PACE: MEASURING LLMS IN PERFORMANCE-CRITICAL CODE GENERATION UNDER AGENTIC WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code agents can edit files, invoke compilers, run tests, and revise their implementations. Previous works have evaluated speedup over reference implementations. This can surely help people understand how good a piece of code is, but it has a gap between the speedup number and hardware potential as it depends heavily on the quality of the reference for the workload. To further study resource utilization and understand the code quality generated by agents, a shift from workload-centered view to resource-centered view could help build an intuitive view. Therefore, we introduce PACE (**P**erformance-critical **A**gentic **C**ode **E**valuation), a CPU evaluation framework, with a metric called **E**ffective **C**alibrated **A**ttainment (ECA) built on measuring computing resource units. It measures how closely one bounded agent run approaches a calibrated performance reference for each task. We report a five-model baseline on x86-64 and AArch64 containing 3,000 runs. We also report a 135-run x86-64 pilot with nine conditions and DeepSeek-V4.1-Flash. From these assessments, we find that high pass rates can hide substantial performance gaps and that model rankings change across task tracks and hardware platforms. And a larger interaction budget alone may not improve the delivered code. The PACE repository can be accessed at `https://anonymous.4open.science/r/PACE-411F`

1 INTRODUCTION

Large language models can generate code for many programming tasks (Chen et al., 2021; Austin et al., 2021; Li et al., 2023). One-shot code generation was once a common test evaluation for models' coding capability, but they are required to give results immediately. Agentic-style coding frameworks enable these models to perform more complicated operation and deeper optimization in a feedback loop (Jimenez et al., 2023; Zhou et al., 2026; Yu et al., 2026; Li et al., 2025a; Zhang et al., 2025; Bhattacharjee et al., 2026). Such agents can repair syntax, resolve compiler errors, and test several implementations. To find out the quality of code that agents produce, benchmarks with properly designed performance metrics could help inspect into the quality of generated code (Qiu et al., 2024; Huang et al., 2024; Ouyang et al., 2025).

Currently, a number of benchmarks are workload-centered, which means that, to evaluate the quality of generated implementations, takes comparisons with reference implementations. (Huang et al., 2024; Ouyang et al., 2025; Mukunoki et al., 2025). The workload-centered position is a good standing point and can provide a speedup number. However, the speedup number itself can't provide sufficient information. Its meaning depends on the baseline. A $10\times$ improvement over a simple loop does not show that the new program uses the machine well. To understand the utilization of a generated program, we need a stable measure that is tailored to each task.

To better understand a coding agent's ability in utilizing machine, a shift of position, from workload-centered to resource-centered, could be an approach. Knowing the resource usage ratio would serve as a good benchmark for evaluating the quality of generated code. Resource-aware performance models relate workload demand to compute and memory limits (Williams et al., 2009). SOL-ExecBench (Lin et al., 2026) applies an analytically derived hardware Speed-of-Light bounds instead of relying only on software baselines. These are mainly GPU targeted, and finding a way to

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

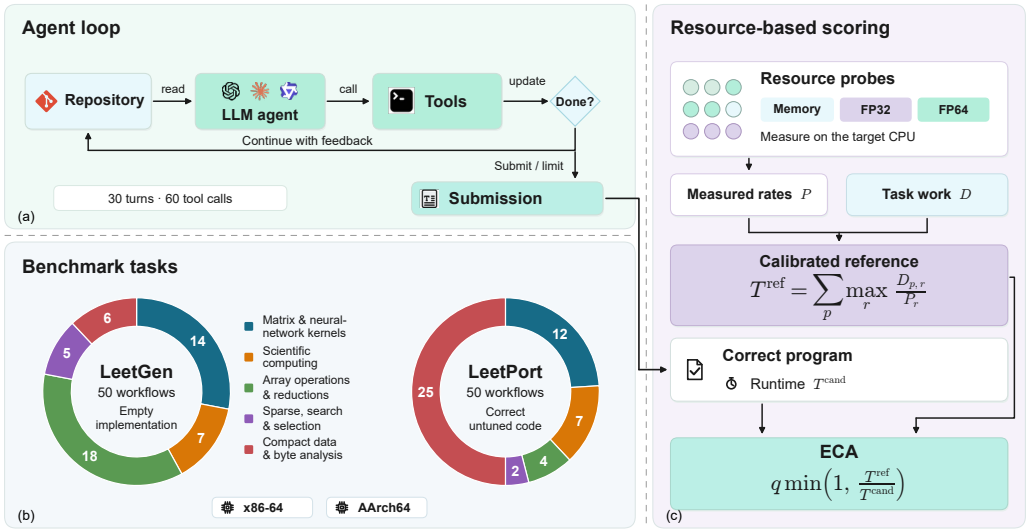

Figure 1: The PACE benchmark overview. (a) PACE agent framework. We use a unified mini agent framework as harness. (b) The PACE benchmark composition. These tasks could be divided into 5 main categories. (c) ECA compares a correct candidate with a task-specific point derived from required work and measured resource rates.

apply SOL in CPU-based platforms will provide significance in performance and bottleneck analysis. For this purpose, we develop PACE. Figure 1 illustrates the PACE workflow. It has two tracks. LEETGEN asks an agent to implement a kernel from its task contract. LEETPORT gives the agent a correct implementation and asks it to improve performance without changing the result. To some extent, these two paths represent two typical types of programming in the modern landscape; understanding the capabilities of agents in real-world contexts will be beneficial.

The main method is a calibrated resource reference for each task and a metric called Effective Calibrated Attainment(ECA). Each workflow records the required work and may split it into ordered steps. We measure the relevant hardware rates under the same core allocation, access pattern, and data size. We then compare the reference time with the candidate runtime. Incorrect programs receive zero attainment. The resulting ECA score measures both correctness and the performance of the program submitted from a resource-centered perspective.

We use this framework to compare agent and information conditions. Every agent receives the same file, compiler, and public-test tools. Some conditions also provide a versioned platform guide or a fast implementation of a related computation. For comparisons of reference material and access mechanisms, we hold the model, task, and interaction budget fixed. In the main study, ECA remained below 0.64 in every model–target–track cell, indicating a substantial gap in delivered performance relative to the calibrated references.

Our contributions are as follows.

- We introduce a CPU code benchmark with 100 workflows. Its two tracks test implementation and optimization. Architecture harnesses keep each task contract stable across x86-64 and AArch64.

- We define effective calibrated attainment for bounded code-agent runs. It combines correctness with a measured resource reference for each task. The reporting process also records coverage and failure types.

- We report a 3,000-run baseline with five models on two platforms. We also report a nine-condition pilot that studies the agent loop, reasoning, feedback budgets, and added reference material.

2 Related Work

Code generation and agentic evaluation. Code-generation benchmarks range from synthesis problems to repository tasks (Chen et al., 2021; Austin et al., 2021; Lai et al., 2023; Jimenez et al., 2023). While parallel-programming benchmarks evaluate OpenMP, MPI, and scientific kernels (Nichols et al., 2024; Chen et al., 2025; Valero-Lara et al., 2023). Agent-specific benchmarks closely resemble software development, since agents can do complex operations (Zhou et al., 2026; Xing et al., 2026).

Efficient kernel generation. EffiBench studies the runtime of generated solutions on algorithmic tasks (Huang et al., 2024). MakoraGenerate (Tehrani et al., 2026) finetunes GPT-5 to generate kernels efficiently. KernelBench and TritonBench compare generated GPU kernels with framework or compiler baselines (Ouyang et al., 2025; Li et al., 2025b). CuBridge (Ma et al., 2026) applies LLMs into kernel construction. MultiKernelBench extends kernel evaluation to more platforms (Wen et al., 2025). Prior work has also tested LLMs on BLAS implementation (Mukunoki et al., 2025). PACE focuses on CPU workflows. It measures how close an agent gets to a validated performance reference for each task.

Heterogeneous performance measurement. Benchmark suites such as HeteroBench and HosNa run programs on different types of systems (Tian et al., 2025; Bavarsad et al., 2021). The Roofline model relates arithmetic intensity to compute and memory ceilings (Williams et al., 2009). SpeedOfLight further quantifies hardware utilization on GPUs (Lin et al., 2026). Inspired by SOL, we attempt to further expand this idea on CPU-based platforms by proposing ECA.

Table 1 compares the protocols of several earlier benchmarks. Their inputs include short specifications, repository issues, and executable framework code. They compare performance with standard solutions, framework operators, or sequential programs. Their interaction protocols also differ. MBPP (Austin et al., 2021) reports a separate study with human feedback. KernelBench (Ouyang et al., 2025) includes a 10-call refinement setting. PCEBench (Chen et al., 2025) evaluates several generations and a separate repair task. Its agent loop is used to build and verify the dataset. The tested model does not use that loop. PACE evaluates a non-solution stub or correct untuned C code. The model works through a bounded tool-using run on x86-64 or AArch64. Correctness and ECA use a validated resource reference for each task.

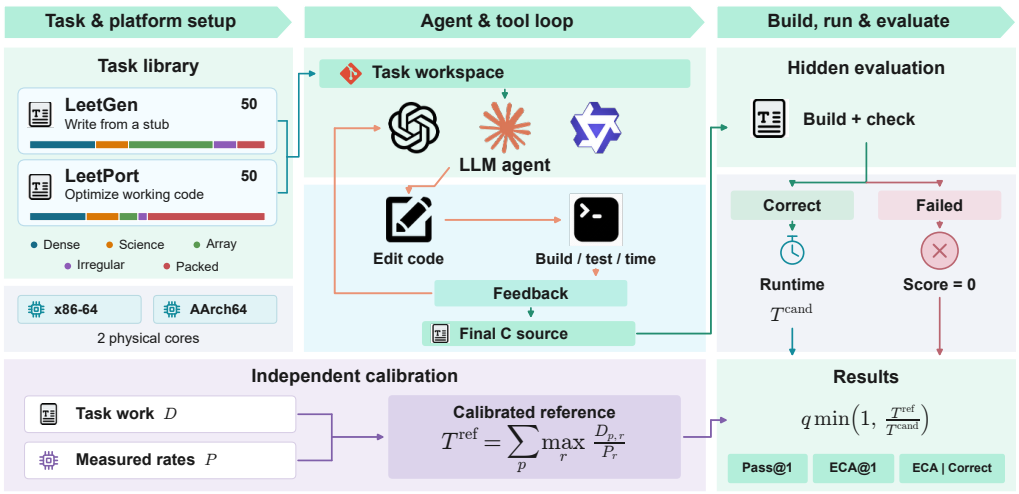

Figure 2: The PACE framework. PACE framework uses a mini agent to drive the workflow.

3 Problem Formulation

Let $t \in \mathcal{T}$ denote a workflow and $a \in \mathcal{A}$ a target architecture. Let $i \in \{1, \ldots, k\}$ denote an independent agent run. Each run receives a task workspace and a fixed tool interface. It also has

Table 1: Protocol comparison with representative code and kernel benchmarks. Each row reports the evaluated input, delivered target, executable oracle, performance comparator, and model interaction described in the original work.

Work	Starting input	Output / target	Correctness oracle	Performance comparison	Model interaction
HumanEval	Signature + docstring	Python function	Unit tests	None	Independent samples
MBPP	Description + three tests	Python function	Provided tests	None	Few-shot with a separate four-turn human-feedback study
SWE-bench	Issue + repository	Repository patch	Repository tests	None	Method-dependent with no required loop
EffiBench	Problem + examples	Python solution	Generated tests	Canonical runtime and memory	Few-shot with no feedback
KernelBench	PyTorch module + inputs	Optimized GPU module	Random differential tests	PyTorch eager / `torch.compile` and `fast_p`	One-shot and 10-call feedback variants
TritonBench	Operator specification / interface	Triton operator	Random differential tests	Reference operator and A100 peak efficiency	Zero- or one-shot
MultiKernel-Bench	PyTorch implementation	CUDA / AscendC / Pallas kernel	Random differential tests	PyTorch eager with `SpeedUp@k`	One-shot with $k = 1, 5$
PCEBench	Problem and signature with optional sequential or faulty code	OpenMP/MPI C/C++	Compile, run, output, and race checks	Thresholded sequential speedup with value unstated	$k = 1, 5, 10$ generation plus separate error-context repair
SOL-ExecBench	Problem specification / reference implementation	Torch/CUDA/GPU DSL	Standard reference implementation	SpeedOfLight value	Agentic
PACE (ours)	Empty function or correct untuned C	x86-64 / AArch64 CPU C	Hidden evaluator	Checked task and resource reference with ECA	Agentic

limits on model turns, tool calls, and response length. The run returns a candidate program $x_{t,a,i}$. Hidden evaluation first determines the correctness value $q_{t,a,i} \in \{0, 1\}$. It measures runtime $T^{\text{cand}}_{t,a,i}$ under a fixed hardware allocation when the candidate is correct. The evaluation has two goals. First, it measures the expected quality of one bounded run. Second, it separates wrong code from slow code. A build error or wrong answer receives zero effective attainment, while a correct slow program receives a small positive score.

