ASCENDINFRABENCH: BENCHMARKING CODING AGENTS FOR FULL-LIFECYCLE LLM INFRASTRUCTURE ENGINEERING ON ASCEND NPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM infrastructure must coordinate model computation, communication, and state management. We introduce a capability-oriented task construction method that specifies required work, permitted changes, and performance objectives while leaving implementation choices open. We use this method to construct ASCENDINFRABENCH, with 24 tasks spanning Optimization, Model Adaptation, and Open Build across training, reinforcement learning, and serving infrastructure on Ascend NPUs. We independently rebuild each submission, check numerical correctness and state handling, measure performance, and review the source against task requirements. State checks verify that training resumes from checkpoints, rollout uses updated policies, and serving preserves request-cache isolation. We evaluate 9 coding agents across 216 development runs, each with a 2-hour budget. GPT-6 Astra delivers valid implementations on all 24 tasks, including all 6 Model Adaptation and Open Build tasks, achieving a $1.63\times$ task-weighted geometric-mean speedup over task baselines. Across agents, valid implementations meet the same task requirements through different choices in execution, model integration, and system organization. ASCENDINFRABENCH assesses agents' ability to optimize existing code, integrate model support, and build complete LLM systems.

1 INTRODUCTION

LLM infrastructure engineering coordinates model computation, communication, and state across training, reinforcement learning (RL), and serving. Distributed training partitions computation and optimizer state (Shoeybi et al., 2020; Rajbhandari et al., 2020); RL transfers updated policies from training to rollout (Sheng et al., 2025); serving manages concurrent requests and their caches (Kwon et al., 2023; Zheng et al., 2024). Common engineering tasks include optimizing an existing system, extending a framework to support a new model architecture, and building a custom system for a target model and workload. These tasks motivate evaluating coding agents along two dimensions: **infrastructure stage**, which determines the workflow, and **engineering scope**, which defines the starting point and the work the agent must complete.

Existing benchmarks evaluate repository-level changes and terminal workflows (Jimenez et al., 2024; Deng et al., 2025; Merrill et al., 2026), accelerator kernels (Ouyang et al., 2025), and, more recently, inference optimization and complete serving systems (Nangia et al., 2026; Yeon et al., 2026; Kamahori et al., 2026). We study how well coding agents optimize, adapt, and build systems across training, RL, and serving infrastructure. As tasks expand from modifying existing code to building complete systems, agents gain more control over system design. Evaluation must therefore accommodate this design freedom while checking numerical correctness and state handling: training must resume with optimizer state intact, RL rollout must use updated policies, and serving must preserve request isolation.

We introduce a capability-oriented task construction method that specifies required computation, state transitions, and performance objectives while leaving implementation choices open within a defined scope. We use this method to construct ASCENDINFRABENCH, with 24 tasks on Ascend NPUs covering the three infrastructure stages and four engineering scopes: Implementation Optimization, Joint Optimization, Model Adaptation, and Open Build. Independent evaluation rebuilds each

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.

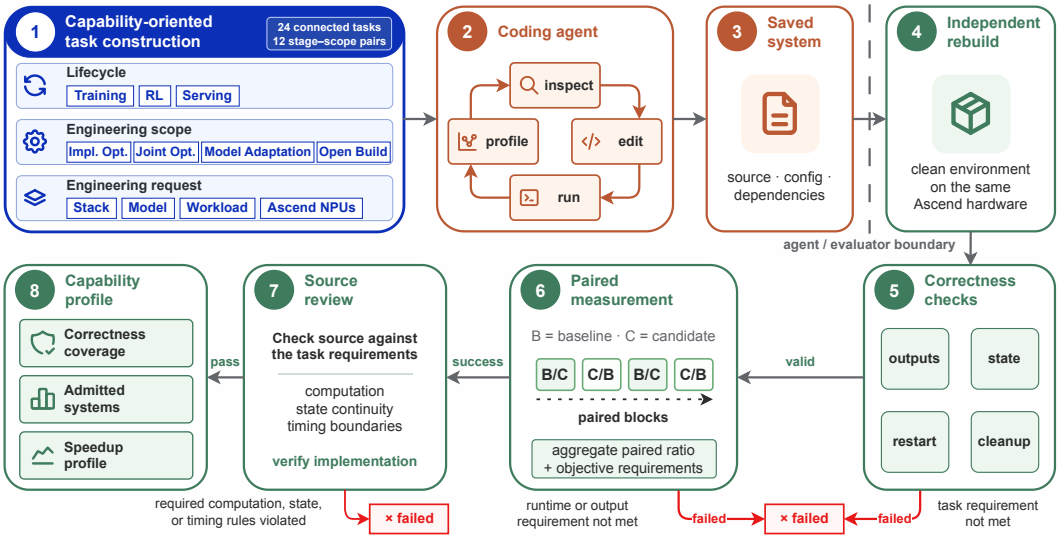

Figure 1: **From engineering tasks to evaluated systems.** Each agent develops an implementation; independent evaluation rebuilds it, checks correctness, measures performance, and reviews the source.

submission, checks its computation and state handling, measures performance, and reviews its source against task requirements (Figure 1).

We evaluate 9 coding agents across 216 development runs, each with a 2-hour budget. GPT-6 Astra delivers valid implementations on all 24 tasks, including all 6 Model Adaptation and Open Build tasks, achieving a 1.63× task-weighted geometric-mean speedup over task baselines. GPT-5.6 Sol and DeepSeek-V4-Pro each complete 17 Optimization tasks, but their completion counts differ for Model Adaptation tasks (3 versus 2) and Open Build tasks (3 versus 1). This contrast highlights the value of evaluating agent capabilities by engineering scope.

Across the benchmark, agents deliver different valid solutions to the same tasks, with implementation choices spanning execution, model integration, and system organization. Our analysis of development trajectories shows how agents develop and refine these solutions through code reuse, performance measurements, and end-to-end testing, providing an empirical basis for improving coding agents.

Contributions. **(i) A capability-oriented task construction method** defines required computation, state transitions, and performance objectives while allowing different implementations within each task's permitted scope. **(ii) AscendInfraBench** provides 24 tasks spanning three infrastructure stages and four engineering scopes, with independent reconstruction, correctness checks, performance measurement, and source review. **(iii) A 9-agent empirical study** compares agent capabilities across stages and scopes and examines both their implementation strategies and development practices.

2 Related Work

Code and system optimization. Repository and terminal benchmarks test issue resolution, workflows, and program reconstruction (Jimenez et al., 2024; Deng et al., 2025; Merrill et al., 2026; Yang et al., 2026). Performance tasks span codebase optimization (He et al., 2026; Ma et al., 2026; Sehgal et al., 2026), numerical programs, and accelerator kernels (Shetty et al., 2025; Press et al., 2025; Ouyang et al., 2025; Xing et al., 2026). Other benchmarks cover research engineering, system operations, and cloud infrastructure edits (Wijk et al., 2025; Gao et al., 2026b; Tarasova et al., 2026).

LLM infrastructure engineering. ISO-Bench studies performance patches in vLLM and SGLang. InferenceBench evaluates server deployment and optimization. VibeServe studies bespoke serving systems (Nangia et al., 2026; Yeon et al., 2026; Kamahori et al., 2026). CCL-Bench uses execution traces to characterize compute, memory, and communication efficiency (Ding et al., 2026). Φ-Bench and FrontierSWE v2 include broader engineering portfolios (Φ-Bench Contributors, 2026; Kondra

et al., 2026). ASCENDINFRABENCH tests Optimization, Model Adaptation, and Open Build in each of training, RL, and serving (Table 1). Appendix B.2 gives the task-level comparison criteria.

Ascend and CANN. LiveCANNBench evaluates output prediction and code repair in CANN repositories. CANN Bench and AscendKernelGen evaluate kernel generation on Ascend, while MultiKernelBench spans accelerator platforms (Wang et al., 2026; Gao et al., 2026a; Cao et al., 2026; Wen et al., 2025). ASCENDINFRABENCH evaluates NPU code and framework changes through complete workloads, including distributed execution and state recovery.

Learning algorithms and post-training. MLAgentBench and MLE-bench evaluate machine-learning experimentation; Agent^2 RL-Bench, PostTrainBench, AI4AI-Bench, and RSIBench-Data evaluate training algorithms or data strategies (Huang et al., 2024; Chan et al., 2025; Chen et al., 2026; Rank et al., 2026; Chi et al., 2026; Meng et al., 2026). Our tasks evaluate how agents improve infrastructure execution while preserving the specified learning and workload requirements.

Table 1: **Infrastructure coverage across stages and engineering scopes.** Opt.: Implementation and Joint Optimization; Adapt: Model Adaptation; Open: Open Build.

Study	# Tasks	Training			RL			Serving			Score target
		Opt.	Adapt	Open	Opt.	Adapt	Open	Opt.	Adapt	Open	
ISO-Bench	54	—	—	—	—	—	—	✓	—	—	Inference perf.
InferenceBench	4	—	—	—	—	—	—	✓	—	✓	Serving perf.
VibeServe	6	—	—	—	—	—	—	—	—	✓	Serving perf.
Φ-Bench	85	✓	—	—	—	—	—	✓	—	—	Task-specific
FrontierSWE v2	34	✓	—	✓	—	—	—	✓	✓	—	Task-specific
AscendInfraBench	**24**	✓	✓	✓	✓	✓	✓	✓	✓	✓	**System perf.**

3 BENCHMARK DESIGN AND EVALUATION

3.1 CAPABILITY-ORIENTED TASK CONSTRUCTION

Fix the required work and open the system choices. We build each task around an engineering goal, specifying the model computation, input workload, device budget, required outputs, starting point, and permitted changes. The starting point and editable scope determine what the agent must implement or integrate. Different system designs can answer the same request by completing the prescribed work within these constraints.

State requirements describe what must happen next: a new process restores a checkpoint and continues training; rollout uses the updated policy; a service keeps each request's cached state separate. We translate these requirements into numerical comparisons and tests that save, reload, or reuse the state. We also define timing boundaries and performance objectives, so each task specifies both the behavior to deliver and how its performance is measured. For example, T06 asks agents to build a Qwen3-4B trainer on 2 NPUs. The task fixes the model, initialization, ordered data, and optimizer mathematics, while leaving parallel decomposition, communication, memory management, and checkpoint representation open. Evaluation combines complete-job timing with numerical and optimizer-state checks, including a fresh-process update 31 after checkpoint restoration. Appendix E provides the complete task prompt.

3.2 ASCENDINFRABENCH COVERAGE AND ENGINEERING SCOPES

ASCENDINFRABENCH contains 24 tasks: 6 training, 5 RL, and 13 serving tasks (Figure 2). They use MindSpeed-LLM, verl, SGLang, and vLLM-Ascend, with dense and mixture-of-experts (MoE) models. Tasks use 1, 2, or 4 Ascend 910B NPUs. Infrastructure stages identify the workflow, while engineering scopes specify the starting point and the agent's responsibility. Appendix B.1 gives the full catalog.

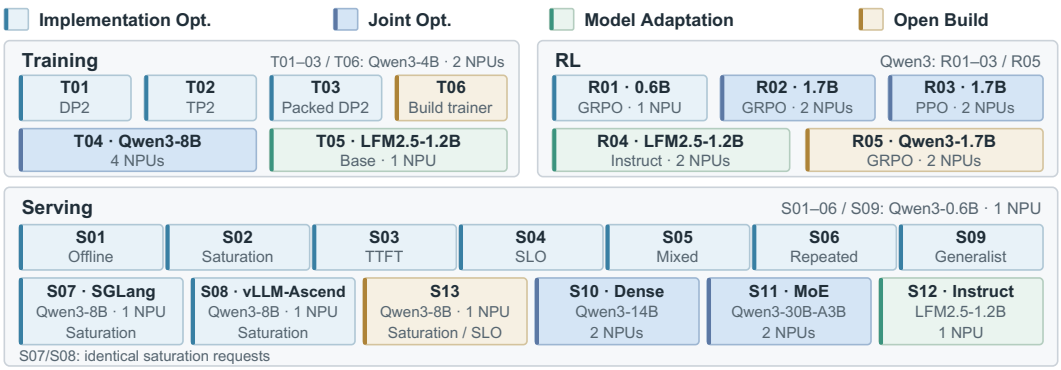

Figure 2: **Related tasks expose different system choices.** Colours identify the 4 engineering scopes.

A correctness reference supplies expected numerical results or behavior, whereas a performance baseline is a runnable system that supplies the denominator for speedup. The 4 scopes differ in their starting systems and permitted changes.

