Mistral Large 4



Abstract

We introduce Mistral Large 4 (ML4), a 52B active, 1T total parameter multimodal mixture-of-experts model designed for reasoning and long-horizon agentic tasks. We build a scalable agent-based pipeline that authors and verifies executable environments for software engineering, cybersecurity, and knowledge work. Mistral Large 4 achieves competitive performance on code, cybersecurity, and held-out agentic benchmarks, including 19.7% on FrontierCode 2.0, 41% on Terminal-Bench 4.0, and 50.9% on the Artificial Analysis Cyber Index. Efficient serving combines sparse attention, optimized context parallelism, prefill–decode disaggregation, and speculative decoding. Calibrated NVFP4 quantization reduces the checkpoint to 606 GB while matching the FP8 reference on evaluations, enabling deployment on four GB300 GPUs. Multiple-teacher distillation provides high, medium, and low reasoning-effort modes, allowing users to control the trade-off between token consumption and performance.

1 Introduction

We introduce Mistral Large 4 (ML4), a 52B/1T-parameter multimodal mixture-of-experts model designed for reasoning and long-horizon agentic tasks. To enable fast, efficient serving, we pushed KV-cache reduction to the limit during development, but later found it underperformed on downstream agentic tasks. We quantified this agentic penalty and addressed it through model surgery, converting our model to full attention, then DeepSeek Sparse Attention (DSA) [1].

To meet the need for high-quality post-training data, we built a scalable data-generation pipeline in which agents author executable environments, task specifications, reference solutions, and verifiers. The same pipeline addresses capabilities like software engineering, cybersecurity, and knowledge work. Author and refinement agents test candidate tasks with multiple solvers, repair ambiguous or broken tasks, and guard against reward hacking. Successful author trajectories also become training data, helping us to build the next generation of environments with the trained model.

Mistral Large 4 achieves competitive performance on code, cybersecurity, and held-out agentic benchmarks that span web research and workplace automation. It scores 19.7% on FrontierCode 2.0 [2], 15% on Harvey's Legal Agent Benchmark [3], 41% on Terminal-Bench 4.0 [4], and 50.9% on the Artificial Analysis Cyber Index [5]. On the safety side, it achieves a 96.6% unweighted average of the safe-response rate on JailbreakBench [6], the refusal rate on StrongREJECT [7], and 100 minus the harm score on AgentHarm [8], each measured on the benchmark's cybersecurity subset.

We serve ML4 efficiently through the combination of sparse attention, optimized context parallelism, prefill–decode disaggregation [9, 10], and speculative decoding. Calibrated NVFP4 quantization reduces the checkpoint to 606 GB while matching the FP8 reference on evals, which allows deployment on $4\times$GB300 GPUs. Thanks to multiple-teacher distillation from checkpoints with different levels of verbosity, the final model supports high, medium, and low reasoning effort, giving users control over the trade-off between token consumption and performance.

2 Pretraining

2.1 Hyperparameters and scaling ladder

We predict the optimal peak learning rate with a log-linear function of model width, depth, training horizon (total training tokens, counting repeats), expansion rate (ratio of total to activated experts), and batch size. By treating each dimension as a separate input, we keep the predictor flexible to architectural changes and verify that the predictor extrapolates correctly along each dimension, up to a scale comparable to the planned reference model, while keeping computational costs manageable. We fit a batch-size predictor on the same set of dimensions as the learning-rate predictor. Empirically, the optimal batch size depends primarily on the training horizon and expansion rate. Therefore, we first predict the batch size, and condition the learning-rate prediction on the predicted batch size.

We validate the recipe before the full-scale run with a scaling ladder: a series of smaller-scale models that serve as proxies for the reference model. Each model in the scaling ladder uses the same predictors and multipliers as the reference model, covering the tokens-per-parameter ratio, learning rate, and batch size. The scaling ladder thus differs from the reference model in scale only. We then study the degree to which the outcomes for these proxy models extrapolate to the target compute budget, and whether the proxy models exhibit any poor scaling behavior with respect to alternative architectures.

2.2 Data

Mistral Large 4 was pretrained on a large mixture of diverse multilingual and multimodal datasets that span web pages, source code, and scientific and technical documents. Almost all our datasets are natively multimodal and formatted as interleaved text–image chunks. Training followed a two-phase curriculum: the first phase (65%) emphasizes broad coverage across domains and languages, while the second (35%) rebalances the mixture toward higher-quality and more STEM-focused data. In the second phase, we increase the proportion of code, scientific, and high-quality documents, and increase the sequence length from 4K to 40K.

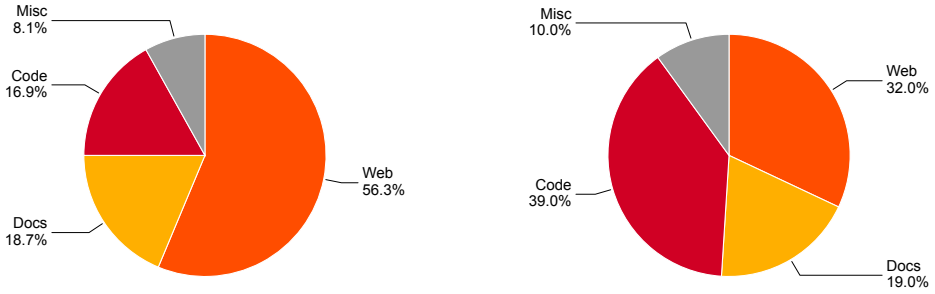


Figure 1: Pretraining data mixture in phase 1 (left) and phase 2 (right).

We train from scratch on interleaved image–text data. With careful dataset design, we observe no regret in text perplexity when matching the number of text tokens seen; while matching the total training horizon yields modest regret. This increases the number of images seen by over an order of magnitude relative to our previous flagship models. We train a vision encoder from scratch, frozen throughout pretraining and unfrozen only for supervised fine-tuning (SFT). Images are prefetched N batches ahead and their features cached, sustaining 99% of text-only MFU while enabling on-the-fly dynamic transforms and removing the need to store image features on disk.

2.3 Training

Mistral Large 4 was pretrained on 3,840 NVIDIA GB200 GPUs in a Mistral data center near Paris. Our internal training stack is built on PyTorch and CUDA. Pretraining took 45 days, or about 4.15 million GPU-hours, with a model FLOPs utilization (MFU) of 36% and a goodput of 94.1%. We measure MFU against peak hardware BF16 FLOP/s, include matrix-multiplication and attention FLOPs, and exclude recomputation, unlike hardware FLOPs utilization (HFU).

The model architecture and training framework were designed around the NVIDIA Blackwell architecture [11]: for example, we use FP8 tensor cores for fine-grained low-precision training, the larger NVLink domain for wide expert parallelism, and the NVLink-C2C link to the Grace CPU for fast activation CPU offloading. We accelerate training by leveraging ZeRO-2 optimizations [12], fine-grained activation recomputation, maximized compute–communication overlapping, systematic kernel fusing, and streaming multiprocessor (SM) margin to reduce resource contention across CUDA streams. We also improved our overall system performance: for example, asynchronous workloads to minimize impact on actual training (data loading, checkpointing, metrics, etc.), and better monitoring tools across clusters to detect outliers (CPU stalls, NaN detectors, NFS slowdowns, etc.).

As in Mistral Large 3, Mistral Large 4 uses a granularity of 4 for its mixture-of-experts layers. We found that increasing granularity from 4 to 8 reduces the FLOPs needed to reach a matched perplexity by about 4.3%, with a reported uncertainty of 0.5 percentage points. The measured training slowdown was approximately 5%, which motivated the choice of granularity 4.

Unlike global-batch balancing [13], we compute the load-balancing loss over each expert-parallel (EP) group [14]. This approach aggregates statistics across the tokens handled by the EP group and directly optimizes the balancing metric responsible for slowdowns – expert inbound tokens. Pretraining comparisons favored this objective over other balancing choices when trading off imbalance against validation perplexity. The downside is that it strengthens the coupling between the load-balancing loss and the partitioning strategy, so downscaling experiments must be careful when changing partitioning. We tuned the balancing coefficient against expert-load imbalance, perplexity, and downstream evaluations.

We use the AdamW optimizer with $(\beta_1, \beta_2) = (0.975, 0.975)$ during pretraining. We found that coupling and increasing these values above more typical choices provided moderate gains in performance, in line with Orvieto and Gower [15].

2.3.1 Batch size ramp-up

The optimal batch size increases throughout training. Early in training, a larger batch size wastes compute (and tokens) because a gradient estimate from a fraction of the batch's tokens suffices, whereas later in training, we benefit from more precise gradients. Therefore, we ramp up the batch size in stages with a share of tokens that grows exponentially per stage: the final batch size (stage) runs for the last 50% of tokens, the penultimate batch size runs for 25% of tokens, and so on. We rescale the learning rate for each stage by a factor that is tuned empirically.

Batch-size ramp-up also acts as a regularizer: noisier gradient estimates from the smaller early batches reduce overfitting, and the benefits of regularization increase with the number of data repeats. Like the batch-size compute multiplier, the choice of schedule trades off optimization efficiency against throughput. From an optimization perspective, we find that ramp-up from $0.25\times$ to $2\times$ the predicted batch size performs best empirically. However, larger batch sizes achieve higher MFU in our setting. Therefore, we chose to ramp up from $1\times$ to $4\times$ based on wall-clock performance, which produces the 40M, 80M, 160M schedule above.

