Accelerate Bench: Evaluating Agents for Application-Level Program Optimization

Tencent Hy Team HKUST HKUST(GZ) THU PKU

Application optimization requires coding agents to discover useful changes and coordinate them across a complete program while preserving the required computation. We introduce Accelerate Bench, a benchmark of 32 CPU optimization tasks adapted from scientific applications and supercomputing competitions. Each task provides a working program and specifies the workload, correctness requirements, timing boundaries, and permitted edits, leaving agents to choose where and how to optimize. Public testing supports iterative development, and a separate verifier scores the final submission. We evaluate 12 coding agents, each pairing a model with a harness. Of their 325 correct outcomes, 161 (49.5%) stay below $1.25\times$ speedup. Controlled replays show why coordination across an application matters: identical source code runs between $0.29\times$ and $2.12\times$ depending only on build and runtime settings, and one code change is worth $1.01\times$ or $6.4\times$ depending on another. Accelerate Bench pairs executable workloads with agents' programs and trajectories to evaluate both their gains and the process behind them.

1. Introduction

Software performance engineering improves application performance by identifying and addressing inefficiencies in software and the way it executes (Woodside et al., 2007; Leiserson et al., 2020). Effective optimization often requires interventions across multiple layers of the software stack, from compiler and build configurations to algorithms, data structures, dependencies, and hardware parallelism (Leiserson et al., 2020; Balaprakash et al., 2018). For an existing application, this is typically an iterative engineering process: measuring execution, locating performance bottlenecks, implementing targeted changes, and evaluating their end-to-end impact while preserving correctness (Woodside et al., 2007; Balaprakash et al., 2018).

Application-level optimization therefore extends beyond localized code editing. A valid patch may leave end-to-end runtime unchanged, and effective speedups can depend on interactions among changes made across different layers of the software stack. Success requires discovering where performance is lost, deciding where to intervene, and verifying that the resulting program is both correct and faster at the workload level.

Coding agents offer a path toward automating this process. Existing benchmarks such as SWE-Perf, GSO, and SWE-fficiency evaluate performance optimization in real software repositories using tasks derived from human optimization commits or pull requests (He et al., 2026; Shetty et al., 2025; Ma et al., 2026). These benchmarks anchor evaluation around known optimization instances, providing realistic repositories and expert optimization references.

We study a complementary setting: can an agent optimize an application when the required behavior and workload are specified, but no known optimization patch determines where or how the agent should intervene? Inspired by behavioral specification in ProgramBench (Yang et al., 2026), we define optimization tasks by the computation that must be preserved and the workload that must become faster. Within a permitted edit scope, the agent may modify algorithms, implementation, build settings, dependencies, and runtime configuration.

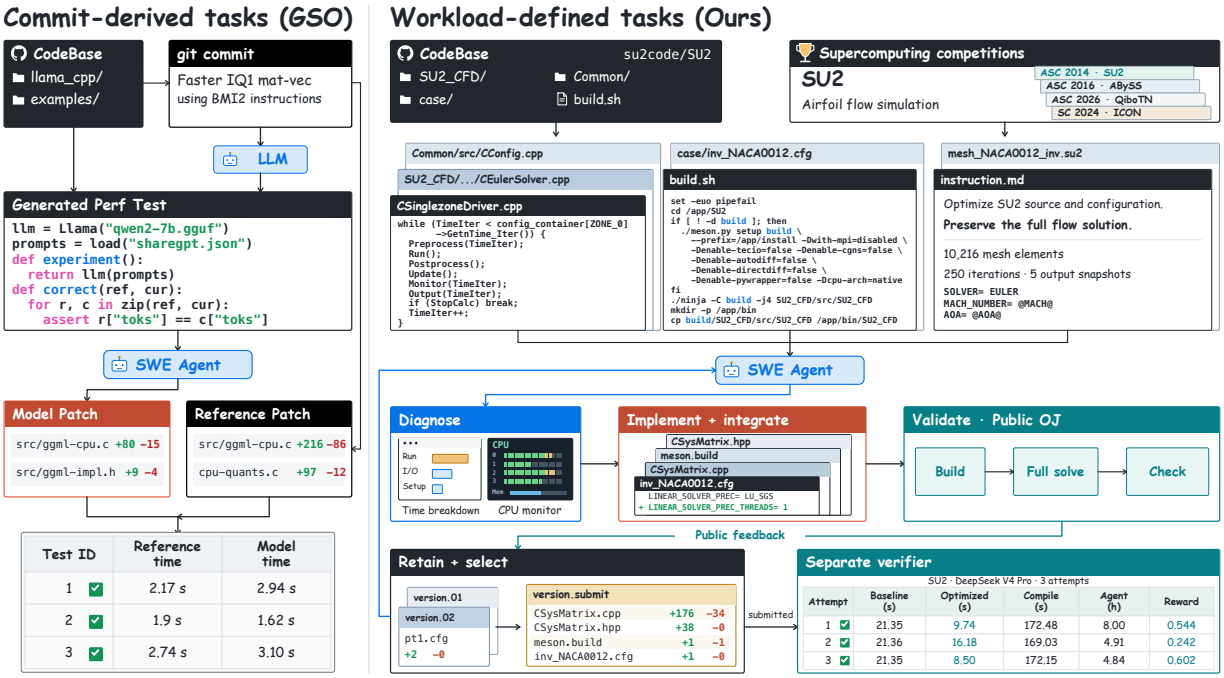


Figure 1 | Application performance engineering in Accelerate Bench. Left: GSO derives optimization tasks and reference patches from human commits (Shetty et al., 2025). Right: Accelerate Bench defines tasks by an application and its workload, illustrated with `SU2`. The agent diagnoses performance, modifies implementation, compiler options, and runtime configuration, and uses complete public tests to guide revisions and candidate selection. A separate verifier rebuilds and scores the final submission. GSO's table shows three example tests; the `SU2` table reports three DeepSeek V4 Pro runs. `SU2` runtime sums the mean solver times over three verification cases. Compile time is reported separately and excluded from reward; Agent time covers the agent-execution phase. The diagnostic displays illustrate the tools used in this workflow.

We introduce **Accelerate Bench**, a benchmark of 32 CPU optimization tasks adapted from scientific applications and supercomputing competition problems. Each task provides a working initial program together with correctness requirements, timing boundaries, permitted edits, and a public testing interface. Agents iteratively edit, build, execute, and evaluate candidate programs, while a separate verifier scores the final submission. This setup requires agents to discover optimization opportunities and convert them into verified end-to-end acceleration.

We evaluate 12 coding agents on Accelerate Bench, comparing correctness and end-to-end acceleration across the 32 tasks. Their outcomes range from correct programs with little or no acceleration to substantial workload-level speedups. Nearly half of the correct outcomes remain below 1.25× speedup, quantifying the gap between a correct program and a substantial runtime reduction.

Accelerate Bench also supports detailed analysis of the optimization decisions behind these results. We address three questions: what changes agents make to an application, how they develop and select candidate programs, and how optimization choices interact to affect end-to-end performance. Submitted programs, trajectories, and controlled replays supply the evidence. Our contributions are:

- **A workload-defined optimization benchmark.** Each of the 32 tasks fixes the computation, complete outputs, stateful checks such as checkpoint recovery, and timing boundaries; public tests support iteration, and a private verifier scores the final submission. In 14 planned controls, all eight invalid variants fail the check they violate (Section 2).

- **An evaluation of 12 coding agents.** Each agent is correct on 24 to 30 of the 32 tasks, but the best Score@1 is 40.3%, and 161 of 325 correct outcomes stay below 1.25× speedup (Section 4).
- **An analysis of how agents optimize.** From 480 runs of five agents and controlled replays of their programs, we find that these agents mostly change the computation and rarely the build, even though build and runtime settings alone can substantially speed up an unchanged program. Code changes also interact: a change that gains little on its own can yield a large speedup once another change is in place. During development, their first edited programs rarely pass the public tests, and they sometimes submit a slower program than the best one they tested (Section 5).

We will release the versioned task packages, including public tests and verifiers.

2. Benchmark Construction

Accelerate Bench comprises 32 CPU optimization tasks adapted from mature scientific applications across six established supercomputing competitions. We adapt each task into an executable workload running in a resource-constrained environment, with explicit permitted modifications, timing boundaries, and output requirements. We pair interactive public feedback during development with held-out private verification. Task designs were reviewed by experienced finalists from the ASC Student Supercomputer Challenge. Appendix B.1 documents task origins and adaptations.

2.1. Task Definition and Optimization Scope

Unlike benchmarks built around isolated historical commits or patches (Shetty et al., 2025; He et al., 2026), our tasks are defined by complete application workloads and required outputs. Each task fixes the target computation and leaves agents to decide where and how to optimize. We evaluate performance across the full workflow, so local improvements count only if they reduce total runtime.

Each task provides a working baseline implementation, build and execution scripts, task instructions, and a public testing interface in a container with fixed CPU core and memory limits (Appendix B). Agents may modify application source code, compiler options, and dependencies within the permitted directory, while preserving the required computations and outputs. Application-level optimization means reducing the end-to-end runtime of this prescribed workflow under these resource limits.

Figure 1 illustrates this workflow with SU2: agents optimize an airfoil flow simulation by editing solver routines, common libraries, and compilation flags. To pass, each run must complete all 250 iterations on the supplied 10,216-element mesh and produce five complete flow-field snapshots. The public test rebuilds the candidate, runs the simulation, and checks field residuals and run-to-run consistency against reference solutions. The verifier alone controls the validation meshes, physical configurations, and the checking code used to score submissions.

Timing boundaries follow each task's computational workflow, excluding one-time setup and build steps. For example, GROMACS times the `mdrun` simulation, excluding build time and topology preparation; the HPCG-derived task times sparse matrix assembly, initialization, multigrid solves, and output; and paged generation times model loading, autoregressive decoding, checkpoint saving, and recovery in a new process. Because these stages are timed and checked together, a faster kernel counts only in proportion to its share of the workflow, and a shortcut that skips a required output or recovery step fails correctness rather than earning a speedup. Appendix Table 7 gives representative contracts, and Table 5 lists each task's CPU cores, memory, and time limits; every task package specifies its own outputs and checks.

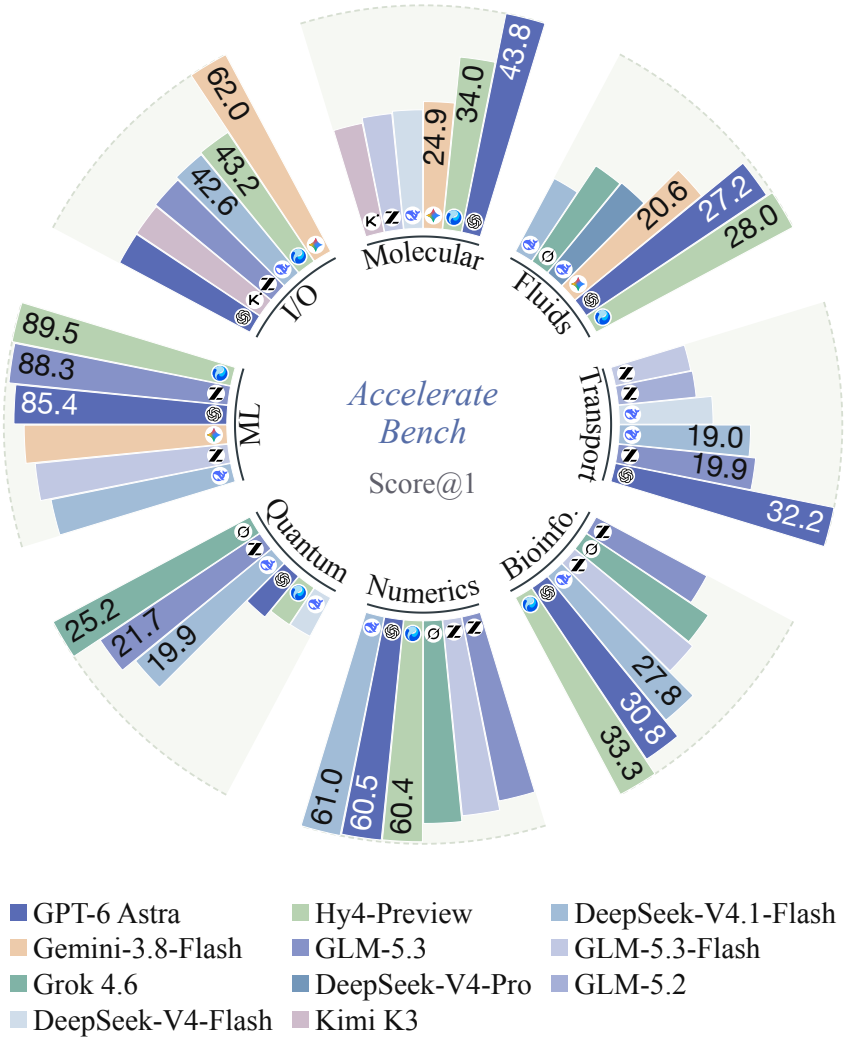


Figure 2 | Score@1 by application area. Top six systems per area, increasing clockwise. Labels give the top three scores (%); each sector is scaled to its own maximum (dashed arc).

2.2. Task Adaptation, Domain Coverage, and Workload Scaling

The 32 tasks span eight application areas (Figure 2) curated from six supercomputing competitions: ASC, PAC, the ACM-China International Parallel Computing Challenge (IPCC), SCC at SC and ISC, and IndySCC (Appendix Table 8). Each task adapts a competition's target application and optimization challenge into a bounded, self-contained workload rather than reproducing the original input (Appendix B.1). We standardize dependencies, build scripts, and execution entry points in isolated containers, using inputs from canonical benchmarks, public datasets, or synthetic generators.

Workload scaling. Competition workloads typically run on multi-node clusters for hours. To allow agents to iteratively edit, build, and test within a practical evaluation budget, we adapt these workloads to single-node CPU environments by reducing input sizes and iteration counts. The adapted tasks retain the main computational stages, inter-stage dependencies, and numerical correctness criteria of the original applications (adaptation parameters in Appendix B.1).

2.3. Public Tests, Private Verification, and Checks

Agents develop against a public test; a private verifier scores the submission (Figure 1). During the run, the agent can call the public test repeatedly; it rebuilds the current program, runs it on a few public inputs (typically two), checks the outputs, and reports correctness, runtime and, in most tasks, the baseline runtime or speedup, or the failing step. The agent sees whether its program builds, produces the required outputs, and runs faster, but not the checking code. After the run, the hidden verifier rebuilds the submitted files and runs them on private inputs of the same kind, typically four. Most tasks include the public inputs and held-out inputs that are often larger or freshly randomized per evaluation, such as other Mach numbers and angles of attack for SU2. Only the verifier's repeated timings and checks set the reward (Section 3), so the score depends on inputs the agent has not seen.

To prevent invalid optimizations—such as skipping required computations or omitting output fields—the verifier enforces correctness checks. Submissions must meet task-specific numerical tolerances, generate complete output files, and satisfy stateful requirements such as recovering correctly from a saved checkpoint in a new process; only submissions passing all checks are scored on runtime. In 14 planned controls on HPCG-derived and paged generation (Appendix C), every valid variant passes and all eight invalid variants fail the check they violate: omitting the second solution fails the output-size check, and four attention variants that pass component tests fail application checks.

3. Evaluation Metrics

Using the task-provided implementation as the baseline, we measure the runtime agents save while preserving the required computation. All metrics weight the 32 tasks equally.