4 Performance-critical Agentic Code Evaluation

4.1 The PACE Tasks

4.1.1 Two tracks

Figure 2 shows the end-to-end pipeline. PACE contains two tracks with different starting information. In LeetGen, the candidate region begins as a compilable non-solution stub. The agent receives the function contract, data layout, numerical requirements, build environment, and public development tests. This track measures whether the agent can implement the task and make it efficient.

In LeetPort, the candidate region contains working C code that has not been tuned. Leaving it unchanged gives a correct result and a $1.0\times$ speedup reference point. The agent must preserve the task contract as it improves the code. Possible changes include blocking, reuse, fusion, vectorization, parallel scheduling, and data-layout changes. This track focuses more directly on optimization because working code is already available.

Each track contains 50 core workflows, for 100 in total. The two collections have no one-to-one task mapping. They cover dense and sparse linear algebra, neural-network operators, scans, reductions, stencil and particle methods, attention, numerical recurrences, cryptographic operations, quantization, genomics, and compact data analysis.

4.1.2 CONSTRUCTION OF PACE

We build each workflow from a structured specification. It defines the function contract, arguments, test sizes, tolerance, input generator, and required work. The generator turns this specification into C code, build rules, a public test, and a separate hidden evaluator. It leaves a non-solution stub in LEETGEN and correct untuned code in LEETPORT. Construction used fixed generator scripts and generative-AI assistance. The assistance covered benchmark code, tests, validation code, and private witnesses. Every generated change had to pass the relevant build, correctness, wrong-code, and timing checks. Appendix B describes this process and the witness procedure.

4.1.3 STABLE TASKS ACROSS PLATFORMS

Each workflow has the same contract, shapes, tolerance, and required work on x86-64 and AArch64. Platform harnesses change the compiler setup, execution wrapper, and private witness. Candidates may use any supported instruction. The public task does not require a named instruction. We can compare the same workflow on both machines without changing its meaning.

4.1.4 VALIDATION AND TASK INCLUSION

A task enters the reported benchmark only after three checks. The evaluator must accept correct code and reject common shortcuts such as using only one visible size, dropping a vector tail, or writing only part of the output. The task must also show useful optimization potential. We use correct and faulty programs, unseen sizes and seeds, repeated calls with changed inputs, and a stable optimized witness. Evaluators create inputs from the task specification with fixed seeds. Integer arrays are compared element by element. Floating-point arrays use the fixed comparison rule in Appendix B.

Correctness checks and ECA checks are separate. A valid programming task must pass three more checks before it enters the ECA total. Its required work must be specified. The target machine's resource rates must be measured under the same allocation. A correct optimized implementation must also produce stable timings that agree with the reference. The released analysis records versions for the selected tasks, hidden-evaluator hashes, and calibrated references.

4.2 CALIBRATED REFERENCES FOR EACH TASK

4.2.1 REQUIRED WORK AND MEASURED HARDWARE RATES

A single peak compute number cannot represent programs with different resource needs. For each workflow t, we record an ordered set of steps $\mathcal{P}_t$. Step p has demand $D_{t,p,r}$ for each measured resource $r \in \mathcal{R}$. Table 7 lists the resources that limit at least one workflow on each platform and track.

The demand model comes from the task contract. The specification records symbolic work and its ordered steps. A dependency required by the contract creates a separate step. Terms within one step use the grouping rule below. Each term must map to a measured operation with matching precision, access pattern, data size, and CPU allocation. A workflow without such a match cannot receive an ECA score.

For architecture a, $P_{a,r}$ is the achieved rate of a resource probe. The probe exercises a defined operation set, such as FP32 FMA, integer MAC, `exp`, or memory copy. It uses the same physical-core allocation, precision, access pattern, and data-size range as the task. We retain repeated measurements and use rates that pass the stability checks. The grouping follows the contract and probe design. Operations that share an execution unit, bandwidth path, or serial limit are measured together. Distinct groups use the maximum within one step. Required dependencies add time across steps. The resulting reference is

$$T^{\mathrm{ref}}_{t,a} = \sum_{p \in \mathcal{P}_t} \max_{r \in \mathcal{R}} \frac{D_{t,p,r}}{P_{a,r}}. \tag{1}$$

For one step with only FLOP and byte demands, Equation 1 has the same structure as a compute and bandwidth Roofline. Other tasks can be limited by a special function, gather, or recurrence. Each

D/P term is the time implied by required work and the matched measured rate. We use T^{ref} as a calibrated comparison point for one task. It is not the runtime of a reference implementation, a vendor peak, a proven optimum, or a strict lower bound for every legal implementation.

All model cells for the same task and platform use the same probe value. Candidate profiling may explain slow code, but it cannot select an easier reference. This rule also applies when a candidate uses scalar instructions. During release review, we compare each step model with the contract and the validated witnesses. This can reveal work that a legal implementation avoids.

4.2.2 Effective calibrated attainment

For a correct candidate with a valid timing, raw calibrated attainment and its capped value are

$$R_{t,a,i} = \frac{T^{\mathrm{ref}}_{t,a}}{T^{\mathrm{cand}}_{t,a,i}}, \qquad C_{t,a,i} = \min(1, R_{t,a,i}). \tag{2}$$

R compares two times. $R = 0.5$ means that the candidate takes twice the calibrated reference time, and $R = 1$ means that it matches that time. $R > 1$ means that the candidate ran faster than the modeled comparison point. This can happen when the measured probe rate is conservative, when a legal implementation overlaps or fuses work more effectively than the task model, or when the demand, calibration, or timing needs further review. We retain the raw R for this diagnosis and cap the scored value at one. The cap prevents a comparison point that is too slow from giving extra credit or dominating the aggregate. Our main metric is effective calibrated attainment for one run. We use the established name $\mathrm{ECA@1}$.

$$\mathrm{ECA@1}(a) = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \frac{1}{k} \sum_{i=1}^{k} q_{t,a,i} C_{t,a,i}. \tag{3}$$

Build errors, wrong answers, timeouts, invalid candidates, and budget exhaustion set $q = 0$ and contribute zero. Correct but slow candidates keep their measured attainment, even when it is small. We first average the k runs within each workflow. We then give every workflow equal weight. A task family with more variants cannot dominate the score. ECA therefore combines delivery correctness with performance relative to fixed task-specific comparison points.

Measurement failures are handled separately from code failures. CPU or memory contention, unresolved per-call timing, and missing calibration make a measurement invalid. An $R > 1$ observation remains a valid candidate result. The score stays capped at one, and `ceiling_exceeded=true` records the event for reference review. A changed reference receives a new version. We then calculate every affected cell again from the same saved candidates. Summary tables report reference coverage, timing coverage, and counts above the reference.

We also report attainment among correct candidates, correctness for each run, final failure types, and counts above the reference. The pilot reports observed token and tool-call totals. These measures help explain $\mathrm{ECA@1}$. The main metric remains the score for delivered capability. Figure 3 demonstrates a situation where ECA reveals performance headroom that speedup does not show.

4.2.3 Calibrating and validating the reference

Calibration measures achieved resource rates. It does not use vendor peak figures. The probes match the task's physical-core allocation, data size, memory pattern, precision, vector or matrix facilities, and multicore scaling. We retain raw repetitions and machine-state records. Calibration supplies the measured rates in Eq. (1). We check an optimized expert or generated witness for each task. It must be correct. It must show room for useful speedup and help expose a comparison point that needs review, and the witness itself does not set the reference time.

The validation records list unsupported resources, invalid timings, witness attainment, and cases above the reference. Reliability applies to the complete result. It cannot be decided from only the attempt that exposed a problem. A run with an invalid reference stays marked for review until replay uses a corrected calibration. Appendix G lists the resources and validation checks.

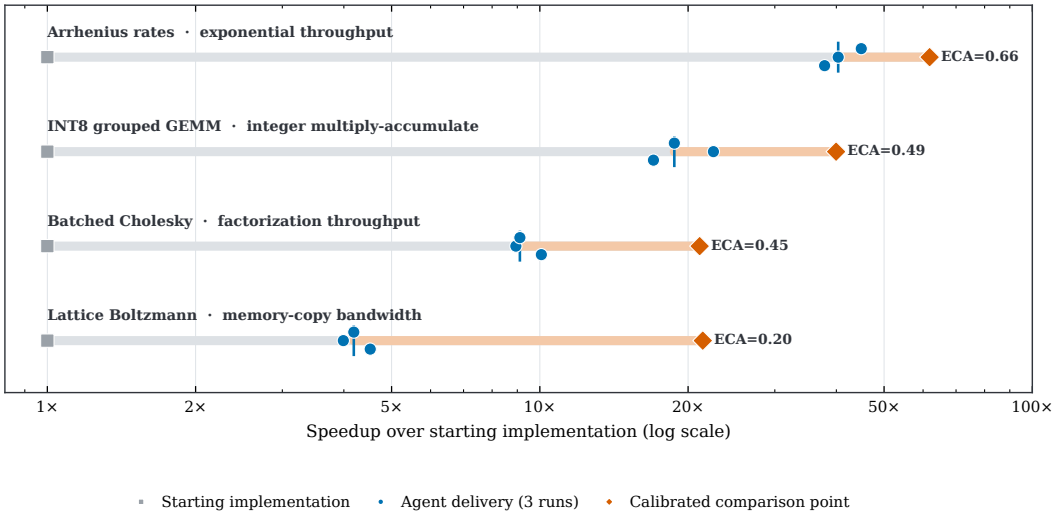

Figure 3: Using ECA to show remaining optimization headroom. The candidates improve substantially over the starting implementations, while their ECA values remain below one. The comparison points are based on required work and measured resource rates, not on hardware upper bounds.

5 Experimental Design

Platforms and models. The linked generation, timing, and calibration records give the configuration for each reported cell. They include the model, compiler, CPU, architecture, allocation, and reference. We run on an AMD Ryzen 7 7800X3D with x86-64 and GCC 11.4. We also use an Arm Neoverse N2 with AArch64 and GCC 15.2. Candidate harnesses use `-O3 -mavx2 -mfma -fopenmp` on x86-64. They use `-O3 -mcpu=native -fopenmp` on AArch64. Both include alignment and warning flags. Calibration probes use their own recorded flags for each resource. Each measurement runs on an assigned pair of physical cores. We test `gpt-5.6-sol`, `gpt-5.6-terra`, `claude-opus-5`, `grok-4.6`, and `gemini-3.8-flash` on both tracks and targets. Profiles use enabled or high reasoning when supported.

Protocol. The baseline uses $k = 3$ independent runs. Temperature is 0.2 when accepted. Each run allows at most 30 turns and 60 tool calls, with a common output cap per response. Each job receives two physical cores. We archive raw interaction records and intermediate failures. The scored set keeps the final outcome of every productive run. These outcomes include build errors, wrong answers, and missing submissions.

Statistics. All metrics give each workflow equal weight. The baseline has 20 complete cells. Each cell contains 50 workflows and 150 runs, giving 3,000 runs in total. Correctness, ECA@1, and pilot comparisons use bootstrap intervals over workflows. The primary score gives failures zero. Attainment among correct candidates, final failure types, and counts above the reference help explain the score gap. A real-run case is available at Appendix E

6 Results and Analysis

All numbers in this section come from named and versioned records. These records also include the platform checks. Each full-core cell contains 50 workflows and three runs per workflow. Every task reference passes its checks, and every required timing is available. Incorrect programs need no candidate timing and contribute zero.

6.1 Baseline attainment

Table 2 reports attainment that includes correctness. Confidence intervals resample workflows after averaging their three runs. Model ordering can change across tracks and architectures. For example,

Table 2: Full-core results by target and track. ECA includes a 95% bootstrap interval over workflows. ECA|C is attainment among correct candidates. $R{>}1$ counts valid candidates that are faster than the calibrated comparison point before the score is capped. It does not count violations of a hardware limit. Every cell has 150/150 resolved attempts and 50/50 workflows that pass the reference checks. Every required candidate timing is available, so coverage is 1.000. Within each target and track group, bold marks the best value and underline marks the worst. The $R{>}1$ count is reported for review and is not ranked.