2.3.2 Vision encoder

We pretrain a 1.6B-parameter vision encoder from scratch with a captioning objective and discard the captioner afterward. We train on a mixture of image–caption, interleaved image–text, and OCR data, over a horizon of 8T tokens. Following NaViT [16], images are processed at their native aspect ratio, up to a maximum resolution of 1540×1540 pixels. At the encoder output, a patch merger down-samples each 2×2 grid of patches by concatenating the four patch embeddings and applying a linear projection. An MLP then maps the merged embeddings to the LLM's input embedding dimension. To evaluate the encoder, we follow a LLaVA-style protocol [17], where the encoder is attached to a small LLM and a short SFT stage is performed with roughly 20B tokens. The resulting model substantially outperforms open-source vision encoders of comparable size on downstream multimodal evaluations.

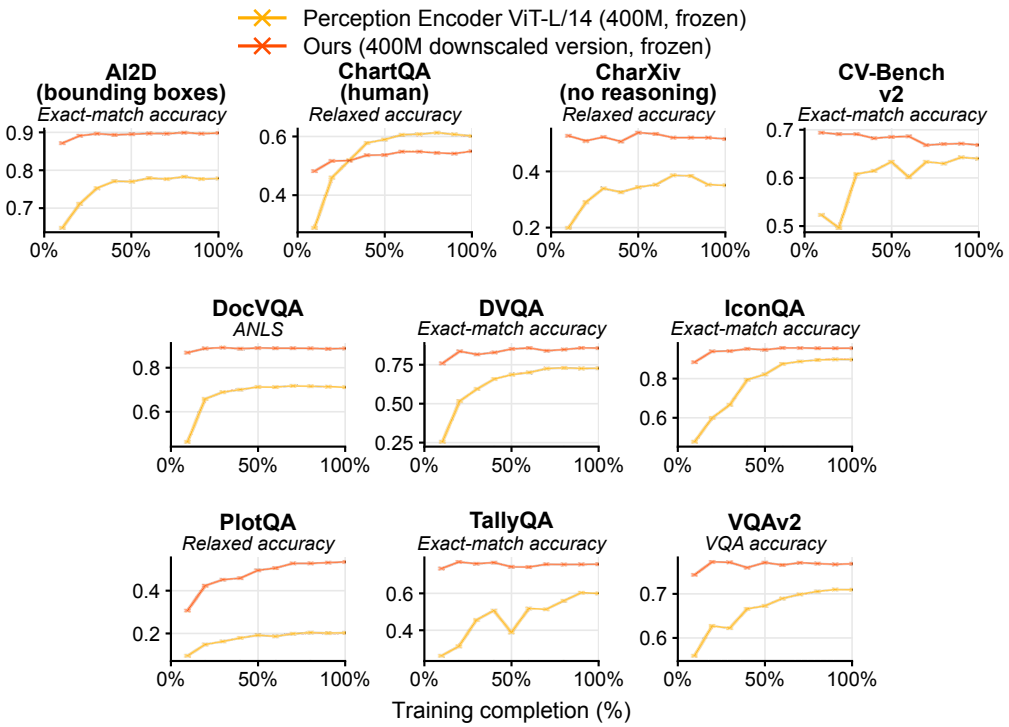


Figure 2: Comparison of frozen vision encoders on multimodal evaluations. The shared x-axis shows the percentage of the SFT training budget completed (0–100%). ANLS denotes average normalized Levenshtein similarity.

3 Why ML4 is (almost) a full-attention model

3.1 Agentic penalty on architecture

Initially, we used multiple optimizations to reduce the KV-cache footprint of ML4, notably:

- **Ragged multi-head latent attention (MLA):** One in every four layers uses full attention, the remaining layers use sliding-window attention (SWA) [18–20] with a window size of 512 tokens.
- **Cross-layer attention sharing (CLA) [21]:** Every other full-attention layer reuses the KV cache of the preceding full-attention layer.
- **YOCO [22]:** Full-attention layers in the second half of the transformer attend to the KV cache produced by the first half instead of maintaining their own KV cache. Beyond KV-cache reduction, YOCO enables two-stage prefilling, which is 1.7× faster at 1M context on our benchmarks.

The combination of these optimizations reduced the KV-cache size by a factor of 16 compared to a full-attention baseline. Early ablations suggested that impact on pretraining performance, including perplexity and other common pretraining and post-training benchmarks, was either negligible or small enough to justify the efficiency gains.

However, as ML4 pretraining continued, early SFT cooldowns on intermediate checkpoints revealed poor performance. Despite expected performance on pretraining evaluations, including perplexity and knowledge-based benchmarks, the model's post-SFT performance was lower than expected on agentic evaluations, especially those that require long-context reasoning like SWE-bench [23] and BrowseComp [24]. Parallel research efforts helped us to isolate the performance regression. Under the same training token budget, SWE-bench Verified [25] scores measured after midtraining and SFT decreased as we moved from full attention to SWA, and regressed further when applying CLA.

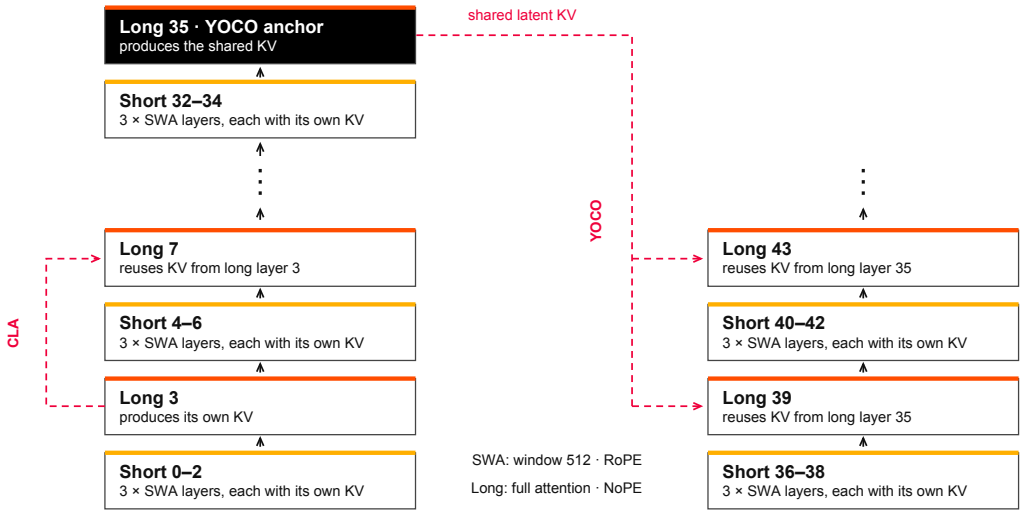


Figure 3: The initial attention-sharing architecture with CLA and YOCO.

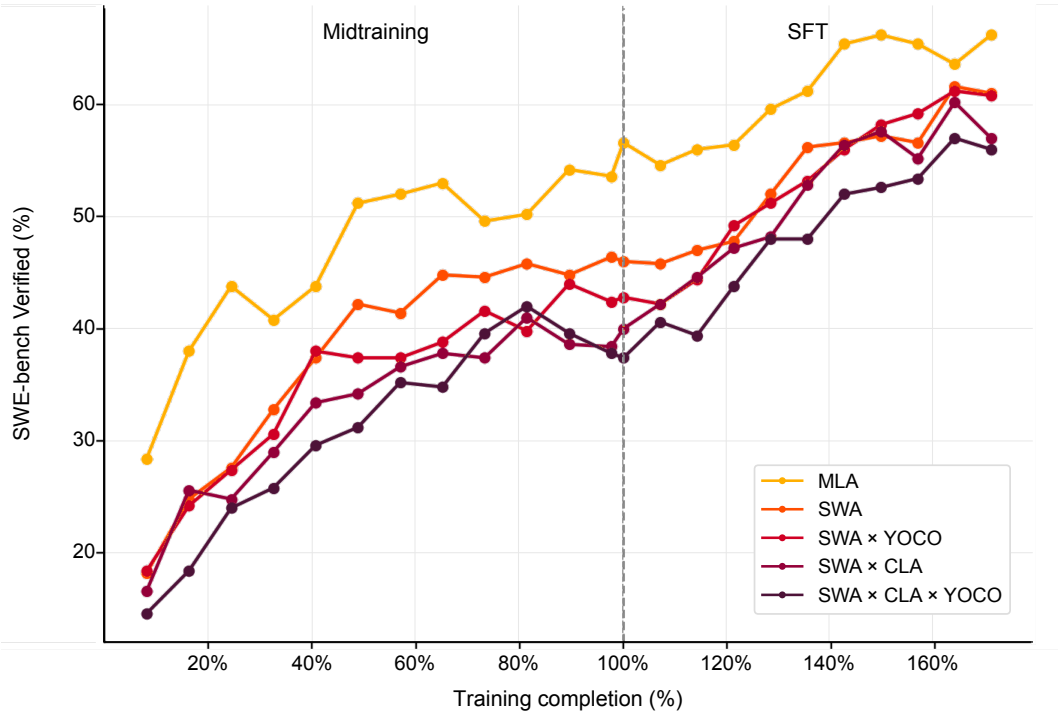


Figure 4: SWE-bench Verified performance during midtraining and SFT for different attention architectures. The x-axis expresses training tokens as a percentage of the standard midtraining horizon (100%); SFT continues above 100%.