Runtime savings. For task t, let B_t be the baseline runtime and $T_t(a)$ the runtime of a submitted program a, measured under the same timing rule. Let $C_t(a) \in \{0, 1\}$ indicate whether the program passes the final correctness checks. For outcomes with a valid runtime, the reward is defined as

$$r_t(a) = C_t(a) \max\left(0, 1 - \frac{T_t(a)}{B_t}\right). \tag{1}$$

Failed outcomes and those without a valid runtime receive zero. Thus, a correct 2× speedup earns 0.5 and a correct but no faster program earns zero; the reward is the fraction of baseline runtime saved.

Scored run. Each run gives the agent eight hours to inspect, edit, build and test the program before its final submission is scored. We denote the run of agent m on task t by $a^*_{m,t}$; all metrics and usage counters refer to its final submission. Our primary metric, **Score@1**, is the mean reward over tasks:

$$\text{Score@1}_m = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} r_t(a^*_{m,t}). \tag{2}$$

Speedup targets. We report A_τ, the fraction of tasks whose run is correct and achieves at least τ-fold speedup. We use $\tau = 1.25$ and 2, corresponding to runtime savings of at least 20% and 50%, respectively. KernelBench uses a similar joint correctness-and-speed criterion (Ouyang et al., 2025).

Geometric mean speedup. **GM** is the geometric mean of the runs' speedups across tasks; incorrect or unaccelerated outcomes are assigned 1×. Replays of unchanged source on three tasks measure within 2% of 1×, far below the 1.25× target; Appendix E reports this calibration together with sensitivity analyses for the scoring design and baseline choice.

4. Main Results

Experimental setup. We compare 12 coding agents on 32 tasks. Each agent pairs a model with a harness: GPT-6 Astra uses Codex, Gemini uses Gemini CLI, Grok uses Grok Build, and the remaining models use Claude Code. Table 1 reports performance and recorded usage.

Table 1 | Performance and recorded usage on 32 tasks. Scores and rates are percentages; GM is in ×. Effort denotes the requested reasoning setting. Steps and tokens are means per task; tokens include repeated and cached input. Usage is not reported for GPT-6 Astra, Gemini-3.8-Flash, and Grok 4.6 (—). Bold marks performance maxima. The 0910 label abbreviates the recorded DeepSeek deployment identifier (Appendix D.1).

Model	Effort	Score@1	GM	$A_{1.25}$	A_2	Steps	Tokens (M)
GPT-6 Astra	high	**40.3**	2.11	**68.8**	**43.8**	—	—
Hy4-Preview	high	36.5	**2.13**	56.2	37.5	538.0	49.48
DeepSeek-V4.1-Flash^{0910}	high	29.9	1.78	50.0	31.2	676.4	63.35
Gemini-3.8-Flash	high	29.9	1.71	50.0	25.0	—	—
GLM-5.3	max	28.1	1.77	40.6	21.9	967.1	92.89
GLM-5.3-Flash	high	27.3	1.65	43.8	21.9	773.8	74.98
Grok 4.6	high	26.2	1.66	43.8	21.9	—	—
DeepSeek-V4-Pro	high	20.6	1.46	43.8	12.5	471.7	46.17
GLM-5.2	high	19.9	1.46	34.4	15.6	529.8	45.61
DeepSeek-V4-Flash	high	19.1	1.44	34.4	12.5	404.0	38.70
Kimi K3	max	19.0	1.34	34.4	15.6	433.0	37.37
Hy3	high	9.8	1.25	12.5	6.2	254.3	20.92

Correct programs are common; substantial speedups are not. Of the 384 agent–task outcomes, 325 pass correctness checks, but 161 of these (49.5%) fall short of 1.25× speedup (Figure 3).

Each agent is correct on 24 to 30 of the 32 tasks, while Score@1 ranges from 9.8% to 40.3%. DeepSeek-V4-Flash and GLM-5.3-Flash both pass 29 tasks but score 19.1% and 27.3%; GLM-5.3 reaches 28.1% with 24. Runtime savings distinguish otherwise correct outcomes.

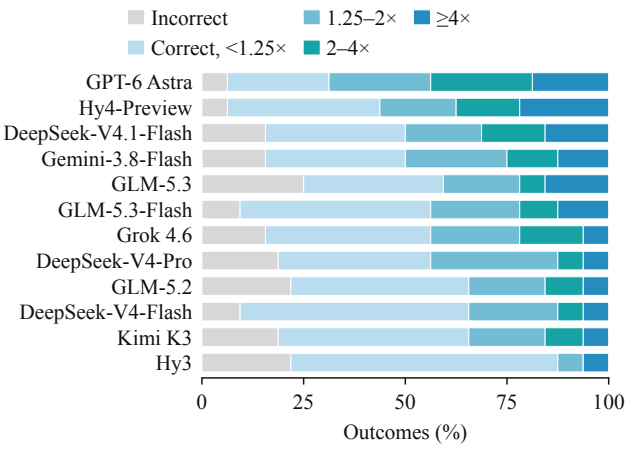


Figure 3 | Correctness and acceleration. Each bar splits an agent's 32 task outcomes into incorrect programs and correct programs binned by speedup over the baseline.

GPT-6 Astra leads, and the lead holds under most scoring rules. GPT-6 Astra reaches the highest Score@1 (40.3%), followed by Hy4-Preview (36.5%); their programs reach at least 2× speedup on 43.8% and 37.5% of tasks. Six of seven alternative score transformations keep GPT-6 Astra first; uncapped log speedup ranks Hy4-Preview first, consistent with its higher GM, and the other changes reorder only agents within 2 percentage points of each other (Appendix E.4). Recorded usage does

not follow the ranking: GLM-5.3 averages 967.1 steps per task and Hy4-Preview 538.0.

5. Analysis

We answer the three questions from Section 1: what changes agents make to an application (Section 5.1), how they develop and select candidate programs (Section 5.2), and how optimization choices interact to affect end-to-end performance (Section 5.3).

Analysis sample. For this analysis, we run five agents, DeepSeek-V4.1-Flash, Grok 4.6, DeepSeek-V4-Pro, DeepSeek-V4-Flash, and Hy3, three times on each task at high reasoning effort, giving 480 runs. A trajectory is the complete record of a run's messages, tool calls, and outputs; a candidate is a program the agent sends to the public tests. GPT-5.6 Luna labels the optimization techniques in each run's final submission (476 runs), and DeepSeek-V4.1-Flash annotates every tool call's action, reason, and cited feedback, together with the relationships between changes (Appendix G).

5.1. What Agents Change

We group fifteen techniques found in final submissions into six categories (Figure 4).

① **Agents mostly change the computation, and rarely the build.** Computation or reuse changes appear in 225 of the 476 labeled runs, across 29 tasks, and parallelism or vectorization in 132 runs across 22 tasks. Compiler and build tuning appears in only 40 runs.

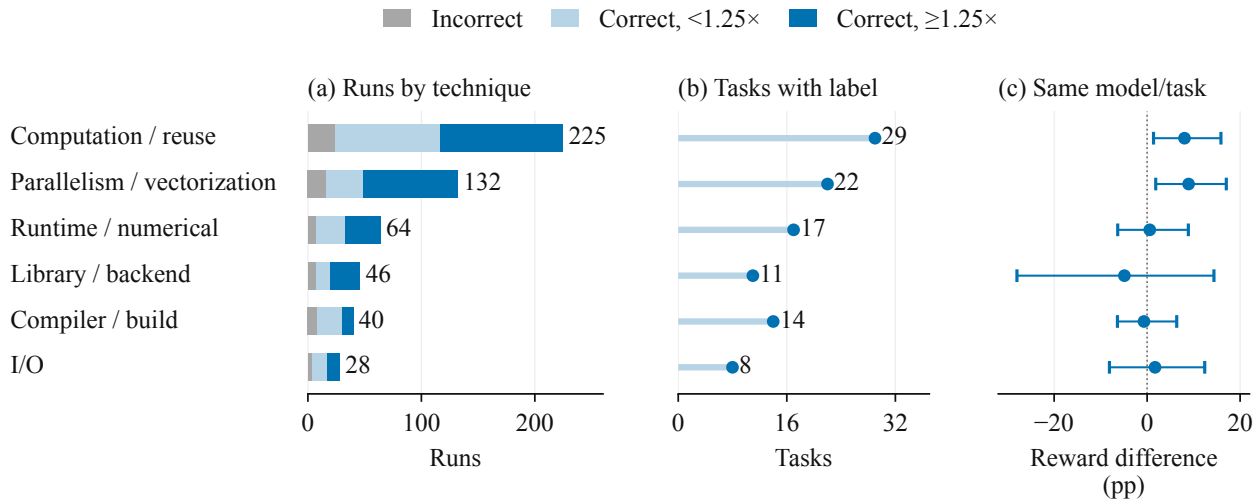


Figure 4 | Optimization techniques and outcomes. (a) Runs using each technique among 476 labeled runs, by final outcome. (b) Tasks in which each technique appears. (c) Mean reward difference between runs with and without the technique, from the same model on the same task; tasks are weighted equally, with 95% task-bootstrap intervals.

② **Runs with parallelism or vectorization most often reach the target speedup.** Of the runs using them, 62.9% are correct and at least 1.25× faster, the highest rate among the six categories, compared with 48.0% for computation or reuse changes. Both keep correctness near 88% (Table 18).

③ **Runs using either leading technique earn about eight more points of reward.** Against runs of the same model on the same task without them, mean reward is 8.0 percentage points higher with computation or reuse changes (27 tasks; 95% interval [1.4, 15.9]) and 8.9 points higher with parallelism or vectorization (19 tasks; [1.9, 17.0]; Appendix H.2).

5.2. How Agents Develop and Select Candidates

① **All agents test more as their runs progress, but more testing does not guarantee success.** Every agent's share of test calls rises from the early to the late third of its trajectory (Figure 5b). Hy3 has the highest late-stage share, 21.7%, yet reaches 1.25× in only 16 of 96 runs, the fewest (Figure 5d). Agents differ more in tool calling: Grok 4.6, in the Grok Build harness, issues several calls in 57.1% of tool-calling turns, versus 1.6–10.0% for the Claude Code agents (Figure 5c).

② **Most runs measure performance, and those that do reach the target more often.** Of the 480 runs, only 61 contain no detected timing or profiling command (P0), 251 contain timing commands without profiling (P1), and 168 contain profiling-related commands (P2). The fractions that are correct and at least 1.25× faster are 11.5%, 37.1%, and 44.6%, respectively. Much of this gap reflects which tasks and models measure: after adjusting for task and model, the P1–P0 reward difference is +4.4 percentage points (95% interval $[-4.0, 12.8]$; Figure 6a–b, Appendix H.4).

③ **The first public test of an edited program usually fails.** The first public test of an edited program passes in only 32 of 155 source-confirmed runs (20.6%), across 26 tasks. Build and execution failures account for 36 tests and program-check failures for 32; 27 more responses are infrastructure or reference failures and 12 name no stage, which do not indicate program defects (Table 2).

Table 2 | First public-test responses for edited programs in 155 runs, by reported stage.

Response	Pass	Package / source contract	Build	Candidate execution	Program checks	Infrastructure / reference	Unspecified stage
Runs	32	16	21	15	32	27	12

④ **Iteration pays off, but not monotonically.** We follow 31 version histories, runs with at least two correct candidates under one timing protocol, across seven applications. Relative to the first correct candidate, the best has a median speedup of 1.26×, and the last is more than 1.05× faster in nineteen histories. Progress is uneven: of 70 consecutive correct-candidate pairs with different source, 36 are faster than their predecessor, but only 26 set a new best (Figure 6c–d; Appendix H.6).

⑤ **Agents sometimes submit a slower program than the best one they found.** In eleven of the 31 histories, the last correct candidate is more than 5% slower than the best (eight by more than 10%), and in ten of these the submitted source matches that slower candidate. Across histories, the median speedup over the first correct candidate falls from 1.26× for the best to 1.18× for the last. Agents do manage candidates explicitly: restore or select operations appear in 206 of 480 trajectories, and of the 583 turns with them, 95.2% state a reason and 93.0% cite earlier feedback (Appendix H.7).

5.3. How Optimization Choices Interact

Controlled replays of submitted programs show that a change's gain depends on the settings and other changes around it, and sampled trajectories show that agents meet such dependencies in most runs.

① **The same source can run slower or twice as fast, depending only on build and runtime settings.** In one GROMACS run, the submission changes only the build and runtime configuration. Replaying this source under four settings, all correct, gives speedups from 0.289× to 2.123×: AVX2 vectorization reaches 2.123× with four threads but 0.709× with one (Table 26).

② **One code change can be worth** 1.01× **or** 6.4×**, depending on another.** In a DeepSeek-V4-Flash QiboTN program, we switch three changes on and off in a full factorial design: broader dispatch to a fast exact simulator (B), reuse of a computed state (R), and deferred loading of tensor libraries

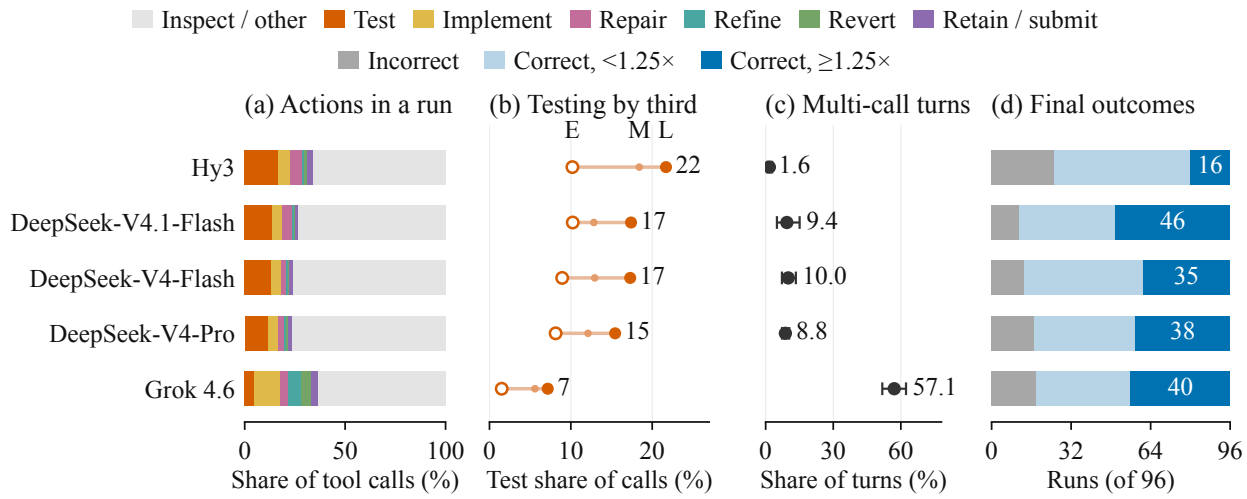


Figure 5 | Every agent tests more late in a run, but Hy3, which tests most, reaches 1.25× least often. Each tool call has one decision label from the automatic annotator (Appendix G). (a) Mean share of a run's calls with each label. (b) Share of calls labeled *test* in the early (E), middle (M), and late (L) third of a run; rows are sorted by L. (c) Share of tool-calling turns with two or more calls, with 95% task-bootstrap intervals. (d) Outcomes of all 96 runs per agent, colored as in Figure 4. Definitions: Appendix H.3; all trajectories: Figure 8.

(L). Together, they make the program 9.6–9.7× faster than with all three off. Deferred loading alone improves runtime by only 1.01×, but by 6.3–6.4× once B is on (three rounds; Appendix J.2).

③ **An optimization helps only if the application executes it.** In an HPL run, DeepSeek-V4-Flash optimizes `HPL_dgemm.c`, then finds by comparing binaries that its executable is unchanged because the application calls `cblas_dgemm`; it moves the optimization to that entry point (Appendix K).