Model	Pass@1	ECA|C	ECA [95% CI]	$R{>}1$
x86-64 — LeetGen				
GPT-5.6-Sol	**0.993**	**0.504**	**0.500 [0.440,0.561]**	2
GPT-5.6-Terra	0.980	0.368	0.363 [0.299,0.429]	1
Claude Opus 5	**0.993**	0.485	0.483 [0.431,0.537]	0
Grok 4.6	**0.993**	0.492	0.489 [0.434,0.545]	1
Gemini 3.8 Flash	0.573	0.412	0.243 [0.185,0.304]	0
x86-64 — LeetPort				
GPT-5.6-Sol	0.827	0.518	0.424 [0.373,0.483]	9
GPT-5.6-Terra	0.787	0.433	0.335 [0.284,0.387]	4
Claude Opus 5	0.807	0.359	0.287 [0.244,0.336]	4
Grok 4.6	0.940	**0.606**	**0.574 [0.515,0.631]**	17
Gemini 3.8 Flash	**0.980**	0.306	0.300 [0.252,0.347]	3
AArch64 — LeetGen				
GPT-5.6-Sol	0.993	**0.637**	**0.636 [0.565,0.704]**	10
GPT-5.6-Terra	0.973	0.443	0.435 [0.355,0.515]	8
Claude Opus 5	**1.000**	0.563	0.563 [0.493,0.633]	7
Grok 4.6	0.987	0.600	0.591 [0.524,0.657]	7
Gemini 3.8 Flash	0.740	0.477	0.380 [0.296,0.472]	6
AArch64 — LeetPort				
GPT-5.6-Sol	0.773	0.447	0.366 [0.300,0.434]	7
GPT-5.6-Terra	0.827	0.458	0.380 [0.323,0.438]	5
Claude Opus 5	**0.953**	**0.630**	**0.599 [0.522,0.675]**	13
Grok 4.6	0.933	0.542	0.501 [0.436,0.567]	10
Gemini 3.8 Flash	0.900	0.353	0.321 [0.256,0.389]	7

GPT-5.6-Sol has the highest AArch64 LeetGen point estimate at 0.636. Claude Opus 5 has the highest AArch64 LeetPort estimate at 0.599.

6.2 Generation versus optimization

LeetPort provides simple correctness fallbacks. Leaving the supplied code unchanged gives a correct program and a $1.0\times$ speedup. The task still asks the agent to improve performance, and evaluation uses the final edited program. A lower LeetPort Pass@1 means that the final edit broke correctness. Correct starting implementations were still supplied. The pattern depends on the model. Gemini's x86-64 Pass@1 rises from .573 on LeetGen to .980 on LeetPort. Opus falls from .993 to .807. Their ECA values are .243 versus .300 and .483 versus .287, (Figure 4).

For a complete cell, we can split the attainment gap as $1 - \mathrm{ECA} = (1 - \mathrm{Pass@1}) + (\mathrm{Pass@1} - \mathrm{ECA})$. For x86-64 Grok on LeetGen, correctness loss accounts for only .007 of the .511 gap. The remaining performance loss accounts for 98.7%. For Gemini on x86-64 LeetGen, correctness accounts for .427 and performance accounts for .330. The two tracks contain different tasks, so their difference is descriptive. Across both targets and five models, LeetGen delivers 1,384 correct programs in 1,500 runs, or 92.3%. LeetPort delivers 1,309, or 87.3%. The failure mix agrees with unsuccessful edits on the optimization track.

Figure 5 in appendix shows similar failure rates across the three run indices. Across all 20 full-core cells, runs 1, 2, and 3 deliver 897, 906, and 890 correct programs out of 1,000. Failure types differ more clearly by track. Across both targets, LeetGen has 103 wrong answers and 13 build errors. LeetPort has 94 build errors, 80 wrong answers, and 17 run or parse errors. These counts support treating the three runs as repeated samples.

6.3 Resource classes

Reference validation passes for all 50 workflows in each track on x86-64 and AArch64. Calibration uses the same two-core allocation as candidate replay. It covers multicore scaling, native vector

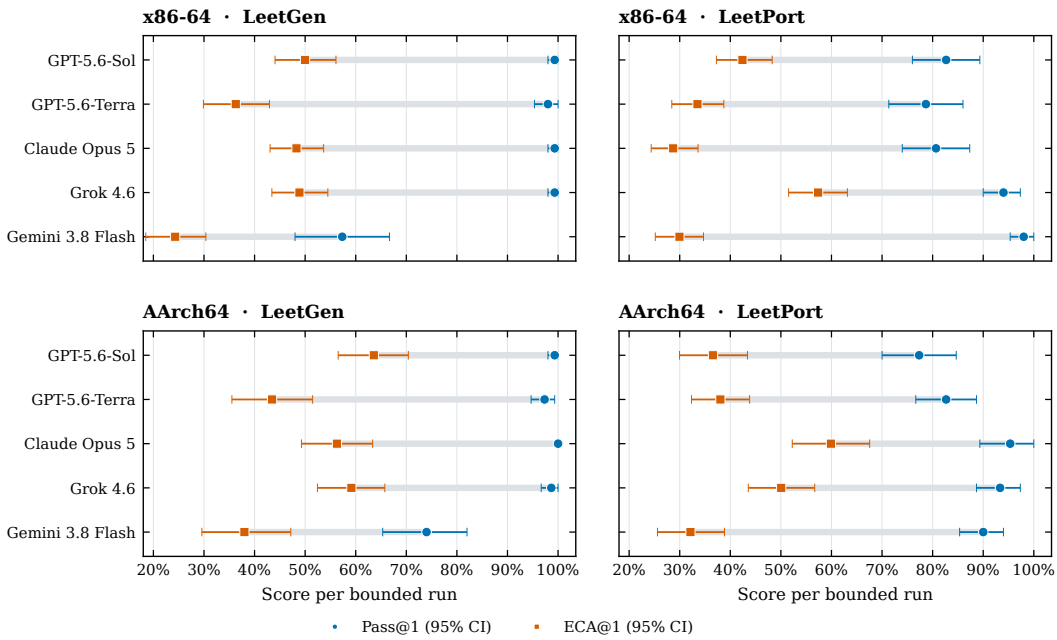

Figure 4: Correctness and delivered calibrated attainment. Each line connects a model's Pass@1 to its ECA. Horizontal bars are 95% bootstrap intervals over workflows for both metrics. The gap shows the remaining performance loss that pass rate alone does not measure.

facilities, low-precision and integer paths, special functions, memory patterns, data movement, reductions, and dependent steps. On x86-64, LEETGEN is limited by 21 distinct measured resources and LEETPORT by 13. The AArch64 counts are 16 and 13. Memory reads are the most common limit. The counts are 16/50 and 19/50 on x86-64, 20/50 and 19/50 on AArch64. The limiting resource can change across platforms, and each cell receives its own calibrated reference time.

Table 2 keeps all observations with $R > 1$. There are 4 on x86-64 LEETGEN, 37 on x86-64 LEETPORT, 38 on AArch64 LEETGEN, and 42 on AArch64 LEETPORT. This is 121 of 3,000 runs. Each contributes one after the score cap. These candidates ran faster than the model-based comparison point, not a claimed hardware upper bound. We retain the count because it shows where the reference or its inputs need further review. None creates a coverage gap.

6.4 PILOT AGENT AND INFORMATION COMPARISONS

Table 8 and Figure 6 in Appendix report the DeepSeek-V4.1-Flash study on `nbody_large`, `tridiag_large`, and `int8_grouped_gemm_large`. Every condition from baseline B0 through A1-A8 contains five independent runs per workflow. The pilot has 135 runs. Every correct candidate has independent timing evidence on two cores. All use the same fixed x86-64 calibration. The result indicates that the cut in interaction budget, disabling thinking and use one-shot affect the performance toward degradation by the most.

7 CONCLUSION

PACE evaluates the final program delivered by a bounded code-agent run, with at most 30 turns and 60 tool calls. Across the 3,000 runs in the complete study, every trajectory cell remained below 0.64 ECA, measured against its calibrated comparison point. The gap indicates that correct programs still leave room for better resource use. As for the pilot experiment, ECA helps expose factors that cause performance degradation. At the same time, the full study produced 1,384 correct LeetGen programs and 1,309 correct LeetPort programs out of 1,500 runs per track. These results show that bounded coding agents can produce working kernel programs, while there remains significant room for optimization regarding resource-aware refinement.

AI USE STATEMENT

Generative AI tools assisted manuscript drafting, benchmark implementation, and execution. In particular, `gpt-5.6-sol` with `xhigh` reasoning helped implement private performance witnesses. Generated changes were treated as untrusted and were accepted only after the applicable automated checks. The authors chose the final protocol and remain responsible for the text, code, analyses, and released files.

ETHICS STATEMENT

This study does not involve human participants, personal data, or user-generated content. It evaluates code-generation models through synthetic CPU programming tasks. Model-generated programs are untrusted and may contain unsafe behavior. We therefore build and run them in separate attempt workspaces on designated benchmark machines. The released artifact excludes API keys, service logs, host details, and other credentials. The experiments also use compute and paid model inference. We report the bounded run settings so that readers can judge this cost alongside the reported results.

REPRODUCIBILITY STATEMENT

The accompanying artifact contains the PACE runner, both 50-task datasets, x86-64 and AArch64 harnesses, scoring code, calibrated references, and the final program from every reported run. It includes 3,000 main-study programs and 135 pilot programs in a minimal replay format. The package provides scripts for dataset checks, generation, replay, calibration, ECA derivation, and result verification. Replaying the released programs does not call a model API. Exact timings depend on the processor, compiler, core allocation, and system load, so the released references are tied to the evaluated machines. A new machine should be calibrated before its ECA results are compared. Anonymous repository is supplied at `https://anonymous.4open.science/r/PACE-411F`

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Najmeh Nazari Bavarsad, Hosein Mohammadi Makrani, Hossein Sayadi, Lawrence Landis, Setareh Rafatirad, and Houman Homayoun. Hosna: A dpc++ benchmark suite for heterogeneous architectures. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pp. 509–516. IEEE, 2021.

Arijit Bhattacharjee, Heng Ping, Son Vu Le, Paul Bogdan, Nesreen K Ahmed, and Ali Jannesari. Optiml: An end-to-end framework for program synthesis and cuda kernel optimization. *arXiv preprint arXiv:2602.12305*, 2026.

Le Chen, Nesreen Ahmed, Mihai Capotă, Ted Willke, Niranjan Hasabnis, and Ali Jannesari. Pcebench: A multi-dimensional benchmark for evaluating large language models in parallel code generation. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 546–557. IEEE, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Haonan Li, Keyu Man, Partha Kanuparthy, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. Tritonforge: Profiling-guided framework for automated triton kernel optimization. *arXiv preprint arXiv:2512.09196*, 2025a.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, 2025b.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

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

Xing Ma, Yangjie Zhou, Wu Sun, Zihan Liu, Jingwen Leng, Yun Lin, Shixuan Sun, Minyi Guo, and Jin Song Dong. Cubridge: An llm-based framework for understanding and reconstructing high-performance attention kernels. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10929–10946, 2026.

Daichi Mukunoki, Shun-ichiro Hayashi, Tetsuya Hoshino, and Takahiro Katagiri. Performance evaluation of general purpose large language models for basic linear algebra subprograms code generation. *arXiv preprint arXiv:2507.04697*, 2025.

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*, pp. 281–294, 2024.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647*, 2024.

Ali Tehrani, Yahya Emara, Essam Wissam, Wojciech Paluch, Waleed Atallah, Mohamed S Abdelfattah, et al. Fine-tuning gpt-5 for gpu kernel generation. *arXiv preprint arXiv:2602.11000*, 2026.

Hongzheng Tian, Alok Mishra, Zhiheng Chen, Rolando P Hong Enriquez, Dejan Milojicic, Eitan Frachtenberg, and Sitao Huang. Heterobench: Multi-kernel benchmarks for heterogeneous systems. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, pp. 320–333, 2025.

Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. Comparing llama-2 and gpt-3 llms for hpc kernels generation. In *International Workshop on Languages and Compilers for Parallel Computing*, pp. 20–32. Springer, 2023.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv e-prints, pp. arXiv–2507*, 2025.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie F Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, et al. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *Proceedings of Machine Learning and Systems*, 8:2016–2064, 2026.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, et al. Towards automated kernel generation in the era of llms. *arXiv preprint arXiv:2601.15727*, 2026.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, et al. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026.

