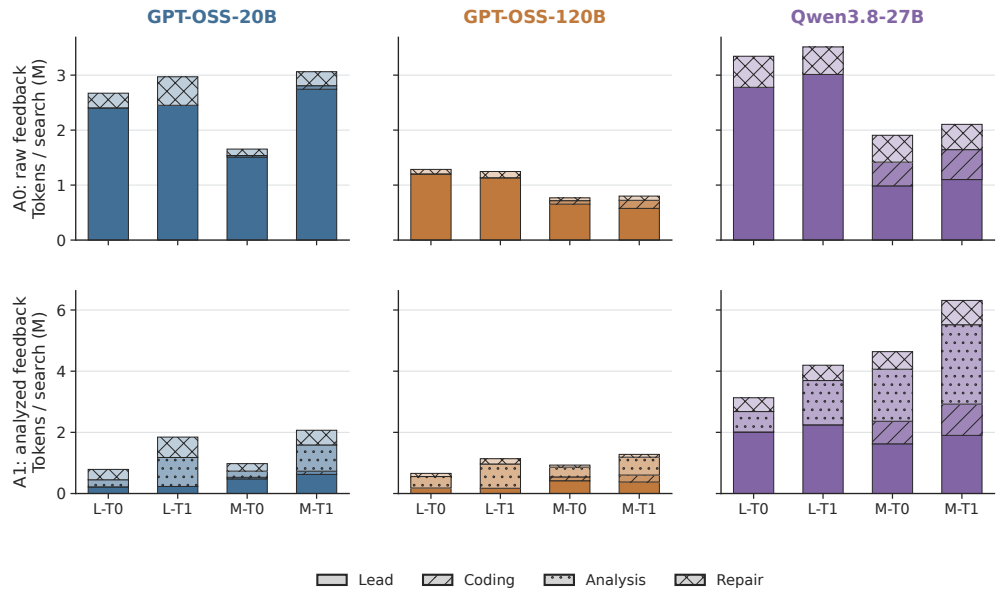

Figure 13: Mean tokens per search by role over the full suite. Row labels give the analysis setting; bar labels give coding and tools settings using Table 1. Each bar averages `off`, `turn`, and `opt` history. Colors identify models; shading and hatching identify roles. Vertical scales match across models within each row. These are group means, not matched component effects.

H Reference source length and search cost

H.1 Source-length groups

Reference source length describes the amount of code presented to the agents. We count nonblank lines that do not begin with a comment in the original NumPy file, including imports and helper functions. The count concerns the reference, not the generated C++ program. Sorting all 54 references by length and dividing them into three approximately equal groups gives boundaries of six and 13 lines, with equal-length references kept together. We use the same groups for every model and panel: short references have at most six lines, medium references seven to 13, and long references at least 14. Table 4 lists the individual counts.

H.2 Longer references require more search resources

Figure 14 relates reference length to token cost, search time, and evaluator time. Each dot first averages searches for one kernel; the black marks summarize the resulting kernel-level values. This prevents kernels with more recorded searches from dominating a group. Table 16 reports the corresponding rank correlations across kernels.