④ **Agents meet such dependencies in most runs.** We use the trajectory annotations to locate candidate evidence and review randomly sampled trajectories, both with and without annotated dependencies: a reviewer confirms a dependency in 34 of 40 flagged and 15 of 40 unflagged trajectories. Weighting the two strata by size, in an estimated 72% of the 480 runs (95% interval 63–80%), tool output shows that a change's effect or correctness depended on build settings, runtime settings, or the code path actually executed; in 52% (41–63%), the dependency concerns an optimization's speed or correctness rather than getting the program to build or run (Appendix I.2). In a Palabos run, the agent's OpenMP program runs for 11 minutes with the default thread count, which sees all 192 host CPUs, and for 1.5 s with four threads. In SIESTA, a build with `-march=native` differs from the reference in 662 of 868 values until fused multiply–add contraction is disabled. In OpenFOAM, rebuilding the solver with `-march=native` leaves runtime unchanged (3.20 versus 3.14 s), because the executed kernels sit in precompiled shared libraries.

6. Related Work

Code optimization benchmarks. Earlier benchmarks measure program efficiency at function scale: PIE studies performance-improving edits to competitive-programming solutions, EffiBench, Mercury and ECCO compare the runtime of generated solutions under correctness tests, EvalPerf and ENAMEL add inputs that expose inefficient algorithms and compare against graded or expert references, and ParEval tests whether models write correct parallel code (Shypula et al., 2024; Huang et al., 2024; Du et al., 2024; Waghjale et al., 2024; Liu et al., 2024; Qiu et al., 2025; Nichols et al., 2024). SWE-Perf supplies target functions and measures test runtimes; GSO compares repository optimizations with

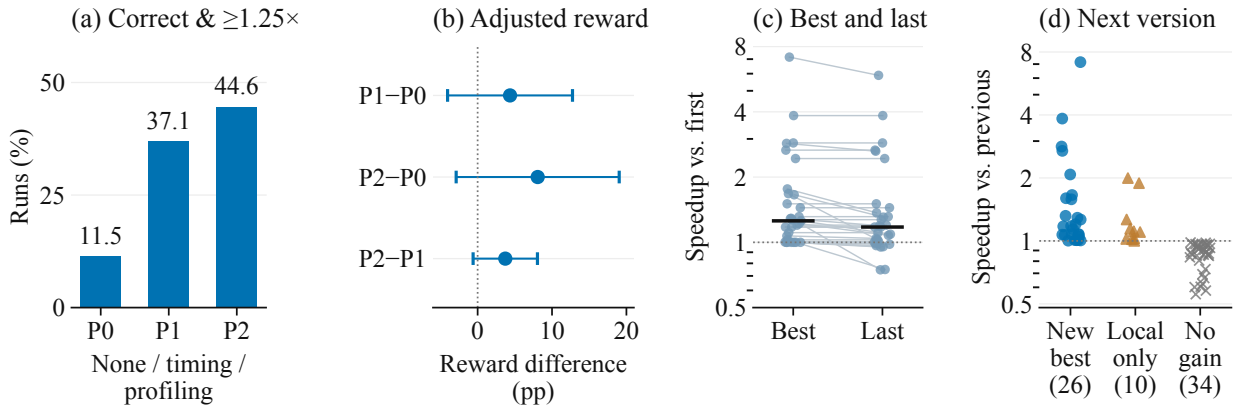


Figure 6 | Timing and profiling, search progress, and candidate selection. (a) Share of runs correct and at least 1.25× faster, for runs with no timing or profiling command (P0), timing only (P1), or profiling (P2). (b) Reward differences adjusted for task and model, with 95% task-cluster intervals. (c) Best and last correct candidates versus the first in 31 histories; lines join one history, black bars mark medians. (d) Speedup over the preceding correct candidate in 70 pairs with different source; "Local only" is faster without setting a new best. Speedup axes are logarithmic.

expert implementations; SWE-fficiency asks agents to locate bottlenecks for a supplied workload (He et al., 2026; Shetty et al., 2025; Ma et al., 2026). A cross-machine replay of their reference patches finds that many tasks do not consistently satisfy the original validity rules (Chen et al., 2026); our timing calibration and scoring sensitivity analyses address the same concern (Appendix E). KernelBench targets efficient GPU operators and model forward passes, and AlgoTune evaluates numerical solvers through a problem–solution interface; both score speedup over a reference implementation of a self-contained function (Ouyang et al., 2025; Press et al., 2025). In production, ECO selects code through fleet-wide profiling and checks model-generated performance edits with tests and post-deployment monitoring (Lin et al., 2026). Accelerate Bench instead fixes one complete application workflow for all agents and times it under application-specific correctness constraints; as in SWE-fficiency, agents must first find where runtime is spent, and they may change algorithms, implementations, and build and runtime settings.

Long-horizon agent evaluation. RE-Bench is closest in format: agents improve working starting solutions within budgets of up to eight hours, and several of its ML research-engineering tasks score runtime reductions against expert references (Wijk et al., 2025). It normalizes scores between the starting and expert solutions; we report runtime ratios to the provided baseline and verify final submissions on private inputs. AlphaEvolve improves programs through model-proposed edits scored by automated evaluators (Novikov et al., 2025). ALE-Bench evaluates iterative algorithm engineering through public testing and final private evaluation (Imajuku et al., 2025). AutoLab studies sustained improvement of working implementations, and MLS-Bench tests whether improvements to ML components generalize across settings and scales (Xu et al., 2026a; Lyu et al., 2026). Accelerate Bench applies this iterative loop to application-level optimization, combining candidate programs, comparable timings, and evaluation feedback to analyze how agents achieve new gains, recover earlier performance, and retain measured improvements.

Scientific coding benchmarks. SciCode evaluates scientific reasoning and code synthesis, while ScienceAgentBench asks agents to implement data-driven tasks from research publications (Tian et al., 2024; Chen et al., 2025). AInsteinBench evaluates development in scientific repositories; SWE-bench Science covers scientific software repair and integration (Duston et al., 2025; Xu et al.,

2026b). Accelerate Bench starts from working applications and asks agents to accelerate the prescribed computations, subject to constraints on numerical outputs and workflow behavior.

7. Limitations

Task coverage and baselines. The evaluation covers CPU workflows at the specified scales and resource budgets. Speedups are relative to task-provided baselines with varying degrees of tuning; they measure improvement from those starting points. Results at other scales, on other hardware, and in production environments require separate evaluation.

Agent and execution settings. As in other agent benchmarks, each result reflects a model running in its harness, so score differences combine model capability with harness behavior, inference compute, and execution conditions. Every agent receives the same task container, resource and time limits, and timing and correctness rules, and scores are runtime ratios to the task baseline. Separating models from harnesses and measuring sensitivity to hardware placement are natural extensions that the retained programs and containers support.

Trajectory samples. The trajectory analyses cover five agents. Source labels combine model-assisted annotation with targeted source checks, and prevalence estimates rest on reviewed samples. Version histories cover 31 runs in seven applications, most candidates have one timing observation, and controlled experiments establish effects for the tested artifacts, workloads, and settings.

8. Conclusion

Accelerate Bench evaluates application-level optimization on 32 workload-defined CPU tasks with public feedback and private verification. The best agent reaches a Score@1 of 40.3%, and 161 of 325 correct outcomes stay below 1.25× speedup. Computation or reuse changes appear in 225 of 476 labeled runs, compiler and build tuning in only 40, yet controlled replays show that gains depend on build and runtime settings and on other code changes: identical source runs from 0.29× to 2.12×, and agents encounter such dependencies in an estimated 72% of analyzed runs. The first public test of an edited program passes in only 32 of 155 source-confirmed runs, and in 10 of 31 version histories the submitted program is more than 5% slower than the best candidate.

Authors and Affiliations

Bowen Liu[1], Yuhan Li[1,‡], Nuo Chen[1,†], Weile Luo[1], Miao Peng[1,3], Qintong Zhang[1,5], Qifan Zhang[1,3], Jia Li[2,3], Ziqi Gao[4], Lydell Huang[1]

[1]Tencent Hy Team
[2]The Hong Kong University of Science and Technology
[3]The Hong Kong University of Science and Technology (Guangzhou)
[4]Tsinghua University
[5]Peking University

[‡]Project Lead [†]Corresponding author
chennuo26@gmail.com

Contact: Bowen Liu
bliu690@connect.hkust-gz.edu.cn

Reproducibility Statement

Evaluation metrics, run-inclusion and aggregation rules, task resources, and model configurations are described in Section 3 and Appendices B–E; Appendix A lists the data snapshot behind each result. Each score is stored with its task and verifier versions, the scored program, and the execution settings; process analyses use retained trajectories and evaluator responses. Reproduction requires the same workload definitions, task and verifier versions, execution resources, and agent settings. We will release the versioned task packages, including public tests and verifiers, with the published version of the paper.

AI Use Statement

The 12 coding agents evaluated in this paper are the objects of study. Beyond them, we used large language models in two roles.

As research instruments. GPT-5.6 Luna labels the optimization techniques in each run's final implementation, and DeepSeek-V4.1-Flash annotates every tool call in the 480 analyzed trajectories (Appendix G). These labels support the analyses in Section 5; prevalence estimates come from a separate review of random samples rather than from label counts (Appendix I.2).

As research and writing assistants. AI assistants supported literature search, discussion of benchmark positioning and analysis design, drafting and editing of the manuscript, and code for data processing, figure generation, and evaluation infrastructure.

The authors reviewed all AI-assisted outputs, checked each citation against its source, and verified the reported numbers against the frozen evaluation outputs. The authors take full responsibility for the paper's content.

References

Anastasios N Angelopoulos, Stephen Bates, Clara Fannjiang, Michael I Jordan, and Tijana Zrnic. Prediction-powered inference. *Science*, 382(6671):669–674, November 2023.

Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in High-Performance Computing Applications. *Proceedings of the IEEE*, 106(11):2068–2083, November 2018. ISSN 1558-2256. doi: 10.1109/jproc.2018.2841200. URL http://dx.doi.org/10.1109/JPROC.2018.2841200.

Zhi Chen, Zhensu Sun, Yuling Shi, David Lo, and Lingxiao Jiang. Are Performance-Optimization Benchmarks Reliably Measuring Coding Agents?, 2026. URL https://arxiv.org/abs/2607.01211.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. ScienceAgentBench: Toward Rigorous Assessment of Language Agents for Data-Driven Scientific Discovery. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 96934–96990, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f12b4df26344f3be803c06b555252efe-Paper-Conference.pdf.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A Code Efficiency Benchmark for Code Large Language Models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 16601–16622. Curran Associates, Inc., 2024. doi: 10.52202/079017-0529. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/1df1df43b58845650b8dada00fca9772-Paper-Datasets_and_Benchmarks_Track.pdf.

Titouan Duston, Shuo Xin, Yang Sun, Daoguang Zan, Aoyan Li, Shulin Xin, Kai Shen, Yixiao Chen, Qiming Sun, Ge Zhang, Jiashuo Liu, Huan Zhou, Jingkai Liu, Zhichen Pu, Yuanheng Wang, Bo-Xuan Ge, Xin Tong, Fei Ye, Zhi-Chao Zhao, Wen-Biao Han, Zhoujian Cao, Yueran Zhao, Weiluo Ren, Qingshen Long, Yuxiao Liu, Anni Huang, Yidi Du, Yuanyuan Rong, and Jiahao Peng. AInsteinBench: Benchmarking Coding Agents on Scientific Repositories, 2025. URL https://arxiv.org/abs/2512.21373.

Naoki Egami, Musashi Hinck, Brandon Stewart, and Hanying Wei. Using Imperfect Surrogates for Downstream Inference: Design-based Supervised Learning for Social Science Applications of Large Language Models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 68589–68601. Curran Associates, Inc., 2023. doi: 10.52202/075280-3000. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d862f7f5445255090de13b825b880d59-Paper-Conference.pdf.

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? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=pNxw4i5Dp2.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie Zhang. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 11506–11544. Curran Associates, Inc., 2024. doi: 10.52202/079017-0367.

URL https://proceedings.neurips.cc/paper_files/paper/2024/file/15807b6e09d691fe5e96cdecde6d7b80-Paper-Datasets_and_Benchmarks_Track.pdf.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A Benchmark for Long-Horizon Objective-Driven Algorithm Engineering. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-5226. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/e555eab2c68f886052f404397ea52001-Paper-Datasets_and_Benchmarks_Track.pdf.

Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495), 2020. ISSN 0036-8075. doi: 10.1126/science.aam9744.

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

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating Language Models for Efficient Code Generation. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=IBCBMeAhmC.

Bohan Lyu, Yucheng Yang, Siqiao Huang, Jiaru Zhang, Qixin Xu, Xinghan Li, Xinyang Han, Yicheng Zhang, Huaqing Zhang, Runhan Huang, Kaicheng Yang, Zitao Chen, Wentao Guo, Junlin Yang, Xinyue Ai, Wenhao Chai, Yadi Cao, Ziran Yang, Kun Wang, Dapeng Jiang, Huan ang Gao, Shange Tang, Chengshuai Shi, Simon S. Du, Max Simchowitz, Jiantao Jiao, Dawn Song, and Chi Jin. MLS-Bench: A Holistic and Rigorous Assessment of AI Systems on Building Better AI, 2026. URL https://arxiv.org/abs/2605.08678.

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? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0pyFbZSfbT.

Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can Large Language Models Write Parallel Code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, pp. 281–294. ACM, June 2024. doi: 10.1145/3625549.3658689. URL http://dx.doi.org/10.1145/3625549.3658689.

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, 2025. URL https://arxiv.org/abs/2506.13131.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs Write Efficient GPU Kernels? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 13–19 Jul 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? In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-3585. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/9a648e8e1014c2427156dcb5465cd488-Paper-Datasets_and_Benchmarks_Track.pdf.

Ruizhong Qiu, Weiliang Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is LLM-generated code? A rigorous & high-standard benchmark. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 2233–2261, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/06694da057cb15fef11542270a592627-Paper-Conference.pdf.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging Software Optimization Tasks for Evaluating SWE-Agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-5054. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/ddd983890041619 43e646322a16348c3-Paper-Datasets_and_Benchmarks_Track.pdf.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 47277–47301, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/ce65173b994cf7c925c71b482ee14a8d-Paper-Conference.pdf.

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. SciCode: A Research Coding Benchmark Curated by Scientists. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 30624–30650. Curran Associates, Inc., 2024. doi: 10.52202/079017-0963. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/36850592258c8c41cecdaa3dea5ff7de-Paper-Datasets_and_Benchmarks_Track.pdf.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Yaser Al-

Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15362–15376, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.859. URL https://aclanthology.org/2024.emnlp-main.859/.

Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua M Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-Bench: Evaluating Frontier AI R&D Capabilities of Language Model Agents against Human Experts. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 66772–66832. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/wijk25a.html.

Murray Woodside, Greg Franks, and Dorina C. Petriu. The Future of Software Performance Engineering. In *Future of Software Engineering (FOSE '07)*, pp. 171–187. IEEE, May 2007. doi: 10.1109/fose.2007.32. URL http://dx.doi.org/10.1109/FOSE.2007.32.

Zhangchen Xu, Junda Chen, Yue Huang, Dongfu Jiang, Jiefeng Chen, Hang Hua, Zijian Wu, Zheyuan Liu, Zexue He, Lichi Li, Shizhe Diao, Jiaxin Pei, Jinsung Yoon, Hao Zhang, Mengdi Wang, Radha Poovendran, Misha Sra, Alex Pentland, and Zichen Chen. AutoLab: Can Frontier Models Solve Long-Horizon Auto Research and Engineering Tasks?, 2026a. URL https://arxiv.org/abs/2606.05080.