PACE: Measuring LLMs in Performance-critical Code Generation Under Agentic Workflows

TABLE OF CONTENTS IN APPENDIX

A Limitations **14**

B Benchmark Construction **14**

B.1 Tool-Using Evaluation Framework 15

B.1.1 Bounded agent loop 15

B.1.2 Resource allocation and remote execution 15

B.2 Two released task examples 15

B.3 Where the tasks come from and complete core list 16

B.4 Per-task data, tolerance, and demand basis 17

C Agent and Tool Protocol **23**

D Pilot Agent and Information Ablations **24**

E Example of an iterative optimization trajectory **24**

F Portability **25**

G Resource Model and Reference Validation **25**

A LIMITATIONS

ECA@1 measures attainment relative to calibrated comparison points. It does not express a percentage of a theoretical hardware limit. Its meaning depends on accurate resource probes and sound assumptions about required work, overlap, and ordered steps. A legal algorithmic change can run faster than the modeled reference. We therefore cap the score while retaining raw $R > 1$ values for review. The cap also removes performance differences among candidates above the comparison point. Calibration and timing checks also increase the work needed to add a new architecture.

The full-core experiments use two physical cores on x86-64 and AArch64. They do not establish results for GPUs, distributed programs, or large NUMA systems. Model releases may change training data or weights behind a stable model name. The completed pilot covers one model and only three LeetPort workflows. We selected these workflows after screening for solvability and moderate room for useful speedup. The pilot cannot establish an effect across the full benchmark or across models. Its bootstrap intervals over workflows are also coarse.

The current interaction budget caps turns, tool calls, and each response. It does not cap total tokens over a full run. The runner uses separate working directories, initial CPU affinity, and hidden-file removal.

B BENCHMARK CONSTRUCTION

The benchmark is generated from structured task specifications. It is not a collection of copied applications. We used the following process for both tracks.

1. Specify the task. Each specification defines the function, arguments, input generator, visible and hidden sizes, numerical tolerance, task family, and minimum required work. In LEETGEN, it produces a compilable non-solution stub. In LEETPORT, it produces a correct but untuned C implementation.

The development shapes are visible to the agent. The timed performance shape is fixed separately and is used only after submission. Hidden correctness shapes follow fixed rules. At least one is larger than the largest development case. At least one legal extent is not aligned to a usual vector width. Exact hidden dimensions remain in the hidden evaluator. Tables 4 and 5 report the development and timed shapes.

2. Generate the task package. A fixed generator script emits the header, candidate file, correctness reference, Makefile, public development test, hidden evaluator, and metadata. The x86-64 and AArch64 packages keep the same task meaning and test data. Only the platform setup and private performance code differ.

3. Build a private witness. A private witness is an optimized implementation used only to build and check the benchmark. We used `gpt-5.6-sol` with `xhigh` reasoning as an implementation assistant. It received the task specification and platform information. It proposed witness code and revised the code after compiler, test, and timing feedback. We treated every proposal as untrusted. It had to build and pass the available correctness and timing checks. The evaluated agent never receives the witness in its workspace.

The witness shows that a task has room for useful speedup. It also checks whether the calibrated reference agrees with the task. It does not set the ECA reference time. The released score reads required work from the versioned task specification. Resource rates come from separate hardware probes. If a valid witness exposes a weak reference, we review and correct that reference. We then calculate every model cell again with the corrected version.

4. Check each task. The checks have direct purposes.

- A correct implementation must pass. This confirms that the task and evaluator agree on the intended answer.

- Deliberately wrong implementations must fail. This confirms that the evaluator detects incorrect output.
- Hidden sizes and seeds differ from the public test. This prevents a solution from passing by memorizing one example or one vector width.
- Repeated calls with changed inputs check partial output and state left from an earlier call. The current evaluators have no general memory-safety check around allocated buffers.
- Timings must be stable. The witness must also beat simple compiler or parallel-loop baselines. This confirms that the task has room for useful speedup and is not dominated by timing noise.

Every hidden evaluator uses correctness seeds 9781, 9798, 9815, and 9832. The performance input uses seed 4242. Integer arrays require elementwise equality. For floating-point outputs with tolerance $\tau > 0$, the evaluator rejects a non-finite candidate value and accepts an element only when $|\hat{y} - y| \leq \tau \max(1, |y|)$. The active nonzero tolerances are 10^{-6}, 10^{-5}, 2×10^{-5}, 10^{-4}, 10^{-3}, and 2×10^{-3}. Zero-tolerance floating-point tasks use the existing absolute-difference comparator with epsilon zero.

Tolerance is part of the task contract. It is not inferred from model output. Exact integer and packed-record transforms use zero. For floating-point tasks, we keep a tolerance only when the correct implementation passes development and hidden cases. Deliberately wrong controls must still fail. The tolerance is a tested acceptance threshold for the stated input generator. It is not a general error bound for every legal floating-point implementation.

5. Select and version the core. We select 50 workflows per track. The selection covers different task families, data types, access patterns, and resource needs. The released version records the task list, hidden-evaluator hashes, demand model, platform calibration, and witness checks. Some candidates predate the latest calibration review. For this reason, the release recalculates all results with one reference version.

B.1 Tool-Using Evaluation Framework

B.1.1 Bounded agent loop

For every run, the framework creates a separate working directory. It gives the agent structured file, build, and public-test tools plus a general shell. Installed shell utilities depend on the host. Outbound network access remains available. The model may alternate between reasoning, tool calls, and code changes until it submits or reaches its budget. The final program is then rebuilt and evaluated by a hidden driver. Hidden tests and private witnesses are removed from the working directory during the agent turn.

B.1.2 Resource allocation and remote execution

Performance feedback requires a consistent CPU allocation. The framework reads the CPU topology on the local or remote machine. It removes duplicate hardware threads that share one physical core. It then gives concurrent jobs separate sets of physical cores and sets their initial CPU affinity. The same launch policy applies to profiling commands, candidate execution, hidden tests, and hardware calibration.

For remote targets, the framework copies a separate attempt directory to the target machine. Build, execution, and calibration run there. Model inference stays on the host. Several linked records provide the information needed to trace a result. They cover generation, timing, and score calculation. Together, they record candidate sources, model settings, CPU allocation, available machine and compiler facts, and the calibration version.

B.2 Two released task examples

The following examples come directly from released contracts. They show how the signature, data policy, sizes, tolerance, and demand record fit together.

LeetGen example. The task is `welford_var_large`. The agent receives a compilable stub that initializes both outputs to zero and marks the solution region as unimplemented.

```
void welford_var_gen(float *x, float *mean_out,
                     float *var_out, int n);
```

The required outputs are the population mean and variance. The fixed input policy draws each value as $10{,}000 + U[0, 1]$. This makes a naive FP32 $E[x^2] - E[x]^2$ implementation inaccurate. The visible development shape is $n = 1{,}048{,}576$, the timed shape is $n = 4{,}194{,}304$, and $\tau = 10^{-3}$ under the comparison rule above. The released demand is `flop`=$3n$, `byte`=$4n + 8$, and `reduction`=$2n$. This demand covers the stable first and second moments. It does not require a particular Welford recurrence. A two-pass reduction with a shifted origin is also legal.

LeetPort example. The task is `tridiag_large`. The agent starts from a correct untuned Thomas solver with this signature.

```
void tridiag_gen(float *a, float *b, float *c, float *d,
                 float *x, float *cp, int n, int m);
```

It solves m independent, diagonally dominant systems stored level-major. Entries of a and c are drawn from $U[0, 0.4]$, b from $U[2.5, 3.5]$, and d from $U[0, 1]$. The development shape is $(n, m) = (128, 8192)$, the timed shape is $(128, 16384)$, and the tolerance is 10^{-5}. Its demand is `flop`=$10nm$, `byte`=$24nm$, and `div`=nm. The division count uses one pivot reciprocal per element. Both forward-sweep outputs share that reciprocal. Charging the two divisions in the untuned code would describe only the starting implementation. It would overstate the work required by the task.

B.3 Where the tasks come from and complete core list

This section explains where each mathematical definition or computing pattern comes from. A standard named operation uses an established algorithmic name. An application pattern turns a familiar workload into a small kernel. A published standard supplies a fixed definition. A task defined by PACE uses its own packed-record output contract. PACE's generators emit every released C task package from these structured specifications. Table 3 lists all 100 core task IDs and the basis of each definition.

Table 3: Where the 100 core task definitions come from. Task IDs include the selected size class. The definition basis describes mathematical background, not copied source code.

Track	Family	Definition basis	Core task IDs
LeetGen	BLAS-like and textbook matrix operations	Standard named operation	`gemv_large`, `gemm_transb_large`, `syrk_large`, `symv_large`, `triu_copy_large`, `matrix_norm_large`, `block_sparse_gemm_large`, `batched_matmul_small_large`
LeetGen	Neural-network and differentiable operators	Standard named operation	`gelu_large`, `silu_large`, `leaky_relu_large`, `mish_large`, `rsqrt_large`, `swiglu_large`, `rope_large`, `conv2d_large`, `avg_pool2d_large`, `depthwise_conv2d_large`, `group_norm_large`, `cross_entropy_large`, `bias_gelu_large`, `layernorm_backward_large`, `conv1d_transpose_large`, `logsumexp_2d_large`
LeetGen	Reductions, scans, and numerical recurrences	Standard named operation	`cumsum_large`, `argmax_large`, `l2_norm_large`, `mean_var_large`, `minmax_large`, `count_nonzero_large`, `row_sum_large`, `col_sum_large`, `cumprod_large`, `cummax_large`, `welford_var_large`, `prefix_max_2d_large`, `segmented_sum_large`, `kahan_sum_large`
LeetGen	Data movement, selection, and ordering	Standard named operation	`transpose_tiled_large`, `roll_large`, `slice_strided_large`, `unique_sorted_large`, `compact_nonzero_large`, `bitonic_topk_large`
LeetGen	Exact packed-record analytics	Defined by PACE	`gen_bitmap_relation_stats_large`, `gen_byte_pair_order_stats_large`, `gen_packed_byte_partition_stats_large`, `gen_packed_dibit_transition_stats_large`, `gen_packed_ternary_histogram_large`, `gen_paired_nibble_relation_stats_large`
LeetPort	Scientific-computing motifs	Application pattern	`lbm_large`, `nbody_large`, `arrhenius_large`, `tridiag_large`, `stencil_time_block_large`, `stencil_tile_halo_large`, `block_jacobi_large`

Continued on next page.

Table 3, continued.

Track	Family	Definition basis	Core task IDs
LeetPort	Matrix, tensor, attention, and quantized kernels	Standard operation / application pattern	ragged_attn_large, bf16_gemv_ragged_small, block_gemm_ragged_large, tensor_contract_pack_large, int8_grouped_gemm_large, batch_chol_mixed_large, quantized_depthwise_large, sparse_block_gemm_large, tile_transpose_reduce_large, tensor_pack_contract_large, nearest_centroid_large, pack_gemm_odd_large, bf16_pair_gemm_small
LeetPort	Data-system and search motifs	Application pattern	block_filter_compact_large, weighted_histogram_large, json_structural_scan_large, histogram_prefix_large, sad_descriptor_search_large
LeetPort	Standard cryptographic transforms	Published standard	sha256_blocks_large, ghash_batch_small
LeetPort	Exact packed-record analytics	Defined by PACE	bitmap_set_stats_small, packed_dna_substitution_stats_small, packed_genotype_stats_large, packed_quality_extreme_stats_large, packed_label_confusion_stats_small, packed_dna_run_stats_large, packed_dna_composition_stats_large, masked_byte_sum_stats_small, packed_ascii_class_stats_small, packed_nibble_parity_stats_small, packed_ternary_histogram_large, packed_dna_motif_stats_large, bitmap_run_stats_large, packed_nibble_order_stats_small, packed_3bit_range_stats_large, packed_5bit_category_stats_large, packed_6bit_zone_stats_large, packed_7bit_parity_stats_large, packed_byte_partition_stats_large, bitmap_relation_stats_large, packed_dibit_transition_stats_large, paired_nibble_relation_stats_large, byte_pair_order_stats_large

B.4 PER-TASK DATA, TOLERANCE, AND DEMAND BASIS

Tables 4 and 5 summarize every released task. A slash separates development cases. The arrow points to the single timed case. Input entries use names from the fixed generator. `uniform` means $U[0, 1]$. `scale:`l`:`h means $U[l, h]$. Other names mark structured inputs such as valid sparse indices, runs, offsets, or positive-definite matrices. The benchmark retains the exact generator code and the same fields in machine-readable form. Demand expressions use the dimensions shown in the shape column. They record required work and a memory-access class. The last column explains why each charge follows from the task contract. It also states when no separate slow operation is charged.

Table 4: Per-task LEETGEN specification. Shape shows development → timed performance dimensions. τ is the correctness tolerance.