Optimization covers 2 scopes that start from working systems and target performance improvements. **Implementation Optimization** allows agents to change framework or operator code while preserving the prescribed computation and execution plan. Its 13 tasks cover graph execution, scheduling, cache management, and training data movement. **Joint Optimization** additionally allows configuration and execution-plan changes. Its 5 tasks let agents combine code changes with different parallel layouts, actor/rollout placement, or serving replicas under the same workload and device budget. Both scopes use the unmodified system under its original execution plan as the performance baseline.

Model Adaptation asks agents to add missing model support to a supplied framework. Training must support the model, weight conversion, and resumed updates; RL must also transfer the trained policy to rollout; serving must maintain attention and convolution state through prefill and decode. The baselines are an adapted MindSpeed-LLM trainer, a minimally adapted verl/vLLM stack for RL, and a minimal SGLang service for serving.

Open Build starts from an empty solution tree and asks for a complete training, RL, or serving system. Agents choose reusable components and supply preparation and launch procedures. The 3 performance baselines are the task-configured MindSpeed-LLM trainer, verl GRPO system, and SGLang system. Appendix E provides a complete task prompt.

Inputs and development feedback. All task workloads are available to agents during development, but the feedback provided differs by engineering scope. For Optimization tasks, agents can compare outputs from the supplied system with those from their modified implementation. Model Adaptation and Open Build tasks provide reference values, expected responses, or executable development checks rather than a working implementation of the target task. This feedback guides development; final evaluation independently tests each submitted implementation. Appendix D.1 describes the agent-visible feedback and evaluator-held checks.

3.3 Independent Evaluation

The evaluator rebuilds the captured final submission in a separate environment and applies 3 gates: Correctness, Measurement, and Review. Correctness checks algorithms, outputs, and state in a rule-based manner. Measurement records performance. Review checks the code behind the observed behavior.

Correctness: compare results and test the next operation. Training runs use the same model, initialization, and ordered inputs as trusted reference runs. We compare loss and pre-clipping global gradient norm at each update, test packed-document boundaries, and restore saved training state in a fresh process for the next update. Open Build training also checks optimizer state transitions.

RL evaluation checks the computation on the candidate's actual trajectories and policy state. We independently recompute rewards, advantages/returns, and losses, and compare selected-token log probabilities or PPO critic values. Numerical replay also compares loss-input derivatives, sampled parameter gradients, and optimizer updates against calibrated references; Appendix D.3 details these replay checks. We test whether rollout uses the updated policy and whether checkpoint recovery supports the next update.

Serving checks generated results and request handling. For the 12 Optimization and Model Adaptation tasks, an independent reference implementation of the same model recomputes the log probability of each service-generated token, given the original input and the service's preceding tokens. We check that the service-reported log probabilities match reference values within tolerance. We also check that each generated token's reference log probability falls within the allowed margin of the highest log probability at that position. Task checks also exercise outputs, cache reuse, and request behavior. Open Build serving uses task-specific output and HTTP checks for streaming, termination, token accounting, and concurrency. Appendix D gives the references, tolerances, and state tests.

Measurement: time the same work against the task baseline. We run 4 alternating baseline/candidate pairs on the same physical NPU allocation, with shared workloads and task-defined warmup rules and timing boundaries. Training measures throughput, RL measures accepted response-token rate, and serving measures throughput, latency, or SLO goodput. Ratios above 1 indicate better performance. For training and serving, we take the median of the 4 paired performance ratios. For RL, we compute each implementation's rate as total accepted response tokens divided by total elapsed time across its 4 runs, then divide the candidate rate by the baseline rate. Task-prescribed weights combine the objective ratios into task speedup through a weighted geometric mean (Appendix C).

Review: follow the code that produces the result. Review traces the actual model computation, optimizer updates, parameter transfers, checkpoint loading, and timed work. It checks that reported observations come from these operations and that changes stay within the permitted scope. The authors decide final outcomes with model-assisted analysis (Appendix D.6).

A **valid** submission passes correctness and source review and completes performance measurement.

4 Experimental Evaluation

Setup. We evaluate 9 coding agents, each specified by a model–harness configuration. We conduct one development run per agent–task pair with a 2-hour budget, for 216 runs in total. Table 2 and Appendix B detail the agent configurations and environments.

4.1 Performance across infrastructure stages and engineering scopes

Agents deliver 178 valid implementations across 216 submissions. GPT-6 Astra completes all 24 tasks at $1.63\times$ task-weighted geometric-mean speedup. By stage, GPT-5.6 Sol leads training at $1.45\times$ geometric-mean speedup, while GPT-6 Astra leads RL at $3.74\times$ and serving at $1.28\times$. Table 2 summarizes performance and development usage and defines the aggregate speedups (Appendix C.4); Figure 3 shows all outcomes.

Delivery varies across engineering scopes. Implementation Optimization yields 110/117 valid submissions, versus 15/27 for Open Build. GPT-6 Astra and GPT-5.6 Sol each deliver all 6 Model Adaptation and Open Build systems. GPT-5.6 Sol, Claude Opus 5, and DeepSeek-V4-Pro each complete 17/18 Optimization tasks, but their completion counts differ for Model Adaptation (3, 2, and 2) and Open Build (3, 2, and 1) (Appendix H.1). Scope-level results reveal differences that Optimization completion counts alone miss.

Coverage leaders change with the speedup threshold. Performance coverage counts valid tasks meeting all per-objective constraints and achieving speedup $S \geq \tau$ (Ouyang et al., 2025). Unchanged Optimization baselines score $1.00\times$ but do not count toward coverage when $\tau > 1$. At $1.10\times$, GPT-5.6 Sol and Claude Fable 5.1 each cover 13 tasks, versus 12 for GPT-6 Astra (Figure 4). At $2\times$, GPT-6 Astra covers 6 tasks and Kimi K3 covers 5. Together, these thresholds show both how broadly agents improve performance and how often they deliver larger gains.

Table 2: **Performance and development usage.** GM: geometric mean. Overall is the geometric mean of the stage or scope. "—" indicates there is no valid task submission. Subscripts count valid tasks. Impl./Joint: Implementation/Joint Optimization. Totals: Train/RL/Serve 6/5/13; Impl./Joint/Adapt/Open 13/5/3/3. Usage reports per-run medians. Input includes cached input. Bold and underline indicate the best and second-best values, respectively. Higher counts and speedups and lower usage are preferred.

Configuration	Valid	Task GM	Infrastructure stage				Engineering scope					Usage / run		
			Train	RL	Serve	Overall	Impl.	Joint	Adapt	Open	Overall	Input (M)	Output (k)	Tools
GPT-6 Astra Codex · max	**24/24**	**1.63**	1.36_{6}	$\mathbf{3.74}_{5}$	$\mathbf{1.28}_{13}$	**1.87**	$\mathbf{1.24}_{13}$	$\mathbf{3.10}_{5}$	1.25_{3}	$\mathbf{2.38}_{3}$	**1.84**	20.83	125.53	257
Claude Fable 5.1 Claude Code · max	22/24	1.37	$\underline{1.38}_{6}$	2.28_{4}	1.16_{12}	1.54	1.10_{12}	2.18_{5}	$\underline{1.34}_{2}$	1.58_{3}	1.50	$\underline{17.85}$	173.44	277
GPT-5.6 Sol Codex · max	$\underline{23/24}$	$\underline{1.39}$	$\mathbf{1.45}_{6}$	2.36_{4}	1.15_{13}	$\underline{1.58}$	1.15_{13}	2.18_{4}	$\mathbf{1.37}_{3}$	1.71_{3}	$\underline{1.56}$	30.96	**65.59**	**152.5**
Claude Opus 5 Claude Code · max	21/24	1.22	1.37_{6}	1.23_{2}	1.15_{13}	1.25	1.11_{13}	1.39_{4}	1.31_{2}	1.64_{2}	1.35	**17.05**	154.04	$\underline{169}$
Kimi K3 Kimi Code · max	20/24	1.31	1.30_{6}	2.33_{3}	1.13_{11}	1.51	1.13_{11}	1.58_{4}	1.24_{2}	$\underline{1.84}_{3}$	1.42	24.99	108.73	192.5
DeepSeek-V4-Pro DeepSeek · max	20/24	1.21	1.19_{5}	1.24_{3}	$\underline{1.21}_{12}$	1.22	$\underline{1.18}_{13}$	1.57_{4}	1.21_{2}	0.63_{1}	1.09	33.95	78.42	182
GLM-5.3 Claude Code · max	19/24	1.34	1.14_{5}	$\underline{2.74}_{3}$	1.18_{11}	1.54	1.14_{12}	$\underline{2.43}_{4}$	1.14_{3}	—	—	18.98	84.67	178
MiniMax M3 Claude Code · max	15/24	1.12	1.02_{4}	1.21_{2}	1.15_{9}	1.12	1.13_{12}	1.00_{2}	1.26_{1}	—	—	45.17	$\underline{70.22}$	302.5
Gemini 3.7 Flash Antigravity · high	14/24	1.13	1.05_{4}	1.47_{1}	1.13_{9}	1.21	1.13_{11}	0.99_{1}	1.18_{2}	—	—	36.76	104.98	241

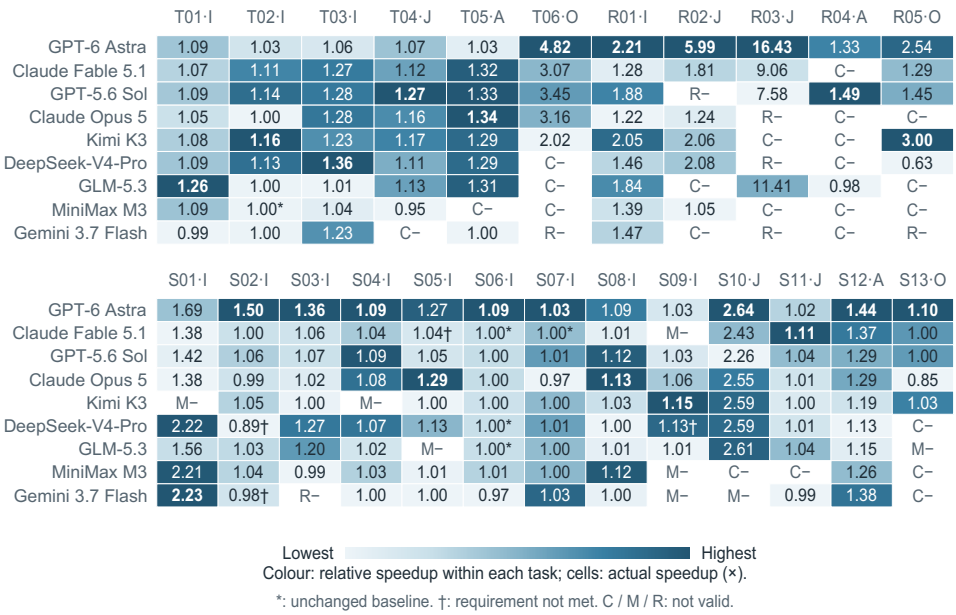

Figure 3: **All 216 task outcomes.** Cells show speedups; colours compare agents within each task. I/J/A/O: Implementation/Joint Optimization, Model Adaptation, Open Build. C/M/R: Correctness/Measurement/Source review failure. *: unchanged baseline; †: unmet performance constraint.

4.2 Optimizing execution paths and system layouts

Serving improvements combine kernels, scheduling, and graph replay. In offline Qwen3-0.6B serving (S01), Gemini 3.7 Flash, DeepSeek-V4-Pro, and MiniMax M3 capture decode graphs for larger batches to avoid falling back to eager execution, achieving 2.21–2.23× speedups.

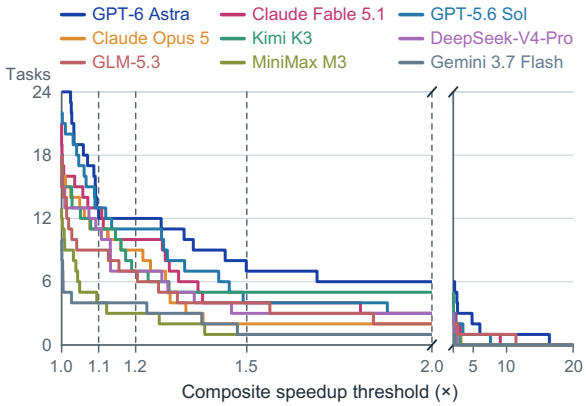

Figure 4: **Performance coverage.** Valid tasks meeting all per-objective constraints and reaching the speedup threshold; the 1–2× range is expanded.