Zhipeng Xu, Jiahao Lu, Yining Zheng, Yuxin Wang, and Xipeng Qiu. SWE-bench Science: Can Coding Agents Resolve Engineering Tasks in Science?, 2026b. URL https://arxiv.org/abs/2608.19799.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can Language Models Rebuild Programs From Scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

A. Appendix Overview and Evidence Sources

The appendices follow the order of the main text (Table 3). Each numerical result is computed from one of the frozen snapshots listed in Table 4.

Table 3 | Where the appendices support the main text.

Main text	Appendix
Benchmark construction (§2)	Tasks, sources, resources and contracts (B); verifier validation controls (C)
Evaluation metrics (§3)	Agents, eligibility, usage (D); timing calibration, score sensitivity (E)
Main results (§4)	Per-task rewards of the reported runs (F)
Analysis sample and labels (§5)	Annotation procedures and their audits (G)
Changes, candidates (§5.1–§5.2)	Technique and workflow analysis (H)
Interacting choices (§5.3)	Dependency review (I); controlled experiments (J); case records (K)
All trajectories	Action sequences and outcomes of the 480 runs (L)

Table 4 | Evidence sources. Each row is a frozen snapshot; the earlier snapshot supports only the timing and aggregation checks named here.

Snapshot	Used for
Main evaluation: 12 agents, 384 runs	Table 1, Figure 3, Figure 2, Table 16; scoring sensitivity (E.4)
Analysis: 480 runs of five agents, three per task, with trajectories and public-test records	Section 5; Appendices G–I, K and L
Earlier 842-run snapshot, per-case timings	Baseline perturbation (E.1); workload aggregation check (E.3)
Controlled replays of fixed programs	Appendices C, E and J; conditions are stated per experiment

Terms. A *run* (also called an attempt) is one agent's optimization session on one task. Its *trajectory* records the agent's messages, tool calls and outputs. A *candidate* is a program the agent sends to the public tests during the run; the *final submission* is the frozen program the verifier rebuilds and scores.

Conventions for interpreting the analyses. Comparisons between runs are observational. Where stated, they hold the model and task fixed. Other changes still co-occur within a run, so these comparisons measure associations rather than the effect of a single technique or behavior. Unless stated otherwise, intervals are pointwise 95% percentile intervals obtained by resampling the 32 tasks, keeping all runs of a task together. They are conditional on the observed runs and are not adjusted for multiple comparisons. Controlled replays measure the named programs under the stated workloads, resources and settings, and their timings are reported separately from the public-test feedback that the agent saw.

B. Tasks, Resources, and Contracts

Task definition. A task t specifies an initial program a_t^0, a permitted edit space $\mathcal{A}_t$, a public test P_t, a verifier V_t, and an interaction budget b_t. The task instruction defines the required computation, outputs and editable paths. During a run, the agent produces candidates $a_t^1, a_t^2, \ldots \in \mathcal{A}_t$, guided by code inspection, its own executions and the public responses $P_t(a_t^k)$, and leaves a final submission $\hat{a}_t$. The verifier scores $\hat{a}_t$ on private inputs; the agent does not see this reward during the run. The score therefore evaluates the submitted program under the task contract, independent of what the agent claims about it.

Resources. Each task package fixes its CPU cores, memory, agent interaction limit and verifier time limit. Table 5 lists these allocations and the task-package version, with memory in GiB, interaction limits in hours and verifier limits in minutes.

Table 5 | Resource allocations and task-package versions.

Task	Version	CPUs	GiB	Agent h	Verify min
GROMACS	12.0.3	4	6	8	390
LAMMPS	4.1.3	4	8	8	30
GPAW	1.0.2	4	6	8	233.3
Quantum ESPRESSO	1.2.2	4	6	8	233.3
SIESTA	1.2.2	4	6	8	233.3
Fluidity	1.0.2	4	8	8	233.3
ICON	1.0.2	4	6	8	200

Task	Version	CPUs	GiB	Agent h	Verify min
MPAS-A	1.0.2	4	6	8	233.3
NEMO	1.0.2	4	8	8	233.3
OpenFOAM	1.0.3	4	8	8	40
Palabos	2.0.2	4	6	8	120
SU2	2.0.2	4	8	8	60
OpenMC	3.0.4	4	6	8	120
ShengBTE	1.0.2	4	6	8	233.3
VPIC	1.2.2	4	6	8	150
PRESTO	2.0.2	4	6	8	120
ABySS	1.0.3	4	8	8	30
MHM2	1.0.2	4	6	8	150
MrBayes	2.0.2	4	6	8	120
wtdbg2	2.0.3	4	8	8	30
HPCG	12.0.3	4	6	8	120
HPL	11.0.2	4	6	8	120
Graph sparsification	3.0.2	4	4	8	120
miniVite	11.0.2	4	6	8	120
NUFFT Imaging	1.0.2	4	6	8	200
Tucker reconstruction	3.1.6	8	16	8	120
QuEST	3.0.5	4	8	8	60
QiboTN	3.2.7	8	32	8	833.3
Paged LM generation	1.0.2	4	6	8	200
ThunderSVM	4.3.4	4	8	8	60
DataLife	1.0.2	4	6	8	233.3
IO500	12.0.2	4	6	8	120

Table 6 | Task collection by application area. All 32 tasks carry equal weight in every aggregate.

Application area	*N*	Tasks
Molecular simulation / electronic structure	5	GROMACS, LAMMPS, GPAW, Quantum ESPRESSO, SIESTA
Fluids / atmosphere / ocean	7	Fluidity, ICON, MPAS-A, NEMO, OpenFOAM, Palabos, SU2
Transport / plasma / astronomy	4	OpenMC, ShengBTE, VPIC, PRESTO
Bioinformatics / phylogenetics	4	ABySS, MHM2, MrBayes, wtdbg2
Numerics / tensors / graphs	6	HPCG-derived, HPL-derived, NUFFT imaging, Tucker reconstruction, Graph sparsification, miniVite
Quantum circuits	2	QuEST, QiboTN
Machine learning	2	Paged LM generation, ThunderSVM
Scientific I/O	2	DataLife, IO500-derived
Total	32	

Computational contracts. Each contract names what the agent may edit, what is timed and what the program must produce. Table 7 gives three representative contracts, and every task package states these three elements for its own workflow.

Table 7 | Example computational contracts. The HPCG-derived task is an application optimization problem built on the HPCG source, not the official HPCG competition benchmark.

Task	Permitted edits	Timed execution	Required outputs and checks
GROMACS	Full source tree and supported build settings	Complete `mdrun`; excludes build and input preparation	Prescribed physical calculation, trajectories, energies, final state, deterministic replay
HPCG-derived	Source tree, solver algorithm, entry point, build settings	Input, matrix construction, setup, two solves, output	Two full solutions; absolute error $\leq 10^{-8}$, relative residual $\leq 10^{-9}$; identical repeated vectors
Paged generation	Application modules and helpers	Model loading, prefill, decoding, checkpoint, fresh-process resume	Full-model states, KV caches, sampling probabilities and tokens; numerical tolerances and exact recovery outputs

B.1. Task sources and reconstruction

Competition provenance. Table 8 assigns each task to the competition family recorded in its construction history, which identifies the application or optimization theme that motivated its reconstruction. An application that also appears in other competitions is counted once, under its recorded source.

Table 8 | Recorded competition sources for all 32 tasks. Names link to organizer or event records.

Competition	*N*	Tasks
Asia Supercomputer Community (ASC) Student Supercomputer Challenge	14	ABySS, QiboTN, Fluidity, HPCG, HPL, NEMO, OpenFOAM, Palabos, PRESTO, QuEST, ShengBTE, Siesta, SU2, wtdbg2
Student Cluster Competition at the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)	10	DataLife, GROMACS, ICON, IO500, LAMMPS, MPAS-A, MrBayes, OpenMC, Quantum ESPRESSO, VPIC
Parallel Application Challenge	4	NUFFT, Paged LM generation, Tensor, ThunderSVM
Student Cluster Competition at the International Supercomputing Conference (ISC)	2	GPAW, MHM2
IndySCC, the remote Student Cluster Competition at SC	1	miniVite
ACM-China International Parallel Computing Challenge	1	Graph sparsification

From source material to a runnable task. Competition descriptions provided application names and optimization themes; upstream repositories and examples supplied reusable code and scientific inputs. We adapted these materials into CPU workloads with explicit build and execution entry points, collected outputs, and permitted edit spaces. Some tasks extend earlier component tests into broader application workflows. For example, HPL now performs an MPI dense solve and exports the full solution, and wtdbg2 performs graph assembly and consensus. Where the package specifies it, the earlier component tests remain as regression checks. Where original competition inputs were missing, we used upstream examples, public datasets, or generated inputs meeting the requirements.

Source scope and workload scope. A complete source tree does not mean the workload exercises every application feature. For example, the full miniVite program runs the first distributed Louvain

phase, GPAW performs two geometry steps, and ShengBTE covers bulk transport. Other tasks expose selected source components. OpenFOAM permits changes to its `pisoFoam` solver and case over fixed system libraries, SU2 submissions contain specified solver subtrees and a case, and DataLife permits changes to monitoring and analysis around a fixed I/O driver. These boundaries determine which implementation choices an agent can change.

Some workloads were built from an optimization theme rather than reconstructed from the original competition program. Paged generation runs SmolLM2 with task-authored NumPy code, the tensor task combines TAPP with a Tucker compression and reconstruction driver, and the graph task independently implements a feGRASS-inspired sparsification and solver pipeline. Their contracts specify these computations, which differ from the original PAC or IPCC payloads. Table 9 records the resulting source and workflow choices. The versioned task packages define the evaluated collection, which adapts the competition workloads rather than reproducing them.

Source identities come from the construction records. Ordinary-file comparisons against upstream independently checked the QuEST, QiboTN, TAPP, and Palabos sources; QiboTN includes a CPU backend adaptation. The table marks unresolved commits and archive mappings explicitly.

Table 9 | Task origins and reconstructed workflows. Repository names link to GitHub unless another host is named; † marks a source mirror. Releases are upstream versions, distinct from task-package versions; commits are abbreviated.

Task / source repository	Release / revision	Supplied code and retained computation
ABySS BirolLab/abyss	1.9.0 07629837046e	Full application source; generated short reads drive assembly, graph/coverage outputs and contig statistics.
QiboTN qiboteam/qibotn	QiboTN 0.0.6 283bab987e89	Full QiboTN tree with a disclosed CPU backend patch; task-authored circuits produce complete states and observables.
DataLife pnnl/datalife	Release name unknown ee9d3939a323	FlowMonitor/FlowAnalysis/tools source; monitoring and DAG analysis of a fixed I/O driver. Separate HDF5/DaYu projects excluded.
Fluidity FluidityProject/fluidity	4.1.16 4fc02e3da1cb	Full source; selected MPI redistribution and DG heat workflows with field outputs and checkpoint recovery.
GPAW GitLab: gpaw/gpaw	21.1.0 sdist c5b9c0c91bf0 Historical tag; sdist mapping unverified	Full release sdist and upstream PAW setups; SCF, two BFGS steps and restart. These steps do not establish geometry convergence.
GROMACS gromacs/gromacs†	2020.4 Commit unresolved	Full source with build/numerical preparation patches; prescribed water/argon MD trajectories and energies.
HPCG-derived hpcg-benchmark/hpcg	3.1 e3fe052b572b	Full source plus an authored entry point; a prescribed matrix with two right-hand sides may use replacement solvers. Official HPCG compliance is not claimed.
HPL-derived icl-utk-edu/hpl	2.3 Commit unresolved	Full source plus matrix-input and solution export; bounded MPI dense solves, not a TOP500 run.

Continued on the next page.

Table 9 continued.

Task / source repository	Release / revision	Supplied code and retained computation
ICON spcl/icon-dace†	2024.10 public a1324166fa60	Full recorded source mirror; selected Hadley/JABW atmosphere cases on an upstream global grid, with conservation and restart outputs.
IO500-derived IO500/io500	IO500 sc20 65b0397101a8	Full IO500/IOR/mdtest/pfind sources; all 12 phases, shortened below the official duration rule.
feGRASS-derived feGRASS paper	Independent implementation No author-code import	Independent six-file implementation; generated graph sparsification, preconditioning and multi-RHS PCG. Original authors' source and IPCC payload are not used.
LAMMPS lammps/lammps	patch_23Oct2017 39df9f5d9467	Full serial application source; generated MD inputs, trajectories and restart continuation. Partial reconstruction of the competition workload.
MHM2 Bitbucket: berkeleylab/mhm2	2.0.0 release Commit unresolved	Full release and recorded dependencies; generated two-genome, two-round coassembly with graphs, alignments and abundance outputs; UPC++ smp.
miniVite ECP-ExaGraph/miniVite	Release name unknown 4cbf5beb58c6	Full miniVite source with partition-output additions; distributed Louvain first phase, not the multi-phase Vite application.
MPAS atmosphere MPAS-Dev/MPAS-Model	8.4.1 91c5eac175ee	Full application and pinned physics dependencies; upstream 240-km grid, bounded forecast and restart continuation.
MrBayes NBISweden/MrBayes	3.2.7a 8819c865c312	Full source; upstream and generated alignments, two four-chain runs, midpoint recovery and posterior summaries.
NEMO NEMO forge: nemo/nemo	5.0.2 dcc7fb8c1779	Full source; LOCK_EXCHANGE ocean run with full fields and checkpoint continuation.
OpenFOAM OpenFOAM release	Debian 1912.200626 Commit unresolved	Editable pisoFoam solver and cavity case, plus legacy LDU code; fixed Debian OpenFOAM libraries. Transient full-field CFD.
OpenMC openmc-dev/openmc	0.16.0 617d35a5063c	Full source; upstream C5G7 cross-section constants in task-authored fuel-pin geometry, with full tallies and source-bank recovery.
NUFFT imaging flatironinstitute/finufft	FINUFFT 2.4.1 e7144a5c08cb	Full FINUFFT source plus authored binning and inverse-imaging drivers; complete reconstruction and CG history.
Paged LM generation HF: HuggingFaceTB/SmolLM2-135M	SmolLM2-135M 93efa2f097d5 Model revision; driver commit unknown	Full SmolLM2 weights with authored NumPy code; prefill, sampling and resumed generation. No equivalence to the PAC DeepSeek indexer is shown.
Tucker reconstruction TAPPorg/reference-implementation	TAPP 0.5.0 77c32d744ee6	Full TAPP source plus authored HOSVD/Tucker drivers and bridge; all reconstruction stages. Not the original large PAC contraction workload.
ThunderSVM Xtra-Computing/thundersvm	0.3.4 e71013a88fb5	Full CPU application build/source surface; public a9a-derived inputs, C-SVC training, saved-model reload and prediction.

Continued on the next page.

Table 9 continued.

Task / source repository	Release / revision	Supplied code and retained computation
Palabos omalaspinas/palabos†	2.3.0 bba6dc6464fa	Full source plus authored driver/CMake; upstream cavity-derived and periodic shear simulations with complete field outputs.
PRESTO scottransom/presto	3.0.1 7d569eff79ec	Full source with disclosed planning/build adaptations; generated pulse time series drive FFT, acceleration search and folding.
Quantum ESPRESSO QEF/q-e	6.8 ReleasePack 03a7fa640903	Full release and pinned devxlib; selected SCF/NSCF, density, DOS and checkpoint workflows with upstream pseudopotentials.
QuEST QuEST-Kit/QuEST	4.2.0 9d7618d7263e	Full upstream source plus task runners; state-vector and density-matrix circuits with complete states and probabilities.
ShengBTE Bitbucket: sousaw/shengbte	1.5.1 e9f6fdc3362f	Full source and upstream force constants; bulk RTA and iterative BTE at prescribed settings. Nanowire workflows are outside the task.
SIESTA GitLab: siesta-project/siesta	4.1.5 c3221beccffa	Full source and upstream tests/pseudopotentials; LDA calculations and density-matrix restart. The older matrix proxy is only retained regression coverage.
SU2 su2code/SU2	8.5.0 12eb826f049e	Only Common/SU2_CFD and the case are submitted over a fixed full build tree; serial NACA0012 Euler flow and five complete snapshots.
VPIC lanl/vpic	1.2 f29f1545d339	Full source; bounded upstream Harris reconnection case with complete diagnostic dumps and checkpoint recovery.
wtdbg2 ruanjue/wtdbg2	2.5 b77c5657c809	Full assembler and consensus source; generated noisy PacBio/ONT-style reads, graph/layout outputs and consensus. Original competition reads are not reproduced.