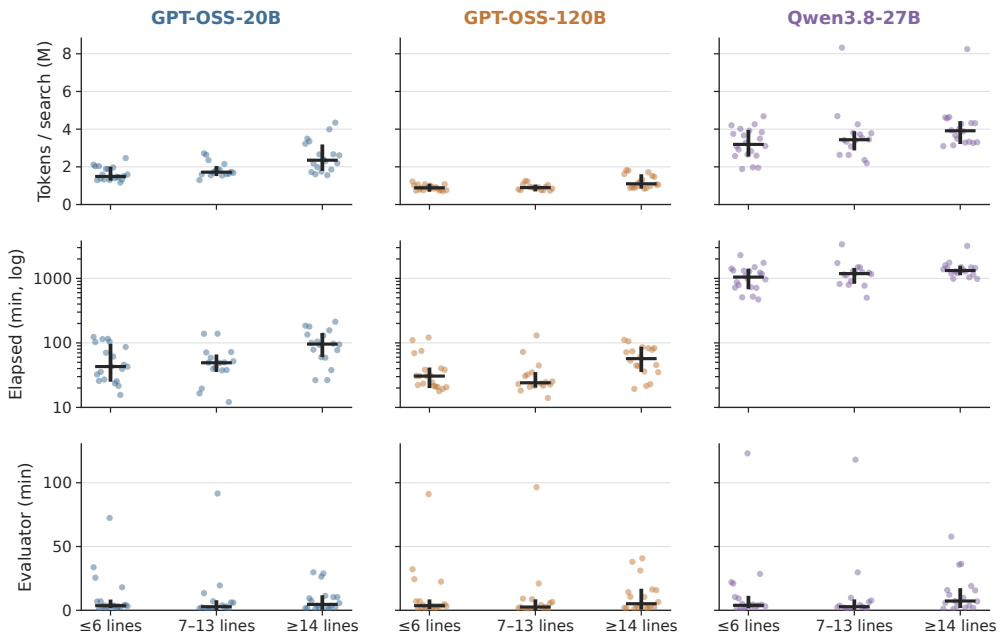

Figure 14: Search cost by reference-source length over the full suite. Each colored dot is one kernel's mean over configurations with the corresponding quantity recorded. Black ticks and vertical segments show the median and interquartile range across kernels. Group boundaries are shared by all three models. Axis limits are matched within each row; elapsed time uses a logarithmic scale.

Longer NumPy references are associated with higher token cost ($\rho = 0.56$ and 0.41 for 20B and 120B) and longer searches ($\rho = 0.33$ and 0.24). They also tend to produce larger lead inputs and more candidate evaluations. Reference length is thus related to the amount of work done during optimization.

The evaluator-time correlations, however, are close to zero (-0.07 and -0.05): a longer source file does not necessarily take longer to compile, validate, and benchmark over the search. Native-validity correlations are also weak (-0.10 and -0.07). Source length describes search cost in this suite, but it is not a general measure of runtime cost or optimization difficulty. The spread within each group further shows that similarly sized references can have very different costs.

Table 16: Spearman correlations between reference lines of code and kernel-level search outcomes. Parentheses give the number of kernels. Tokens, times, and lead occupancy are means over configurations; validity and timing availability are fractions of the 24 configurations. Log-speed variation is the standard deviation of log speed-ups across configurations with usable timings.

Metric	20B	120B	Q3.8-27B
Tokens	+0.56 (54)	+0.41 (54)	+0.34 (54)
Elapsed time	+0.33 (54)	+0.24 (54)	+0.33 (54)
Evaluator time	-0.07 (54)	-0.05 (54)	+0.06 (51)
Lead occupancy	+0.48 (54)	+0.55 (54)	+0.16 (54)
Native validity	-0.10 (54)	-0.07 (54)	+0.08 (54)
Usable timing	-0.18 (54)	-0.24 (54)	-0.13 (54)
Evaluations	+0.26 (54)	+0.35 (54)	+0.13 (54)
Log-speed variation	+0.19 (42)	+0.48 (43)	+0.45 (48)

For Qwen3.8, reference length also correlates with token cost ($\rho = 0.34$) and elapsed search time (0.33), but not evaluator time (0.06). This descriptive pattern spans all three models; measured search time also depends on serving conditions.

I SENSITIVITY TO OUTPUT-TOKEN WEIGHT

The main evaluation adds recorded input and output tokens with equal weight. To expose the role of this convention, let I_j and O_j sum input and output tokens over the same timing-eligible matched pairs for setting j. Define

$$C_w = I + wO, \qquad R_C(w) = \frac{I_1 + wO_1}{I_0 + wO_0}. \tag{3}$$

The weight changes accounting only: the programs, selected candidates, and optimization trajectories remain fixed. Figure 15 reports the direct-coding comparisons on the 28-kernel panel.