	Graph	Attn.	Req.	KV	Ops.	Setup
GPT-6 Astra	7	8	3	5	4	3
Claude Fable 5.1	4	5	2	3	1	4
GPT-5.6 Sol	4	1	2	1	4	2
Claude Opus 5	4	3	5	1	0	3
Kimi K3	5	9	3	2	1	3
DeepSeek-V4-Pro	1	0	5	0	0	4
GLM-5.3	5	4	3	1	1	0
MiniMax M3	2	2	5	2	1	3
Gemini 3.7 Flash	3	2	3	2	5	0

Edited tasks / 9; categories overlap

Figure 5: **Serving edits by component.** Counts over 9 Implementation Optimization tasks; categories overlap.

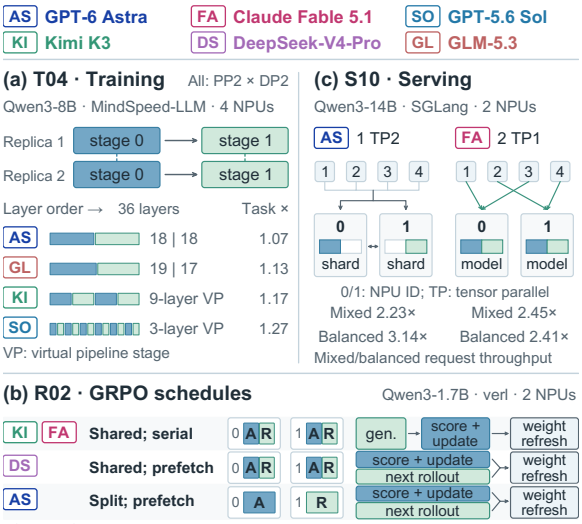

Figure 6: **Joint optimization of system layouts.** PP/DP/TP: pipeline/data/tensor parallelism; Replica 1/2: data-parallel replicas. Ratios compare task performance in (a) and request throughput in (c) with their references.

	GPT-6 Astra	GPT-5.6 Sol	Claude Opus 5	Kimi K3	DeepSeek-V4-Pro	GLM-5.3
Offline	1.01	0.99	1.36	2.31	2.07	1.00
Saturation	1.07	1.04	1.05	1.00	0.98	1.00
TTFT	1.05	1.03	1.01	1.00	1.03	1.01
SLO	1.04	1.06	0.97	0.98	1.01	1.05
Mixed	1.01	1.02	1.03	1.00	1.01	0.99
Repeated	1.02	1.04	1.01	1.00	0.99	1.02
Other 5	1.04	1.04	1.01	1.00	1.00	1.02
Overall	**1.03**	**1.03**	**1.06**	**1.15**	**1.13†**	**1.01**

Task-relative ratios (×)

Figure 7: **S09: 6 serving workloads.** Rows separate the workload gains; † marks an unmet performance requirement.

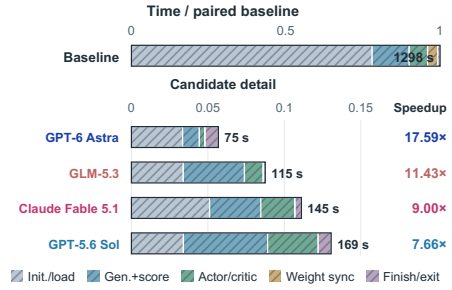

Figure 8: **R03: PPO time by phase.** Paired profiling reruns; each candidate uses its own baseline. The 2 panels use different scales.

For repeated-context serving (S06), GPT-6 Astra batches token rows in its QKV normalization and rotary-embedding kernel, keeps sequence metadata on the host, and reuses a graph-update worker. The system improves shared-prefix and branching-tree throughput, reaching 1.09× overall versus 1.01× for the next-best agent (Appendix F.4). Across the 9 Implementation Optimization serving tasks, edits span scheduling, caches, graph handling, wrappers, and kernels (Figure 5).

Agents change how work is divided and scheduled across devices. On 4 NPUs, the valid Qwen3-8B trainers (T04) all use PP2×DP2 but choose different layer splits and virtual pipeline stages (Figure 6a). For GRPO (R02), shared-device schedules serialize or prefetch rollout work. GPT-6 Astra instead separates actor and rollout devices and overlaps rollout with updates within the allowed policy lag (Figure 6b). The illustrated systems achieve 1.81–5.99× task speedups. For Qwen3-14B serving (S10), Claude Fable 5.1's 2 TP1 replicas are faster on mixed requests, but GPT-6 Astra's TP2 model is faster on balanced requests (Figure 6c).

Serving gains depend on the request mix. Across S09's 6 workloads, Kimi K3's gains concentrate in offline serving, while GPT-5.6 Sol and GPT-6 Astra improve all 5 non-offline workloads (Figure 7).

PPO gains combine faster setup and shutdown with faster online updates. For the complete 8-update R03 job, paired profiling reruns show 2.41–14.42× speedups in the online training loop across the 4 valid systems (Figure 8). Each iteration includes response generation, scoring, actor/critic updates, and policy synchronization; Appendix F.5 reports phase times.

4.3 Adding model support across training, RL, and serving

For Model Adaptation, 8 agents succeed in training, all 9 in serving, and 3 in RL. These tasks add LFM2.5-1.2B Base for training and Instruct for RL and serving (Figure 9).

Valid trainers add model layers and checkpoint conversion, while the failed adaptation does not match the reference loss and global gradient norm. All serving implementations maintain state across prefill and decode, reaching 1.13–1.44×.

RL also requires correct policy updates and transfer of updated weights to rollout. For Gemini 3.7 Flash, a forward pass succeeds, but an unrecognized convolution-parameter key blocks rollout weight loading. Three other agents fail same-input log-probability and sampled-gradient checks at packed-document boundaries. The 3 valid RL adaptations complete both update computation and weight transfer.

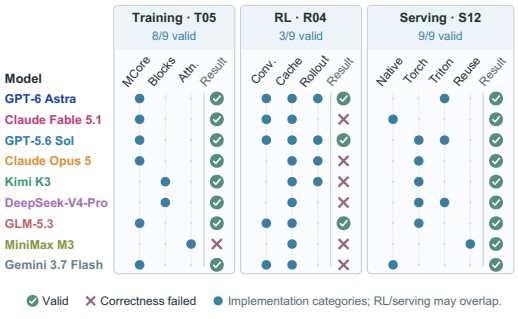

Figure 9: **Adaptation routes and outcomes.** Dots denote implementation choices; Result marks validity or failure. Categories may overlap (Appendix F.1).

4.4 Open Build reveals alternative complete-system designs

One training request yields three effective parallel designs. On the same 2-NPU Qwen3-4B task (T06), agents shard tensors, pipeline layers, or shard optimizer state (Figure 10a). All 5 valid trainers exceed 2× and restore state for update 31 in a fresh process. GPT-6 Astra reaches 4.82× over the evaluated MindSpeed-LLM reference (TP2, microbatch size 1, full-layer activation recomputation). The optimizer-sharded trainers partition FP32 master weights and Adam moments while retaining full BF16 models and full-sized FP32 gradient buffers. GPT-6 Astra pairs bucketed reduce-scatter/all-gather with an AscendC AdamW kernel (Appendix F.2).

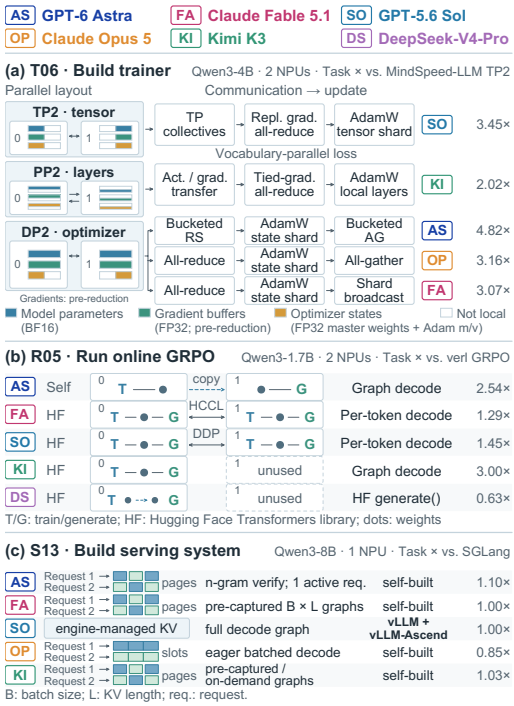

Figure 10: **Open Build designs and task speedups.** Each panel names its reference. T06 shows logical state extents, not memory sizes; DP gradients are pre-reduction. RS/AG: reduce-scatter/all-gather; Act./grad.: activations/gradients. In R05, dots denote weights and double arrows synchronization.

RL builds differ in model sharing, device placement, and decoding. The Open Build RL task (R05) requires an online GRPO training system for Qwen3-1.7B. Of the 5 valid systems, 4 reuse Hugging Face Transformers models but organize training and rollout differently (Figure 10b).

Serving builds differ in request-state management and decode execution. S13 submissions use paged or slot-based KV state with graph or eager decode; agents build new engines or adapt vLLM/vLLM-Ascend (Figure 10c). GPT-6 Astra's service verifies n-gram proposals against the target model. Against the recommended SGLang reference, the 5 valid systems range from $0.85\times$ to $1.10\times$ on the combined throughput, latency, and SLO objectives.

4.5 System development practices and delivery limits

Agents reuse code from individual operators to complete frameworks. GPT-5.6 Sol builds a serving system around vLLM/vLLM-Ascend (S13) and reuses SGLang's position-cache operator to improve an existing service's throughput (S07). When adding support for LFM training (T05), Kimi K3 finds no matching implementation in the upstream code it searches, then adds model blocks and checkpoint conversion around MCore (Appendix G.3).

Measurements guide host work and data movement. For its serving system (S13), GPT-6 Astra times full decode separately from graph replay, then replaces per-item tensor metadata writes with host-array writes. Batch-64 decode time falls from 23.71 to 20.83 ms in the next development probe; the submitted service retains these host-array writes. While building a trainer (T06), GPT-5.6 Sol measures transfer performance in a single trial, observes faster transfers with pinned memory, and retains pinned buffers for checkpoint state.

Agents select execution paths by input size. When optimizing SGLang serving (S07), GPT-6 Astra observes a 2.52% decode slowdown in a development test, then restores the original allocator and raises the QKV tiling threshold from 32 to 128 tokens. On the same task, GPT-5.6 Sol enables position-cached preprocessing above a measured size threshold. Both agents keep the original path for smaller inputs (Appendix G).

Full-system tests uncover integration and recovery errors. GPT-5.6 Sol's trainer (T06) fails an end-to-end training test at a coordinate lookup for a 1-dimensional parameter. After the lookup fix, the trainer passes development checks, including resuming training in a fresh process; independent evaluation confirms the submitted trainer's restart behavior. Appendix G gives the full trainer trace and a restart case from online GRPO training (R05).

Agents build alternative working systems, but delivery remains uneven when execution, updates, and recovery must work together. RL adaptations expose gaps in packed-input computation and rollout weight loading; serving gains in one request regime need not extend to others. PPO profiling separates setup/shutdown gains from online-loop gains. Together, these results distinguish local improvements from complete-system integration.

5 Limitations and Conclusion

AscendInfraBench samples representative models and workloads on a fixed Ascend software stack. Although it already includes multi-device tasks that capture several realistic infrastructure challenges, it does not yet evaluate agents at large-scale production clusters (e.g., over 100 devices). Extending the benchmark to larger clusters and more diverse and evolving model architectures is therefore a natural next step. Future work could also explore long-horizon engineering tasks and sustained system maintenance.

We introduce a capability-oriented task construction method and AscendInfraBench for evaluating coding agents across training, RL, and serving. The method specifies required work, permitted changes, and performance objectives without prescribing a single implementation. Tasks span optimization, model adaptation, and complete-system construction, with independent evaluation of numerical correctness, state handling, and performance, supported by source review. Our 9-agent study reveals different strengths across stages and scopes; GPT-6 Astra delivers valid implementations on all 24 tasks at $1.63\times$ task-weighted geometric-mean speedup over task baselines. Valid solutions to the same tasks differ in execution, model integration, and system organization. Development trajectories show how code reuse, performance measurements, and end-to-end testing shape these choices, providing an empirical basis for improving coding agents for LLM infrastructure engineering.