C. Verifier Validation Controls

The verifier accepts every behavior-preserving control and rejects every invalid shortcut we constructed. The fourteen planned controls on HPCG-derived and paged generation comprise four unchanged-source runs, two legal transformations and eight invalid variants. All unchanged-source runs and both legal transformations pass, and each invalid variant fails the check for the obligation it violates. The controls exercise the named checks and do not estimate a benchmark-wide false-acceptance rate.

HPCG-derived: solution obligations. The workload solves a sparse system for two right-hand sides. A variant that omits the second solution fails the output-size check, and one that emits NaN fails the finite-value check. Stopping after one iteration or reusing the wrong solution fails the solution checks. The unchanged source and a legal I/O change that disables stream synchronization both pass (Figure 7).

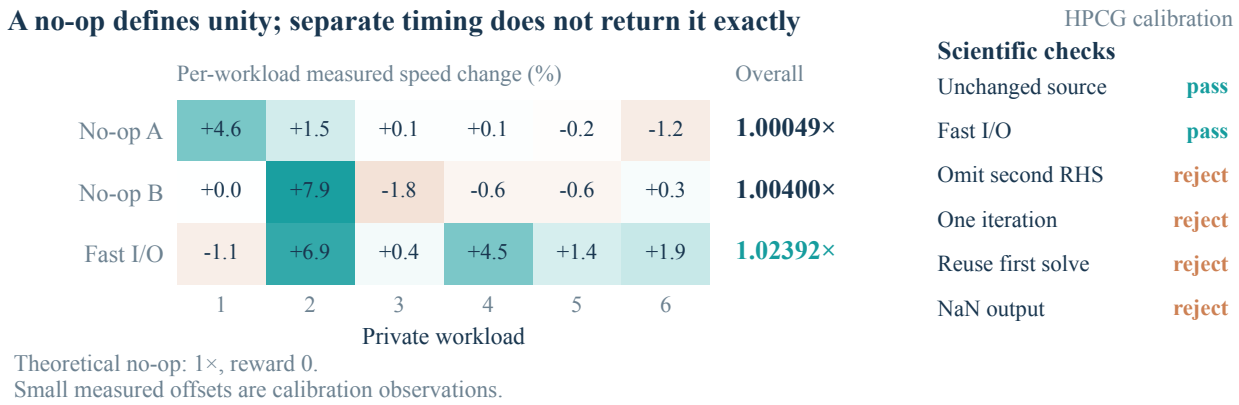


Figure 7 | Unchanged-source controls and verifier challenges on HPCG-derived. Each row compares six private solver workloads under the original three-repeat timing rule. The unchanged-source rows scatter around the theoretical 1× reference. Disabling standard-stream synchronization passes at 1.024× overall. The verifier rejects all four changes that violate the solution obligations.

Paged generation: application checks catch what component checks miss. The workload covers full-model computation, checkpoint creation and continuation in a fresh process. Four invalid variants pass the legacy attention-component checks but fail the application checks (Table 10). Lossless checkpoint compression passes both levels.

Table 10 | In paged generation, application checks reject four invalid variants that component checks accept. All variants run through the original full-workflow verifier. ✓ denotes acceptance, ✗ rejection, and — no speedup for a rejected variant.

Controlled change	Component checks	Application checks	Speedup
Omit a generation step	✓	✗	—
Advance checkpoint cursor incorrectly	✓	✗	—
Change saved cache metadata	✓	✗	—
Emit nonfinite probabilities	✓	✗	—
Lossless checkpoint compression	✓	✓	0.894×

Passing is separate from acceleration. Both legal transformations pass, yet checkpoint compression measures 0.894× and disabling stream synchronization 1.024×. This I/O change therefore does not explain the much larger HPCG-derived speedups in the main results.

Measurement protocol. All validation controls run on a reserved allocation of four physical cores with the task's memory limit and private workload definitions. Baseline and candidate executions alternate within each workload. The controls add no cache eviction or warmup beyond what the task's evaluator performs. Background services stay active, and we record their observed load.

D. Evaluation Protocol

Agents and execution. Table 1 compares twelve agents, each pairing a model with its harness. GPT-6 Astra runs in Codex CLI and Gemini-3.8-Flash in Gemini CLI, both through a model API gateway. Grok 4.6 runs in `grok-build`, and the other nine agents run in `claude-code`. Kimi K3

and GLM-5.3 request `max` reasoning effort and the other agents `high`. This provider-specific setting does not equalize inference compute across providers. An open-source agent-evaluation framework orchestrates the runs, which execute in Docker. Tasks allocate four or eight CPU cores and 4–32 GiB of memory, and each run has an eight-hour limit. Agent runs and verifier evaluations use dual-socket Intel Xeon Platinum 8476C hosts with 48 cores per socket, SMT enabled, two NUMA nodes, AVX-512 and AMX support, and about 2 TB of memory. During a run, the agent inspects the permitted files, builds candidates and queries the public test. The verifier then rebuilds and scores the final submission. The reward times only the application, measured separately from interaction time.

Run eligibility. Runs interrupted by external API failures or infrastructure cancellations are excluded. Agent-caused out-of-memory terminations, including exit code 137, remain eligible and are scored by the verifier like any other final submission.

Reported run. For each agent–task pair with three attempts, we select the median-reward run among the latest three policy-eligible scored attempts, sorting ties by original completion time and trial ID. GPT-6 Astra, Gemini-3.8-Flash, Kimi K3 (max), and GLM-5.3 (max) have one attempt per task, which is used directly. Table 1 therefore contains 384 selected runs; all performance and usage metrics for a pair come from the same selected run.

Metrics. For the reported run $a^*_{m,t}$ of agent m on task t, with correctness C and speedup $s = B_t/T_t$,

$$\mathrm{Acc}_m = \tfrac{1}{32}\sum_t C, \qquad A_{\tau,m} = \tfrac{1}{32}\sum_t C\,\mathbf{1}[s \ge \tau], \qquad \mathrm{GM}_m = \exp\!\big(\tfrac{1}{32}\sum_t \log s^*\big),$$

where $s^* = \max(1, s)$ for correct submissions and $s^* = 1$ otherwise. The value $s^* = 1$ is an aggregation convention and does not correspond to an observed fallback execution. It keeps unsuccessful runs in the summary without crediting them with a speedup. None of these quantities is conditioned on correctness.

Usage counters. A step is one model reply in the archived trajectory. A reply with several tool calls counts once; user messages, system messages and tool returns are excluded. Tokens are the input plus output counts reported by the harness. Input counts include context repeated across calls and cached tokens. Table 1 averages these counters over the 32 reported runs of each agent and prints a dash when a harness does not report complete counters. Because provider tokenizers and reasoning-token conventions differ, the counters describe recorded use rather than a normalized compute measure or a verified bill.

D.1. Model Identifiers

Table 11 records the API identifiers of the twelve agents in Table 1. We remove provider routing prefixes such as `xai/` or `zhipu/`, but show the Gemini identifier as the gateway records it. Some raw Gemini CLI session records retain the client's default alias `gemini-3.5-flash`; the batch gateway binds this alias to the `gemini-3.8-flash` backend and records both identities.

Table 11 | Recorded model identifiers, in the order of Table 1.

Display label	Recorded model identifier
GPT-6 Astra	`gpt-6-astra`
Hy4-Preview	`hy4-preview`
DeepSeek-V4.1-Flash (0910)	`deepseek-v4.1-flash-expires-on-0910`
Gemini-3.8-Flash	`google/gemini-3.8-flash`
GLM-5.3	`glm-5.3`
GLM-5.3-Flash	`glm-5.3-flash`
Grok 4.6	`grok-4.6`
DeepSeek-V4-Pro	`deepseek-v4-pro`
GLM-5.2	`glm-5.2`
DeepSeek-V4-Flash	`deepseek-v4-flash`
Kimi K3	`kimi-k3`
Hy3	`hy3`

E. Timing Calibration and Score Sensitivity

For a correct final submission, speedup is $s = B/T$, where the baseline time B and the submission time T use the same task-specific timing scope. Its reward is $r = \max(0, 1 - 1/s)$, and an incorrect submission receives zero (Equation 1). Each task defines how repeated workload executions are aggregated. The mean reward measures fractional runtime savings over equally weighted tasks, relative to the supplied starting program rather than to an expert implementation. This appendix checks how measurement variation, workload aggregation and alternative score definitions affect these numbers.

E.1. Timing Calibration

Unchanged programs measure within 2% of 1×. An unchanged program has theoretical speedup 1× and reward zero, but two separate executions can measure on either side of 1×. The HPCG-derived unchanged-source controls measure 1.00049× and 1.00400×, the paged-generation controls 1.00939× and 1.00697×, and the GROMACS unchanged-source control 0.99232×. In the HPCG-derived replay of Appendix J.3, nine unchanged-source evaluations have median 0.9997× and range 0.9820–1.0165×. Canonical scores keep these raw measurements; Appendix E.4 examines small positive rewards.

Optimized programs vary more than the baseline. In the same replay, one optimized final submission has a 14.54% coefficient of variation in total runtime across outer rounds. Its median within-shape coefficient of variation across nine process timings is 34.06%, compared with 3.48% for the baseline. Noise estimated from the slower original application therefore understates the variation of a short optimized execution. Unchanged-source controls calibrate only the baseline side.

Small rewards are sensitive to the baseline. Increasing each baseline time by 1% while holding candidate times fixed gives 50 previously zero-reward correct submissions a positive reward. This synthetic perturbation uses the timing records of the earlier 842-run snapshot. Of the 625 correct submissions in that snapshot, six lie within ±1% of the 1.25× target. Small rewards thus depend on a reference that is fixed and measured by the verifier. Recomputing rewards from these timing records reproduces all 625 stored values to within 10^{-9}.

E.2. Relationships Among Speedup Summaries

If incorrect and slower submissions fall back to the baseline ($s^* = 1$) and $s^* = s$ otherwise, the weighted harmonic mean of s^* equals $1/(1-\bar{r})$ under the same weights. The harmonic-mean speedup therefore preserves the Score@1 order and adds no independent criterion. Geometric means and target attainment answer different questions. In Table 1, Hy4-Preview has the highest GM (2.13×), whereas GPT-6 Astra has the highest Score@1, $A_{1.25}$ and A_2.

Neighboring benchmarks aggregate speedup differently. SWE-fficiency aggregates expert-relative speedup ratios with a harmonic mean (Ma et al., 2026). GSO reports success against an expert-relative speed target (Shetty et al., 2025). SWE-Perf measures statistically supported runtime reductions from the original implementation and reports expert patches as references (He et al., 2026). Accelerate Bench measures each agent's improvement over the stated starting program. Expert and configuration-tuned implementations are separate calibration references, not upper bounds.

E.3. Workload Aggregation

Within a task, the ratio of total runtimes is a baseline-weighted harmonic mean of the per-case speedups $s_j = B_j/T_j$:

$$\frac{\sum_j B_j}{\sum_j T_j} = \left(\sum_j w_j/s_j\right)^{-1}, \qquad w_j = \frac{B_j}{\sum_\ell B_\ell}.$$

Equal-case harmonic averaging, as in GSO (Shetty et al., 2025), uses different weights and therefore expresses a different workload objective. Neither guarantees that every case improves. For per-case speedups $[0.75, 1000]$, the equal-case harmonic mean is about 1.50× despite a slowdown on the first case. A per-case non-regression requirement needs an explicit gate and a measurement tolerance.

Submissions reaching 2× improve every measured case. In the earlier 842-run snapshot, timing records give complete paired case timings for 421 correct submissions on 23 tasks (1,892 submission–case observations). Their case-time sums reconstruct the scored totals. All 107 submissions reaching 2× improve every case (Table 12). Of the 175 reaching 1.25×, eight have a slower case: four in `HPL`, three in `NUFFT` and one in `sparsification`. For only one of the eight do all repeated candidate timings of the slower case exceed the baseline timings. Replacing total-time speedup with an equal-case harmonic mean moves four of the 175 below 1.25×. The analysis covers these 421 submissions; the other 204 correct submissions in that snapshot lack paired case timings.

Table 12 | Workload aggregation changes few target decisions. Counts use the 421 correct submissions with paired case timings; threshold rows are nested. Slower cases are identified from median case runtimes; the 5% margin is a sensitivity level, not a noise estimate.

Total speedup	Submissions	Any slower case	Any > 5% slower	Equal-case HM below target
> 1×	342	102	28	5
≥ 1.25×	175	8	5	4
≥ 2×	107	0	0	5

E.4. Score Transformations and Reference Choice

All analyses here use the 384 runs of Table 1, one per agent and task, with tasks weighted equally. Correctness and reward fields are held fixed. Each baseline-fallback speedup is recovered exactly

from its reward r as $s^* = 1/(1-r)$; this inversion does not reconstruct raw runtimes or the size of slowdowns clipped to zero.

Small gains leave the leading agent unchanged. Of the 384 runs, 98 receive zero reward: 59 fail correctness and 39 are correct without acceleration. Another 24 save less than 1% of baseline runtime. Setting these 24 rewards to zero changes any agent's Score@1 by at most 0.09 percentage points and swaps a single pair, DeepSeek-V4-Flash and Kimi K3, whose Score@1 values differ by 0.07 points. Across savings gates of 0%, 1%, 5%, 10% and 20%, each crediting only savings beyond the gate, GPT-6 Astra stays first under both equal-task and equal-area weights. The remaining changes swap agents whose Score@1 values lie within 2 percentage points of each other. We fixed the grid after the outcomes were known, so it is a sensitivity analysis rather than a calibrated noise threshold.

Bounded savings versus log speedup. We compare three transformations of the baseline-fallback speedup s^*:

$$f_{\text{save}}(s^*) = 1 - 1/s^*, \qquad f_{\log}(s^*) = \log_2 s^*, \qquad f_{\log,K}(s^*) = \min\left(1, \frac{\log s^*}{\log K}\right), \quad K > 1.$$

Fractional savings adds progressively less as runtime approaches zero. Log scoring adds the same increment for each doubling, and the capped form stops rewarding gains beyond K.

Table 13 | Score transformations change the emphasis on large gains. Example scores assume correctness. The final column is the share of total additive score from HPCG and paged generation over the 384 runs of Table 1, weighting tasks and agents equally. The caps K are sensitivity choices.

	Score at a given speedup			Contribution
Scoring rule	2×	4×	8×	of the two tasks
Runtime savings	0.500	0.750	0.875	22.1%
Log speedup	1.000	2.000	3.000	38.4%
Capped log, $K = 2$	1.000	1.000	1.000	15.7%
Capped log, $K = 4$	0.500	1.000	1.000	22.7%
Capped log, $K = 8$	0.333	0.667	1.000	29.4%
Capped log, $K = 16$	0.250	0.500	0.750	34.4%
Capped log, $K = 32$	0.200	0.400	0.600	36.9%

Six of the seven rules keep GPT-6 Astra first. Uncapped log speedup ranks Hy4-Preview first, consistent with its higher GM in Table 1. Every other change reorders agents whose Score@1 values lie within 2 percentage points. The rules differ mainly in the weight they give to large gains. With equal task and agent weights, HPCG and paged generation contribute 22.1% of total bounded savings, and uncapped log scoring raises their share to 38.4%; under capped log scores, the share rises from 15.7% at $K = 2$ to 36.9% at $K = 32$ (Table 13). These shares refer to additive scores, not fractions of GM speedup. Bounded savings caps any single task's contribution to the equal-task mean at 1/32.