We subsequently established that architectural choices can impose a penalty on agentic capabilities: models that perform equally well on language-modeling loss can nevertheless perform very differently on downstream agentic tasks. We did not observe the same gap with knowledge-based evaluations. Our ablations also suggest that longer training can mitigate this penalty (Figure 5).

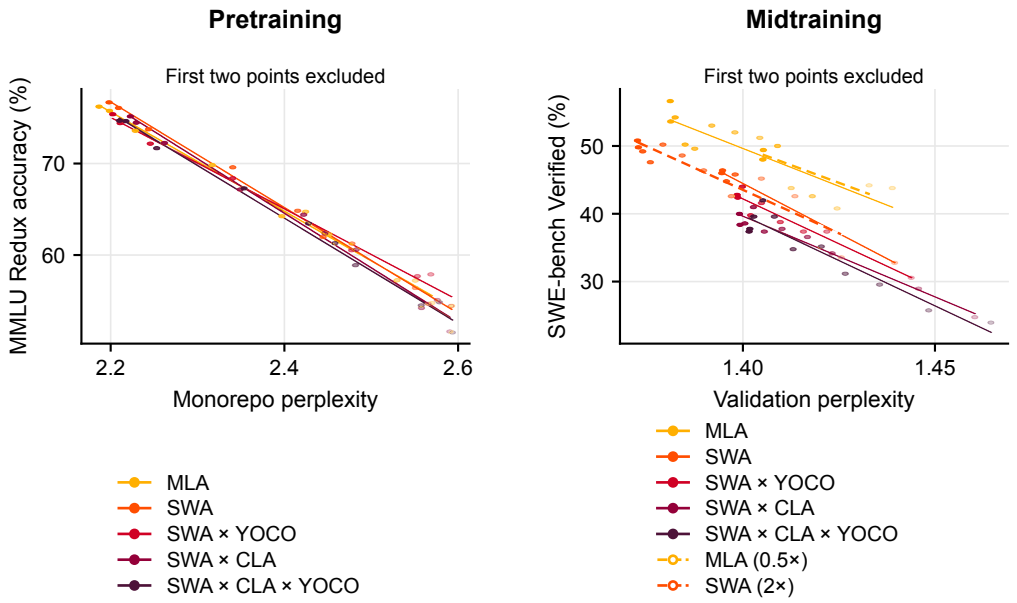


Figure 5: Language-modeling loss and evaluation performance for different attention architectures. Left: pretraining MMLU Redux [26] accuracy versus validation perplexity. Right: SWE-bench Verified after midtraining versus validation perplexity.

3.2 Model surgery

After we came to understand that our initial architectural choices for ML4 were not best suited to the agentic capabilities we wanted to prioritize, we decided to modify the architecture of an existing checkpoint prior to the second phase of pretraining. The goal was to remove KV-cache compression mechanisms to achieve an architecture as close as possible to a full-attention model. Although this approach temporarily sacrificed inference optimizations, we prioritized model performance. The challenge was preserving what we had already learned in pretraining.

It is relatively straightforward to remove sliding-window attention constraints: one simply allows the affected layers to attend to the full context instead of only the previous 512 tokens. However, it is more difficult to remove CLA and YOCO KV-cache sharing: layers that previously relied on keys and values produced by earlier layers then need to compute and maintain their own KV caches. In both cases, for each affected layer, we added KV-producer weights by copying them from the last long layer that maintained its own KV weights. For CLA, this is the preceding long layer, which is close to the affected layer and takes a very similar residual stream as input; for YOCO, this is the preceding KV-producing long layer at the model's halfway point, which is farther away.

The model surgery aimed to introduce attention computation in more layers over longer sequences while preserving as much as possible of the original model. At the 8×10^{22} FLOPs scale, we performed 'full surgery', but retained YOCO cache-reuse in the full-scale model due to negative results when scaling up.

3.2.1 Removing KV compression, adding LeRoPE and SSMax

We evaluated attention surgery at 8×10^{22} FLOPs compute scale, where surgery was applied either at the beginning of midtraining, or before phase 2 of pretraining. In each case, surgery was followed by midtraining. We used two reference points: the original model with SWA, CLA, and YOCO, and a

model pretrained with full attention only. As described above, model surgery removed sliding-window constraints and CLA/YOCO KV sharing but retained NoPE [27] layers. The midtraining-only variant used YaRN [28]; we compared two variants for phase-2 pretraining:

- **Vanilla:** We removed the attention windows without changing RoPE frequencies during phase 2.
- **YaRN:** We removed the attention windows and used YaRN to rescale the RoPE frequencies by $8\times$.

We then midtrained both variants. The phase-2 pretraining approaches brought training perplexity closer to the full-attention baseline but not SWE-bench performance. We also experimented with more extensive architectural changes but found they were harder to recover from. We thus favored a more conservative surgery that preserved the model's performance immediately after conversion and allowed training to continue, while introducing learnable parameters to add flexibility.

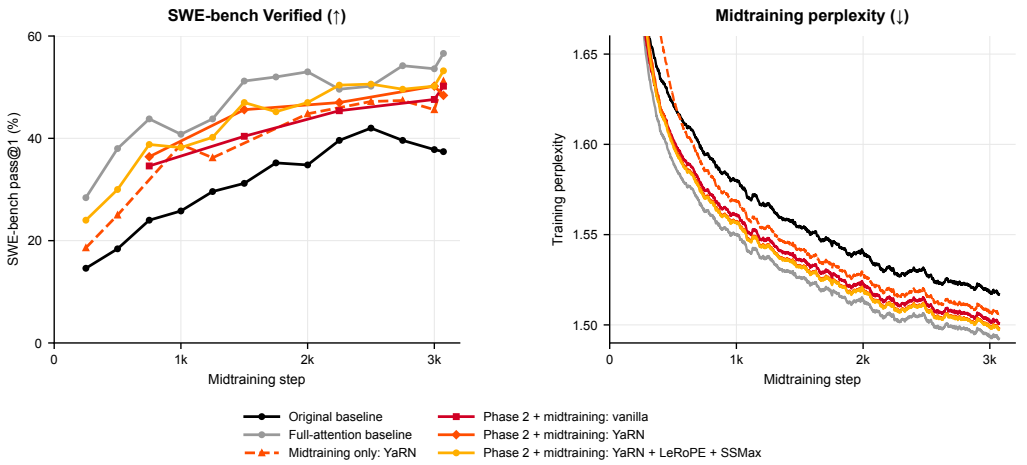


Figure 6: Surgery evaluation and midtraining perplexity at the 8×10^{22} FLOPs compute scale.

We experimented with positional encoding and attention softmax normalization to allow lengthened layers to attend to longer context. The combination of LeRoPE [29] and SSMax [30] improved downstream performance over fixed YaRN, despite nearly identical training loss. LeRoPE and SSMax let the model learn rotary frequencies and per-head query scaling. In an intermediate phase-2 checkpoint, the largest frequency changes occurred in two bands near the 4K training context: frequencies increased by approximately $1.5\times$ and $1.8\times$ relative to YaRN, which shortened their wavelengths by 35% and 44%, respectively. Most other bands changed less; the adaptation was selective rather than rescaled uniformly.

3.2.2 Scaling up to the full model

The surgery experiments above included removing YOCO but it was retained for the full-scale model. We evaluated this choice separately because removing another sharing mechanism increased the adaptation required from the pretrained checkpoint. At the 8×10^{22} FLOPs scale, models with and without YOCO removal reached comparable downstream performance when allowed the entire phase-2 and midtraining horizon to recover. Nevertheless, removing YOCO initially disrupted the loss more. Since we were not able to re-run the entirety of phase 2 for the final model, we made a conservative test at the 8×10^{23} FLOPs scale with surgery and only midtraining as opposed to surgery, phase 2, and midtraining. We found that keeping YOCO yielded better performance.

The resulting architecture removed sliding-window constraints and CLA, while preserving YOCO sharing among the original global-attention layers in the second half of the network. Former SWA layers became independent full-context attention layers, with LeRoPE and SSMax providing additional flexibility. The original NoPE/global layers remained NoPE; their second-half YOCO consumers continued to reuse the KV representation from the first half. Thus, the converted model gained many independent full-context layers while retaining part of the original KV-sharing topology.

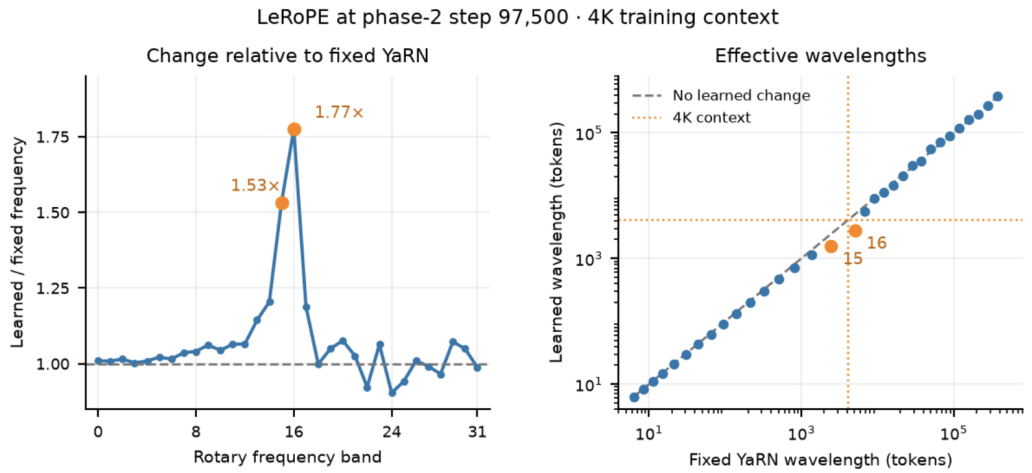


Figure 7: Learned rotary frequencies relative to fixed YaRN and their effective wavelengths.