AI USE STATEMENT

We used large language models to find related work and check references, draft sections of the paper, and revise and polish the writing. LLMs also supported research execution by helping develop analysis scripts and analyze submitted implementations and experimental results, including source review. The evaluated coding agents generated the task submissions studied in the benchmark. The authors reviewed the AI-assisted writing and analyses, verified the reported results, and made the final evaluation decisions. The authors take responsibility for the paper and its accompanying artifacts.

REPRODUCIBILITY STATEMENT

Appendix A links to the benchmark code and website, with de-identified development trajectories and submitted artifacts for all 216 runs. Task definitions, environments, and evaluation checks appear in Appendices B–D. Appendices D.6 and E reproduce the source-review and task prompts; Appendix G.1 follows the resulting trainer development, and Appendix I reports measurement repeatability.

REFERENCES

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Mubingxu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, Dongyang Tao, Feidiao Yang, Chang-Dong Wang, Yutong Lu, Weicheng Xue, Bin Zhou, and Yonghong Tian. Ascendkernelgen: Llm-driven kernel generation for npus. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30693–30718, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. Mle-bench: Evaluating machine learning agents on machine learning engineering. *International Conference on Learning Representations*, 2025:50466–50494, 2025.

Wanyi Chen, Xiao Yang, Xu Yang, Tianming Sha, Qizheng Li, Zhuo Wang, Bowen Xian, Fang Kong, Weiqing Liu, and Jiang Bian. Agent^2 RL-Bench: Can llm agents engineer agentic rl post-training? arXiv preprint arXiv:2604.10547, 2026.

Yizhe Chi, Wenyi Li, Deyao Hong, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Ai4ai-bench: Benchmarking llm agents in algorithmic design for recursive self-improvement. arXiv preprint arXiv:2608.20318, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? arXiv preprint arXiv:2509.16941, 2025.

Eric Ding, Byungsoo Oh, Bhaskar Kataria, Kaiwen Guo, Jelena Gvero, Abhishek Vijaya Kumar, Arjun Devraj, Lindsey Bowen, Atharv Sonwane, Emaad Manzoor, and Rachee Singh. Ccl-bench 1.0: A trace-based benchmark for llm infrastructure. arXiv preprint arXiv:2605.06544, 2026.

Xue-Jian Gao, Deng Pan, Yueming Su, Jiasheng Li, Bin Du, Fengming Zhu, Chengdi Ma, Junyi Fan, Qichen Liao, Chengqiu Hu, Xinxian Chen, Lingchao Zheng, Jun Li, Jiwei Yang, and Yuwei Fan. CANN Bench: Benchmarking Agent Generated Kernels against Real NPU and Algorithmic Limits. arXiv preprint arXiv:2607.20518, 2026a.

Yuan Gao, Zeren Yang, Junnan Li, Shawn (Wanxiang) Zhong, Ahmed Dajani, Mai Zheng, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Infrabench: Evaluating infrastructure agents across layers, lifecycle, and risk. arXiv preprint arXiv:2608.11234, 2026b.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, ZhiJie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. Swe-perf: Can language models optimize code performance on real-world repositories? In *Forty-Third International Conference on Machine Learning*, 2026.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 20271–20309. PMLR, 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? *International Conference on Learning Representations*, 2024:54107–54157, 2024.

Keisuke Kamahori, Shihang Li, Simon Peter, and Baris Kasikci. Vibeserve: Can ai agents build bespoke llm serving systems? arXiv preprint arXiv:2605.06068, 2026.

Rishyanth Kondra, Sanket Mhatre, Akshit Kumar, Evan Chu, Bilal Bakht Ahmad, Ayush Nangia, Rajan Agarwal, Arpan Dasgupta, Animesh Sinha, Bhuvanesh Sridharan, Krupa Dave, Brendan Graham, Guanyu Song, Anirudh Rahul, Wei Hern Lim, Abishek Thangamuthu, Ramneet Singh, Danna Liu, Navid Pour, Calvin Chen, and Justus Mattern. Frontierswe v2: Testing coding agents at the frontier of software engineering, 2026. URL https://www.frontierswe.com. Blog: https://www.frontierswe.com/blog/v2.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0229-7. doi: 10.1145/3600006.3613165.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. Swe-fficiency: Can language models optimize real-world repositories on real workloads? In *Forty-Third International Conference on Machine Learning*, 2026.

Fanqing Meng, Lingxiao Du, Qiguang Chen, Ziqi Zhao, Haocheng Lu, Mengkang Hu, and Michael Qizhe Shieh. RSIBench-Data: Benchmarking Data-Centric Research for Recursive Self-Improvement. arXiv preprint arXiv:2607.25886, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv preprint arXiv:2601.11868, 2026.

Ayush Nangia, Shikhar Mishra, Aman Gokrani, and Paras Chopra. Iso-bench: Can coding agents optimize real-world inference workloads? arXiv preprint arXiv:2602.19594, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 47356–47415. PMLR, 2025.

Φ-Bench Contributors. Φ-Bench: Can Large Language Models Engineer the Infrastructure That Powers Them? Project website, 2026. URL https://faibench.org/.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, Nathanael Bosch, Eli Meril, Albert Steppi, Arman Zharmagambetov, Fangzhao Zhang, David Pérez-Piñeiro, Alberto Mercurio, Ni Zhan,

Talor Abramovich, Kilian Lieret, Hanlin Zhang, Shirley Huang, Matthias Bethge, and Ofir Press. Algotune: Can language models speed up general-purpose numerical programs? In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-3585.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. doi: 10.1109/SC41405.2020.00024.

Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. PostTrainBench: Can LLM agents automate LLM post-training? arXiv preprint arXiv:2603.08640, 2026. URL https://arxiv.org/abs/2603.08640.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 979-8-4007-1196-1. doi: 10.1145/3689031.3696075.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. Gso: Challenging software optimization tasks for evaluating swe-agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-5054.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2020.

Natalia Tarasova, Enrique Balp-Straffon, Aleksei Iancheruk, Yevhenii Sielskyi, Nikita Kozodoi, Liam H. Byrne, Jack Butler, Dayuan Jiang, Marcin Czelej, Andrew Ang, Yash Shah, Roi Blanco, and Sergei Ivanov. Swe-infrabench: Evaluating language models on cloud infrastructure code. arXiv preprint arXiv:2606.05249, 2026.

Sijie Wang, Kai Zhao, Wee Peng Tay, Shuo Zhang, Chengwen Liu, Quanjiang Guo, Ren Junhao, Xin Li, Heng Lian, Jingdi Lei, Rui She, Huacan Wang, and Ronghao Chen. Livecannbench: Benchmark swe ai coding for ascend cann. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 22788–22803, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A Multi-Platform Benchmark for Kernel Generation. arXiv preprint arXiv:2507.17773, 2025.

Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua M. Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. Re-bench: Evaluating frontier ai r&d capabilities of language model agents against human experts. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 66772–66832. PMLR, 2025.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. arXiv preprint arXiv:2601.00227, 2026.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Programbench: Can language models rebuild programs from scratch? arXiv preprint arXiv:2605.03546, 2026.

Jehyeok Yeon, Ben Rank, and Maksym Andriushchenko. Inferencebench: A benchmark for open-ended llm inference optimization by ai agents. arXiv preprint arXiv:2607.20468, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H. Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583, 2024. doi: 10.52202/079017-2000.

A Benchmark Website and Code

Benchmark website. The website brings together task results, de-identified development trajectories, and submitted code artifacts for all 216 runs across 24 tasks and 9 model–harness agent configurations. Readers can browse by task or configuration and inspect the development record and final code changes for each run: https://ascendinfrabench.github.io/

Benchmark code and task specifications. The repository provides the benchmark code and complete task specifications, including prompts, workloads, and development and evaluation settings: https://anonymous.4open.science/r/AscendInfraBench/.

B Tasks and Environments

The 24 tasks run on Ascend 910B NPUs using the vendor images in Table 3. Each evaluation rebuilds the captured submission in the task's recorded software environment, independently of the agent's development environment. Open Build uses the listed base images with their preinstalled frameworks removed; candidates supply their own dependencies, preparation, and launch procedures. The harness versions in Table 3 are recorded in the website archive (Appendix A); each applies to all 24 runs of its configuration.

Table 3: Software, hardware, and coding-agent harnesses used in the experiments. Vendor images are hosted at `quay.io/ascend`; SHA-256 digests are abbreviated to 12 characters.

(a) Task environments

Environment	Base vendor image	Hardware
MindSpeed-LLM	`mindspeed-llm@8bdf32ace55a`	910B
Open Build training	`mindspeed-llm@8bdf32ace55a`	910B
verl	`verl@cf7709b22a25`	910B
Open Build RL	`verl@cf7709b22a25`	910B
SGLang	`sglang@35852210819a`	910B
vLLM-Ascend	`vllm-ascend@471744200c5d`	910B
Open Build serving	`sglang@35852210819a`	910B

(b) Coding-agent harnesses

Configuration	Harness	Version	Effort
GPT-6 Astra	Codex	`0.154.0`	max
Claude Fable 5.1	Claude Code	`2.1.267`	max
GPT-5.6 Sol	Codex	`0.149.1`	max
Claude Opus 5	Claude Code	`2.1.241`	max
Kimi K3	Kimi Code	`0.38.0`	max
DeepSeek-V4-Pro	DeepSeek	`0.1.1-rc.2`	max
GLM-5.3	Claude Code	`2.1.241`	max
MiniMax M3	Claude Code	`2.1.241`	max
Gemini 3.7 Flash	Antigravity	`1.1.19`	high

B.1 Task catalog and shared settings

Table 4 lists every task, its engineering scope, model, framework, and device budget. T01–T03 and T06 share Qwen3-4B and 2 NPUs but vary the execution plan, packed-document workload, and responsibility for building the trainer. S07/S08 use identical Qwen3-8B saturation requests in different frameworks. S01–S06 and S09 share per-regime workloads and configurations; S09 combines the 6 regimes in one service. T05/R04/S12 connect the LFM2.5 architecture family to training checkpoints, policy refresh, and serving state.

Table 4: The 24-task portfolio. Scope: I = Implementation Optimization, J = Joint Optimization, A = Model Adaptation, O = Open Build. The final column gives the NPU allocation.

Code	Scope	Task	# NPUs
T01	I	MindSpeed · Qwen3-4B · DP2 Training	2
T02	I	MindSpeed · Qwen3-4B · TP2 Training	2
T03	I	MindSpeed · Qwen3-4B · DP2 Packed Training	2
T04	J	MindSpeed · Qwen3-8B-Base · Training	4
T05	A	MindSpeed · LFM2.5-1.2B-Base · MCore Training	1
T06	O	Open Build · Qwen3-4B · Training	2
R01	I	verl · Qwen3-0.6B · GRPO/RLVR (Colocated)	1
R02	J	verl · Qwen3-1.7B · GRPO/RLVR	2
R03	J	verl · Qwen3-1.7B · PPO (RM/Critic)	2
R04	A	verl · LFM2.5-1.2B-Instruct · GRPO/RLVR (Colocated)	2
R05	O	Open Build · Qwen3-1.7B · GRPO/RLVR	2
S01	I	SGLang · Qwen3-0.6B · Offline Throughput	1
S02	I	SGLang · Qwen3-0.6B · Online Saturation Throughput	1
S03	I	SGLang · Qwen3-0.6B · Online TTFT	1
S04	I	SGLang · Qwen3-0.6B · Online SLO Goodput	1
S05	I	SGLang · Qwen3-0.6B · Mixed Interactivity	1
S06	I	SGLang · Qwen3-0.6B · Repeated Context	1
S07	I	SGLang · Qwen3-8B · Online Saturation Throughput	1
S08	I	vLLM-Ascend · Qwen3-8B · Online Saturation Throughput	1
S09	I	SGLang · Qwen3-0.6B · Serving Generalist (6 Regimes)	1
S10	J	SGLang · Qwen3-14B · Serving	2
S11	J	vLLM-Ascend · Qwen3-30B-A3B · MoE Serving	2
S12	A	SGLang · LFM2.5-1.2B-Instruct · Serving	1
S13	O	Open Build · Qwen3-8B · Serving	1

Development usage. Table 2 reports per-run median input tokens, output tokens, and tool calls over all submissions. Input includes cached input, and Claude Code usage includes delegated models. Each column is aggregated independently. Per-run values are retained in the companion data archive.

B.2 Criteria for the related-work comparison

Table 1 classifies engineering requests by their starting point, deliverable, and evaluation target. A checkmark denotes a corresponding setting in the cited release; a dash denotes no corresponding setting under these criteria.