Replacing the baseline with the best observed speedup. Let U_t be the largest correct speedup among the 384 reported runs on task t. U_t is a pooled observed reference, not an expert or optimal runtime. All 32 tasks have $U_t > 1$. With the same task weights, we compare

$$H_B = \langle 1/s^* \rangle^{-1}, \qquad H_{\text{pool}} = \langle U_t/s^* \rangle^{-1},$$
$$P_{\text{save}} = \left\langle \frac{1 - 1/s^*}{1 - 1/U_t} \right\rangle, \qquad P_{\log} = \left\langle \frac{\log s^*}{\log U_t} \right\rangle.$$

H_B is the harmonic speedup over the supplied baseline, and H_{pool} is the harmonic ratio to the pooled reference. Unsuccessful runs keep $s^* = 1$ and hence fallback performance $1/U_t$. The progress scores map the baseline to zero and the pooled reference to one.

Table 14 | Reference choice changes the meaning of the score. All columns use the 384 runs of Table 1. H_B measures harmonic speedup from the supplied baseline; H_{pool} measures the harmonic ratio to each task's best observed speedup. P_{save} and $P_{\log}$ normalize progress to that observed reference. The pooled reference is the best speedup among these agents, not an expert baseline.

Agent	Score@1 (%)	H_B (×)	H_{pool}	P_{save} (%)	$P_{\log}$ (%)
GPT-6 Astra	40.3	1.675	0.790	71.3	67.3
Hy4-Preview	36.5	1.574	0.806	62.9	60.4
DeepSeek-V4.1-Flash	29.9	1.427	0.580	50.2	46.2
Gemini-3.8-Flash	29.9	1.427	0.583	53.5	47.4
GLM-5.3	28.1	1.391	0.570	48.0	44.6
GLM-5.3-Flash	27.3	1.376	0.560	53.4	47.2
Grok 4.6	26.2	1.356	0.578	44.9	39.4
DeepSeek-V4-Pro	20.6	1.260	0.473	32.0	26.2
GLM-5.2	19.9	1.249	0.484	31.2	25.4
DeepSeek-V4-Flash	19.1	1.236	0.453	31.1	25.6
Kimi K3	19.0	1.235	0.211	32.9	27.6
Hy3	9.8	1.108	0.394	15.5	12.3

The progress scores keep GPT-6 Astra and Hy4-Preview in the top two places and Hy3 last, but reorder the agents between them (Table 14). The pooled reference ranges from 1.002× on Fluidity to 89.46× on HPCG, so normalizing to it maps a small and a large measured gain to the same endpoint. On ICON, the savings denominator $1 - 1/U_t$ is about 0.021 and amplifies small timing changes. Tasks with large U_t dominate H_{pool}. A run left at baseline on HPCG contributes $1/89.46$, which moves Kimi K3, whose reported HPCG run earns zero reward, to last place. A common per-task reference cancels from geometric means, $\text{GM}(s^*/U_t) = \text{GM}(s^*)/\text{GM}(U_t)$, so it leaves the order on a shared panel unchanged. Harmonic and log normalizations instead reweight tasks.

An independently written reference. An expert implementation with measured speedup e_t could replace U_t, for example as progress $\log s^*/\log e_t$, provided it passes the same checks under the same resource, workload and timing rules and is fixed independently of the agents compared. We built one such reference for HPCG-derived. The task's operator is $A = 27I - S_x \otimes S_y \otimes S_z$, where each S has ones on the main and adjacent diagonals, and the sine basis diagonalizes each factor. Our implementation applies dense orthonormal sine transforms along the three axes and divides by the eigenvalues. It uses no external FFT library and modifies only `src/application_main.cpp`. We wrote it with knowledge of the task and of earlier trajectories, so it is a separate implementation rather than a blind discovery or an expert optimum.

We evaluated three artifacts: the pristine source, this reference and one agent's final submission (DeepSeek-V4-Flash, `45WWLii`). Each receives three clean builds and full private evaluations on four physical cores within one NUMA node, with SMT siblings excluded, 6 GiB of memory and no network. The verifier image, tests and six workload shapes are unchanged; one new right-hand-side seed is used throughout, and artifact order is counterbalanced across rounds. All nine evaluations pass. The reference reaches 21.18–22.22× and the agent's submission 25.86–26.33×. Paired by round, the agent's program is 1.19–1.25× faster than ours in every round (Table 15). The three-round ranges describe these measurements, not population intervals. This replay does not change official scores.

Table 15 | The agent's HPCG-derived solver outperforms an independently written direct solver. The first three rows are speedups over the supplied baseline; the last row divides the reference's runtime by the agent's within each round. All nine evaluations pass.

Comparison	Round 1	Round 2	Round 3
Rebuilt pristine source	1.004×	0.996×	1.001×
Independent direct solver	21.180×	22.216×	21.261×
Agent final submission	26.325×	25.862×	26.209×
Independent time / agent time	1.245×	1.189×	1.200×

F. Per-Task Results

Large speedups concentrate on a subset of tasks. Table 16 lists the per-task reward of every run in Table 1. At least one agent reaches 2× on 15 of the 32 tasks, but all twelve do so only on Paged LM generation. On seven tasks (Fluidity, ICON, miniVite, NEMO, Quantum ESPRESSO, QuEST and VPIC), no agent reaches 1.25×. Correctness failures are most frequent on Fluidity, where seven of the twelve runs fail.

G. Annotation Procedures

Section 5 uses three kinds of labels (Table 17). Technique labels describe what a run's final implementation changes relative to the supplied source. Full-trajectory annotations describe each tool call in a run and the relationships among the optimization choices its trajectory records. Measurement-command groups come from fixed lexical rules (Appendix H.4).

G.1. Technique Labels

GPT-5.6 Luna, run through Codex CLI at medium reasoning effort, reviews one task at a time. Each review covers every run of that task in a labeling panel that includes all 480 runs analyzed here. The reviewer reads the supplied source, the retained candidate source and launch configuration, and the trajectory. Each source excerpt cited as direct evidence keeps its file path and line range.

Each review has two phases. First, the reviewer writes a task map: what the application computes, what must remain correct, and what the supplied implementation executes. It then defines task-local algorithm routes and mechanism tests and labels every run. These labels are fixed before the reviewer sees the scored outcomes. Second, the reviewer checks each outcome record against the saved labels and records any conflict as a separate proposed correction.

For each run, the review records a class (optimization, original scheme, or unresolved), mechanism tags from a shared vocabulary of fifteen optimization mechanisms and one correctness-repair tag, implementation status, confidence, source-access level, and evidence locators. A change counts only if it is active in the final execution path, so unwired, compiled-out, or reverted code does not qualify. A correctness or build repair alone is not an optimization. Changing a thread count is not labeled as a new parallel algorithm. In the 480-run panel, 464 labels rest on directly read final source and 15 on source excerpts bound to the final state; 438 have high confidence.

An independent source audit then mapped the fifteen mechanism tags into six categories and checked tags against recorded behavior. It corrected tags in nine runs and withheld three labels whose active execution path could not be confirmed. With these three labels withheld and one left unresolved by the review, 476 runs are classified.

Table 16 | Per-task reward of the runs in Table 1. Entries are runtime savings r_t (%); × marks a run that fails correctness checks, and 0 a correct run without acceleration. Bold entries reach at least 2× speedup ($r_t \geq 50$); entries just below the 20 and 50 thresholds show decimals. Agents are ordered by Score@1, the column mean.

Task	GPT-6 Astra	Hy4-Preview	DeepSeek-V4.1-Flash	Gemini-3.8-Flash	GLM-5.3	GLM-5.3-Flash	Grok 4.6	DeepSeek-V4-Pro	GLM-5.2	DeepSeek-V4-Flash	Kimi K3	Hy3
GROMACS	**65**	**63**	0	**58**	**52**	**53**	**53**	**52**	**53**	**53**	**53**	0
LAMMPS	**54**	47	**56**	1	3	7	11	0	1	0	7	3
GPAW	31	5	4	23	×	×	×	12	×	12	×	0
Quantum ESPRESSO	9	×	0	4	12	4	×	1	1	1	0	×
SIESTA	**61**	**55**	×	39	2	49.8	×	×	×	**51**	49.7	×
Fluidity	0	0	×	×	×	0	×	×	0	×	0	×
ICON	0	2	×	0	0	2	×	×	0	0	0	0
MPAS-A	23	25	4	24	19	14	1	×	0	2	24	×
NEMO	0	3	1	×	×	3	2	×	3	1	1	3
OpenFOAM	23	12	7	12	0	2	2	25	0	13	×	0
Palabos	**88**	**91**	0	**62**	×	16	**72**	23	×	1	11	0
SU2	**56**	**64**	**61**	45	44	29	25	**54**	32	13	31	8
OpenMC	47	30	21	26	37	3	4	3	9	17	15	3
ShengBTE	**54**	×	**55**	×	43	38	37	36	36	35	15	×
VPIC	0	0	×	5	×	3	2	0	×	×	×	0
PRESTO	27	0	0	1	×	×	0	0	×	2	×	×
ABySS	×	42	49	×	49.9	45	43	36	10	17	37	9
MHM2	37	2	×	33	×	27	2	0	×	0	0	3
MrBayes	29	32	23	12	11	19	9	14	12	7	17	8
wtdbg2	**58**	**58**	39	16	13	0	34	22	0	24	0	0
HPCG-derived	**97**	**99**	**98**	**92**	**99**	**97**	**99**	**97**	**98**	**98**	×	**97**
HPL-derived	45	34	32	34	47	34	**68**	32	48	22	36	7
NUFFT imaging	**80**	**52**	**75**	**76**	×	44	47	38	32	33	**54**	33
Tucker reconstruction	**68**	**82**	**63**	**74**	**70**	**58**	41	36	33	30	×	31
Graph sparsification	**63**	**79**	**79**	16	**79**	**78**	**58**	32	**61**	34	32	14
miniVite	10	18	19	8	14	12	19.95	4	7	3	6	2
QuEST	12	10	11	9	12	0	16	0	7	9	0	2
QiboTN	×	0	29	×	31	×	34	2	0	1	0	0
Paged LM generation	**94**	**94**	**94**	**93**	**94**	**91**	**93**	**93**	**91**	**90**	**85**	**87**
ThunderSVM	**77**	**85**	**53**	**70**	**83**	**65**	8	48	43	40	**52**	×
DataLife	**78**	**84**	**83**	**87**	**83**	**79**	**59**	×	**62**	1	**82**	1
IO500-derived	4	2	3	37	0	1	1	0	×	×	0	1
Score@1	40.3	36.5	29.9	29.9	28.1	27.3	26.2	20.6	19.9	19.1	19.0	9.8

Table 17 | Label sources used in Section 5. All three cover the same 480 runs.

Labels	Annotator	Unit	Input	Used in
Technique	GPT-5.6 Luna, medium effort, Codex CLI	Final implementation of a run	Supplied and candidate source, build and launch files, trajectory excerpts	§5.1
Full trajectory	DeepSeek-V4.1-Flash, high effort, Responses API	Tool call; relationship record	Complete trajectory text	§5.2, §5.3
Measurement command	Lexical rules	Run	Shell-command arguments	§5.2

The six categories are computation and reuse (A1–A7), parallelism and vectorization (T1–T3), runtime and numerical configuration (R1–R2), library/backend selection (L1), compiler/build tuning (B1), and I/O (I1). Of the 476 classified runs, 361 have at least one retained optimization label; the other 115 keep the original scheme, possibly with repairs or other edits. The four unclassified runs remain in the performance and command analyses and are excluded from technique-label contrasts. A run may carry multiple labels.

Boundaries between related labels. Numerical reformulation (A2) changes the mathematical formulation or equations, whereas numerical parameter choice (R2) changes exposed tolerances, discretization, preconditioner, or solver settings. Memoization/precomputation (A4) retains computed values or hoists invariant work; workspace reuse (A6) retains allocated storage across iterations. The task codebooks state these distinctions explicitly, for example in the `OpenFOAM` definitions. CPU parallelism (T1) records changes to active parallel execution, while runtime resource tuning (R1) records resource settings. This boundary varies across codebooks. `OpenFOAM` separates explicit solver threading from resource settings, whereas `paged LM generation` places active thread-pool and BLAS/OpenMP controls under T1. The reported counts keep these task-local assignments.

G.2. Full-Trajectory Annotation

DeepSeek-V4.1-Flash annotates the complete trajectories of all 480 runs at high reasoning effort, through a Responses API gateway with a strict JSON output schema and up to 131,072 output tokens per request. The model receives the full trajectory text, including every step, message, reasoning record, tool argument, and tool return, but not final scores or earlier annotations.

Trajectories are split without dropping text. Each field is cut into segments of at most 10,000 characters, and each segment keeps its position in the original field. The segments are grouped into 11,820 core chunks of about 144,000 characters, keeping whole steps together where possible. Each request adds 24,000 characters of context per side. A parser assigns every tool call an action ID and every message without a tool call a statement ID; the annotator labels each ID in its core chunk exactly once.

For each action, the annotator records its facets, such as source edit, build configuration, correctness test, performance measurement, or profiling; a decision label (investigate, propose, implement, refine, repair, revert, retain/select, test, submit, continue, no new decision, or unclear); the basis for that decision; the visible effect of the action; and a selection label. Reasons, cited feedback, and results point to earlier text spans. An action is labeled successful when the requested operation succeeded; correctness and speed are taken from returned tests. A revert is labeled only when visible in an action. A relationship record connects two distinct optimization choices, or an optimization and

an execution-path condition that enables, blocks, or bypasses it. It carries one of six types and a basis in visible source, returned feedback, or an agent statement. Performance hypotheses are recorded as agent claims.

Before the full run, we validated the protocol on 24 inputs: 22 real chunks, a task-description negative control, and a repeated chunk. On the repeated chunk, decision labels agree for 19 of 21 actions and selection labels for all 21. All 11,820 chunk outputs pass checks for schema validity, exact action coverage, valid evidence IDs, source-role constraints, and chronology.

Chunk outputs are assembled into one evidence graph per trajectory, giving 203,472 tool actions, 417 agent-statement records, and 23,845 relationship records. Exact duplicate records are merged, and simultaneous tool calls share their original agent turn. Every cited text span is checked against its source. Targeted source reviews check the distinctions the analysis relies on: plans versus executed operations, agent hypotheses versus returned observations, temporary working copies versus the submitted tree, and tool success versus application correctness. Recomputing from the saved evidence graphs reproduces every count and association in Sections 5.2 and 5.3.

Decision labels are correct for 77 of 80 calls (96%; 95% Wilson interval 90–99%), and the test, performance-measurement, profiling, and source-edit facets for 73 (91%; 83–96%). Cited reasons explain the labeled call in 67 of 71 applicable cases (94%), and cited feedback is output the call responds to in 63 of 66 (95%). Errors concentrate in the measurement stratum (16 of 20 facets correct); most are calls that prepare profiling, such as adding `-pg`, but are labeled as profiling. Relationship records are less reliable, so Section 5.3 uses them only to locate evidence for review (Appendix I.2).