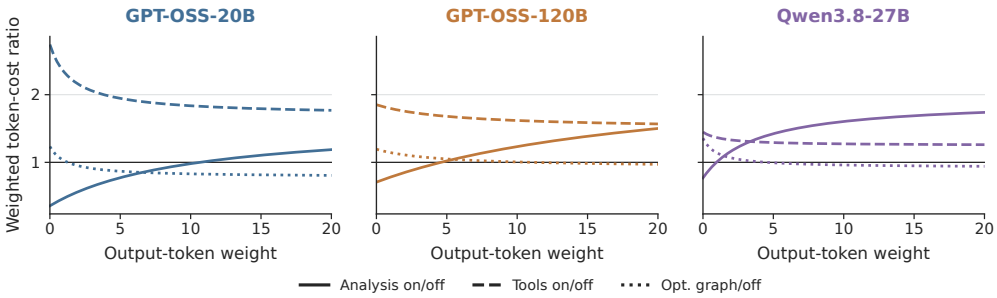

Figure 15: Weighted cost ratios for direct coding. Analysis compares on/off; tools compares on/off with analysis enabled; history compares the optimization graph with history off, with analysis enabled and tools disabled. Pair counts are 165/165, 82/83, and 28/27 for 20B/120B. The corresponding Qwen3.8 counts are 167, 84, and 28. $w = 1$ is equal token weighting. These are accounting weights, not model prices or cache discounts.

Analysis reduces direct-coding input tokens but increases output tokens. At the illustrative accounting weight $w = 4$, its cost ratios are 0.71 for 20B and 0.97 for 120B, savings of about 29% and 3%. The equal-cost crossings are $w = 10.64$ and $w = 4.58$, respectively, so the 120B saving disappears at a smaller output weight.

Enabling tools and skills increases both input and output tokens in these direct, analysis-enabled comparisons. Its weighted ratio remains above one for every nonnegative output weight. The optimization graph instead exchanges additional input for lower output: its equal-cost crossings are $w = 1.26$ for 20B and $w = 10.97$ for 120B. These crossings solve Equation 3 on the recorded totals and do not change the measured speed-up effects.

For Qwen3.8 direct coding, the analysis equal-cost crossing is $w = 0.95$. Its input reduction is already offset by additional output near equal weighting, and the weighted ratio rises further as output receives more weight. The optimization-graph crossing is $w = 4.49$; tools and skills increase both input and output, so their cost ratio stays above one for all nonnegative weights.

J GENERATED PROGRAMS BEHIND STABLE AND VARYING OUTCOMES

To connect the configuration comparisons to generated code, we hold the model fixed at 120B and examine three kernels. CRC16 gives similar speed-ups across configurations; convolution and `ludcmp` (LU decomposition followed by a triangular solve) show much larger differences. Figure 16 places all configurations in the same order so that these patterns can be compared directly. The implementations discussed below are the selected programs for the stated configurations.