Stages and scopes. Training covers language-model training and fine-tuning. RL couples generated experience, rewards, and policy updates. Serving covers complete inference and generation, including offline execution and model/backend integration. Optimization improves a supplied working implementation or execution plan; Model Adaptation adds specified model support; Open Build supplies a complete runnable system from an empty or incomplete starter, with component reuse permitted.

Relation to prior settings. ISO-Bench evaluates serving-performance patches; InferenceBench includes deployment and optimization settings; VibeServe studies complete serving construction (Nangia et al., 2026; Yeon et al., 2026; Kamahori et al., 2026). Broader portfolios include training optimization, workflow construction, and model/backend integration (Φ-Bench Contributors, 2026; Kondra et al., 2026). Our comparison distinguishes these complete workflows from isolated kernels or storage components.

C Timing and Scoring

C.1 Timing and preparation are part of the task contract

Each measurement uses 4 alternating baseline/candidate pairs on the same physical NPU allocation. Workloads, timing boundaries, and preparation follow the task contract (Table 5). PPO includes process startup; Open Build GRPO measures online execution after preparation.

Table 5: Timing boundaries for representative tasks.

Tasks	Timed work or observation	Outside the timed interval
T04: 4-device training	Primary: receipt of update 10 through process completion, counting the remaining 20 updates. Secondary: complete process duration for all 30 updates.	Native-extension build; separate Correctness and cleanup checks.
T06: Open Build training	Before offline bootstrap through 30 updates, checkpoint commitment, and training-process exit, for each workload profile.	Independent checkpoint verification and fresh-process continuation.
R03: PPO	Native or framework process launch through exit, including startup and 8 actor/critic updates. The framework candidate's live final probe is inside its interval.	Task-side input/reward-model preparation; fresh-restart Correctness; post-exit cleanup.
R05: Open Build GRPO	Execute-process launch through the required online work, final policy refresh/freshness observation, and foreground completion.	Bootstrap, model loading and warmup in preparation; restart Correctness; cleanup.
S02/S03/S05	Request-batch throughput or request latency after service readiness and task-owned warmup.	Service loading; unscored preparation trials; correctness; cleanup. Request state is flushed before each scored case or wave.

For S02, S03, and S05, measurements with warmed compiler caches use 2 preparation passes per implementation before the 4 measured pairs. Compiler caches are isolated by implementation and reused across trials. S02 warms each throughput case before its scored batch; S05 warms the workload before 8 scored waves. S03 repeats the workload until 2 consecutive warmup rounds leave the compiler cache unchanged. Request state is flushed before each scored batch or wave. Preparation is outside the score; compilation during scored requests contributes to the measured time.

The S03 A/A controls isolate compiler caches per trial and use the same warmup convergence rule; the reported task results retain the compiler-cache configurations used for each evaluated implementation. The S04 A/A controls and task measurements both use 4 unscored warmup steps. Appendix I reports the control results.

C.2 Aggregating paired measurements

For throughput and goodput, the paired performance ratio is candidate/baseline; for latency, it is baseline/candidate. Let r_{jk} denote this ratio for objective j in pair k. For training and serving, the aggregate objective ratio s_j is the median across the 4 complete pairs:

$$s_j = \mathrm{median}_{k=1}^{4}\, r_{jk}. \tag{1}$$

For 4 observations, the median averages the middle 2 sorted ratios. S05 computes each run's p90 over the exact short requests pooled across its waves, then takes the median of the 4 baseline/candidate ratios. These reductions operate at different levels.

All 5 RL tasks use pooled accepted response-token rates. Let W_{jk}^{B} and W_{jk}^{C} be accepted response-token counts, and let T_{jk}^{B} and T_{jk}^{C} be elapsed time, on the baseline and candidate sides of pair k. Their pooled rates are

$$v_j^{\mathrm{B}} = \frac{\sum_{k=1}^{4} W_{jk}^{\mathrm{B}}}{\sum_{k=1}^{4} T_{jk}^{\mathrm{B}}}, \qquad v_j^{\mathrm{C}} = \frac{\sum_{k=1}^{4} W_{jk}^{\mathrm{C}}}{\sum_{k=1}^{4} T_{jk}^{\mathrm{C}}}, \qquad s_j^{\mathrm{pool}} = \frac{v_j^{\mathrm{C}}}{v_j^{\mathrm{B}}}. \tag{2}$$

Pooling compares total accepted work per unit elapsed time. For RL, we set $s_j = s_j^{\mathrm{pool}}$ in the task-level weighted geometric mean. Task speedup is $S = \prod_j s_j^{w_j}$, with positive objective weights satisfying $\sum_j w_j = 1$. Performance requirement checks use their specified objective basis, including auxiliary metrics with 0 score weight.

Table 6: Representative within-task weighting patterns. Exact objectives and gates remain task-specific.

Pattern	Objectives	Weights
Single objective	Latency, throughput, goodput, or accepted response-token rate	1.0
Training	Post-warmup; end-to-end throughput	0.8 / 0.2
Packed training	2 post-warmup and 2 end-to-end profiles	0.4 / 0.4 / 0.1 / 0.1
2-profile serving	2 declared workload profiles	0.5 / 0.5
Open Build serving	Prefill; decode; saturation group; SLO	0.25 per group

C.3 Performance requirements and performance baselines

Table 7 separates per-objective performance requirements from score weights. Each performance requirement uses its specified aggregation rule and can apply to auxiliary objectives with zero score weight. T04, for example, specifies no performance requirement on its objective ratios: its post-warmup completion and complete-process objectives contribute with weights 0.8 and 0.2. Model Adaptation and Open Build use computation, state, and process-lifecycle requirements to assess delivery, with performance baselines supplying the speedup denominators.

Performance coverage counts a task only when the submission is valid, meets every applicable performance requirement, and reaches the displayed task-speedup threshold. Validity and performance coverage are therefore recorded separately.

Table 7: **Per-objective performance requirements.** Score weights and performance requirement checks are separate; auxiliary controls may have 0 score weight. All listed thresholds apply to direction-normalized objective ratios.

Task group	Objectives	Minimum ratio
S01	Both offline profiles	$0.95\times$
S02	All 3 saturation cases	$0.95\times$
S03	Both TTFT profiles	$0.95\times$
S05	Short-request throughput, short p95 end-to-end latency, long maximum end-to-end latency	$0.95\times$
S06	Independent-context throughput control	$0.95\times$
S09	All 15 objectives, including auxiliary controls	$0.95\times$
Other Optimization	No performance requirement on objective ratios	—
Model Adaptation / Open Build	No performance requirement on objective ratios	—

C.4 Aggregation across tasks, stages, and scopes

Table 2 reports conditional geometric-mean speedups with valid counts in subscripts. For configuration a, let V_a be its valid task set and S_{at} the task speedup on task t. The task-weighted mean (Task GM) is

$$G_{a,\text{task}} = \exp\left(\frac{1}{|V_a|}\sum_{t\in V_a}\log S_{at}\right). \tag{3}$$

For a partition into categories $\mathcal{C}$, each category mean G_{ac} uses $V_a \cap c$. The category-balanced Overall is

$$G_{a,\mathcal{C}} = \left(\prod_{c\in\mathcal{C}} G_{ac}\right)^{1/|\mathcal{C}|}. \tag{4}$$

Infrastructure stage uses 3 equally weighted categories: training, RL, and serving (6, 5, and 13 tasks). Engineering scope uses 4 equally weighted categories: Implementation Optimization, Joint Optimization, Model Adaptation, and Open Build (13, 5, 3, and 3 tasks). An Overall is omitted if any category has no valid result. Means are computed from unrounded task speedups; displayed category means are not used as calculation inputs.

Unchanged Optimization baselines contribute $1.00\times$. Only valid results enter these geometric means; the accompanying counts identify the tasks represented.

D Numerical and State Evaluation

Evaluation compares the submitted system's computation with correctness references and tests the next operation that consumes its state: a resumed update, rollout with an updated policy, or a request using cached state. Source review follows the code behind these observations. The checks below specify the reference inputs, numerical tolerances, and state transitions.

D.1 References and development feedback

Table 8 separates correctness references from performance baselines. Optimization supplies a working system that agents can run before and after changes. Model Adaptation and Open Build supply reference outputs or development checkers while the agent builds a solution.

Table 8: Reference origins by task family. Numerical and state checks are detailed below; the performance baseline supplies the speedup denominator.

Tasks	Correctness reference	Performance baseline
T01–T04	Pristine training implementation	Unmodified MindSpeed-LLM and baseline plan
T05	Reviewed, frozen LFM/MCore implementation	Working LFM/MCore adaptation
T06	Trusted numerical values and sampled AdamW recomputation	Task-configured MindSpeed-LLM TP2 trainer
R01–R03	Same-trajectory algorithm and model comparisons	Task-configured verl
R04	RL checks with independent LFM model references	Minimal verl/vLLM LFM adaptation; FSDP2/TP2
R05	Same-trajectory GRPO and policy-state checks	Task-configured verl GRPO
S01–S11	Independent Transformers token scoring and request checks	Supplied SGLang or vLLM-Ascend configuration
S12	Independent LFM inference and state checks	Minimal SGLang LFM adaptation
S13	Task output and HTTP oracles	Task-configured SGLang

T05 publishes 3 reference losses with absolute tolerance 0.005 for its 4-observation correctness run. T06 supplies a separate deterministic development input with loss, gradient-norm, and tensor references; evaluation retains a small numerical holdout. S12 publishes expected output digests, and S13 exposes HTTP response checks. R04/R05 publish most reward-verifier answers and retain a small answer holdout; advantages and losses are recomputed from the responses and policy actually used by the candidate.

Table 9 complements these task-family references with a stage-by-scope view of all three evaluation gates. Its Optimization column groups Implementation Optimization and Joint Optimization; the result aggregates continue to distinguish all four scopes.

D.2 Training and serving comparisons

Training. Candidate and reference runs share the model, initialization, ordered inputs, and task conditions. At each checked update, the evaluator compares loss and pre-clipping global gradient norm. At least 2 trusted executions establish reference variability; tolerances also account for rounding in recorded observations. The correctness runs cover 4 updates for T01/T02/T03/T05, 3 for T04, and a separate 3-update many-boundary workload for T03. T06 covers both 30-update profiles and fresh-process update 31. Restart tests load model and optimizer state and execute the next update.

Serving. For the 12 Optimization and Model Adaptation tasks, an independent Transformers execution scores every generated token from the same model, input, and preceding tokens, without a KV cache. Let ℓ_i^{C} be the service's log probability for token y_i, ℓ_i^{R} the reference score for that token,

Table 9: **Three evaluation gates across infrastructure stages and engineering scopes.** C: Correctness; M: Measurement (performance baseline shown); R: Source review. Task-specific numerical and state checks are detailed in the accompanying text.

Stage	Optimization (Implementation + Joint)	Model Adaptation	Open Build
Training	**C:** Compare pristine MindSpeed-LLM loss and pre-clipping global gradient norm; test packed-document boundaries and resumed updates. **M:** Unmodified MindSpeed-LLM and its baseline execution plan. **R:** Trace forward, backward, updates, and saved state; check allowed plan changes.	**C:** Compare loss and norm with the frozen LFM/MCore reference; test weight conversion and resumed updates. **M:** Frozen working LFM/MCore adaptation. **R:** Trace model layers, short convolution, weight mapping, and their training path.	**C:** Compare trusted reference loss/norm; recompute sampled AdamW transitions and test fresh-process update 31. **M:** Task-configured MindSpeed-LLM TP2 trainer on both workloads. **R:** Trace full-model computation, global reductions, optimizer state, and recovery.
RL	**C:** Recompute advantages/returns and losses on recorded trajectories; compare log probabilities or critic values; test policy refresh and restart. **M:** Task-configured verl performance baseline and its timing interval. **R:** Trace rollout, rewards, actor/critic updates, weight consumption, and allowed policy lag.	**C:** Apply RL checks with independent LFM model references; test packed-document boundaries and attention/convolution state reuse. **M:** Minimum working verl/vLLM LFM adaptation; fixed FSDP2/TP2 plan. **R:** Trace convolution/cache layout, model arithmetic, and weight installation.	**C:** Recompute advantages and losses from recorded trajectories; compare actor/rollout log probabilities; test updates, policy refresh, and restart. **M:** Task-configured verl GRPO performance baseline. **R:** Trace trajectory consumption, actual updates, and subsequent generation in the delivered system.
Serving	**C:** Compare token log probabilities and greedy choices with Transformers; test outputs, cache reuse, and request handling. **M:** Task-provided SGLang or vLLM-Ascend and its baseline configuration. **R:** Trace model execution, scheduling, KV state, and timed work.	**C:** Compare token scores with independent LFM inference; test prefill, decode, and reuse of request state. **M:** Minimum working SGLang LFM adaptation. **R:** Trace attention/convolution state layouts and NPU operator calls.	**C:** Use task output and HTTP oracles for generation, streaming, termination, accounting, and concurrency. **M:** Task-configured SGLang performance baseline. **R:** Trace actual Qwen3 inference, request state, and the complete service lifecycle.

and $\ell_i^{\mathrm{R},*}$ the reference's highest token score. Both checks must hold:

$$\max_i \left|\ell_i^{\mathrm{C}} - \ell_i^{\mathrm{R}}\right| \le \epsilon_{\log p}, \qquad \max_i \left(\ell_i^{\mathrm{R},*} - \ell_i^{\mathrm{R}}\right) \le \epsilon_{\mathrm{greedy}}. \tag{5}$$

Bounds are established for each model/runtime using trusted repeats, held-out requests, and wrong-token controls. Checks preserve published requests, order, generation settings, concurrency, and output assertions. Repeated-context tasks exercise both cache configurations; S09 exercises all 6 regimes. Open Build serving uses its output and HTTP oracle for generation, streaming, termination, token accounting, and concurrency.