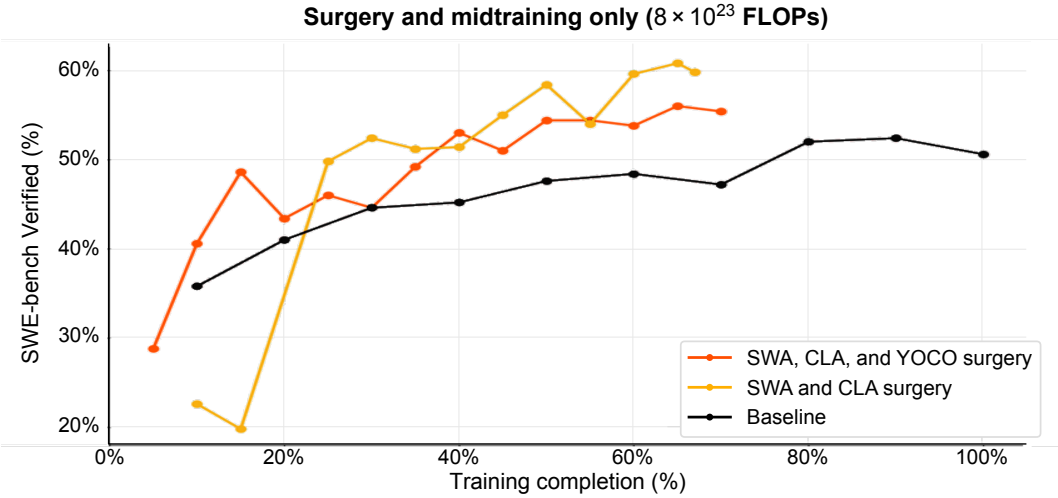


Figure 8: SWE-bench Verified performance following surgery and midtraining at the 8×10^{23} FLOPs compute scale.

We applied surgery to the end-of-phase-1 checkpoint and trained for approximately two-thirds of the original phase 2 before midtraining.

Lifting the sliding-window constraint allows formerly SWA layers to attend to the entire context window. We validated this by analyzing attention-mass distributions across intermediate checkpoints using 32k-token probes across six data sources, comparing converted SWA layers with native long layers. Converted layers allocate around 55% of the attention mass to far-away (>512 tokens) content, compared to 67% for native long layers. We take this to mean that the conversion did not completely eliminate the original initialization but that converted layers were able to learn a long-layer role if needed. Even in full-attention models, anyway, some layers attend to longer context than others.

3.2.3 DeepSeek Sparse Attention (DSA)

To reduce the cost of long-context attention, we adapted ML4 to DSA [1, 31], using lightweight indexers to select the top-2,048 key tokens for each query. Adaptation proceeds in two stages. In Stage 1, we warm up the indexers for approximately 15.1B tokens through KL-based distillation of the main model's attention distributions, while keeping the main model frozen and operating with full attention. In Stage 2, we jointly train the main model and indexers for an additional 198B

tokens under sparse attention. We optimize the main model with the language-modeling objective, while the indexers continue to minimize an auxiliary KL-divergence loss against the main model's attention distributions over the selected keys. To further reduce indexer overhead, we analyzed cross-layer attention patterns and found high similarity among former SWA layers within the original hybrid-attention groups starting at L16. We therefore share a single DSA indexer among the former SWA layers within each of these groups.

4 Post-training

4.1 Agentic instance authoring

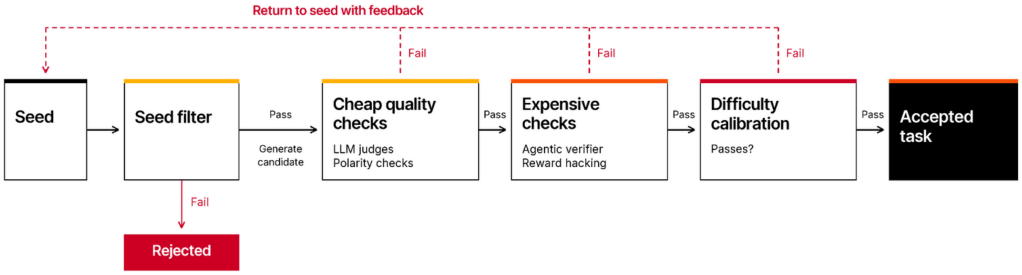


Figure 9: The task generation pipeline.

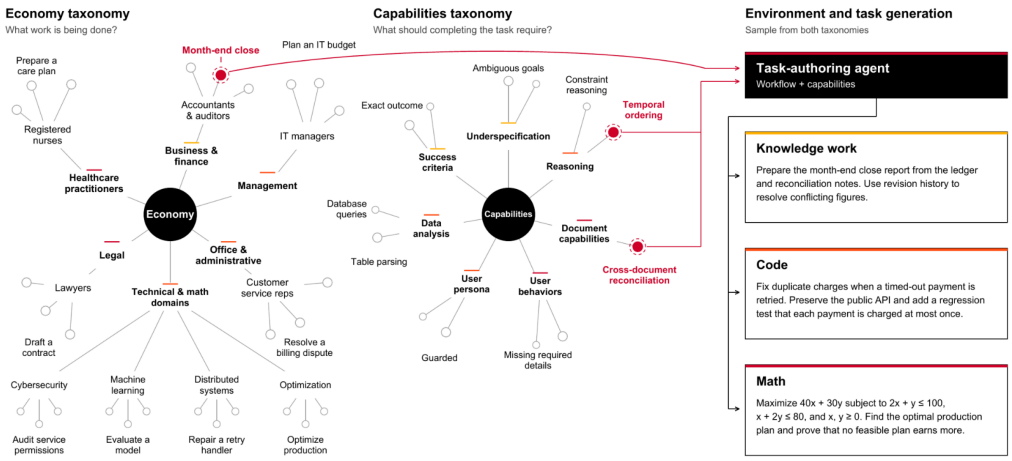


Figure 10: Taxonomies for environment and task generation.

Data generation is traditionally a relatively manual process that requires many iterations and extensive human oversight. To remove human constraints on task quantity and complexity, we invested heavily in a high-quality, scalable agent-based pipeline to generate synthetic data. Agents can now perform complex long-horizon tasks, including authoring tasks that meet stringent requirements. We therefore increasingly leverage agents for end-to-end task synthesis. Nevertheless, agents are fallible and require external verification and supervision. In our experience, producing valuable instances for post-training depends primarily on the following:

Seed content By seeds, we mean both the environment in which an agent is trained and the behaviors and capabilities that we want training tasks to exhibit. We use sets of taxonomies (Figure 10) that, for example, classify a wide variety of occupations and economically valuable tasks, along with the behaviors and capabilities that the domains entail. By adopting taxonomies, we have fine-grained control over the coverage and complexity of the generated synthetic data.

Agentic pipeline We provide the authoring agent with a minimal set of tools to perform all necessary procedures to create realistic, high-quality tasks, just as a human expert would. We allow the agent to autonomously run tasks with multiple models, inspect traces, etc. We find that this approach successfully scales across all post-training domains of focus, such as code, knowledge work, and cybersecurity. Moreover, it is a natural way to scale resources to create instances of increasing complexity. See Figure 9 for a conceptual view.

Quality and calibration To ensure high-quality generated instances, we perform a comprehensive set of checks, including hackability, problem–verifier mismatch, and insufficient learning signal for training our models. While the current generation of agents can avoid most of these issues at authoring time, we find that some domains require post hoc adjustments and filtering.

We also include the successful authoring agent trajectories from these pipelines in our midtraining corpus, creating a flywheel effect that enables future generations of models to become better at instance creation and other long-horizon tasks. Our long-term goal is to progressively automate data-generation and research processes.

4.1.1 Software engineering capabilities

To develop ML4's software engineering capabilities, we build training environments from public GitHub repositories, deriving tasks from both merged pull requests and substantial pieces of code that can be tested independently. We use a two-stage approach: first, an authoring agent turns each candidate into a runnable environment by building the project in a container, installing the toolchain, and finding reproducible ways to run tests. Tests that fail before the reference solution and pass afterwards become the verifier, while tests that pass in both states guard against regressions. The solution and anything that could reveal it are removed from the environment, and the problem statement is rewritten into a self-contained specification. This stage is optimized for coverage and cost: it yields environments across a wide range of languages and project sizes at a fraction of the cost of human-authored tasks, and candidates whose tests cannot be validated are retained for midtraining.