G.3. From Labels to Analysis Variables

Each analysis converts labels into run-level variables before comparing outcomes. Technique contrasts compare labeled and unlabeled runs of the same model and task, average within tasks, and weight eligible tasks equally, with task-bootstrap intervals (Appendix H.2). Measurement-command groups enter a regression with additive task and model effects (Appendix H.4). Trajectory-feature models add a fixed effect per model–task cell (Appendix H.8). All three report task-clustered 95% intervals.

H. Technique and Workflow Analysis

This appendix gives the sample, definitions and full tables behind Sections 5.1 and 5.2, in the order of their observations. Appendix G describes how the labels are assigned.

H.1. Analysis Sample

Each of the five agents of Section 5 runs three times on all 32 tasks, giving 480 runs that represent every model–task pair equally and have complete annotation and command records. Of the final submissions, 396 are correct and 84 incorrect; 175 of the correct ones reach 1.25×. Final performance uses the scored outcomes, and workflow analyses use the public-test responses and timings recorded during each run. Figure 8 shows every annotated action sequence alongside its outcome.

H.2. Technique Counts and Outcome Contrasts

Tables 18 and 19 give technique counts by final outcome and by model for the 476 classified runs.

For each category, we compare mean final reward between runs with and without the label,

Table 18 | Optimization labels and final outcomes in the 480-run panel. Categories overlap; 476 runs have classified technique labels. Speedup requires final correctness.

Technique	Runs	Tasks	Correct	≥ 1.25×
Computation / reuse	225	29	201 (89.3%)	108 (48.0%)
Parallel / vectorized	132	22	116 (87.9%)	83 (62.9%)
Runtime / numerical	64	17	57 (89.1%)	31 (48.4%)
Library / backend	46	11	39 (84.8%)	26 (56.5%)
Compiler / build	40	14	32 (80.0%)	10 (25.0%)
I/O / serialization	28	8	24 (85.7%)	11 (39.3%)

Table 19 | Eligible source labels and technique counts by model. C: computation/reuse; P: parallelism/vectorization; R: runtime/numerical configuration; L: library/backend; B: compiler/build; I: I/O. Each model has 96 scored runs; categories overlap.

Model	Labeled	C	P	R	L	B	I
DeepSeek-V4.1-Flash	96	45	34	8	7	5	7
Grok 4.6	95	67	36	6	7	3	13
DeepSeek-V4-Pro	93	33	26	17	8	13	3
DeepSeek-V4-Flash	96	46	22	16	11	8	2
Hy3	96	34	14	17	13	11	3

within each model–task group that contains both. Group contrasts are averaged within each task, and eligible tasks receive equal weight; incorrect runs keep their scored reward. Intervals resample task means 10,000 times with seed 20260922. Table 20 reports each category's group and task counts, estimate and interval.

Table 20 | Exploratory reward contrasts within model–task groups, averaged over eligible tasks. Intervals use 10,000 task-bootstrap resamples and are pointwise, not simultaneous.

Technique	Groups	Tasks	Difference (pp) [95% CI]
Computation / reuse	53	27	+8.0 [1.4, 15.9]
Parallel / vectorized	46	19	+8.9 [1.9, 17.0]
Runtime / numerical	35	17	+0.6 [-6.3, 8.9]
Library / backend	15	10	-4.9 [-28.0, 14.4]
Compiler / build	25	13	-0.7 [-6.4, 6.4]
I/O / serialization	9	7	+1.7 [-8.1, 12.4]

H.3. Five-Agent Process Comparison

Figure 5 uses all 203,472 tool-action annotations. Within each trajectory, we compute category fractions and split the tool-call sequence into three consecutive, nearly equal parts; the test-label fraction of a part uses that part's calls as the denominator. Each agent's value weights its 96 trajectories equally. Retention and submission form one displayed category, and investigation, continuation, proposals, unclear actions and other labels form the inspection/other category.

A multi-tool turn is a tool-bearing step with more than one call. Each run's fraction divides these turns by all its tool-bearing steps. Averaged over runs, the fraction is 57.1% for Grok 4.6, compared with 10.0% for DeepSeek-V4-Flash, 8.8% for DeepSeek-V4-Pro, 9.4% for DeepSeek-V4.1-Flash and 1.6% for Hy3. Intervals use 10,000 task-bootstrap draws with seed 20260924.

The DeepSeek and Hy3 records identify Claude Code 2.1.265 (one DeepSeek-V4-Pro record lacks harness metadata), and all Grok records identify Grok Build 1.0.24. Grok comparisons therefore vary both model and harness. Within Claude Code, DeepSeek-V4.1-Flash reaches 1.25× in 46 of 96 runs and DeepSeek-V4-Flash in 35 (+11.5 percentage points; 95% interval $[-2.1, 24.0]$). The outcome panel counts all 96 runs per agent, including incorrect ones.

H.4. Timing and Profiling Groups

Lexical rules scan executable shell-command arguments over each full trajectory. Runs with a profiling-related match form P2 and runs with only timing matches form P1; P0 runs have no matched command but may measure performance in other ways. A matched command need not complete or address the scored workload. Table 21 covers all 480 runs. Raw reward intervals resample the 32 tasks 10,000 times. Adjusted contrasts regress reward on group indicators with additive task and model effects, using a CR1 task-cluster covariance and t_{31} intervals.

Adjusted mean reward differences are +4.4 percentage points for P1–P0 (95% interval $[-4.0, 12.8]$), +8.1 for P2–P0 ($[-2.9, 19.0]$) and +3.7 for P2–P1 ($[-0.6, 8.0]$). Adjustment reduces the P2–P0 difference from 17.9 points to 8.1, so more than half of the raw gap reflects which tasks and models profile.

Table 21 | Command-based measurement groups across all 480 trajectories. P0, P1, and P2 are observed tool patterns, not ordered levels of diagnosis quality.

Group	Runs	Correct	≥ 1.25×	Mean reward
P0	61	48	7	8.1%
P1	251	197	93	22.2%
P2	168	151	75	26.0%

H.5. First Public Test of an Edited Program

We reconstruct public requests for 479 of the 480 runs; the missing record is a NEMO run. We compare readable source and configuration contents with the initial image before inspecting feedback; a change confined to build outputs, metadata or omitted files does not count as a source edit. This identifies 385 first edited submissions. Seventy-seven runs send no public request, 16 send requests without an observed source or configuration edit, and one lacks the initial image. Of the 385, 235 have the original request bytes or match a stored snapshot exactly, and 155 of these also have feedback, covering 26 tasks. Requiring confirmed identities for all preceding requests leaves 147 tests, including 31 passes.

Stage labels in Table 2 are mutually exclusive. Program checks include regression, full-field and exact-replay mismatches; infrastructure/reference responses include evaluator-internal errors and baseline-stage failures. Fourteen of the 27 infrastructure/reference responses come from LAMMPS, and fourteen of the fifteen candidate-execution failures from ThunderSVM. Four initially failing tests are followed by a correct response. Later feedback is incomplete, so this count is not a repair rate.

H.6. Comparable Version Histories

A version history is the sequence of correct candidates in one run that share a timing protocol. Timing comparisons require an original correct response, a finite positive runtime in the same field, source-confirmed contents and matching protocol fields: profiles, case counts, grid shapes, graph

sizes, right-hand-side counts and prompt lengths where reported. Two candidates count as the same program when their files and links are identical. In each run, we keep the largest comparable group, breaking ties by first observation. Thirty-one histories contain at least two distinct programs: ten each from DeepSeek-V4-Pro and DeepSeek-V4-Flash, seven from DeepSeek-V4.1-Flash and four from Grok 4.6. Hy3 has none.

Endpoints. Each program's runtime is the median of its observed runtimes. The first and last programs follow request order, and the best program has the lowest median. We report $T_{\text{first}}/T_{\text{best}}$ and $T_{\text{first}}/T_{\text{last}}$, and the last/best gap $T_{\text{last}}/T_{\text{best}}$. At threshold δ, a first-to-last gain requires $T_{\text{first}}/T_{\text{last}} > 1+\delta$, a regression $T_{\text{last}}/T_{\text{first}} > 1+\delta$, and a last/best gap $T_{\text{last}}/T_{\text{best}} > 1+\delta$. The first program is the first correct candidate, not the task baseline. The last correct public test can be followed by failed requests or untested edits. The final submission's source files match the last program in 25 histories, ten of them with a last/best gap above 5%.

Consecutive changes. The 70 changed-program pairs come from 26 of these histories. Both programs in a pair must be adjacent in the original sequence, source-confirmed, correct and protocol-comparable; resubmitted programs are excluded. Each ratio compares the two observed runtimes. A revision is faster when $T_{\text{previous}}/T_{\text{current}} > 1$, and it sets a new best when it beats every earlier comparable correct observation.

Table 22 | Public iteration outcomes within the 480-run panel. Rates first average revisions within runs, then runs within applications; 95% intervals resample the seven represented applications.

Observation	Pooled count	App.-weighted % [95% CI]
First-to-last speedup > 1.05×	19/31	65.2 [40.5, 86.6]
Last is > 5% slower than best	11/31	31.3 [9.5, 58.1]
Last is the best tested version	14/31	51.3 [29.7, 73.2]
Revision improves over predecessor	36/70	62.9 [45.8, 81.2]
Revision sets a new observed best	26/70	58.5 [40.0, 78.4]

Table 23 | Endpoint differences at three thresholds in 31 first/last and last/best program-median pairs.

Threshold	Last faster than first	Last slower than best
5%	19	11
10%	17	8
20%	13	6

Threshold sensitivity. The histories contain 125 distinct programs: 108 observed once, fourteen twice and three three times. Because most programs have a single timing, Table 23 reports the endpoint counts at three thresholds. At 5%, 10% and 20%, there are nineteen, seventeen and thirteen first-to-last gains and eleven, eight and six last/best gaps. Table 22 also reports application-weighted rates, averaging changes within runs, runs within applications and applications equally, with bootstrap intervals over the seven application means.

H.7. Candidate Management

Across 468 trajectories, 19,269 turns contain a modification or restoration/retention label that cites earlier feedback. Modification combines implementation, repair and refinement; restoration/retention

combines reverting and retaining or selecting a candidate. Of these turns, 17,537 contain modification only, 1,596 restoration/retention only, and 136 both; multiple qualifying actions in one turn count once. Reading and administrative tools are excluded from modification counts. A turn's feedback source and rationale come only from its qualifying actions.

The cited feedback comes from tests or measurements in 3,979 turns, build execution in 1,699, inspection or profiling in 8,110, other operations in 3,890, and mixed sources in 1,591. When the call that produced the feedback has several facets, test/measurement takes precedence over build execution; reading a timing log counts as inspection. Every feedback link resolves to an earlier observation and the call that produced it.

The narrower restoration/selection label covers 642 tool records in 206 trajectories. Merging simultaneous records gives 583 turns, of which 555 (95.2%) give a source-bound reason and 542 (93.0%) cite preceding feedback. These operations include restoring a temporary state, promoting a candidate and making backups. Declared intentions, 303 agent statements and 62 tool records, are counted separately. Among later modification or restoration turns that cite a source-bound public correctness response, 42 in fifteen trajectories cite a passing result and 740 in 159 trajectories cite a failing one.

H.8. Trajectory Features and Outcomes

Four trajectory features were fixed before fitting: presence of local restoration/selection; the share of feedback-linked revision turns containing restoration or retention; presence of a relationship record linking several modification layers; and the share of such records among all relationship groups. Presence features use all 480 trajectories. The restoration share uses the 468 with at least one revision turn, and the layer share the 475 with at least one relationship record. The outcome is per-run reward in percentage points, with all 84 incorrect runs at zero.

Each feature enters a separate regression with a fixed effect for every model–task cell, so its coefficient compares the three runs of one model on one task. Standard errors use a task-cluster CR1 covariance with correction $G/(G-1)\,(N-1)/(N-K)$, where $G = 32$ and K is the design rank (161 for the full panel and the layer-share subset, 160 for the restoration-share subset), followed by t_{31} intervals. The feature varies within 100 cells for local restoration/selection, 69 for multi-layer references, and 145 and 150 for the two shares.

All four intervals include zero (Table 24). Before adjustment, the 361 runs with multi-layer references reach 1.25× in 39.6% and are correct in 80.6%, versus 26.9% and 88.2% for the other 119 runs.

Table 24 | Process features and final reward. Each row is a separate within-model–task comparison. Intervals are pointwise 95% task-cluster CR1 intervals with t_{31} critical values. Variable cells contain variation in the indicated feature.

Feature	Runs	Variable cells	Reward pp [95% CI]
Local restoration/selection: present	480	100	−1.83 [−6.73, 3.07]
Restoration/retention share: +10 pp	468	145	−0.37 [−3.04, 2.31]
Multi-layer reference: present	480	69	+5.62 [−3.44, 14.67]
Multi-layer group share: +10 pp	475	150	+0.15 [−2.78, 3.09]

I. Dependencies Across Layers

Section 5.3 estimates how often a change's effect or correctness depends on build settings, runtime settings or the executed code path. The estimate comes from a reviewed random sample (Appendix I.2); the annotator's relationship records are used only to locate candidate evidence and to stratify that sample.

I.1. Relationship Records

The 23,845 relationship records occur in 475 of the 480 trajectories; the five without records remain in the run denominator. Build requirements occur in 410 trajectories, execution-path dependencies in 440, and at least one of the two in 464. Each record has one of the six types in Table 25, and one trajectory can contain several types. Visible source is the most common basis for build requirements, while execution feedback supports 30.6% of incompatibility records and 11.5% of build-requirement records. Performance hypotheses remain agent claims even when their recorded basis is execution feedback, as it is for 153 records; the table reports that basis.

Table 25 | Optimization relationships across 480 trajectories. Runs can contain multiple types. Each record has one annotated basis: an agent statement, visible source, or execution feedback.

Relationship	Runs	Records	Agent	Source	Feedback
Build requirement	410	3,956	1,284	2,216	456
Execution-path dependency	440	4,267	1,711	2,004	552
Incompatibility	440	3,923	2,326	397	1,200
Replacement	451	4,331	1,790	2,224	317
Bypass	432	3,819	1,832	1,371	616
Performance hypothesis	418	3,549	3,390	6	153

For layer summaries, records are grouped by trajectory and cited source turn: the latest cited assistant reasoning or message turn, or otherwise the latest cited turn. This gives 19,191 groups, 705 of which span several annotation chunks. Exact identifiers connect each record's references to annotated actions. A group's layers are the union of the code-editing, build-configuration and runtime-configuration facets it references. At least one layer is linked in 7,737 groups, and 1,814 link two or more (23.4% of linked groups, 9.5% of all groups). These multi-layer groups occur in 361 trajectories: 1,245 link code and build, 425 code and runtime, 50 build and runtime settings, and 94 all three.

I.2. Reviewed Dependency Estimates

Automatic labels can be accurate case by case and still bias population counts, so we estimate prevalence from a reviewed random sample rather than from annotator counts (Chowdhury et al., 2024; Egami et al., 2023; Angelopoulos et al., 2023). A trajectory is *flagged* if the annotator records a build, execution-path, or bypass dependency supported by execution feedback; 344 of the 480 trajectories are flagged. We sample flagged and unflagged trajectories separately in a fixed pseudorandom order.

A trajectory counts as confirmed if tool output (a build or link error, profile, timing, log, or public-test response) shows that an agent change's effect or correctness depended on a build setting, toolchain, or library; on a runtime setting, input, or thread count; or on which code path the application executed, including a change that is never executed. An agent's own statement, source text alone, or an environment fact unrelated to any change does not count. Each confirmed trajectory is further classified as optimization-related, when the dependency concerns an optimization's speed

or correctness, or as an environment workaround, when it concerns only building, loading, or running the program in the evaluation container.