D.3 RL NUMERICAL REPLAY

RL replay records generated tokens, masks, rewards, old-policy log probabilities, and pre-update policy state. The evaluator recomputes group advantages or GAE returns and policy/value losses from those operands. Model comparisons use actor selected-token log probabilities and PPO critic values. For supported model/precision paths, calibrated replay at updates 1, 8, and fresh-process update 9 compares loss-input derivatives, sampled parameter gradients, and optimizer updates. Small derivative arrays are compared in full; parameter gradients use declared sample coordinates. State checks compare parameters used by generation and the next fixed-batch update after continuous execution or restoration.

D.4 OPEN BUILD TRAINING: FROM GRADIENTS TO PERSISTENT STATE

T06 trains the full 36-layer Qwen3-4B model from a partition-invariant deterministic initialization. Each workload processes the supplied global records in order for 30 updates and 983,040 tokens. The long-sequence workload uses sequence length 2,048 and global batch 16; the update-heavy workload uses length 512 and batch 64. The loss is the sum of masked token negative log-likelihoods divided by the global mask count, using the supplied pre-shifted labels and FP32 softmax. Model storage is BF16; gradient accumulation, master parameters, and optimizer moments are FP32.

Recomputing the optimizer update. Let g_t be a pre-clipping gradient and G_t the global L2 norm over unique canonical parameters. For each selected coordinate, the checker reads the gradient and previous optimizer state and recomputes

$$\begin{aligned} \widetilde{g}_t &= g_t \min\{1, 1/(G_t + 10^{-6})\}, \\ m_t &= 0.9 m_{t-1} + 0.1\widetilde{g}_t, \qquad v_t = 0.95 v_{t-1} + 0.05\widetilde{g}_t^2, \\ p_t &= p_{t-1} - \eta_t \left(\frac{m_t/(1-0.9^t)}{\sqrt{v_t/(1-0.95^t)} + 10^{-8}} + \lambda_p p_{t-1} \right). \end{aligned}$$

The cosine learning rate decreases from 1.25×10^{-6} to 1.25×10^{-7} over updates 1–30. Weight decay is $\lambda_p = 0.1$ for matrix parameters and 0 for normalization weights. The BF16 model value must equal ties-to-even rounding of the updated FP32 master value. The implementation evaluates these formulas with explicit FP32 rounding at each arithmetic stage. The observed moments, master parameter, and BF16 model value are compared with this recomputation.

Numerical reference feedback. The separate public development input provides reference loss and global gradient norm at every update 1–31, plus selected logits, BF16 parameters, and FP32 master parameters at updates 2, 11, 23, and 31. Its absolute tolerances are 0.0012717247 for loss, 0.0019385990 for global norm, 0.15234375 for logits, 0 for selected BF16 parameters, and 1.1920929×10^{-6} for selected master parameters; relative tolerances are 0. Raw gradient and moment projections are also available for comparison; transition acceptance follows the optimizer equations above.

Tensor and coordinate coverage. Table 11 lists the exact parameter selectors. The same coordinates apply to both workload profiles. The 32 transition checks comprise 4 mathematical checks at 4 updates in each of 2 workloads.

Agents receive the development input, selectors, and reference feedback before implementation. Final evaluation runs the submitted artifact on a separate input. Its selectors cover early, middle, and final layers and the output normalization, and follow those coordinates through checkpoint restoration.

Table 10: T06 optimizer-transition, trajectory, and continuation checks. A transition bundle follows 1 declared parameter coordinate through its previous state, current gradient, and updated state.

Check	Acceptance rule	Coverage
First/second moments	At most 8 FP32 ULPs, or absolute error at most $8 \cdot 2^{-24}$ times the sum of the absolute update terms	Updates 1, 15, 30, and fresh 31; both workloads
FP32 master update	At most 32 FP32 ULPs from the recomputed AdamW transition	Same transition bundles
BF16 model update	Exact BF16 rounding of the observed master parameter	Same transition bundles
Trajectory and data state	30 ordered committed-update records; matching profile and selector identities; checkpoint at update 30	Every committed update
Fresh-process continuation	Reload the read-only checkpoint and compute the would-be update 31 without committing it; apply the same transition rules	New process, both workloads

Table 11: **T06 parameter selectors.** Layer indices and flattened tensor indices are 0-based. Each coordinate identifies the corresponding model value, master value, gradient, and moments; logits use separately specified batch, sequence, and vocabulary coordinates.

Input	Update	Parameter tensor	Flat index
Evaluation	1	Layer 0 attention Q-projection weight	17
Evaluation	15	Layer 17 MLP down-projection weight	123,457
Evaluation	30	Layer 35 attention Q-normalization weight	37
Evaluation	31	Final model normalization weight	2,559
Development	2	Layer 1 attention K-projection weight	123
Development	11	Layer 9 MLP up-projection weight	34,567
Development	23	Layer 27 input normalization weight	901
Development	31	Layer 34 attention output-projection weight	765,432

D.5 RL: ACCEPTED TRAJECTORIES AND FRESH POLICIES

Open Build GRPO. R05 performs 8 full-parameter AdamW updates, each using 8 prompts and 4 responses per prompt. The host checks all 256 trajectory identities, their sampling seeds, token/text correspondence, reward-service calls, and response-token counts. For every group it recomputes $A_i = (r_i - \bar{r})/(s_r + 10^{-6})$, where s_r is the sample standard deviation; a constant-reward group has 0 advantages. Recorded advantages must match within 10^{-5} absolute error.

An additional fixed-trajectory algorithm probe supplies rewards and old/current token log probabilities. The checker independently computes group advantages, the token-mean clipped policy loss with ratio bounds $[0.8, 1.2]$, approximate KL $\mathrm{mean}(e^{-\Delta} + \Delta - 1)$ for $\Delta = \log \pi_{\mathrm{current}} - \log \pi_{\mathrm{old}}$, and the fraction of ratios outside the clipping interval. Every returned value must match within 10^{-6} absolute error. The prescribed optimizer uses learning rate 10^{-6}, weight decay 0.01, moments $(0.9, 0.999)$, FP32 state, and global gradient clipping at 1; the contract uses 1 policy epoch, token-mean loss, and no critic or reference-KL term.

Policy-version checks cover all 8 updates and permit a trajectory lag of at most 1 update. At updates 1 and 8, actor and rollout log probabilities are compared at sampled response positions with absolute tolerance 0.5; an old-log-probability check covers update 1. Fresh-process restart loads the checkpoint at update 8, generates 4 new responses, commits update 9, and repeats reward, advantage, and policy-freshness checks.

PPO. R03 specifies GAE with $\gamma = 1$ and $\lambda = 0.95$, a reference-policy KL coefficient of 0.001 in token rewards, actor clipping at 0.2, value clipping at 0.5, and 1 unshuffled actor/critic epoch per update. AdamW learning rates are 10^{-6} for the actor and 10^{-5} for the critic. The actor and critic each commit 8 updates over 256 trajectories, while the reference policy and reward model remain frozen.

The host verifies the complete update/prompt trajectory set and tests policy handoff with a selected continuation. Live actor and rollout log probabilities at its first, middle, and final token positions must agree within 10^{-4}. A separate framework process loads the portable actor checkpoint and recomputes every continuation-token log probability; these must agree with the live actor within 0.02. Its maximum change from the initial actor must be at least 10^{-6}. A fresh runtime resumes both actor and critic at update 9 and repeats the live-policy probe. For GPT-5.6 Sol, GLM-5.3, GPT-6 Astra, and Claude Fable 5.1, the final probe and portable export read the updated FP32 master parameters. The comparisons above therefore check that representation and its checkpoint/restart correspondence. The optimizer also installs the BF16 model values used by training and rollout; Section D.3 describes the numerical replay checks. Kimi K3 and Claude Fable 5.1 use PPO without advantage whitening, which the task permits.

D.6 SOURCE-REVIEW PROTOCOL

Source review traces model output, loss, backward computation, optimizer writes, policy installation, checkpoint loading, and timed work against the permitted scope. GPT-6 Astra assists the analysis; the authors decide final outcomes.

Review pipeline. The review tooling exports a packet containing the task contract and captured source, plus explicitly selected evaluation reports and supporting files. A manifest records hashes for these inputs. The reviewer follows entrypoints and called implementations to check four aspects: required computation, permitted specialization, state continuity, and timing boundaries. This includes tracing checkpoint loading and policy transfer to the writes that install state for subsequent execution. Each finding cites source paths and line ranges and explains how the code relates to a task requirement. The reviewer returns `accept`, `reject`, or `needs_evidence`, separating demonstrated violations from specific unresolved questions. Before importing the review, the host checks input identity, coverage of all four criteria, cited paths and line bounds, and consistency between findings and verdict. Source-review findings remain separate from correctness and measurement results.

In Sol's T06 trainer, reduced FP32 gradients feed AdamW, and saved masters and moments are installed into live tensors before the fresh update. In Kimi's R05 engine, generated trajectories enter the clipped-loss update, and the updated BF16 storage is used by rollout. In Sol's R03 system, rollout reads the updated actor object and checkpoint loading restores actor/critic parameters and optimizer states. These traces connect logged observations to the tensors used by subsequent execution. Another evaluated submission illustrates the distinction: Gemini's R05 passes BF16 parameters directly to AdamW, leaving its moment tensors in BF16 despite an output field reporting FP32; source review rejects the state-precision violation.

Source-review prompt. We reproduce the complete source-review prompt used in our experiments. The review packet contains this text in `REVIEW_REQUEST.md`, the output template `decision.template.json`, and the task-specific inputs. Only line wrapping is changed for print.

```
# Independent source acceptance review

Review this packet in a fresh session, without prior verdicts or other
submissions. Use the supplied task contract, candidate source and host
evidence. Do not execute candidate code. Expanded archives are readable
views; original payloads are retained.

Review responsibility: inspect semantic gaps in the implemented
computation and state transitions. Accept the supplied, candidate-bound
host gate results for candidate identity, captured edit scope,
numerical checks, performance floors and score reduction. Do not re-run
these decisions, recalculate performance, or reject source because a
performance floor fails. Source acceptance and performance eligibility
are separate decisions. Do not introduce new tolerances, bitwise/token
equality requirements, or additional validation workloads beyond the
task contract. Host reports are optional. Their absence alone is not a
source violation or a reason to block a source-only review. Request
```

```
evidence only for a specific unresolved computation or state question
relevant to the task contract.

Cover all four criteria:
1. assigned_computation: preserve model/data, real computation, token
   work, updates, optimizer rules and required numerical behavior.
2. permitted_specialization: no omitted work, fabricated outputs,
   model/data substitution or prohibited precomputation. Trust
   host-validated captured scope; inspect executable attempts to bypass
   it rather than re-auditing file allowlists.
3. state_continuity: trace checkpoint, optimizer state, policy refresh
   and restart through the receiving implementation to actual
   state/tensor writes. A completed RPC, transfer acknowledgement or
   advancing version is not itself a state write.
4. performance_boundary: no moving required work outside the prescribed
   measured interval, replacing the host workload/metrics, or selecting
   a check-only path that bypasses requirements on the actual measured
   path. Do not judge speedups, floors, statistical significance, or
   formal-versus-scout eligibility.

Follow actual entrypoints and delegated callees, including archived
framework code. Judge the fixed workload: distinguish an unused
conditional capability from a demonstrated change to required
computation. Record unresolved activation separately. Ground findings
in executable behavior and contract clauses, not diagnostic names.
Trace whether the checked observations come from the actual required
computation: an automatic pass does not excuse fabricated probes,
incorrect gradients followed by self-consistent updates, or
acknowledged transfers without actual tensor writes. Question a passed
gate only with a concrete source-grounded bypass or uncovered semantic
concern on the fixed workload; generic demands for more tests, repeat
measurements or broader numerical coverage do not block source
acceptance.

Copy decision.template.json to a separate decision.json and fill every
field. Use accept, reject or needs_evidence; give source paths, line
numbers and explanations for all four criteria. An accept must have no
violation; reject needs a concrete violation with the original
requirement and an activation basis on the fixed workload;
needs_evidence needs a specific blocking question about those semantics
or missing bound evidence. Keep nonblocking questions separate from
demonstrated violations. Identify human and LLM reviewers accurately;
for LLM review record model and optional effort. Never claim human
adjudication when no human performed it. Do not change packet inputs or
seek a prescribed verdict.

The host verifies and imports your decision separately. Input hashing
binds a review to source and evidence; it does not independently prove
a verdict correct.
```

E AN ORIGINAL OPEN BUILD TASK PROMPT

This appendix reproduces the complete initial message delivered to GPT-5.6 Sol on T06 and the complete `TASK.md` from its first task-file read. The text preserves the original paths and deadline; lines are wrapped for print. Appendix G.1 follows the implementation developed from this request.