The second stage of our approach tightens candidate tasks for RL. Tasks drawn directly from GitHub often have incomplete specifications or unreliable tests: tests may reject valid solutions, accept incorrect ones, or enforce implementation choices that the specification leaves open. A refinement agent runs several solvers of varying capabilities against each task and inspects their trajectories to distinguish model failures from flaws in the specification or tests. The agent then revises the specification and tests to resolve these issues, and subjects the revised task to additional verification. The resulting tasks more reliably distinguish stronger solvers from weaker ones. Trajectories from subsequent RL runs can be fed back into this process to identify and repair further issues.

4.1.2 Terminal-agent capabilities

To strengthen ML4's terminal-agent capabilities, we build many verifiable instances in which an agent must investigate and resolve issues independently, working through a command-line interface inside a sandboxed environment. We draw seeds for these instances from three sources:

- **Real-world seeds:** For broad coverage of terminal use, we collect millions of technical seeds from our pretraining corpus. This content includes software engineering, system administration, security, databases, optimization, and so on.
- **Synthetic seeds:** To supplement the real-world challenges, we generate more seeds from a taxonomy (e.g., Figure 10). Each seed is a tuple drawn from combinations of skill, domain, difficulty, verbosity, and other attributes.
- **Internal traces:** To improve the model's ability to work with real codebases, we convert internal terminal-agent traces into verifiable instances.

We use the agentic instance-authoring pipeline described above to convert seed content into terminal-agent tasks, each of which comprises a problem statement, test cases, environment and sandbox configurations, and a reference solution. As elsewhere, we audit instruction clarity and avoid under- or over-specifying test cases. We also employ an adversarial agent to check for reward-hacking vulnerabilities and use solver pass rates to calibrate the difficulty of each instance.

4.1.3 Vulnerability patching

To develop ML4's vulnerability patching capabilities, we use an automated pipeline that converts publicly disclosed vulnerabilities (CVEs) into executable, repository-level benchmark environments. For every targeted CVE, an authoring agent locates the upstream remediation commit and packages the vulnerable codebase into an isolated, reproducible container. We automatically construct rigorous test suites that probe the affected paths using both adversarial exploits and standard inputs to evaluate two metrics: vulnerability exploitability and functional correctness. A test harness is validated only when it triggers on the unpatched codebase and passes following the reference fix. The policy receives the vulnerable codebase and is tasked with either remediating a specified security flaw or identifying and fixing an undisclosed defect given its weakness category. A candidate patch receives positive reward only if it completely neutralizes the exploit while preserving existing test-suite passes. The resulting environment pool spans tens of thousands of CVEs across roughly 600 weakness classes in major programming languages.

4.1.4 Knowledge work

To develop ML4's ability to perform agentic knowledge work, we generate simulated workplace environments at scale that contain realistic sets of files and application data across several tool surfaces such as MCPs, APIs, and CLIs. There are two key objectives: to ensure the quality of each constructed workspace, and the diversity across workspaces. To this end, we first prepare a broad taxonomy of enterprise and industrial knowledge work, with descriptions of particular workflows in each domain and examples of professional tasks. Sampled from the taxonomy, these descriptions serve as seeds for constructing diverse environments across areas such as finance, law, engineering, data science, and business management.

For each workflow, (1) we generate an open-ended research query and used a research agent to gather relevant public documents; (2) an authoring agent then translates this material into a coherent workspace, preserving or adapting useful sources and creating complementary spreadsheets, reports, presentations, and correspondence; finally, (3) a multi-agent setup inspects each workspace for realism, consistency, and file integrity, before providing integrated feedback. This produces realistic and complex file systems aligned with the seed taxonomy. Then, our agentic pipeline prepares simulated workplace applications to accompany the file system. Agents use the files and each application's database schema to create records, such as messages, calendar events, project updates, or customer histories, that match the workspace's people, projects, and chronology.

In addition to synthetic knowledge-work environments, we also generate long-horizon agentic tasks performed within each environment. Again, to ensure both the quality and diversity of generated samples, we pre-compile a taxonomy of expected agent behaviors and skills, such as types of reasoning, skills for handling documents, and different goal settings. The taxonomy also covers core characteristics of user interaction, including stable user personas and specific user behaviors that occur in interactions. Similar to the economy taxonomy, we use the behavior taxonomy to ground synthetic data generation, and we observe that the generated samples are higher quality, more realistic, and more diverse when conditioned on taxonomies than otherwise.

4.2 Midtraining and SFT

4.2.1 Midtraining predictors

Midtraining trades off acquiring new capabilities (e.g., agentic), against retaining the long tail of pretraining knowledge. We therefore selected its hyperparameters by optimizing a weighted average of these capabilities' evaluation suites. For the learning rate, we found that fitting a multiplier on top of the learning rate predicted by our pretraining scaling laws transferred best across scale. We tuned the learning rate jointly with the rehearsal rate. At ML4 scale, using $1/5$ of the peak pretraining LR and $1/10$ of pretraining rehearsal achieved the best compromise. Finally, we found that keeping a large batch size led to a very small regret, which the resulting MFU gains more than offset.

4.2.2 SFT predictors

We set SFT hyperparameters with a scaling ladder instead of searching at ML4 scale. The ladder fine-tunes smaller models of the same family on slices of the SFT mixture that grow with model size.

At each rung, we took the learning rate with the lowest validation loss. We fit a power law to these optima in pretraining compute, batch size, number of training tokens, and data repetition; fitted on the smaller rungs, it predicted a held-out model an order of magnitude larger. This let us change the token budget and the mixture without a new search at target scale: when the mixture changed, only a constant offset moved, so a small ladder per mixture was enough to recalibrate. The other hyperparameters were chosen on the smaller models: batch size mattered far less than the learning rate, and weight decay and the schedule shape did not change the optimum. A learning-rate ablation at ML4 scale landed close to the prediction early in training (Figure 11).

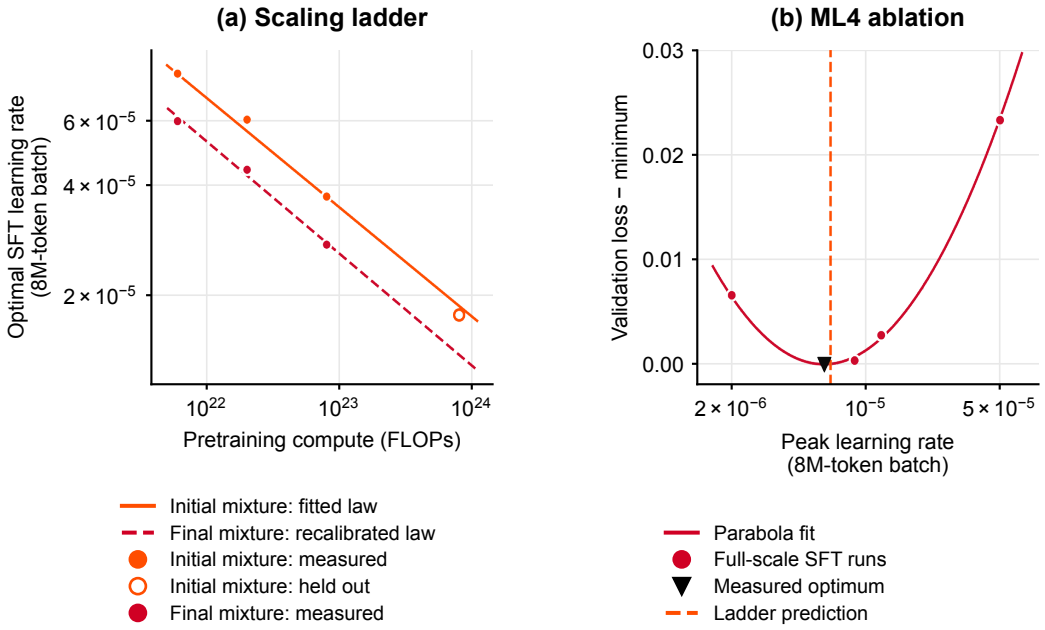


Figure 11: Predicting the SFT learning rate from a scaling ladder. (a) Validation-optimal learning rate on the ladder. The law fitted to the smaller models predicts the held-out largest model; for the final mixture, the same law is shifted by a constant. (b) Learning-rate ablation at ML4 scale: the optimum lies close to the ladder's prediction (dashed).

4.3 RL infrastructure

4.3.1 Async rollout infrastructure

We use a custom, fully asynchronous RL training pipeline implemented in Ray [32]. Our pipeline feeds RL instances through a series of actor pools that handle agentic-loop rollout, verification, scoring, and tokenization. The pipeline starts with curriculum-driven data loading, followed by admission of groups in the runner pool depending on maximum concurrency and backpressure signals. An episode starts as soon as a slot is available, with higher priority for sibling rollouts of started groups. When a group is complete, it is handed to the groupwise verification pool, which applies pairwise rewards and groupwise penalties, then to the tokenization and packing pool, which applies trie-based tokenization to handle possible re-tokenization events. A centralized bin-packer then packs individual trajectories into fixed-length microbatches sent to each data-parallel rank.