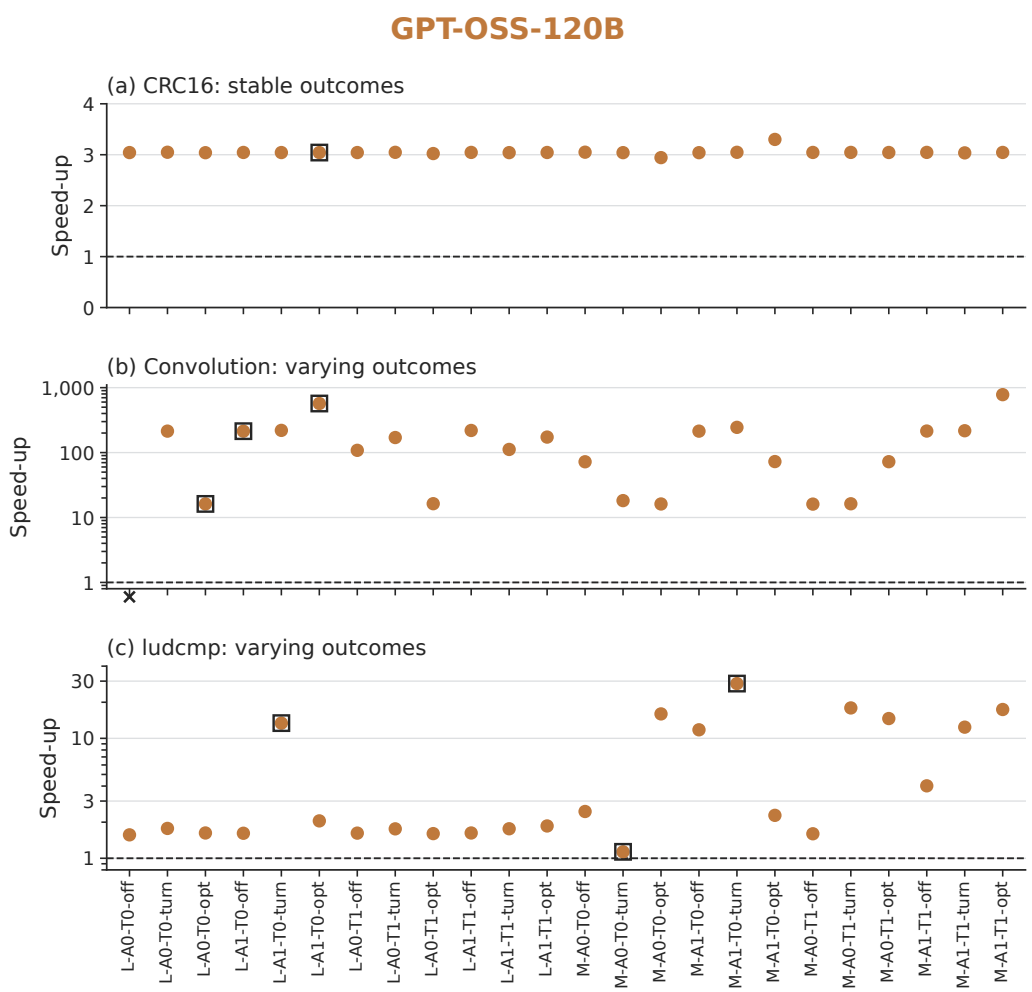

Figure 16: GPT-OSS-120B speed-ups across all 24 configurations for three example kernels. CRC16 uses a linear scale; convolution and `ludcmp` use logarithmic scales. The dashed line marks baseline-equivalent performance. A cross below the convolution axis marks a configuration without a usable timing. Configuration codes follow Table 1; highlighted squares identify the implementations discussed in the text.

CRC16: similar performance from a shared update scheme. All 24 configurations produce speed-ups between 2.94× and 3.30×. The ten inspected selected programs use lookup tables to replace bit-by-bit polynomial updates. For example, `L-A1-T0-opt` constructs a 256-entry table at compile time and processes two bytes per loop iteration, reaching 3.04×. Other inspected variants change the unrolling or table construction while retaining the same byte-wise update scheme. The small performance range contrasts with the large differences between agent workflows: changing the workflow need not lead to a substantially different program.

Convolution: changing the traversal changes reuse. The timed configurations range from 16.08× to 780.18×. Three inspected implementations illustrate different traversals. `L-A0-T0-opt` computes one output channel at a time and revisits the input patch for each channel, reaching 16.19×. `L-A1-T0-off` instead accumulates output channels together, broadcasting each input value across two eight-wide vectors of channels; it reaches 213.61×. `L-A1-T0-opt`

additionally keeps a 4×4 tile of output pixels in progress and reuses each weight vector across that tile, reaching $569.85\times$. The programs differ in which results are accumulated together.

`ludcmp`: organizing parallel work matters. The `ludcmp` configurations span $1.13\times$ to $28.62\times$. `M-A0-T0-turn` computes factor entries through serial row–column dot products and reaches $1.13\times$. `M-A1-T0-turn` changes the factorization to parallel updates of the trailing matrix, with workers traversing contiguous rows after each pivot; it reaches $28.62\times$. A third implementation, `L-A1-T0-turn`, factors panels and parallelizes the trailing-block update, reaching $13.40\times$. The different outcomes expose choices about work distribution and data traversal that are hidden by a configuration's aggregate speed-up.