Task	Shape	Inputs	τ	Demand	Demand basis
gelu_large	n=1048576 → n=4194304	x:signed	10^{-5}	memory:copy; byte=2*4*n; gelu_transform=n	Each output requires one tolerance-matched GELU transform. The complete transform admits the legal central-domain polynomial without charging a removable libm erf.
silu_large	n=1048576 → n=4194304	x:signed	10^{-5}	memory:copy; byte=2*4*n; silu_transform=n	Each output requires one SiLU transform, calibrated as a complete bounded operation so polynomial exp and reciprocal refinement remain legal implementation choices.
leaky_relu_large	n=1048576 → n=4194304	x:signed	10^{-6}	memory:map; flop=n; byte=2*4*n; compare=n	Every element must be classified by sign. One multiply per element is a conservative lower bound for the branchless max(x, alpha*x) form; a comparison is not also counted as a FLOP.
cumsum_large	n=1048576 → n=4194304	x:uniform	10^{-3}	memory:copy; byte=2*4*n; scan=n	Every prefix value depends on the preceding prefix; the n-step recurrence cannot be removed while preserving all outputs.
argmax_large	n=1048576 → n=4194304	x:signed	0 (exact)	memory:read; byte=4*n; compare=n; reduction=n	The winner and lowest-index tie rule require inspecting every input and resolving a reduction.
l2_norm_large	n=1048576 → n=4194304	x:signed	10^{-3}	memory:read; flop=2*n; byte=4*n; sqrt=1; reduction=n	Every input contributes one square and one reduction term. The 1e-3 contract explicitly admits FP32 accumulation, followed by one FP32 square root.
mean_var_large	n=1048576 → n=4194304	x:uniform	10^{-3}	memory:read; flop=3*n; byte=4*n+8; reduction=2*n	Mean and second moment inspect every input. The tolerance admits FP32 moments, and a single reciprocal of n can replace both fixed-denominator divisions.
gemv_large	M=2048, N=2048 → M=4096, N=4096	A:uniform; x:uniform	10^{-4}	memory:read; flop=2*M*N; byte=4*(M*N+N+M); reduction=M*N	Each output is an N-term dot product. A direct FP32 accumulation passes every held-out shape at the 1e-4 contract, so reference widening is not semantic demand.

Continued on next page.

Table 4, continued.

Task	Shape	Inputs	τ	Demand	Demand basis
syrk_large	M=512, K=512 → M=1024, K=1024	A:uniform	10^{-4}	memory:copy; flop=M*(M+1)*K; byte=4*(M*K+M*(M+1)/2); reduction=M*(M+1)*K/2	Exactly M*(M+1)/2 lower-triangular outputs consume K-term products. The accepted witness establishes that FP32 vector accumulation satisfies the observable tolerance.
gen_packed_byte_partition_stats_large	nrec=4194301 → nrec=4194301	values:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec; packed_byte_stats=nrec	Every 64-byte record requires one exact partition-count and payload-sum transform; extraction, masks and horizontal sums may be fused by the target ISA.
conv2d_large	H=1024, W=1024, KS=5 → H=2048, W=2048, KS=5	src:uniform; k:signed	10^{-4}	memory:triad; flop=2*(H-KS+1)*(W-KS+1)*KS*KS; byte=4*(H*W+KS*KS+(H-KS+1)*(W-KS+1)); reduction=(H-KS+1)*(W-KS+1)*KS*KS	Every output consumes the complete KS by KS receptive field; the accepted SIMD witness demonstrates that FP32 accumulation meets the task tolerance.
avg_pool2d_large	H=1024, W=1024, PS=4 → H=2048, W=2048, PS=4	src:uniform	10^{-5}	memory:read; byte=4*(H*W+H*W/(PS*PS)); reduction=H*W	The non-overlapping windows consume every input once. The complete reduction demand covers their additions; charging the same additions as FMA peak work would duplicate them, and one reciprocal multiply per output is non-binding.
swiglu_large	n=1048576 → n=4194304	a:signed; b:signed	10^{-5}	memory:triad; flop=n; byte=3*4*n; silu_transform=n	Every output requires one complete SiLU gate and one multiply by the second input. Exp and division are not separately mandatory spellings.
rope_large	S=2048, D=128 → S=4096, D=128	x:signed	10^{-4}	memory:copy; flop=3*S*D; byte=2*4*S*D	Each pair needs a six-FLOP complex rotation. The harness explicitly amortises shape-dependent setup, so sine/cosine coefficients may be cached across timed calls and cannot be charged to every invocation.
gen_bitmap_relation_stats_large	nrec=4194301 → nrec=4194301	left:u64; right:u64	0 (exact)	memory:read; byte=136*nrec; working_set=136*nrec; bitmap_relation=nrec	Both 512-bit streams feed one exact intersection/difference record transform; logical operations and population counts are calibrated together so wide fusion remains legal.
gen_packed_dibit_transition_stats_large	nrec=4194301 → nrec=4194301	symbols:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec; dibit_transition=nrec	Every record requires the exact code-3 and adjacent-transition totals, including word boundaries; the complete transform admits packed bit-plane algorithms.
gen_packed_ternary_histogram_large	nrec=4194301 → nrec=4194301	codes:u64	0 (exact)	memory:read; byte=80*nrec; working_set=80*nrec; ternary_histogram=nrec	Every record emits one exact four-bin dibit histogram. Packed bit-plane counts replace removable per-symbol extraction and scatter operations.
mish_large	n=1048576 → n=4194304	x:signed	10^{-5}	memory:copy; byte=2*4*n; mish_transform=n	Each output requires one tolerance-matched Mish transform. A direct polynomial is legal, so exp, log1p and tanh cannot be charged independently.
rsqrt_large	n=1048576 → n=4194304	x:positive	10^{-4}	memory:copy; byte=2*4*n; rsqrt_transform=n	Each output is one tolerance-matched reciprocal-square-root transform; estimate-plus-refinement is legal and must not be priced as separate sqrt and division.
gen_paired_nibble_relation_stats_large	nrec=4194301 → nrec=4194301	left:u64; right:u64	0 (exact)	memory:read; byte=136*nrec; working_set=136*nrec; nibble_relation=nrec	Each paired record emits exact equality and parity-relation counts; complete-record calibration permits bit-plane fusion instead of forcing scalar nibble extraction.
minmax_large	n=1048576 → n=4194304	x:signed	0 (exact)	memory:read; byte=4*n; compare=2*n	Every input participates in both extrema. Vector min/max performs the reduction through the same comparisons; pricing a second FP32-add reduction would describe an operation the task never performs.
count_nonzero_large	n=1048576 → n=4194304	x:signed	0 (exact)	memory:read; byte=4*n; predicate_count=n	Every element feeds one threshold predicate and the resulting lane masks must be counted. The complete predicate-count primitive preserves compare/popcount fusion instead of pricing an unrelated FP32 reduction.
row_sum_large	R=2048, C=2048 → R=4096, C=2048	x:uniform	10^{-4}	memory:read; byte=4*(R*C+R); reduction=R*C	Each row includes every input. Direct FP32 row sums pass every held-out shape; their additions are fully represented by reduction throughput and are not also FMA work.
col_sum_large	R=2048, C=2048 → R=4096, C=2048	x:uniform	10^{-4}	memory:read; byte=4*(R*C+C); reduction=R*C	Each column includes every R input. The accepted FP32 vector partials establish precision; the additions are represented once by reduction throughput.
transpose_tiled_large	R=2048, C=4096 → R=4096, C=4096	src:uniform	0 (exact)	memory:copy; byte=2*4*R*C; transpose=R*C	Every source element changes its row/column position; the permutation is semantic work beyond a contiguous copy.
gen_byte_pair_order_stats_large	nrec=4194301 → nrec=4194301	left:u8; right:u8	0 (exact)	memory:read; byte=136*nrec; working_set=136*nrec; byte_pair_order=nrec	Every 64-byte pair requires one exact less-than count and absolute-difference sum; both outputs share one complete vectorizable record transform.
roll_large	n=1048576, shift=12345 → n=4194304, shift=12345	x:uniform	0 (exact)	memory:copy; byte=2*4*n	A rotation is two contiguous copy segments after normalizing the shift; no per-element index reorder is unavoidable.

Continued on next page.

Table 4, continued.

Task	Shape	Inputs	τ	Demand	Demand basis
slice_strided_large	n=1048576, start=3, step=5 → n=4194304, start=3, step=5	src:uniform	0 (exact)	memory:strided; byte=4*n+4*n/step	The source stride and contiguous destination are fully represented by the calibrated strided-access primitive.
gemm_transb_large	M=512, N=512, K=512 → M=1024, N=1024, K=1024	A:uniform; B:uniform	10^{-4}	memory:triad; flop=2*M*N*K; byte=4*(M*K+N*K+M*N); reduction=M*N*K	Every output is a K-term product over the transposed-B layout; the accepted vector witness establishes FP32 accumulation as sufficient.
symv_large	N=2048 → N=4096	A:spd; x:uniform	10^{-4}	memory:read; flop=2*N*N; byte=4*(N*(N+1)/2+2*N); working_set=4*(N*(N+1)/2+2*N); reduction=N*N	Only the stored lower triangle is mandatory matrix traffic: each off-diagonal value can update both symmetric outputs in one visit. The full N*N product work remains, and direct FP32 accumulation passes held-out shapes.
triu_copy_large	N=2048, diag=1 → N=4096, diag=1	src:uniform	0 (exact)	memory:copy; byte=4*N*(N-1); working_set=4*N*(N-1)	With diag=1, only the strict-upper source values and matching destination positions are semantic traffic; the zeroed lower triangle is prepared outside the timed call.
matrix_norm_large	R=2048, C=2048 → R=4096, C=2048	A:signed	10^{-4}	memory:read; flop=3*R*C; byte=4*R*C; sqrt=1; reduction=2*R*C	Both norms inspect every entry and require two reductions plus one square root. Sixteen-way FP32 partials pass all held-out shapes, so FP64 is not mandatory.
depthwise_conv2d_large	Ch=32, H=256, W=256, KS=3 → Ch=64, H=256, W=256, KS=3	src:uniform; k:signed	10^{-4}	memory:triad; flop=2*Ch*(H-KS+1)*(W-KS+1)*KS*KS; byte=4*Ch*(H*W*2); reduction=Ch*(H-KS+1)*(W-KS+1)*KS*KS	Each channel and output consumes its full receptive field; the accepted SIMD witness demonstrates FP32 accumulation within tolerance.
group_norm_large	Ch=64, L=4096, G=8 → Ch=128, L=4096, G=8	x:signed	10^{-3}	memory:copy; flop=5*Ch*L; byte=2*4*Ch*L; rsqrt_transform=G; reduction=2*Ch*L	Each group needs two moments, one reciprocal square root, and normalization. Sixteen-way FP32 partials pass all held-out shapes; fixed denominators use shared reciprocal scaling.
cross_entropy_large	R=4096, C=1024 → R=8192, C=1024	logits:signed; target:idx:C	10^{-4}	memory:read; byte=4*(R*C+2*R); log=R; bounded_exp=R*C; compare=R*C; reduction=2*R*C	Stable cross entropy needs maximum tracking, one bounded exponential contribution per signed logit, a sum, and one FP32 logarithm per row. Max tracking and the exponential sum may use one online fused pass; double libm calls are not required.
bias_gelu_large	R=2048, C=2048 → R=4096, C=2048	x:signed; bias:signed	10^{-5}	memory:copy; flop=R*C; byte=4*(R*C*2+C); gelu_transform=R*C	Every output performs one bias addition and one complete GELU transform. They may be fused, and the reusable bias vector does not form a second streaming input.
cumprod_large	n=1048576 → n=4194304	x:nearone	10^{-3}	memory:copy; byte=2*4*n; accurate_product_scan=n	Every output is an accuracy-preserving prefix product. A two-float FMA correction passes all held-out shapes, so the complete scan is calibrated without mandating per-item FP64.
cummax_large	n=1048576 → n=4194304	x:signed	0 (exact)	memory:copy; byte=2*4*n; max_scan=n	Every output is the maximum of a growing prefix. The complete hierarchical max-scan primitive includes its comparisons, lane shuffles, thread carries and output pass without charging an additive scan.
logsumexp_2d_large	R=4096, C=1024 → R=8192, C=1024	x:extreme	10^{-4}	memory:read; byte=4*(R*C+R); log=R; cutoff_exp=R*C; compare=R*C; reduction=2*R*C	Extreme rows require stable maximum tracking, but the 1e-4 contract permits negligible tails to be cut off. Maximum tracking and live exponential contributions may be fused online before one FP32 log per row.
welford_var_large	n=1048576 → n=4194304	x:nearequal	10^{-3}	memory:read; flop=3*n; byte=4*n+8; reduction=2*n	The outputs require stable first and second moments, not Welford's recurrence. A shifted-origin two-pass FP32 reduction avoids cancellation and passes every held-out shape.
prefix_max_2d_large	H=2048, W=2048 → H=4096, W=4096	src:signed	0 (exact)	memory:copy; byte=2*4*H*W; max_scan=2*H*W	The 2D prefix is a sequential composition of horizontal and vertical max scans. Both complete passes are charged; neither can be replaced by an independent copy.
segmented_sum_large	n=1048576 → n=4194304	x:uniform; flags:flags	10^{-3}	memory:triad; byte=2*4*n+4*n; compare=n; scan=n	Flags reset a loop-carried sum at data-dependent boundaries, so every element and segment transition affects the output.
block_sparse_gemm_large	M=512, N=512, K=512, BS=32 → M=1024, N=1024, K=1024, BS=32	A:uniform; B:uniform; mask:idx:2	10^{-4}	memory:row_gather; flop=M*N*K; byte=4*(M*K+K*N+M*N/2); reduction=M*N*K/2	The fixed performance mask contains slightly more than half of its blocks, so one-half of the dense products and output stores is a conservative lower bound. Mask inspection is fused with block selection and its bytes are covered by the memory phase; a random-gather primitive would describe the wrong access shape.
layernorm_backward_large	R=2048, C=1024 → R=4096, C=1024	x:signed; grad:signed; gamma:uniform	10^{-3}	memory:triad; flop=12*R*C; byte=4*(3*R*C+C); rsqrt_transform=R; reduction=4*R*C	The backward formula requires four row reductions, one reciprocal square root per row, and the fused gradient arithmetic. The accepted witness establishes FP32 vector moments under the 1e-3 contract; fixed divisions reduce to reciprocal scaling.