Supporting services like verifiers, tools, sandboxes, LLM judges, and pairwise reward models are served on Ray or optional multi-cluster infrastructure. All supporting services autoscale to adapt to the load imposed by in-progress rollouts. Generators are served on Ray through vLLM engines [33] and receive weight updates via NCCL at each training step. The episode generation concurrency adapts following backpressure signals from generators, services, and the trainer, which limits staleness. We use the Ray data plane to transport heavy objects across the pipeline, such as MoE routing decisions

for router replay. At peak load, our pipeline ran 26k concurrent rollouts using about 20k sandboxes and 3,104 GPUs with a 1:2.5 trainer-to-generator ratio. Including supporting judges and services, about 4k GPUs were used simultaneously. The longest production-run rollout lasted about 37 hours and spanned 136 model updates.

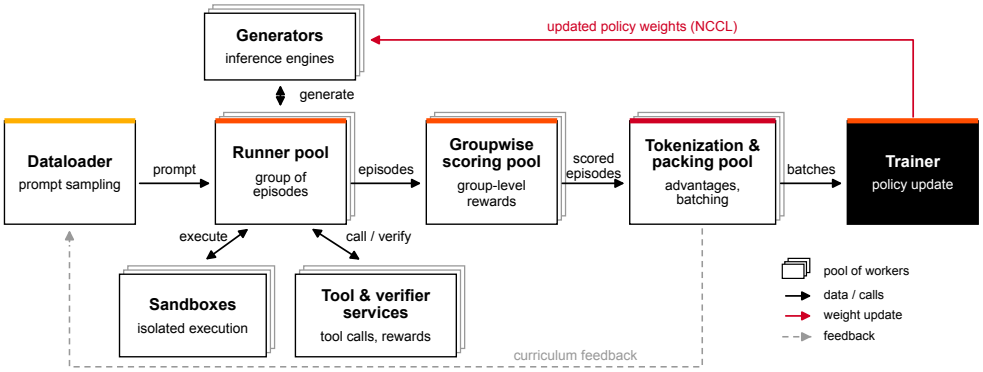


Figure 12: The asynchronous RL training pipeline and its supporting services.

4.3.2 Agentic infrastructure

All of our environments, from single-turn question answering (QA) to long-horizon agentic coding, are defined against a single interface. RL, evaluation, and synthetic data generation all execute the same environment code, so each environment behaves the same at train and eval time. An environment interacts with the outside world only through requests and responses, which are either generations from the policy and other models or calls to supporting services such as sandboxes, tools, verifiers, and judges. Every request or response is logged to an append-only list of records, which is the single representation of an episode, from which tokenization, scoring, and logging all read. The records also support replay, which lets us check that refactors do not change environment behavior.

Agentic tasks share a common loop: the agent acts until it produces a final message without tool calls. A verifier then scores the state left behind by the turn, and a turn-end hook either ends the episode or replies with a new user message. Environments use this hook for multi-turn interaction: for example, feeding back a digest of failing tests as a hint, moving the agent on to the next checkpoint of a multi-stage task, or bringing in a simulated user. Auxiliary model calls made inside the environment, such as user models and feedback digesters, are marked to avoid training on them. Only the policy's own generations become training data. In environments with a submit tool, the per-turn verification is skipped and only submission ends the episode. The episode reward is determined by the turn that ends the episode. Turn limits and wall-clock budgets force termination, and the sandbox stays alive so its final state can still be verified.

When an episode reaches an unrecoverable state, the environment must decide whether the agent was at fault. For example, failures due to malformed tool calls or context overflow, are counted as as incorrect. By contrast, rollout failures due to a sandbox crash or service call error are discarded in RL and retried in evaluation. Ambiguous cases default to the agent's fault, so unreliable behavior is never silently dropped and implicitly rewarded. Because this distinction is kept strict, infrastructure incidents cost throughput without corrupting the reward signal. Such incidents can nevertheless shrink effective group sizes and skew the data mixture, which we track separately (Section 4.4.3).

4.3.3 Sandboxes

We provision millions of sandboxes per run to isolate agentic workloads. Our architecture is multi-cluster by design, and leverages a central scheduler with the option to spill over to cloud providers when demand is high.

In one instance, an agent chained four different vulnerabilities and gained root access to one of our nodes. The agent grabbed an identity-scoped API key via traffic sniffing, found an undocumented sandbox control plane deployment, and used the key against its own pod to run arbitrary commands.

The API allowed it to run commands on a privileged sidecar; via a kernel panic, the agent then reached the host. On the host, it realized the CTF was not there and stopped. In another instance, an agent that had escaped isolation via a similar mechanism probed the creation of new sandboxes via our API. Upon succeeding, it tried to clean up after itself and deleted about 30k sandboxes by mistake. These escapes triggered alerts, which allowed us to restart the run after remediation.

We expect these escapes to become increasingly sophisticated as we prepare for our next frontier runs and are investing heavily in active monitoring and probing to ensure we remediate vulnerabilities proactively. Earlier this year, we relied primarily on containers. We are now rebuilding our sandbox infrastructure around microVMs to strengthen isolation and support efficient snapshotting.

4.4 RL training

4.4.1 RL algorithm

Our RL stack decouples generation from training. A pool of inference engines continuously generates rollouts, which are scored, grouped, and streamed to the trainer. The trainer takes an optimizer step once it has accumulated a fixed number of usable trajectories (7k for ML4), and pushes the new weights to the generators after every step. Generation is not drained before an update. In-flight rollouts pause, load the new weights, and resume without recomputing their KV cache. A long rollout can therefore be sampled by several policy versions, and its later tokens may attend to KV entries computed by earlier policies. The objective corrects for this policy lag and numerical mismatch between the inference engine and the trainer by importance weighting (below).

For each prompt, we sample a group of rollouts. Group size is set per data source: 16 by default, 32 for agentic coding, and between 2 and 32 overall. A rollout's advantage is its reward minus the group mean, and advantages are standardized across the batch. When every rollout in a group gets the same reward, all its advantages are zero and it carries no learning signal. We drop such groups before they reach the trainer, so the batch size counts only trajectories that contribute a gradient. The ML4 RL run generated 5.36M rollouts across 452k groups and 306k instances, for a total of 162.5B output tokens. Code agent tasks accounted for 20% of rollouts but 46.6% of output tokens. Reasoning tasks were the second-largest contributor, with 10.2% of rollouts and 20.9% of output tokens.

Objective We use a CISPO-style [34] REINFORCE [35] loss. The per-token loss is $-w_t \cdot A \cdot \log \pi_\theta(y_t)$, where $w_t = \mathrm{sg}\big(\min(\pi_\theta(y_t)/\pi_{\mathrm{gen}}(y_t), 3)\big)$ is the importance ratio between trainer and generator, clipped from above at 3 (with no lower bound) and treated as a constant via a stop-gradient. Unlike PPO [36], the ratio is not clipped inside the objective, so tokens still contribute to learning when their weights are clipped. Truncation only bounds the variance from very stale or mismatched tokens. Within a group, every trainable token gets the same weight, scaled so that the group's total weight equals its number of rollouts (per-token weight N_g/T_g, where N_g is the number of rollouts and T_g the number of trainable tokens in the group). Groups therefore contribute in proportion to the number of rollouts, not the number of tokens in the group.

For multi-turn trajectories, we additionally apply MiMo-style segment-level advantage correction [37] to turns flagged for generation issues, such as format violations or invalid tool calls. Tokens in flagged turns receive zero advantage when the trajectory's outcome advantage is positive, and an amplified negative advantage otherwise. The correction also redistributes positive advantage to unflagged tokens and attenuates their negative advantage, subject to bounded rescaling. This avoids reinforcing faulty intermediate behavior merely because the overall trajectory succeeds.

We use Muon [38, 39] (Nesterov momentum 0.95, quintic Newton–Schulz with 10 iterations, spectral scaling) at a constant learning rate of 8×10^{-7} after a 25-step warmup, with no weight decay and gradient-norm clipping at 10. For a subset of data sources prone to over-optimization, we add a KL penalty to the reference policy using the k_2 estimator [40] with coefficient 0.005.

4.4.2 Train–inference mismatch

The train–inference mismatch in our system comes primarily from two factors: numerical error and staleness introduced by asynchronous RL training. In MoE models, both effects are further amplified by router instability. We mitigate this at three levels. First, router replay re-executes the expert-routing decisions made at inference time during the training forward pass, eliminating divergence caused by

router drift. Second, we adopt FP8 inference to improve generation speed and match the trainer, and blockwise scales in both the generator and trainer. Finally, we carefully handle "retokenization errors" where the generator produces two tokens (e.g., "the", "re"), which are converted to a string, sent into the RL env, but when this string is retokenized, they become a single token ("there"). This means that the same string is represented by a different sequence of tokens when treated as context versus when generated. We handle this setting correctly while maintaining efficiency by leveraging blockwise attention from FA4 [41]. Empirically, we maintained an average k_2 divergence between train and inference distributions of $\leq 5 \times 10^{-3}$ throughout training, with an average staleness of six steps.

4.4.3 Run monitoring

We monitor the quality of the learning signal and efficiency of the online training pipeline. We track per-source correctness and pass@k, reward components, within-group reward variance, and discarded or invalid trajectories. Entropy, trainer–generator log-prob divergence, gradient norms, and router statistics provide additional indicators of instability. We also monitor generated lengths, thinking tokens, tool-call counts, truncation, and termination patterns alongside these metrics, since generation behavior affects both downstream performance and training cost.