These examples identify concrete implementation differences behind observed outcomes. They also show why a single workflow need not dominate every kernel: several workflows can converge on a similar update scheme, or they can produce programs with different decompositions of the same computation. Each configuration contributes one search, so the examples describe its selected program rather than the probability that a component discovers a particular strategy.

K Implementation Details

This section follows a candidate through the four stages of our framework: planning a change, generating code, evaluating it, and preparing feedback for the next turn. Figure 17 expands this workflow and locates the four configuration choices A–D within it.

K.1 Stage 1: planning the next change

The benchmark adapter prepares a workspace and a task prompt that establish the required behavior, candidate interface, execution constraints, and reference implementation. Within this contract, the lead decides which implementation to build on and which change to attempt. It can continue from the incumbent—the best valid, timed candidate under the evaluation objective—or revisit an earlier candidate, for example when an analysis report identifies a promising direction.

To carry its reasoning between turns, the lead can update an optimization plan and save working notes. The plan records completed work and proposed next steps; the notes can retain observations about previous attempts, constraints to preserve, and ideas worth revisiting. The harness reuses both when constructing the lead's next prompt, together with the available feedback. This working state and the saved candidates remain available under every history setting.

K.2 Stage 2: direct or delegated coding (A)

With direct coding, the lead writes and submits a complete candidate. With delegation, it requests one coding agent to implement the chosen change, supplying the starting implementation, task contract, current plan, and relevant skills. The coding agent can return up to three initial variants, which the controller evaluates in sequence. These variants share a budget of five evaluations: each evaluation of an initial or repaired version consumes one. For example, spending three evaluations on the first variant leaves two for the remaining variants. The controller returns the best measured valid candidate and retains feedback on the evaluated alternatives. A direct submission has the same limit of five evaluations for its candidate and repairs.

All configurations allow up to ten numbered lead turns. Delegated searches first use one coding agent to produce a correctness-oriented implementation at turn zero. Within each numbered turn, the lead can make up to forty model-response iterations, using tools between responses until it submits, delegates, or stops. This limit applies to the lead's conversation loop; coding, repair, and analysis requests have their own loops. The four roles use separate prompts and the same model within each configuration. Table 17 distinguishes these limits.

K.3 Stage 3: evaluation and repair

Both coding routes pass their candidates to the same evaluator. For NPBench, a candidate is a complete C++20 implementation matching the task's generated C interface. The evaluator checks