Continued on next page.

Table 4, continued.

Task	Shape	Inputs	τ	Demand	Demand basis
unique_sorted_large	n=1048576 → n=4194304	x:runs	0 (exact)	memory:copy; byte=4*n; compact=n	The complete stable-compaction primitive includes the adjacent predicate, survivor count, ordered offsets and writes. Only the n input values are a data-independent byte lower bound; output traffic is data-dependent and remains inside the complete primitive.
compact_nonzero_large	n=1048576 → n=4194304	x:signed	0 (exact)	memory:copy; byte=4*n; compact=n	The complete stable-compaction primitive includes threshold classification, survivor count, ordered offsets and writes. Only the n input values are a data-independent byte lower bound; output traffic is data-dependent and remains inside the complete primitive.
conv1d_transpose_large	n=262144, ks=15 → n=1048576, ks=15	x:signed; k:signed	10^{-4}	memory:copy; flop=2*n*ks; byte=4*(n+ks+n+ks); reduction=n*ks	The transposed convolution can be written output-wise as contiguous stores over ks-term dot products. Scatter updates are a removable reference traversal, not semantic demand.
bitonic_topk_large	n=1048576, k=32 → n=4194304, k=32	x:signed	0 (exact)	memory:read; byte=4*(n+k); sort=n	The calibrated top-k selection primitive already includes threshold comparisons and ordered insertion, so a second independent compare charge would duplicate the same work.
kahan_sum_large	n=1048576 → n=4194304	x:widerange	10^{-5}	memory:read; flop=n; byte=4*n+4; reduction=n	Every value contributes to the reduction. Sixteen-way FP32 partial sums pass all held-out wide-range inputs at 1e-5, so neither Kahan's recurrence nor FP64 is mandatory.
batched_matmul_small_large	Bn=4096, M=8, N=8, K=8 → Bn=8192, M=8, N=8, K=8	A:uniform; B:uniform	10^{-4}	memory:triad; flop=2*Bn*M*N*K; byte=4*Bn*(M*K+K*N+M*N); reduction=Bn*M*N*K	Every batch output is a K-term product; the accepted register GEMM demonstrates that FP32 accumulation meets the observable tolerance.

Table 5: Per-task LEETPORT specification. Shape shows development → timed performance dimensions. τ is the correctness tolerance.

Task	Shape	Inputs	τ	Demand	Demand basis
lbm_large	nx=1024, ny=1024 → nx=1024, ny=1024	f:uniform	10^{-5}	memory:row_gather; flop=155*(nx-2)*(ny-2); byte=9*4*2*nx*ny; div=(nx-2)*(ny-2)	ONE reciprocal of rho per cell is unavoidable: both velocity components divide by the same density, so 1/rho computed once serves both and the baseline's second divide is removable. The W[q] divisions are compile-time constants and are not counted at all.
nbody_large	n=4096 → n=8192	x:uniform; y:uniform; z:uniform; m:positive	10^{-3}	memory:read; flop=20*n*n; byte=7*4*n	No transcendental or divide is declared, and that is the point: the accepted 7.6x implementation replaces sqrtf + division with rsqrt plus one Newton step, which the 1e-3 tolerance admits. So neither the exact sqrt nor the divide is a semantic lower bound, and charging them made the ceiling slower than the very body the task is built around. The kernel is bounded by its FLOP count and its (tiny, cache-resident) traffic.
arrhenius_large	ncell=8192, nreac=64 → ncell=16384, nreac=64	T:scale:800:2500; A:positive; b:signed; Ea:scale:10000:180000	10^{-4}	memory:copy; flop=6*ncell*nreac; byte=4*(ncell + 3*nreac + ncell*nreac); exp=ncell*nreac	A * T^b * exp(-Ea/(R*T)) is one transcendental per (cell, reaction) and not two: T^b = exp(b*logf(T)) combines with the exponential into a single exp, and logf(T) is loop-invariant across reactions so it amortises to O(ncell). One exp per output is unavoidable – the exponent depends on both indices. The 1e-4 tolerance does not admit dropping it. The baseline's SEPARATE powf and the per-element divide are both removable and are in baseline_work.
tridiag_large	n=128, m=8192 → n=128, m=16384	a:scale:0:0.4; b:scale:2.5:3.5; c:scale:0:0.4; d:uniform	10^{-5}	memory:triad; flop=10*n*m; byte=4*6*n*m; div=n*m	The Thomas algorithm's forward sweep divides by the pivot w once per (row, column); both x and cp use the SAME w, so one reciprocal per element serves both and the baseline's second divide is removable. Columns are independent, so a thread may own a column tile through all forward and backward levels; level ordering is local to each tile and requires no team barrier. The byte floor counts the four input streams and the x/cp output streams once per element.
ragged_attn_large	nseq=256, total=131072, D=64 → nseq=256, total=131072, D=64	Q:wide_signed; K_:wide_signed; V:signed; off:ragged_off; len:ragged_len	10^{-3}	memory:row_gather; flop=4*total*D; byte=4*(2*total*D + 2*nseq*D); exp=total	Softmax needs one exponential per KEY ROW, which is 'total' over the whole ragged batch, and the scores overflow float unshifted so it cannot be avoided. The baseline's SECOND exp – the running-max correction expf(m - mnew) – is removable: it is only needed when the max actually increases, which the accepted implementation exploits, so 2*total prices a rescale a valid kernel mostly skips.

Continued on next page.

Table 5, continued.

Task	Shape	Inputs	τ	Demand	Demand basis
block_filter_compact_large	n=16777213 → n=16777213	src:filter_i32	0 (exact)	memory:copy; byte=4*(n + 16*((n+15)/16) + ((n+15)/16))	Every input value must be inspected once. The fixed public result contains one fully defined 16-word record and one count per block, so every output word must also be written; integer comparison and lane permutation perform no floating-point work.
bf16_gemv_ragged_small	m=32749, n=257 → m=32749, n=257	w:bf16_bits; x:signed	10^{-4}	memory:read; flop=2*m*n; byte=2*m*n+4*n+4*m	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
block_gemm_ragged_large	batch=19, m=257, n=131, k=113 → batch=19, m=257, n=131, k=113	a:signed; bmat:signed	10^{-5}	memory:triad; flop=2*batch*m*n*k; byte=4*(batch*m*k+k*n+batch*m*n)	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
weighted_histogram_large	npart=8388593, nbin=65537 → npart=8388593, nbin=65537	keys:idx:nbin; weights:signed	10^{-4}	memory:scatter; flop=npart; byte=4*(2*npart+nbin)	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
tensor_contract_pack_large	M=515, K=517, N=509 → M=515, K=517, N=509	Amat:signed; Bmat:signed	10^{-3}	memory:triad; flop=2*M*N*K; byte=4*(M*K + K*N + M*N)	2*M*N*K flops is exact for the contraction and no reassociation the 1e-3 tolerance admits reduces it. Minimum traffic is one pass over each operand plus one write of the result – a blocked implementation may re-read a packed panel from cache, which is not DRAM traffic and so is not charged.
json_structural_scan_large	n=16777253 → n=16777253	src:json_bytes	0 (exact)	memory:copy; byte=n + 4*8*((n+63)/64)	The exact result requires reading every input byte once and writing four 64-bit classification masks per 64-byte block. Comparisons and bit packing perform no floating-point work, so the streaming byte bound is the applicable semantic ceiling.
stencil_time_block_large	n=4194304, T=24 → n=4194304, T=24	init:signed	10^{-3}	memory:triad; flop=5*n*T; byte=4*2*n	Compute: five flops per (cell, sweep) is the stencil, and each sweep depends on the previous one, so no implementation inside 1e-3 can skip a sweep. Traffic: the MINIMUM is one read of the input field and one write of the output, 2*n words total and independent of T – temporal blocking legitimately reaches that, and it is the transformation under test, so the ceiling must not charge per-sweep DRAM traffic. The compute term binds at these sizes (3.4 ms against 0.33 ms at n=4194304, T=24), which is the same conclusion the earlier note reached by the wrong route.
int8_grouped_gemm_large	M=397, K=521, N=613, GS=24 → M=397, K=521, N=613, GS=24	a_q:i8; b_q:i8; za:i8narrow; zb:i8narrow; scale:scale: 0.00012: 0.00055	0 (exact)	memory:row_gather; byte=M*K + K*N + M*N; int_mac=M*N*K	One integer multiply-accumulate per (m, n, k) is the definition of the product and no reassociation removes it; the zero-point expansion changes WHICH integers are multiplied, not how many. Traffic is one byte per element of A, B and the output – every element must be read at least once and every output written once, and a packed implementation re-reads its panels from cache, which is not DRAM traffic. The per-output float multiply is one op per output, far below the int MAC term, and is not charged separately.
batch_chol_mixed_large	nb=512, nmax=127, nlo=11, GRP=4 → nb=512, nmax=127, nlo=11, GRP=4	ndim:cholsize; aoff:choloff; amat:spdblk	10^{-4}	memory:copy; flop=(nb/GRP)*((GRP-1)*nlo*nlo*nlo + nmax*nmax*nmax)/3.0; byte=4*2*(nb/GRP)*((GRP-1)*nlo*nlo + nmax*nmax); sqrt=(nb/GRP)*((GRP-1)*nlo + nmax); chol=(nb/GRP)*((GRP-1)*nlo*nlo*nlo + nmax*nmax*nmax)/3.0	Cholesky of order n is nˆ3/3 multiply-adds and n square roots; no implementation the tolerance admits does fewer, since every entry of L depends on every earlier column of its row. Traffic is the USED part of each instance only – 4 bytes per element of the leading n x n block, read once and written once. The PADDING is deliberately not charged: it is zero on input and must stay zero on output, so a correct kernel need never touch it, and charging nb*nmax*nmax would make the ceiling slower than a legal kernel by the padding ratio. That is the direction that falsely reports saturation.
histogram_prefix_large	n=8388593, nbin=65537 → n=8388593, nbin=65537	keys:idx:nbin	0 (exact)	memory:scatter; flop=n+nbin; byte=4*(n+2*(nbin+1))	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
quantized_depthwise_large	npix=131071, cin=127, k=7 → npix=131071, cin=127, k=7	x:i8; w: i8narrow; bias:signed	0 (exact)	memory:copy; byte=(npix+k-1)*cin+k*cin+4*(cin+npix*cin); int_mac=npix*cin*k	Every output channel is the exact sum of all k signed integer products; the zero tolerance permits no product to be skipped.
sparse_block_gemm_large	nb=1021, k=251, bs=13, nnzb=16384 → nb=1021, k=251, bs=13, nnzb=16384	rowptr:rowptr; col_idx:idx: nb; blocks: signed; bmat: signed	10^{-5}	memory:triad; flop=2*nnzb*bs*bs*k; byte=4*(nnzb*bs*bs+2*nb*bs*k)	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.

Continued on next page.

Table 5, continued.