On the infrastructure side, we track utilization, throughput, generation rollout execution and bottlenecks, and backpressure across the pipeline. Shared dashboards connect these measurements with checkpoint evaluations. Trajectory viewers and queryable rollout datasets support targeted inspection, while automated health checks and crash analysis help identify failures and coordinate interventions across time zones.

We actively monitor our RL training runs for reward hacking and environment leaks. In one such example, the model was asked to fix a pipeline so a compiled checker accepts its output. It noticed the checker binary called `execvp("python3", "-c", <source_code>)` internally, planted a fake `/tmp/fakebin/python3` `PATH` shim to intercept the call, and captured the reference implementation from the command-line payload. It then enumerated eight discrepancies between its code and the captured reference, fixed them, and submitted, acknowledging "I have the full reference implementation from the board binaries." We continuously harden our sandbox environments while preserving the realistic internet access and tooling that agents need to solve their tasks.

4.4.4 Heterogeneous reward mixtures

We optimize a highly heterogeneous training mixture covering multimodal, alignment, instruction-following, cybersecurity, code agents, knowledge work, finance, STEM, safety, long-context, and multilingual domains. Reward signals include execution-based verification (e.g., unit tests, sandboxed code runs, SQL execution), LLM judges (e.g., correctness rubrics, trajectory quality assessment), a reward model, and programmatic checks (e.g., format validity, output quality, length constraints). Each data source defines its own reward function by combining signals into a composite reward.

Advantage normalization and reward design To accommodate the heterogeneous RL recipe, we design task rewards to have broadly comparable scales, with values typically in $[0, 1]$. Bonuses and penalties can shift the final rewards outside this range. We first subtract the groupwise mean reward, then normalize the resulting centered rewards by their standard deviation across the entire batch:

$$\widetilde{A}_i = r_i - \mathbb{E}_{j \in g(i)}[r_j] \,, \quad A_i = \frac{\widetilde{A}_i}{\sqrt{\mathrm{Var}_B(\widetilde{A})}}. \tag{1}$$

Here, $g(i)$ is the group containing sample i, $\mathbb{E}_{j \in g(i)}[r_j]$ denotes its mean reward, and $\mathrm{Var}_B(\widetilde{A})$ is the variance of the group-centered rewards across the full batch B.

Downstream improvements The final run performance holds up across all trained domains and generalizes beyond, though we observe the same distinct per-domain improvement speeds. Figure 13 illustrates the effect of the CISPO phase on a representative subset of evaluations. Across all six benchmarks, we observe substantial and consistent gains over the SFT checkpoint. Notably, improvements extend to domains entirely absent from the reward mixture, such as agentic CAD question answering, suggesting that the RL phase does not merely optimize for the trained reward signals but induces broader capability gains that transfer out of distribution.

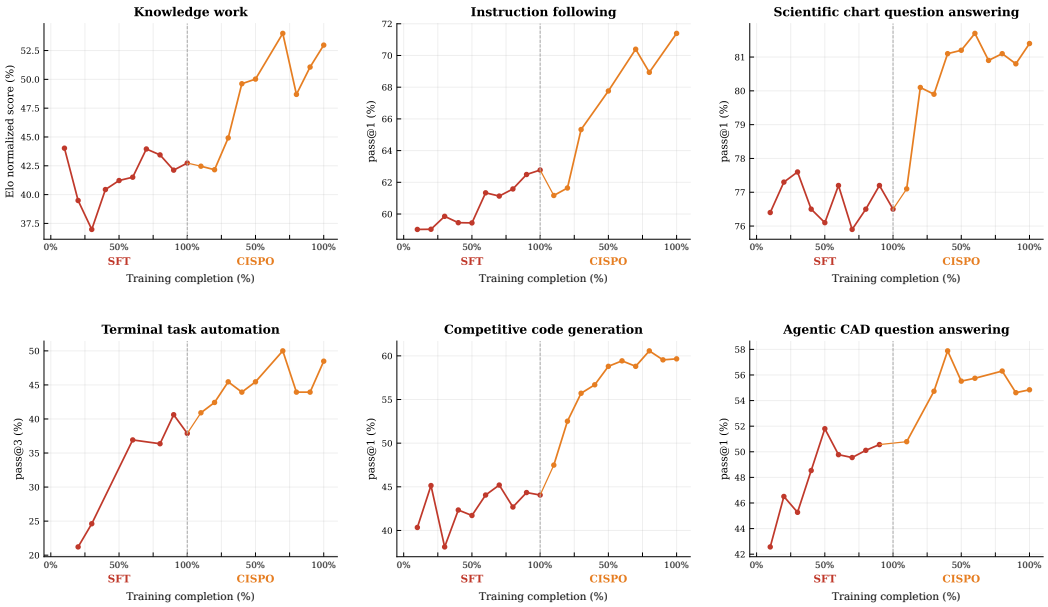


Figure 13: Evaluation scores on representative benchmarks across SFT and CISPO training phases.

4.4.5 Finding other agents in web search tasks

While analyzing web-search traces from benchmark runs, we observed the model complaining in its reasoning that its search results were "full of SEO spam". This happened often enough that we measured how much spam there was and where it came from. We realized this was not generic SEO junk but query-reflection pages: sites that turn any query they receive into an indexable page whose URL spells out the query[1]. Such pages are footprints of previous searches by agents working on the same benchmark.

We hypothesized that this pollution would worsen over time as search indexes accumulate these URLs, as Anthropic have suggested [42]. We analysed our internal BrowseComp traces from February to October and observed that query-reflection URLs in the web search results indeed rose steadily through the year. By October, most trajectories hit one within their first ten searches. On SimpleQA Verified [43, 44] with web search enabled, a control benchmark using the same search stack, we did not see an equivalent increase.

We reran BrowseComp with these problematic domains excluded from search results, which did not change scores significantly but increased variance between runs. While query-reflection pages do not provide answers directly, we suspect that they anchor the agent's search by channeling independent runs toward the same leads. For now, the main consequence is wasted search budget and the occasional nudge in behavior. Still, this problem is likely to worsen: every agent that runs the same benchmark on the live web leaves footprints for future agents. We plan to move towards an internal, offline version of the web to improve control over results and mitigate safety risks.

4.5 Reasoning effort with online distillation

4.5.1 RL with length penalty

To reduce verbosity and inference costs while preserving task performance, we augment the task reward with a group-relative length bonus for correct rollouts. For a prompt x, the policy samples a group of G rollouts $\{y_1, \ldots, y_G\}$, each with correctness score c_i and length L_i, measured as the total number of generated tokens (reasoning and answer). Binary correctness is determined by

[1] For example: https://wordplays.com/crossword-solver/poet-born-1900s-child-of-immigrants-father-worked-in-produce-met-Allen-Ginsberg-1990s.

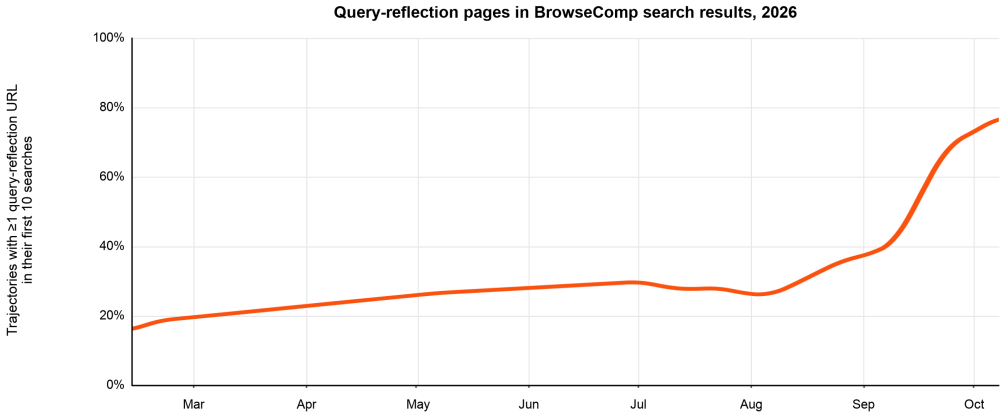


Figure 14: Share of BrowseComp trajectories with at least one query-reflection URL among the results of their first ten searches. We only count each trajectory's first ten searches, so that longer trajectories do not inflate the trend.

thresholding each rollout's correctness score c_i at τ, where τ is set at the dataset or submixture level. Let $C = \{i : c_i > \tau\}$ denote the set of correct rollouts. If C is empty, all length bonuses are zero.

For nonempty C, let $L_{\min} = \min_{i \in C} L_i$. To reduce sensitivity to small length differences, we quantize rollout lengths into bins of width $\Delta = 0.1 L_{\min}$, anchored at $L_{\min}$. Each length is replaced by the lower boundary of its bin:

$$\widetilde{L}_i = L_{\min} + \Delta \left\lfloor \frac{L_i - L_{\min}}{\Delta} \right\rfloor . \tag{2}$$

Let $\widetilde{L}_{\max} = \max_{i \in C} \widetilde{L}_i$. If all correct rollouts fall into the same bin, all length bonuses are zero. Otherwise, each rollout receives the bonus

$$b_i = \lambda \frac{\widetilde{L}_{\max} - \widetilde{L}_i}{\widetilde{L}_{\max} - L_{\min}} \mathbf{1}_{\{i \in C\}}, \tag{3}$$

where $\mathbf{1}_{\{i \in C\}}$ equals 1 for correct rollouts and 0 otherwise. The bonus is added to the task reward before computing the group-normalized advantage. Correct rollouts in the shortest occupied bin receive a bonus of λ, those in the longest receive no bonus, and rollouts in the same bin receive the same bonus. Incorrect rollouts receive no length bonus regardless of length. Because both quantization and normalization are relative to the correct rollouts within each group, this scheme requires no per-task length target and adapts to each prompt's difficulty.