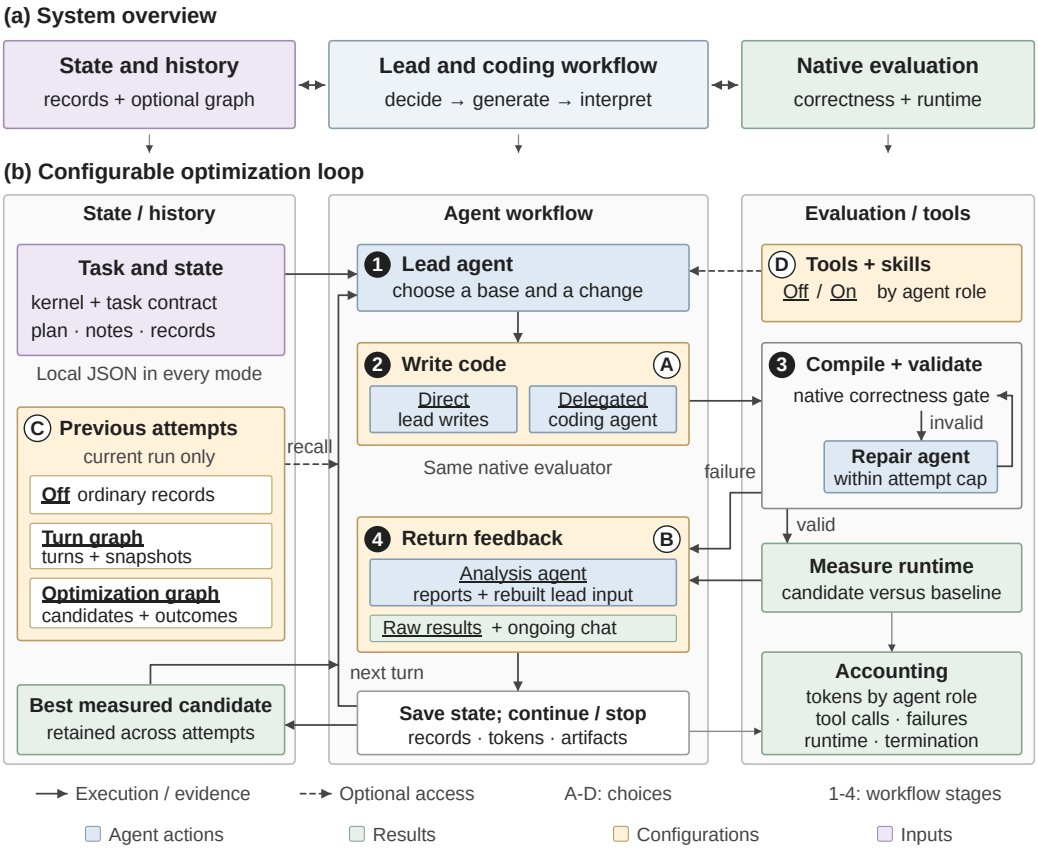

Figure 17: Detailed workflow of the framework. Panel (a) locates the three components expanded in panel (b). Blue boxes denote agent actions, green boxes denote results and accounting, purple boxes denote inputs and retained state, and yellow boxes group configurable alternatives. Filled circles 1–4 mark the workflow stages; outlined circles A–D identify the configuration choices. Solid arrows carry execution or evidence, while dashed arrows indicate optional tool or history access. The return path connects saved state and feedback to the next lead turn.

Table 17: Controller limits. Choice A determines who writes the candidate; variants and repairs consume the same evaluation budget within a submission or delegated request.

Control	Setting	Scope
Lead turns	Up to 10	Numbered turns, after delegated initialization.
Lead iterations	Up to 40	Model-response iterations within one lead turn.
Coding agents (A: delegated)	1	Per delegated turn, including initialization.
Candidate variants (A: delegated)	Up to 3	Initial variants from that coding agent.
Evaluations (A: either)	Up to 5	Per submission or delegated request, including repairs.
Model assignment	One model	Used by the lead, coding, repair, and analysis roles.

its outputs and observable input mutations against the official NumPy implementation (Harris et al., 2020). Only candidates that pass these checks proceed to timing, where the configured Numba implementation supplies the performance baseline (Lam et al., 2015).

A rejected candidate can be sent to a repair agent in either coding mode. Given the source and failure diagnostic, this agent is instructed to correct the failure while preserving the intended optimization. The evaluator then checks the revised candidate, consuming another evaluation from the budget described above. Repair continues until the candidate passes, no corrected candidate is returned, or the budget is exhausted.

The controller retains the valid, timed candidate with the highest measured speed-up, defined as the ratio of baseline median runtime to candidate median runtime from that candidate's evaluation. Analysis can recommend the next change or starting implementation, but does not alter measured outcomes. Appendix B specifies the validation and performance-evaluation protocol.

K.4 Stage 4: raw or analyzed feedback (B)

With raw feedback, evaluation results and diagnostics are appended to the lead's ongoing conversation. With analysis, an analysis agent first receives the candidate code, measured outcomes, and failure evidence; for delegated coding, it considers the evaluated variants together. Its report interprets the measurements, separates observations from possible explanations, and proposes subsequent changes.

The harness then constructs a fresh lead prompt from the task, plan, notes, and accumulated analysis reports. A compact candidate index preserves identities and validity status so that the lead can select a starting implementation, but candidate source, raw timing arrays, and evaluator logs are not automatically copied into this prompt. The analysis reports provide the evaluation feedback. Saved artifacts remain available through explicit candidate access and, when enabled under C, graph exploration.