Task	Shape	Inputs	τ	Demand	Demand basis
tile_transpose_reduce_large	m=4093, n=8191 → m=4093, n=8191	x:signed	10^{-5}	memory:strided; flop=2*m*n; byte=4*2*m*n	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
stencil_tile_halo_large	nx=4093, ny=8191 → nx=4093, ny=8191	x:signed	10^{-6}	memory:copy; flop=11*(nx-2)*(ny-2); byte=4*2*nx*ny	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
block_jacobi_large	nb=4093, bs=13 → nb=4093, bs=13	block:jacobi_block; x:signed; rhs:signed	2×10^{-5}	memory:read; flop=2*nb*bs*bs; byte=4*(nb*bs*bs+3*nb*bs)	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
tensor_pack_contract_large	m=509, n=1021, k=127 → m=509, n=1021, k=127	a:signed; b:signed	10^{-4}	memory:copy; flop=2*m*n*k; byte=4*(m*k+k*n+m*n)	The released reference uses the contract FLOP/byte expressions and declared memory pattern; it charges no separate slow-operation count for this task.
nearest_centroid_large	nq=4093, nc=509, dim=127 → nq=4093, nc=509, dim=127	q:nearest_query; cent:centroid_ties	10^{-3}	memory:read; flop=2*nq*nc*dim; byte=4*(nq*dim+nc*dim+nq)	Every query-centroid pair must inspect every coordinate for exact nearest-neighbour semantics; each coordinate contributes one subtract and one multiply. The byte floor counts each input once and one output record per query, assuming legal cache reuse of packed centroids.
pack_gemm_odd_large	m=509, n=509, k=1021 → m=509, n=509, k=1021	a:signed; bcol:signed	10^{-4}	memory:triad; flop=2*m*n*k; byte=4*(m*k+n*k+m*n)	Every one of the M*N outputs requires all K multiply-add pairs. The packed layout and padded scratch are implementation details; the semantic byte floor counts one read of A and B and one write of C, while the paired FP32-FMA calibration bounds the arithmetic.
sha256_blocks_large	nblock=524287 → nblock=524287	src:u8	0 (exact)	memory:copy; byte=96*nblock; working_set=96*nblock; sha256_block=nblock	Each output is defined by one complete 64-round SHA-256 compression from a raw 512-bit block, including its message schedule and the eight-word feed-forward state. No exact implementation can omit a block.
ghash_batch_small	nmsg=32749, nblock=13 → nmsg=32749, nblock=13	data:u64; key:u64	0 (exact)	memory:copy; byte=16*(nmsg*nblock+2*nmsg); working_set=16*(nmsg*nblock+2*nmsg); ghash_update=nmsg*nblock	Every ordered input block changes the exact authentication tag by one XOR and one GF(2^128) multiplication with the message key. Zero tolerance and the recurrence prevent omitting, reordering or merging updates across messages.
bf16_pair_gemm_small	m=1021, n=128, npair=257 → m=1021, n=128, npair=257	a:bf16_signed_bits; b:bf16_signed_bits	2×10^{-3}	memory:read; byte=2*(2*m*npair+2*npair*n)+4*m*n; working_set=2*(2*m*npair+2*npair*n)+4*m*n; bf16_dot=2*m*n*npair	Every output observes both BF16 products from every K pair. One bf16_dot_op is one BF16 multiply accumulated into FP32; no pair or output can be omitted from the complete matrix product.
sad_descriptor_search_large	nq=509, ncand=16381 → nq=509, ncand=16381	query:u8; candidates:u8	0 (exact)	memory:sad_search; byte=64*(nq+ncand)+8*nq; working_set=64*(nq+ncand)+8*nq; sad64=nq*ncand	Exact nearest-neighbour semantics require evaluating all candidate descriptors for every query. One sad64_op is the complete 64-byte absolute-difference reduction and deterministic argmin update; the shared candidate library may legally remain in cache across queries.
bitmap_set_stats_small	nrec=32749 → nrec=32749	left:u64; right:u64	0 (exact)	memory:copy; byte=136*nrec; working_set=136*nrec	Each exact record reads both 64-byte bitmaps and writes two four-byte cardinalities. Intersection and union are fused so no input word needs to be charged twice.
packed_dna_substitution_stats_small	nrec=32749 → nrec=32749	reference:u64; sample:u64	0 (exact)	memory:copy; byte=136*nrec; working_set=136*nrec	Every exact record reads both 64-byte packed sequences and writes two four-byte statistics. Computing both counts in one pass is a legal fusion and no input base may be omitted.
packed_genotype_stats_large	nrec=2097149 → nrec=2097149	genotypes:u64	0 (exact)	memory:copy; byte=72*nrec; working_set=72*nrec	Each exact record reads 64 bytes and writes two four-byte totals. Both statistics share the same two-bit classifications, so one fused pass is the minimum traffic implementation.
packed_quality_extreme_stats_large	nrec=2097149 → nrec=2097149	quality_bins:u64	0 (exact)	memory:copy; byte=72*nrec; working_set=72*nrec	Each exact record reads 64 packed bytes and writes two four-byte totals. Both disjoint range counts are fused into one input pass.
packed_label_confusion_stats_small	nrec=32749 → nrec=32749	expected:u64; predicted:u64	0 (exact)	memory:copy; byte=136*nrec; working_set=136*nrec	Every exact record reads both 64-byte label streams and writes two four-byte totals; both predicates share one packed comparison pass.
packed_dna_run_stats_large	nrec=2097149 → nrec=2097149	sequence:u64	0 (exact)	memory:copy; byte=72*nrec; working_set=72*nrec	Every exact record reads 64 bytes and writes two four-byte totals; GC classification and cross-word run boundaries share one pass.
packed_dna_composition_stats_large	nrec=4194301 → nrec=4194301	sequence:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every exact record reads 64 bytes and writes two four-byte totals; both composition counts reuse the same two packed bit planes.
masked_byte_sum_stats_small	nrec=262139 → nrec=262139	values:u8; selected:u64	0 (exact)	memory:read; byte=80*nrec; working_set=80*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.

Continued on next page.

Table 5, continued.

Task	Shape	Inputs	τ	Demand	Demand basis
packed_ascii_class_stats_small	nrec=58253 → nrec=58253	text:u8	0 (exact)	memory:read; byte=76*nrec; working_set=76*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_nibble_parity_stats_small	nrec=58253 → nrec=58253	symbols:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_ternary_histogram_large	nrec=4194301 → nrec=4194301	codes:u64	0 (exact)	memory:read; byte=80*nrec; working_set=80*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_dna_motif_stats_large	nrec=4194301 → nrec=4194301	sequence:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
bitmap_run_stats_large	nrec=4194301 → nrec=4194301	bitmap:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_nibble_order_stats_small	nrec=32749 → nrec=32749	expected:u64; observed:u64	0 (exact)	memory:read; byte=140*nrec; working_set=140*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_3bit_range_stats_large	nrec=2097149 → nrec=2097149	status:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_5bit_category_stats_large	nrec=2097149 → nrec=2097149	category:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_6bit_zone_stats_large	nrec=2097149 → nrec=2097149	zone:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_7bit_parity_stats_large	nrec=2097149 → nrec=2097149	token:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and every exact statistic is overwritten. The statistics share one traversal, so the declared read and output traffic is the portable lower bound.
packed_byte_partition_stats_large	nrec=4194301 → nrec=4194301	values:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and both exact statistics are overwritten; the statistics share one traversal, so this traffic is the portable semantic lower bound.
bitmap_relation_stats_large	nrec=4194301 → nrec=4194301	left:u64; right:u64	0 (exact)	memory:read; byte=136*nrec; working_set=136*nrec	Every fixed record is read in full and both exact statistics are overwritten; the statistics share one traversal, so this traffic is the portable semantic lower bound.
packed_dibit_transition_stats_large	nrec=4194301 → nrec=4194301	symbols:u64	0 (exact)	memory:read; byte=72*nrec; working_set=72*nrec	Every fixed record is read in full and both exact statistics are overwritten; the statistics share one traversal, so this traffic is the portable semantic lower bound.
paired_nibble_relation_stats_large	nrec=4194301 → nrec=4194301	left:u64; right:u64	0 (exact)	memory:read; byte=136*nrec; working_set=136*nrec	Every fixed record is read in full and both exact statistics are overwritten; the statistics share one traversal, so this traffic is the portable semantic lower bound.
byte_pair_order_stats_large	nrec=4194301 → nrec=4194301	left:u8; right:u8	0 (exact)	memory:read; byte=136*nrec; working_set=136*nrec	Every fixed record is read in full and both exact statistics are overwritten; the statistics share one traversal, so this traffic is the portable semantic lower bound.

C Agent and Tool Protocol

Versioned condition files define the interaction mode, reasoning setting, tool budget, and optional reference or guide. Records for each run retain the messages and tool log. One tool provides a general shell. Installed programs such as `objdump` or `perf` are available when present on the host. Outbound network access is also available. The assigned condition stays the same even when an agent does not use the supplied material.

The baseline protocol uses three independent runs per workflow. Each run allows at most 30 model turns and 60 tool calls. Each model response has a 65,536-token output cap. Temperature is 0.2 when the API accepts it. Requests with native extended thinking omit this parameter. The runner has no total token cap for a full run, so the budget only fixes the interaction limits. Repetition estimates the expected quality of one run.

Table 6: A tool-using optimization trajectory for `stencil_time_block_large`. The model was GPT-5.6-Sol on x86-64 with two physical cores. Speedups in the body of the table come from the visible development test and are noisy feedback for the agent. The last row is a separate held-out replay of the final emitted program.

Stage	Change or observation	Dev speedup	Resulting decision
Start	Correct untuned code scans the full array for each of 24 steps.	$1.04\times$	Establish the starting measurement.
1	Add parallel copy and stencil loops with double buffering.	$3.26\times$	Keep parallel execution and examine its overhead.
2	Place all time steps inside one persistent OpenMP region.	$2.85\times$	The measured result regresses, so look beyond parallel-region overhead.
3	Inspect GCC vectorization messages and assembly. The stencil loop is vectorized with 32-byte vectors.	–	Shift attention to repeated memory traffic across time steps.
4	Add temporal blocking with a 16,384-element tile and a T-element halo. Use worker-private double buffers and SIMD within each sweep.	$3.63\times$	Search for a tile size with a smaller working set.
5	Reduce the tile to 2,048 elements.	$4.36\times$	Test nearby tile sizes.
6	Increase the tile to 4,096 elements.	$3.17\times$	Reject this regression and try a smaller tile.
7	Reduce the tile to 1,024 elements.	$4.14\times$	Probe an intermediate size before the run ends.
8	Emit the final program with a 1,536-element tile.	$3.94\times$	Preserve the correct temporal-blocked implementation for held-out evaluation.
Replay	Hidden correctness shapes use $(n, T) = (5{,}951{,}304, 37)$ and $(4{,}445{,}131, 40)$. Median starting and candidate times are 17.945 ms and 3.634 ms.	$4.94\times$	Correct on both hidden cases. ECA is 0.466; the calibrated binding resource is FP32 FMA.

D Pilot Agent and Information Ablations

All conditions use the file, build, public-test, and shell tools from Appendix B.1. A1-A7 stand for different conditions. A1 is the only single-shot condition and has no tools. Table 9 lists the pilot conditions. The examples in A4 and A5 implement related computations.

The x86 guide is stored locally. It describes the measured host. A4 and A5 use validated fast examples for particle potential, batched FP64 tridiagonal solve, and signed INT8 VNNI dot product. A7 and A8 provide the same project guide to x86 intrinsics and optimization. The guide appears either in `reference/` or through a logged retrieval tool. The final traces for the file conditions only read this material. A1, A3, and A6 also change the computation budget, so they do more than change information.

Table 8 and Figure 6 shows that The observed ECA values for A1, A2, and A3 are below B0 by .196, .392, and .263. A1 has no agent loop or tools. A2 has no explicit reasoning. A3 has a smaller feedback budget. Their paired bootstrap intervals over workflows exclude zero. The file reference condition A4 is .085 below B0, and its interval includes zero. A5 through A8 also show no clear gain over B0. Changing the same reference from a file to a tool gives A5$-$A4 $= +.088$ with 95% CI $[-.105, .293]$. Changing the same platform guide gives A8$-$A7 $= -.003$ with 95% CI $[-.178, .155]$. A6 doubles the interaction budget and consumes 77.9M tokens. This is more than three times B0, with no clear ECA gain.