Initial message.

```
# Engineering task

Read `$TASK_DIR/TASK.md` and `$TASK_DIR/task.json` before changing
code.
Work directly in the source directories listed below.

## Source scope

- Source `openbuild_solution` root: `/workspace/openbuild-source`
  Editable tree: `solution/**`.
  Protected paths that must not be changed:
  - `solution/.openbuild-root`
  Regular files may be added, modified, or deleted within the editable
tree except at protected paths.

Only regular-file changes inside the declared editable scopes are part
of this deliverable.
Changes outside the declared scope will not be preserved.
Use the task instructions and available checks to complete the
engineering work.

## Hard time limit

This work session lasts exactly 7200 seconds.
The controller deadline is `2026-08-27T09:14:35.417780Z` (UTC).
The controller gives the Agent the full work interval; bounded
dispatch and process-tree cleanup time are outside that budget. There
is no interactive grace period after the work interval.
Keep usable changes saved continuously. The session ends at the
deadline, and only files already written inside the declared source
scope are part of the deliverable.
```

Task file read by the agent.

```
# Deliver a two-NPU Qwen3-4B training system

## Goal

Build a complete offline-replayable training system for
`Qwen3-4B-Base` on
exactly two runtime-logical Ascend 910B devices, then maximize
practical
steady-state token throughput while preserving the public training,
numerical,
checkpoint, restart, cleanup, and resource contracts.

## Required behavior

- Execute both public profiles for exactly 30 committed AdamW updates
and
  983,040 input tokens, using the supplied global records in order.
- Reconstruct the deterministic update-zero state and preserve the
public
  loss, clipping, optimizer, scheduler, weight-decay, BF16, and
```

```
data-cursor
  semantics.
- Emit the complete public loss, norm, projection, trajectory,
checkpoint,
  and fresh-process observations, including the would-be update 31
without
  committing it.
- Execute real work on both assigned devices, keep ranks attached to
the
  foreground entrypoint, exit all ranks, and complete cleanup.
- Package every required implementation and dependency below
`solution/` for
  offline replay.
- Do not use static results, CPU substitution, fabricated
observations,
  replacement batches, skipped or deferred device work, or incomplete
  checkpoints.

## Runtime

- Model: `Qwen3-4B-Base` with BF16 model storage and FP32 optimizer
state.
- Hardware: exactly two runtime-logical Ascend 910B devices, `0,1`.
- The base runtime supplies Python, CANN/HCCL, PyTorch, `torch_npu`,
and
  general-purpose dependencies, but no training or model framework; no
model
  checkpoint or weights are supplied.
- The public formal binary input is generated by
  `tools/generate_formal_input.py`; development feedback uses its
separate
  public development input and oracle.

## Source scope

- The candidate owns the complete delivery below `solution/` and must
stay
  within the task's regular-file size limit.

## Validation

Read [the delivery contract](config/DELIVERY_CONTRACT.v1.md), the
public
workloads, schemas, and `$TASK_DIR/task.json` before implementation.
They define
the exact input ABI, mathematics, timing, output, checkpoint, and
lifecycle
behavior.

Use the public full-shape development loop while building:

```bash
bash "$TASK_DIR/run/development_correctness.sh"
bash "$TASK_DIR/run/development_measurement.sh"
```

It provides the majority of the public semantic and numeric feedback
on a
distinct deterministic input. The final check keeps only a small
controller-
side numeric holdout and does not claim the final same-hardware
baseline
comparison.

Lifecycle
```

````
Run the five task commands from `$TASK_DIR/task.json`:

```text
describe -> smoke -> correctness -> measurement -> cleanup
```

Smoke accepts the initial empty delivery. Correctness and measurement
require
the completed delivery and the explicitly supplied input bundle.
Summarize the
architecture, bottleneck, implementation, and validation in the final
response.
````

Required work and implementation freedom. The original request leads to 2 workloads, each with 30 updates and 983,040 tokens: sequence length 2,048 with global batch 16, and sequence length 512 with global batch 64. Initialization and optimizer mathematics are specified; model execution, parallelism, memory management, checkpoint format, and dependency packaging remain implementation choices. Appendix G.1 summarizes the resulting development history.

F System Designs and Performance

This section supplies the implementation definitions and measurements behind the system comparisons in Section 4. It separates final-submission measurements from the profiling reruns used to explain execution costs.

F.1 Implementation categories

Figure 5 counts edits to source-module families across the 9 Implementation Optimization serving tasks. A task counts once in every family it edits, so categories overlap. These counts describe code changes rather than a decomposition of speedup.

Figure 9 describes the final source of all 27 Model Adaptation submissions, including unsuccessful submissions. Training categories identify the principal layer assembly: MCore TransformerLayer assembly with a supplied convolution mixer; custom decoder/convolution blocks around MCore components; or custom attention and decoder layers with separate Q/K/V projections. RL categories overlap: actor convolution or packed-sequence boundaries; hybrid-cache state or block layout; and rollout lifecycle, readiness, or API control. Serving categories identify added or changed short-convolution code: native NPU operator dispatch, Torch operations, Triton kernels, or reuse of existing kernels with registration/state-layout changes. The Result column separately shows the final evaluation outcome.

F.2 Training layouts and state ownership

For T04, the launchers in Figure 6a use TP1, PP2, and DP2 on 4 NPUs. Astra keeps the default 18/18 layer split; GLM assigns 19 layers to the first stage; Kimi and Sol use 9- and 3-layer virtual pipeline blocks. The layout diagram shows placement, not elapsed time. Task speedups combine the two objectives in Appendix C.

T06 permits a complete trainer design. Sol partitions tensors and FP32 optimizer state across 2 devices, uses vocabulary-parallel loss, and varies activation recomputation by workload. Kimi partitions layers 0–19 and 20–35 across its two pipeline stages. Each stage owns its local parameters and optimizer state; tied gradients are synchronized across stages.

The three optimizer-sharded trainers retain a complete BF16 model and full-sized FP32 gradient buffer on each rank, but partition FP32 master weights and both Adam moments. Opus all-reduces gradients, updates its interval, and all-gathers BF16 weights. Astra packs matching shard intervals into reduce-scatter buckets: global results overwrite only the owned interval of the existing full gradient buffer. An AscendC AdamW kernel updates local state, followed by bucketed all-gather.

Fable all-reduces gradients, updates local state, and broadcasts each BF16 shard from its owner. All five trainers restore the state needed for fresh-process update 31.

The MindSpeed-LLM performance baseline for T06 uses TP2, sequence parallelism, microbatch size 1, and uniform full-layer activation recomputation. Model storage is BF16; gradient accumulation and optimizer state are FP32. Table 13 records absolute evaluation times, including bootstrap and the training process. Table 12 gives paired ratios for the Sol T06 and Kimi R05 cases discussed in Appendix G; training uses the median, while RL uses pooled accepted response-token rates.

Table 12: **Paired performance of 2 valid systems.** Ratios compare training with MindSpeed-LLM and RL with verl.

System	Pair 1	Pair 2	Pair 3	Pair 4	Task speedup
GPT-5.6 Sol / training	3.45	3.45	3.47	3.43	3.45
Kimi K3 / GRPO	3.01	3.06	2.98	2.97	3.00

F.3 RL AND SERVING SYSTEM ORGANIZATION

All nine T06 submissions define their own model execution and training loop. In R05, four of the five valid systems reuse Transformers model objects while supplying their own system loop. Kimi shares model storage between training and generation on one active NPU; DeepSeek uses separate actor and rollout models on one active NPU. Astra places actor and rollout on separate devices and copies updated weights. Sol and Fable use two replicas, with DDP and bucketed reductions, respectively.

Kimi's resident GRPO engine owns graph decode, updates, and reload. It generates four responses per prompt, computes group-normalized advantages from reward-service returns, and commits eight updates. The checkpoint stores model, optimizer, scheduler, update, policy-version, and data-cursor state. Preparation loads and warms the engine; the pooled accepted response-token rate increases from 170.90 to 513.54 tokens/s.

S13 separates request-state management from decode policy. Fable pre-captures batch-size/KV-length graphs and uses eager execution outside that grid. Kimi combines selected pre-captured shapes with a bounded on-demand graph cache. Opus uses contiguous per-request KV slots and eager batched attention. Sol assembles vLLM/vLLM-Ascend with `FULL_DECODE_ONLY` and a local queue change. Astra uses paged KV and decode graphs; with one active request, it verifies up to four n-gram proposals and accepts only the prefix matching target-model greedy decoding.

Astra's S13 prefill, decode, and SLO ratios are $1.175\times$, $1.259\times$, and $1.096\times$; the three saturation ratios are $0.990\times$, $0.885\times$, and $0.840\times$. Prefill, decode, saturation, and SLO each receive one quarter of the task weight, producing a task speedup of $1.100\times$. This is a latency-oriented design with a measured throughput trade-off.

F.4 REPEATED-CONTEXT SERVING

In S06, Astra tiles QKV normalization and rotary preprocessing over token rows and grouped query heads, keeping the library path for unsupported shapes and small batches. It retains host sequence lengths, avoids page-allocation scans when decode needs no new page, and reuses a graph-update worker. Shared-prefix and branching-tree throughput improve by 8.61% and 8.80%; the independent-context control improves by 30.10%. Fable, DeepSeek, and GLM retain the supplied implementation. Other submissions change cache, scheduling, graph-update, or metadata paths without the corresponding NPU preprocessing rewrite; the best other task gain is MiniMax's 0.83%.

F.5 WORKLOAD AND PPO PHASE MEASUREMENTS

S09 gives each of its six request regimes weight $1/6$. The Other 5 row in Figure 7 is the geometric mean of the five non-offline ratios. Performance coverage separately checks the performance requirements on both scored and auxiliary objectives.

Table 14 supplies the R03 phase times and speedups behind Figure 8. Each of the four valid implementations has four alternating profiling pairs on the same 2-NPU allocation. Every run

Table 13: **Absolute time and execution windows.** Mean seconds across the 4 executions on each side. T06 training-process time includes initialization, 30 updates, checkpoint commitment, and exit. R03 separates its online loop from the remaining process time.

Task / workload	Window	Baseline (s)	GPT-5.6 Sol (s)
T06 / long sequence	Complete job	581.17	254.23
	Offline bootstrap	5.61	0.01
	Training process	575.55	254.22
T06 / update heavy	Complete job	1239.70	238.02
	Offline bootstrap	5.62	0.01
	Training process	1234.07	238.00
R03 / PPO	Complete process	1305.73	172.17
	Online loop	275.64	116.17
	Outside online loop	1030.09	56.00

processes 256 trajectories and 16,384 response tokens through eight actor and eight critic updates. Ratios divide mean baseline time by mean candidate time within that implementation's paired group; the displayed baseline example is Sol's group.

Table 14: **PPO times and speedups by phase.** (a) Mean seconds across 4 executions per side; the baseline example is paired with GPT-5.6 Sol. Finish covers finalization with checkpoint saving disabled. (b) Ratios use each implementation's own paired baseline. Outside-loop time is initialization plus finalization; inside-loop time includes generation, scoring, updates, and policy synchronization.