K.5 History within a search (C)

Choice C determines how the lead can revisit earlier attempts in the current search. Ordinary working state remains available with history off. With the turn graph, the harness saves completed-turn snapshots containing the plan, notes, candidate records, feedback, and workspace artifacts. The lead can inspect a turn or retrieve a saved text artifact. With the optimization graph, it instead explores candidate implementations and their evaluation evidence. Candidate source hashes identify implementations, and an explicitly supplied base identifies a parent implementation; repeated source updates the stored evidence rather than creating another implementation node.

Both graphs also supply automatic recall before a lead turn. The query draws on the task, plan, and recent analyses or errors, and matches terms in saved candidate descriptions and evidence. Matching records are ordered by correctness, speed-up, and recency, with up to eight summaries added to the prompt. The lead can request further summaries or use the selected explorer to inspect source and detailed evidence. Thus automatic recall provides a compact entry point, while explicit exploration supports closer examination of a previous attempt. The evaluated history scope is restricted to the current search.

K.6 Tools and skills (D)

Choice D enables tools together with their accompanying skills. Skills explain how to use the available tools and provide guidance on coding practices and optimization strategies. The harness selects them for each role according to the language, task, and tool availability; the lead can also pass relevant guidance when delegating a change.

The lead, coding, repair, and analysis roles can use web search, while the analysis role can request profiling and inspect the resulting artifacts. Disabling D removes these tools and skills while retaining the controls needed to plan, save notes, access candidates, submit or delegate code, and evaluate it. Graph recall and exploration are controlled separately by C. On the GPT-OSS 28-kernel panel, the tools-on searches issued 243 web searches for 20B and 40 for 120B, fewer than one per search. The profiler was called 1,958 times for 20B and 2,626 times for 120B (Appendix K.7).

K.7 Execution resources and timing

Native evaluations acquire CPU allocations from a shared resource manager, which assigns disjoint CPU sets and enforces affinity and thread counts. Baseline and candidate measurements use the same allocation and execution settings, while model inference runs in a separate allocation. Compilation, setup, allocation, and input copies occur outside the measured native call. Warmup, repetitions, and cache handling follow Appendix B.

To reduce the influence of variation in individual baseline measurements, reporting uses a common baseline for each kernel. We pool compatible baseline repetitions across searches, remove duplicate timing batches, and take their arithmetic mean. Compatibility requires the same task preset, baseline implementation, thread count, cache mode, and timing protocol. The reported speed-up divides this common baseline by the selected candidate's mean runtime; online candidate selection maximizes the ratio of baseline median runtime to candidate median runtime from each candidate's evaluation. Pooling reduces reliance on a single baseline batch and gives all configurations of a kernel the same performance reference.

Profiling supplies additional diagnostic evidence in separate executions, leaving the clean timing samples used for candidate ranking unchanged. With tools and skills enabled, every validated candidate is profiled once automatically with hardware counters. For CPU code, the profiling tool supports `perf` hardware counters and can select a configured PAPI measurement wrapper. The analysis agent can request these backends through `profile_performance`, inspect their output, and use it to explain a bottleneck or propose a subsequent change. PAPI requires a wrapper that brackets the measured region; its installation alone does not provide that instrumentation.

K.8 Token accounting and context management

Token cost sums the recorded input and output usage of completed model calls across the lead, coding, repair, and analysis roles. Model work performed through tools is assigned to the invoking role. Repeated input counts each time it is processed, and input and output totals are retained separately alongside evaluation counts and wall time.

The harness also tracks how often each role needs to shorten its conversation to fit the model's context window. Before a request, it estimates the input size and reserves space for the requested output; cleanup can also be triggered by a server context-length rejection. Cleanup retains the system instructions and approximately the newest half of the removable conversation, preserving tool-call/result pairs. Each actual shortening increments that role's cleanup count and records the before-and-after context estimates. These counts distinguish a conversation that stays small from one that repeatedly fills its window and must discard older exchanges; Appendix G.2 reports them across all configurations. Model-serving and context-window settings are given in Appendix A.1.