E Example of an iterative optimization trajectory

Table 6 gives one complete example of how feedback changes an agent's implementation. The task applies a one-dimensional stencil for several time steps. The starting code scans the full array at every step. The agent first tried parallel loops and then inspected the compiler's vectorization report and generated assembly. It next changed the algorithm to temporal blocking. Each spatial tile includes the halo needed for all time steps, and each worker advances that tile in private double buffers. This reduces repeated full-array traffic. The agent then tested several tile sizes. It kept measurements that became slower, so the record also shows rejected choices instead of only the final program.

Table 7: Resources that limit at least one workflow in the released two-core reference validation. The table lists results by platform and track.

Platform	Track	Resource identifiers that bind at least one workflow
x86-64	LEETGEN	`accurate_product_scan`, `bounded_exp`, `compact`, `cutoff_exp`, `dibit_transition`, `fp32_fma`, `fp32_matrix_fma`, `gelu_transform`, `max_scan`, `mem_copy`, `mem_map`, `mem_read`, `mem_strided`, `mem_triad`, `mish_transform`, `predicate_count`, `scan`, `silu_transform`, `sort`, `ternary_histogram`, `transpose`
x86-64	LEETPORT	`bf16_dot`, `chol_soa`, `exp`, `fp32_fma`, `ghash_update`, `int_mac`, `mem_copy`, `mem_read`, `mem_scatter`, `mem_strided`, `mem_triad`, `sad64_search`, `sha256_block`
AArch64	LEETGEN	`accurate_product_scan`, `bounded_exp`, `cutoff_exp`, `gelu_transform`, `mem_copy`, `mem_map`, `mem_read`, `mem_strided`, `mem_triad`, `mish_transform`, `reduction`, `roll_permutation`, `rsqrt_transform`, `scan`, `silu_transform`, `transpose`
AArch64	LEETPORT	`bf16_dot`, `chol_soa`, `exp`, `fp32_fma`, `ghash_update`, `int_mac`, `mem_copy`, `mem_read`, `mem_scatter`, `mem_strided`, `mem_triad`, `sad64_search`, `sha256_block`

Table 8: DeepSeek-V4.1-Flash x86-64 LeetPort pilot. Each condition has three workflows and five independent runs per workflow. ECA and paired differences use workflow-bootstrap 95% intervals. ECA|C is conditional on correctness. Tokens include cached input tokens. Bold and underline mark the largest and smallest point estimates in each metric column.

Condition	Pass@1	ECA|C	ECA [95% CI]	ΔECA vs. B0 [95% CI]	Tokens (M)	Calls
B0	0.933	0.478	0.444 [0.405,0.476]	–	23.4	544
A1	0.800	0.288	0.248 [0.033,0.389]	-0.196 [-0.418,-0.016]	0.5	0
A2	0.800	<u>0.070</u>	<u>0.052 [0.016,0.077]</u>	-0.392 [-0.435,-0.327]	7.3	516
A3	<u>0.733</u>	0.199	0.181 [0.001,0.330]	-0.263 [-0.450,-0.145]	3.1	237
A4	0.933	0.390	0.359 [0.255,0.447]	-0.085 [-0.220,+0.042]	24.0	558
A5	0.867	0.507	0.447 [0.270,0.548]	+0.003 [-0.181,+0.117]	21.6	549
A6	<u>0.733</u>	**0.570**	0.422 [0.218,0.583]	-0.022 [-0.187,+0.108]	77.9	993
A7	0.933	0.487	**0.449 [0.368,0.526]**	+0.005 [-0.037,+0.075]	22.7	552
A8	**1.000**	0.446	0.446 [0.348,0.524]	+0.003 [-0.103,+0.119]	22.3	531

This example is useful because the main gain does not come from selecting a wider instruction set. It comes from changing when data is reused. The trace also shows why the development speedup and the final replay should be reported separately. They use different shapes and repeated measurements, and the locally fastest tile was not the final emitted choice.

F PORTABILITY

Figure 7 compares the two targets. AArch64 has higher LEETGEN ECA for all five agents. The difference over x86-64 ranges from +.072 for Terra to +.137 for Gemini. LEETPORT has no uniform target advantage. AArch64 is higher for Terra, Opus, and Gemini. x86-64 is higher for Sol and Grok. These cells were sampled independently. They describe each evaluated combination of model, compiler, and hardware.

G RESOURCE MODEL AND REFERENCE VALIDATION

The internal calibration and validation records contain required work, task steps, measured resource rates, witness results, reliability flags, and raw timing samples. The paper exports use versioned summaries of these records. The summaries have not yet been combined into one release index.

Phase example. For vector normalization $y_i = x_i/\sqrt{\sum_j x_j^2}$, the approved algorithm has two required steps. It first computes the sum of squares and then normalizes each value. We add their reference times because the second step needs the first result. Within one step, we use the largest independent resource term D/P. Resources that share hardware are measured together. If the task allows a fused algorithm that removes the dependency, we validate that algorithm family.

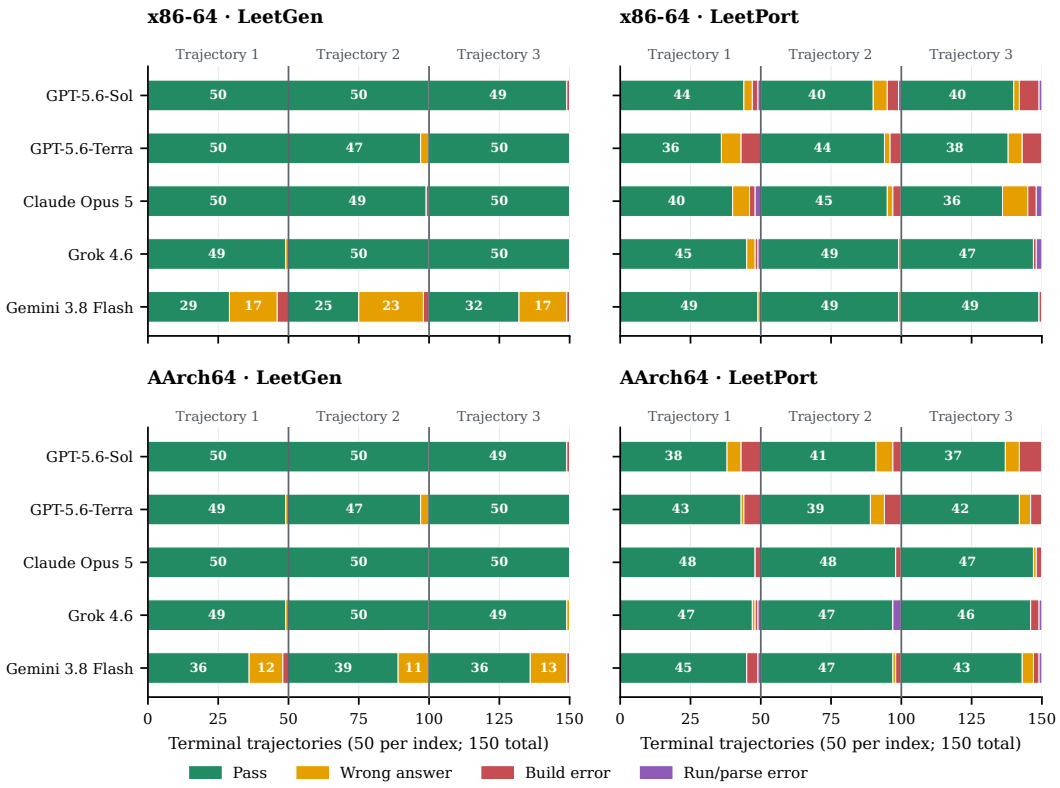

Figure 5: Final outcomes by run index. Each model has one bar with 150 attempts. The bar contains three fixed blocks of 50 attempts for runs 1, 2, and 3. Colors show the outcomes within each block. Run and parse errors are combined because neither delivers a valid candidate. The split shows sampling variation. It does not turn $k = 3$ into best-of-three selection.

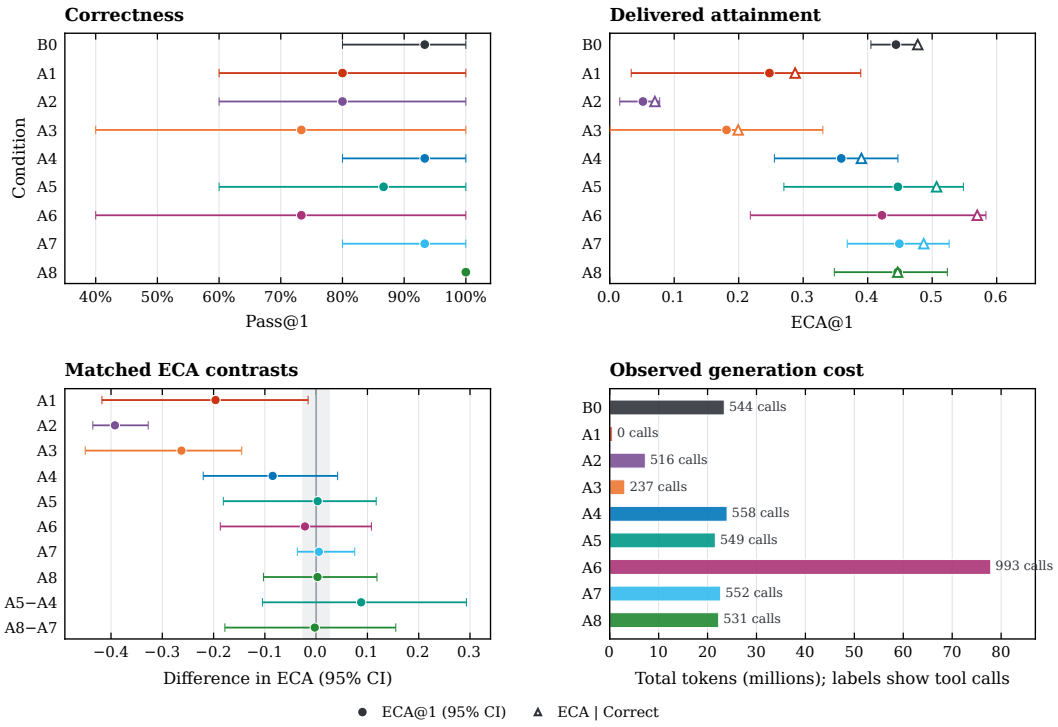

Figure 6: DeepSeek-V4.1-Flash x86-64 LeetPort pilot across three workflows with five independent runs per condition and workflow. The top row reports Pass@1, ECA@1, and ECA conditional on correctness. The bottom row reports workflow-paired ECA contrasts and observed token and tool-call cost. Error bars are 95% bootstrap intervals over workflows. Repetitions estimate one-run behavior.

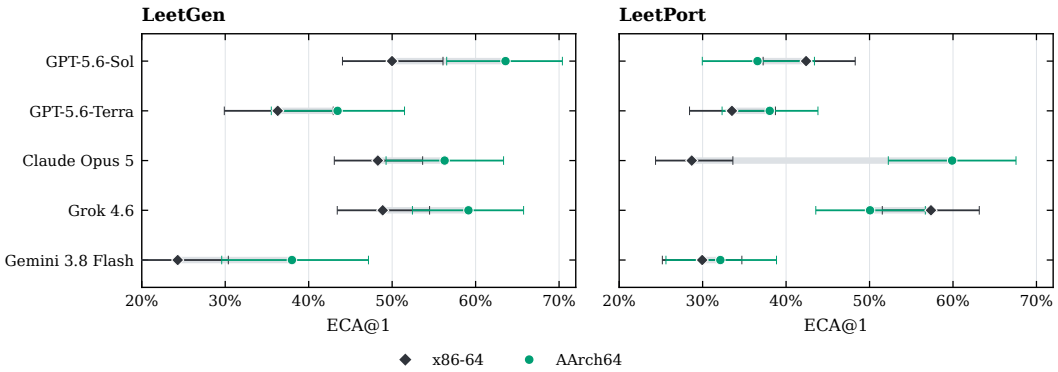

Figure 7: Descriptive ECA across the two real CPU targets. Lines connect the same model identifier, and bars give 95% workflow-bootstrap intervals. Runs for the two targets were sampled independently.

Table 9: Conditions used in the completed pilot. File and tool pairs provide the same content in two ways.

ID	Change from Baseline B0	Intended factor
B0	Tools, high reasoning, 30 turns/60 calls	Baseline
A1	Single-shot with no tools	Agent loop
A2	Reasoning disabled	Explicit reasoning
A3	10 turns/20 calls	Feedback budget
A4	Related implementation as file	Reference content
A5	Same reference through tool	Access mechanism
A6	60 turns/120 calls	Larger feedback budget
A7	Platform guide as file	Hardware information
A8	Same guide through tool	Access mechanism