(a) Mean phase times (s)

System	Init.	Gen./score	Update	Sync	Finish	Total
Baseline	1012.45	155.75	77.32	42.45	9.68	1297.74
GPT-6 Astra	44.12	14.41	4.91	0.03	11.38	74.95
GLM-5.3	44.37	52.35	15.81	0.00	1.48	114.50
Claude Fable 5.1	66.50	43.62	28.93	0.17	5.36	144.65
GPT-5.6 Sol	44.12	71.48	42.84	0.00	10.95	169.48

(b) Speedups ($\times$)

System	Whole job	Outside loop	Inside loop
GPT-6 Astra	17.59	18.70	14.42
GLM-5.3	11.43	22.48	4.05
Claude Fable 5.1	9.00	14.24	3.82
GPT-5.6 Sol	7.66	18.56	2.41

Timing runs from process launch through exit. Initialization ends at online-loop entry; generation/scoring includes policy, reference, reward, value, and advantage work. Concurrent actor/critic intervals form one update union. Explicit rollout publication forms the synchronization interval. Finalization includes result writing, the final policy probe, and exit, with checkpoint saving disabled during timing. Unclassified online gaps are at most 0.50 seconds per row and remain included in loop time.

Lightweight timestamps supply the figure's bars and ratios. CPU/NPU traces of updates 1 and 4 are collected in separate runs, without adding profiler overhead to the displayed times. The original evaluation measurements in Table 13 and the result data remain the basis of task aggregates; these profiling reruns explain the phases rather than replace those observations.

G DEVELOPMENT CASES

The cases connect code inspection or test feedback to choices retained in the submitted systems. Development timings describe the version tested at that point; final task results come from independent evaluation. The website in Appendix A provides the de-identified run records and artifacts, while the companion archive retains the detailed case notes. These examples explain decisions, rather than estimate how often agents follow a particular process.

G.1 FROM A TASK REQUEST TO A COMPLETE TRAINER

Sol's T06 history follows the original request in Appendix E through implementation and evaluation. The retained history contains 36 visible agent messages and 168 completed tool calls.

Build and check the computation. The agent reads the task, delivery interfaces, and numerical references, checks the two-device runtime, and implements a tensor-parallel model, FP32 optimizer state, bootstrap, and launcher. It first tests kernels and short updates. The first full replay then fails at an update-23 coordinate lookup for a one-dimensional parameter; after the lookup fix, development checks pass 172 numerical and 32 optimizer-transition checks, including both fresh-process probes.

Refine execution and checkpoint handling. Memory tests lead to retaining all long-sequence activations and checkpointing nine update-heavy layers. A checkpoint-compression experiment makes local replay slower and is reverted. Transfer measurements instead lead to reusable pinned staging (Table 15). A numerical repeatability failure prompts tests of deterministic execution. A combined development measurement subsequently passes the numerical, transition, restart, and execution checks.

Capture and independently evaluate the final revision. A later edit limits deterministic execution to the first two long-sequence updates and is tested with two four-update probes. The final full development measurement is launched, but its completion is not present in the retained history. The captured revision passes all three gates in independent evaluation, including checks of fresh-process continuation with restored optimizer, scheduler, and data-cursor state. Its task speedup is $3.45\times$. Table 12 reports the four paired ratios.

G.2 CODE CHOICES INFORMED BY DEVELOPMENT MEASUREMENTS

Table 15: **Test feedback and the resulting code choices.** Times refer to the tested development versions; the right column shows the choices in the final systems.

Case	Observation	Choice retained in the delivered system
GPT-6 Astra S13	Timing separates full batch-64 decode from graph replay; the subsequent decode probe changes from 23.71 to 20.83 ms.	Replace per-item tensor metadata writes with array writes on the host.
GPT-5.6 Sol T06	A 512 MB transfer takes 166.53 ms with pageable memory versus 29.51 ms with pinned memory.	Stage FP32 parameters and optimizer state through reusable pinned host buffers.
GPT-6 Astra S07	A clean development measurement reports a 2.52% decode regression.	Restore the original decode allocator; raise the QKV tiling threshold from 32 to 128 tokens.
GPT-5.6 Sol S07	A crossover test returns matching outputs and a gain at 6,144 and 7,680 tokens.	Enable position-cached QKV preprocessing only from 6,144 tokens.

The T06 transfer probe uses one 128,000,000-element FP32 tensor on NPU 0. Pageable, pinned blocking, and pinned nonblocking copies take 166.53, 29.51, and 29.99 ms. Each is a single ordered trial including allocation, transfer, and final synchronization, but not file I/O. The S07 cases use different feedback: Astra changes its threshold after a decode regression, while Sol measures an input-size crossover. Both preserve the original path for smaller inputs.

G.3 Reuse and integration choices

For S13, Sol obtains v0.23.0 checkouts of vLLM and vLLM-Ascend and packages an offline runtime, bootstrap, launcher, and local patches. For S07, it instead inspects the supplied SGLang model and graph-runner code and reuses the existing position-cache operator. Kimi's T05 upstream search finds no tracked paths matching `lfm`; it adds LFM2 layers, short convolution, and checkpoint conversion around MCore. These cases range from adopting a framework to reusing an operator and implementing missing model support.

Restart feedback changes the GRPO trainer. Kimi R05 completes eight updates before a local restart probe runs out of NPU memory. The agent introduces detached-leaf chunked backward and releases the old engine in the test driver before constructing its replacement. The next same-process probe loads the checkpoint and completes update 9. This probe exercises both edits together; independent evaluation separately verifies fresh-process continuation of the submitted trainer and reports a task speedup of $3.00\times$.

H Outcome Accounting

Each of the 216 submissions receives one final outcome (Table 16). The 178 valid results comprise 173 measured implementations and five unchanged Optimization baselines assigned $1.00\times$. The remaining outcomes are 23 Correctness, eight Measurement, and seven Source review failures. Confirmed numerical or behavioral failures on the published workloads take precedence over earlier performance or source records.

Table 16: **Mutually exclusive final outcomes for all submissions.** Measured valid and unchanged baseline together form the valid total. C/M/R identify Correctness, Measurement, and Source review failures.

Configuration	Runs	Measured valid	Unchanged baseline	C	M	R	Valid total
GPT-6 Astra	24	24	0	0	0	0	24
Claude Fable 5.1	24	20	2	1	1	0	22
GPT-5.6 Sol	24	23	0	0	0	1	23
Claude Opus 5	24	21	0	2	0	1	21
Kimi K3	24	20	0	2	2	0	20
DeepSeek-V4-Pro	24	19	1	3	0	1	20
GLM-5.3	24	18	1	3	2	0	19
MiniMax M3	24	14	1	8	1	0	15
Gemini 3.7 Flash	24	14	0	4	2	4	14
Total	216	173	5	23	8	7	178

H.1 Engineering scope and performance requirements

Figure 11 includes all submissions within each scope. Four valid Implementation Optimization results fail an applicable performance requirement: they remain valid but do not contribute to performance coverage. Stage totals are 48 valid training, 27 RL, and 103 serving systems. Only the 173 measured valid implementations enter the execution-repeatability analysis.

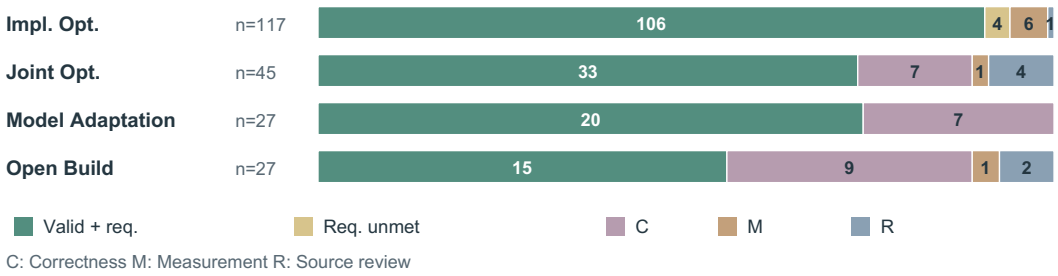

Figure 11: **Final outcomes by engineering scope.** Each bar includes all submissions in its scope; the segments count valid results with/without satisfied performance requirements ("req." in the legend) and Correctness, Measurement, and Source review failures.

H.2 Requirements behind failed outcomes

The 23 Correctness failures comprise eight state-handoff/restart failures, five numerical disagreements, three runtime-integration failures, two NPU-memory failures, two serving-output failures, and one each in training execution, algorithm evidence, and required packaging. The numerical group includes the three R04 packed-boundary log-probability/gradient failures, MiniMax T05 loss/norm failure, and MiniMax S10 generated-token scoring failure. Gemini R04's convolution-key mismatch prevents policy installation; the failed PPO checkpoint comparisons prevent valid continuation despite completed update loops.

Measurement failures arise during scored workloads after initial correctness checks: seven are runtime failures involving timeouts, service or scheduler exits, graph capture, or an NPU exception; one fails a measurement-time output-quality check. Source review rejects seven implementations whose executed computation or state handling does not meet the task contract. Per-run diagnoses remain in the companion archive; the table records every final outcome without adding overlapping failure counts. Researcher implementations and A/A controls are not included in the agent results; repeated measurements are reported separately.

I MEASUREMENT REPEATABILITY

The controls in Figure 12 assess performance measurements of fixed implementations, not variability across independent agent development runs. Baseline/candidate pairs alternate on the same physical NPU allocation using task-specific preparation.

Sensitivity to a measurement pair. For each of the 173 measured valid implementations, remove one complete baseline/candidate pair and recompute the task speedup with its prescribed aggregation rules. If S_{-k} is the task speedup after removing pair k and S is the four-pair task speedup, the plotted range is $100(\max_k S_{-k} - \min_k S_{-k})/S$. Points show individual implementations and diamonds show each configuration's median range. This range measures sensitivity to the observed pairs, not a confidence interval.

Unchanged-baseline controls. A/A comparisons run unchanged baseline code on both sides of the four-pair protocol. The 14 controls cover 11 task workloads, with two controls each for S03, S04, and R01. Ratios range from $0.9848\times$ to $1.0147\times$, all within 2% of parity. S03 isolates compiler caches per trial and warms until two consecutive rounds leave the cache unchanged; S04 uses four unscored warmup steps. Appendix C.1 details these preparation rules and the cache configurations used for the reported task results.

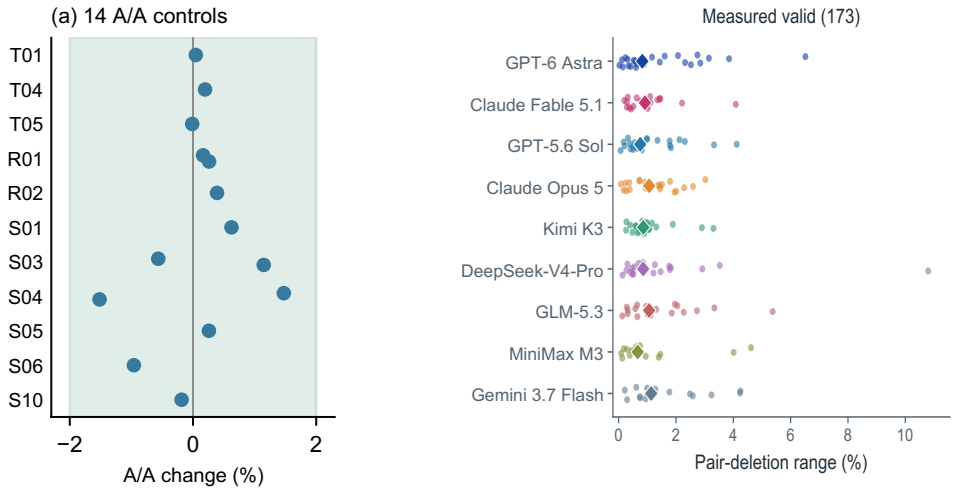

Figure 12: **A/A controls and measurement repeatability.** Left: 14 complete 4-pair controls across 11 task workloads under the preparation rules described above; shading marks $\pm 2\%$ from parity. S03, S04, and R01 each have 2 controls. Right: pair-deletion ranges relative to the 4-pair task speedup for all 173 measured valid implementations, grouped by agent configuration. Diamonds mark each configuration's median range.

Independent repetitions. Three fixed implementations also have complete repeated four-pair evaluations. Initial/repeat task speedups are $1.0275/1.0093$ for GLM S02, $1.0245/1.0416$ for Opus S03, and $1.0462/1.0441$ for Sol S05, under their recorded workload and cache configurations. These repetitions are reported separately; the main results retain the initial measurements.