The reviewer searches each sampled trajectory's complete record of tool calls and outputs, not only the annotator's records. Of 40 flagged trajectories, 34 are confirmed (85%); of 40 unflagged trajectories, 15 are (38%). Weighting the two strata by their sizes gives an estimated 72% of all 480 trajectories (95% interval 63–80%, normal approximation with finite-population correction). Optimization-related dependencies are confirmed in 24 flagged and 13 unflagged trajectories, giving an estimated 52% (41–63%). Flagged trajectories contain about 85% of the estimated confirmed trajectories. A first-pass review of 40 further flagged trajectories, restricted to at most eight annotator records each, confirms 26; pooling it with the full reviews gives a lower estimate of 64% (57–71%). At the record level, 29 of 60 randomly sampled feedback-supported dependency records are confirmed (48%; build 8/13, execution path 7/21, bypass 14/26). Most rejected records either state the dependency only in the agent's words (11) or describe a comparison between settings or an ordinary bug rather than a dependency (13). The record counts in Appendix I.1 therefore locate candidate evidence and do not estimate prevalence.

Confirmed cases span the three layers of Section 5.3. They include a thread-count variable that the linear-algebra library ignores, compiler flags that change numerical results (ShengBTE, SIESTA), an algorithm that is slower on small inputs and faster on large ones (NUFFT), an OpenMP change that slows a program at four threads until shared state is made thread-private (PRESTO), an edited `HPL_dgemm.c` that leaves the binary byte-identical because a macro maps the call to `cblas_dgemm` (HPL), and public tests that return before the edited code runs.

J. Controlled Experiments

Each experiment replays fixed programs under the unchanged private verifier and varies one set of factors. GROMACS varies build and runtime settings on identical source, QiboTN switches three code changes on and off in a full factorial design, and HPCG-derived compares each run's fastest public-test candidate with its final submission. All replays run on dedicated physical cores within one NUMA node, with SMT siblings excluded; background services remain active.

J.1. GROMACS: One Source, Four Configurations

The task rebuilds the submitted molecular-simulation source according to `workflow.json` and times the complete prescribed `mdrun`. In the replayed run, the final submission differs from the pristine source only in `workflow.json`, so the whole gain lies in build and runtime configuration. We replay the identical source under four combinations of SIMD and thread settings on four physical cores. The one-thread variants keep this allocation, and both thread counts reuse the same clean build for each SIMD setting.

Speedups of the same source range from 0.289× to 2.123× across the four configurations, all of which pass the numerical and replay checks on six private physical profiles with three interleaved timing repetitions per profile (Table 26). With SIMD disabled and four threads, the unchanged source measures 0.992×. AVX2 with four threads reaches 2.123×; with one thread, AVX2 gives 0.709× and disabled SIMD 0.289×.

Table 26 | The same GROMACS source ranges from 0.289× to 2.123×. Runtime is the sum of six workload medians. Each row has its own interleaved measurement of the fixed SIMD-disabled, four-thread reference. All rows pass correctness.

SIMD	Threads	Reference (s)	Candidate (s)	Speedup
Disabled	1	30.58	105.70	0.289×
Disabled	4	30.48	30.72	0.992×
AVX2	1	30.69	43.31	0.709×
AVX2	4	30.80	14.51	2.123×

J.2. QiboTN: Interacting Code Changes

The controlled program is DeepSeek-V4-Flash's final QiboTN submission `5KUtPHw`. We vary broad dispatch to a fast exact simulator (B), reuse of a previously computed state for an observable (R), and deferred loading of tensor libraries (L). Level one keeps the submission's behavior and zero disables only that feature. All interventions edit the same backend file and leave the other 66 files unchanged, so the all-off variant still contains the submission's other changes. Each of the eight variants receives three complete application evaluations in a fixed random order.

Workload and checks. Each round runs six fresh-process variational circuits with 9–14 qubits, with identical inputs across variants within a round and different fixed seeds across rounds. We check the full complex state against an independent calculation, its norm, all single-qubit expectations, a symbolic observable, selected probabilities and exact numerical replay. Every recipe runs each role twice, in reference–candidate and candidate–reference order. The reported time sums six per-recipe median wall times, including process startup, imports, execution and full output. The environment uses eight physical cores, a 32 GiB limit, a fixed container image and no network; the source and tests are checked for changes before and after each evaluation. All 24 evaluations pass, giving 576 process timings.

Table 27 | Deferred loading (L) helps QiboTN only when broad dispatch (B) is on. Each row is one variant of the eight-variant factorial design; 1 keeps the submission's behavior and 0 disables it. Candidate time sums six per-recipe median wall times. Gain is the all-off candidate time divided by the variant's candidate time within the same round; share is the fraction of the all-on runtime saving that the variant recovers. Ranges span three rounds with different circuit seeds. All 24 evaluations pass correctness.

B	R	L	Candidate (s)	Gain over all-off	Share of saving (%)
0	0	0	76.99–77.12	1.000×	0.00
1	0	0	50.86–51.00	1.512–1.515×	37.76–37.93
0	1	0	76.94–77.21	0.999–1.002×	−0.14 to 0.18
0	0	1	75.96–76.02	1.014–1.015×	1.50–1.60
1	1	0	50.89–50.93	1.512–1.514×	37.79–37.88
1	0	1	7.94–8.04	9.592–9.711×	99.87–100.08
0	1	1	75.98–76.01	1.013–1.015×	1.44–1.61
1	1	1	7.95–8.01	9.614–9.699×	100.00

Results. Deferred loading (L) gives a large gain only when broad dispatch (B) is on. With R off, adding L to the B-on variant improves runtime by 6.343–6.409×, whereas adding L to the all-off variant improves it by 1.014–1.015× (Table 27). Averaged over both R levels on the log-runtime scale, the B–L interaction is a ratio of gains of 6.278–6.307. All three features together accelerate the all-off

variant by 9.614–9.699×, and no single feature meets the predeclared criterion of recovering 90% of this combined saving in every round. Removing B from the full submission slows it by 9.490–9.560×, and removing L by 6.359–6.405×, so the knockout effects overlap rather than add. The source explains the dependency: deferred loading avoids importing the tensor library only when the execution path does not need it. When B is off, these recipes never reach the state-reuse path, which makes R an unreachable-path control.

Because the rounds use different circuit seeds, the ranges include both input and timing variation. Median within-recipe differences between candidate pairs range from 0.31% to 1.33% across variants, and baseline aggregate times range from 75.793 to 76.192 seconds. The baseline is the supplied pristine QiboTN source with its original application runner, whereas the all-off variant keeps the submission's other edits.

J.3. HPCG-derived: Selected Candidates Versus Final Submissions

For seven HPCG-derived runs, we replay the candidate with the best public-test runtime in each run alongside its final submission. The replay set contains 18 candidate instances covering 16 distinct programs: the seven final submissions, the six best public-test candidates that differ from them, and five further candidates selected from the public feedback available at earlier budget cutoffs. Three outer rounds produce 54 complete private evaluations, 324 case checks, and 972 baseline and 972 candidate process timings; every evaluation passes. All seven final submissions solve the task's constant-coefficient stencil with a sine-transform direct solver for both right-hand sides.

The same algorithm family still yields different runtimes. In six runs, the candidate with the best public-test runtime differs from the final submission: two final submissions are faster in every round, two are slower in every round, and two change direction across rounds. In the seventh run the two are identical. A single timing comparison can therefore misstate which of two candidates is faster.

Replay conditions also shift absolute speedups. The replay fixes a new right-hand-side seed and runs on a different allocation from the original runs. Under these conditions, one run's recorded final speedup of 63.36× becomes 19.73×, mostly through a change in candidate time. Capping that program at four threads gives a three-run median of 20.28× with overlapping timing ranges, so thread oversubscription alone does not explain the difference. The replay does not change official scores.

K. Case Records

The cases below come from the 480 analyzed runs and are grouped by the observation in Section 5 they illustrate. We treat agent statements as hypotheses, and tool output and submitted source as observations. Step numbers refer to the original trajectory.

K.1. Repairing a Candidate After a Failed Check

HPCG-derived: an indexing error found by residuals. The task solves a fixed three-dimensional 27-point Dirichlet system for two right-hand sides; correctness requires small solution error, small independently computed residuals and reproducible vectors. The operator admits transform-based direct solvers. In Hy3 run `u2x9KjE`, the agent's transform-based candidate produces relative residuals of about 1.6–1.9. After the agent changes the slice stride in `apply_y` to `ny*nx` and rebuilds, the same checker reports residuals of 3.0×10^{-15} to 9.1×10^{-15}.

QuEST: an operation applied twice. The task simulates state-vector and density-matrix quantum circuits and checks complete states and probabilities. In Hy3 run `r2HQQna`, step 146 reports a maximum amplitude discrepancy of 0.644 for the agent's fused density-matrix path. At steps 174–175 the agent attributes the discrepancy to applying the ket operation twice and changes the dispatch guard. The patch introduces a build error that the agent repairs by step 180. After further fixes to its test harness, step 201 reports no differing local cases, and step 208 returns a public correctness pass.

K.2. Keeping a Slower Candidate After Numerical Checks

SU2 runs 250 iterations of a transonic airfoil simulation and must write five complete flow-field snapshots within a field-error tolerance of 3×10^{-3}. In DeepSeek-V4.1-Flash run `evDkccR`, the agent compares two preconditioner settings. At step 849, Jacobi-12 gives worst relative field L_2 errors of 2.096×10^{-4} and 1.759×10^{-4} on the two public profiles, and Jacobi-15 raises the first to 1.489×10^{-3} while lowering the second. At step 850, four interleaved local timings per setting give medians of 3.030 s for Jacobi-12 and 4.297 s for LU-SGS with five inner iterations.

At step 851 the agent rejects Jacobi, citing errors that change non-monotonically with the iteration count and uncertainty about other inputs. It keeps LU-SGS, checks that its configuration differs from the reference only in the preconditioner thread setting, and obtains two public correctness passes at step 852. The final submission scores reward 0.456. The discarded Jacobi candidate was not replayed privately, so the private-verifier speed given up by this choice remains unmeasured.

K.3. Changes the Application Does Not Execute

HPL: an optimized kernel bypassed by the build. The task runs a complete MPI dense linear solve, from input distribution to solution export and checks. In DeepSeek-V4-Flash run `hiruimA`, the agent optimizes `HPL_dgemm.c`. A `cmp` check reports the stock and optimized executables as identical, and the expected kernel symbol is absent; a subsequent symbol inspection shows an unresolved `cblas_dgemm` call. The agent then moves its optimization into a strong CBLAS override with a fallback. A source listing read late in the trajectory still contains the override, and the agent's closing summary lists it among the retained changes. Because the final submission also changes application I/O, its reward of 0.290 does not isolate the override's speedup; the case documents the execution-path check.

QiboTN: a local test that misses the changed route. The task executes quantum circuits and must return complete states and observables. In Grok 4.6 run `fXyyQog`, the public workflow passes before the agent changes the tensor-network fallback. The next local smoke run uses the dense-state route and therefore does not exercise the changed fallback. The final submission is correct, so the case concerns what the executed test covered and does not indicate a faulty fallback.

K.4. Build Settings That Change Numerical Output

GPAW converges an electronic state, takes two geometry steps, writes a wavefunction checkpoint and resumes in a fresh process. In DeepSeek-V4-Flash run `Ykcn7Dy`, an x86-64-v3 build produces different output and checkpoint files. The agent first attributes the difference to vectorization and then builds with fused multiply–add disabled. For that build, the same file comparison that had reported differences now reports identical initial, geometry and checkpoint files, and identical resumed output. The agent revises its explanation, calls the measured speed gain marginal, and reports keeping its earlier build. Here diagnosis and selection diverge: the agent reports that the repaired change was not worth keeping, and the source tree does not independently confirm that choice.

L. Optimization Trajectories and Final Outcomes

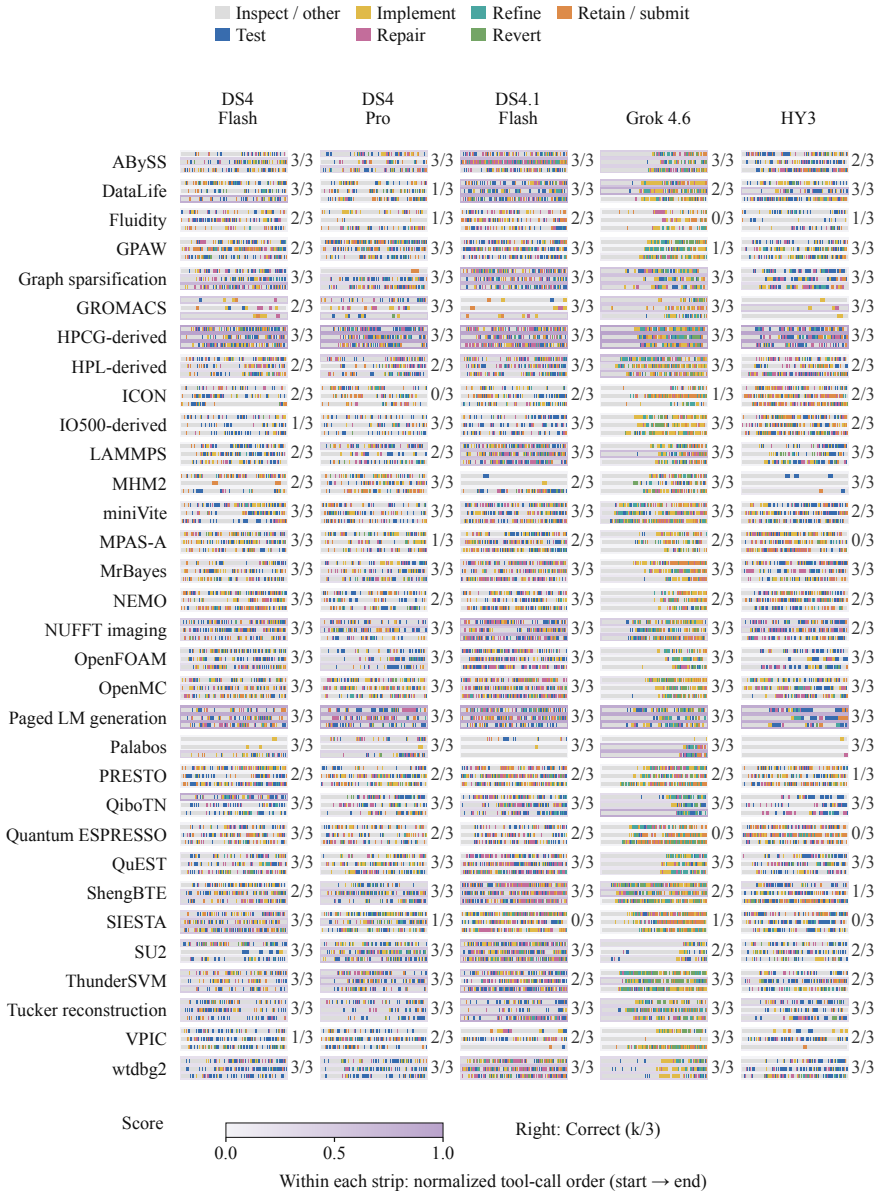


Figure 8 | Action sequences and outcomes across all 480 analysis runs. Each task–agent cell contains three runs, ordered by trial ID. Foreground colors show tool-action annotations; the pale background of each strip shows its final, correctness-gated runtime-saving score on a common 0–1 scale. A score of zero can also indicate a correct program without a runtime saving. Counts at right report correct runs out of three. Horizontal position shows recorded tool-call order, normalized separately for each run, not elapsed time; calls in the same recorded step may overlap. The sequences contain 203,472 tool actions; 417 statement-only annotations are not plotted.