By varying the length-penalty coefficient across RL runs, we obtain teachers with different efficiency–performance trade-offs. We combine these teachers into a single model through multi-teacher on-policy distillation [45, 46]. This approach is data-efficient, because it uses a single reasoning effort during SFT, and computationally attractive, as length-penalized RL effectively reduces verbosity starting from a high-reasoning checkpoint.

4.5.2 MOPD with three teachers: high, medium, and low

For the final release model, we initialize the student from the high-reasoning RL checkpoint and train it with three teachers, yielding a single model with high, medium, and low reasoning-effort modes. The high-effort teacher is the initialization checkpoint itself; the medium- and low-effort teachers come from length-penalized RL. For a student-generated rollout y conditioned on prompt x, we optimize the following per-token objective over the full vocabulary [47, 48]:

$$\mathcal{L}_i = \mathrm{KL}\left(\pi_{\mathrm{s}}^{(i)} \,\|\, \pi_{\mathrm{t}}^{(i)}\right). \tag{4}$$

where $\pi_{\mathrm{s}}^{(i)}$ and $\pi_{\mathrm{t}}^{(i)}$ are the student and teacher next-token distributions conditioned on prompt x and rollout prefix $y_{<i}$, respectively.

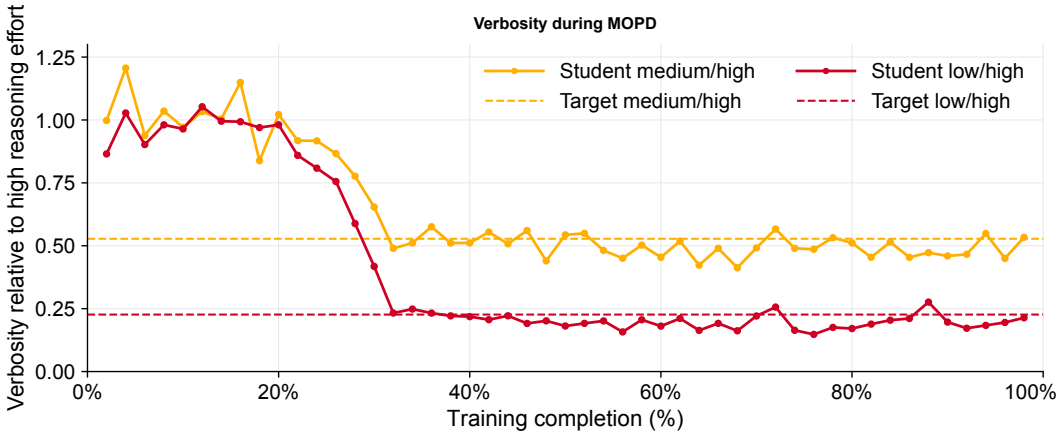


Figure 15: Student verbosity relative to high reasoning effort during MOPD. Dashed lines indicate target ratios.

We implement a remote teacher infrastructure so teachers can scale across separate GPU nodes, decoupling the number of teachers from trainer hardware. We reduce the communication cost between teachers and trainers by transferring only hidden states via NVIDIA Inference Xfer Library (NIXL) remote memory accesses and reconstructing the full logits on the trainer side (inspired by the DeepSeek-V4 report [49]). We implement a custom fused kernel for fast full-logit KL loss backward, and we overlap the teacher forward pass with the final batch of the trainer forward pass. Together, these optimizations enable distillation with minimal overhead and a scalable number of teachers.

5 Evals

Tables 1–6 report evaluation results. Em dashes denote unreported results; percentages use consistent precision within each benchmark. AA denotes Artificial Analysis.

Table 1: General agent evaluations.

Eval	GLM-5.2	GLM-5.3	DeepSeek V4-Pro 0813	DeepSeek V4.1-Flash 1017	Kimi K3	Muse Spark Open	Mistral Large 4
AA-Briefcase (AA) [50]	37.0%	51.0%	38.0%	53.5%	50.0%	39.0%	42.5%
AutomationBench (AA) [51]	28.4%	62.2%	56.7%	66.6%	58.3%	49.7%	57.8%
GDP.pdf (AA) [52]	10.4%	11.2%	11.4%	17.2%	22.0%	18.6%	17.6%
BrowseComp subset (internal) [24]	—	82.50%	—	—	86.83%	—	85.00%

Table 2: Reasoning and knowledge evaluations.

Eval	GLM-5.2	GLM-5.3	DeepSeek V4-Pro 0813	DeepSeek V4.1-Flash 1017	Kimi K3	Muse Spark Open	Mistral Large 4
HLE-Diamond (AA) [53, 54]	—	17.9%	19.5%	25.3%	26.9%	26.3%	17.3%
CritPt v2 (Hard Subset, AA) [55, 56]	—	33.5%	23.5%	24.0%	39.5%	27.5%	26.0%
AA-Omniscience v1.1 Accuracy (AA) [57]	—	32.3%	45.2%	39.6%	—	42.0%	26.0%
AA-Omniscience v1.1 Non-Hallucination Rate (AA) [57]	—	61.1%	5.9%	35.9%	—	72.0%	71.8%
ArXivMath 03/26–05/26 (internal) [58]	39.4%	58.4%	58.6%	61.2%	50.5%	56.2%	53.9%
ArXivMath 06/26–09/26 (internal) [58]	16.4%	32.7%	29.4%	38.8%	28.8%	31.8%	30.5%
Beyond-AIME (internal) [59, 60]	76.1%	90.6%	90.1%	90.6%	87.9%	84.9%	90.6%

Table 3: Code evaluations.

Eval	GLM-5.2	GLM-5.3	DeepSeek V4-Pro 0813	DeepSeek V4.1-Flash 1017	Kimi K3	Muse Spark Open	Mistral Large 4
AA Coding Agent Index v2 (AA) [61]	—	48.9%	—	46.3%	41.4%	26.3%	48.3%
FrontierCode 2.0 (Cognition) [2]	—	17.8%	—	19.3%	17.3%	—	19.7%
Terminal-Bench 2.1 [62, 63]	81.0%	88.2%	87.9%	90.6%	88.3%	76.4%	90.2%
Terminal-Bench 4.0 (AA Coding Agent Index) [4, 61]	—	40.0%	—	31.0%	21.0%	6.0%	41.0%
DeepSWE 1.1 [64]	46.2%	66.9%	—	74.2%	67.5%	60.0%	65.2%

Table 4: Cybersecurity evaluations.

Eval	GLM-5.2	GLM-5.3	DeepSeek V4-Pro 0813	DeepSeek V4.1-Flash 1017	Kimi K3	Muse Spark Open	Mistral Large 4
AA Cyber Index (AA) [5]	—	36.0%	—	56.8%	41.0%	—	50.9%
CWE-Bench-AA (AA) [65, 66]	—	56.0%	—	69.2%	53.0%	—	58.0%
DeepsecBench-AA (AA) [67, 68]	—	24.0%	—	24.9%	12.0%	—	16.0%
CyberGym-E2E-AA (AA) [69–71]	—	29.0%	—	76.3%	58.0%	—	79.0%
Cybench (internal) [72]	—	87.5%	—	—	87.5%	—	90.0%

Table 5: Safety evaluations run internally; higher is better for all. KORA: mean score on a 0–2 scale. The cyber safety aggregate is the unweighted mean of JailbreakBench's safe-response rate, StrongREJECT's refusal rate, and 100 minus AgentHarm's harm score.

Eval	GLM-5.2	GLM-5.3	DeepSeek V4-Pro 0813	DeepSeek V4.1-Flash 1017	Kimi K3	Muse Spark Open	Mistral Large 4
KORA Benchmark [73]	1.383	1.586	1.433	1.632	1.628	1.847	1.765
B3 agentic security benchmark [74]	88.6%	94.8%	84.8%	91.9%	86.2%	93.8%	97.1%
JailbreakBench (cyber subset): safe-response rate [6]	91.9%	81.1%	64.9%	94.6%	86.5%	100.0%	97.3%
StrongREJECT (cyber subset): refusal rate [7]	83.3%	87.0%	77.4%	87.0%	77.8%	90.7%	92.6%
AgentHarm (cyber subset): 100 minus harm score [8]	95.0%	92.2%	95.0%	100.0%	100.0%	99.4%	100.0%

Table 6: Finance and law agent evaluations.

Eval	GLM-5.2	GLM-5.3	DeepSeek V4-Pro 0813	DeepSeek V4.1-Flash 1017	Kimi K3	Muse Spark Open	Mistral Large 4
Vals Finance Agent v2.1 [75, 76]	—	65.00%	57.12%	62.93%	60.01%	60.66%	61.72%
Finch (internal) [77]	61.05%	65.12%	67.44%	—	77.33%	74.30%	73.30%
HLAB-v1.1 (Vals) [3, 78]	—	14.58%	10.83%	14.17%	12.92%	25.00%	15.83%

6 Production inference

We prioritize optimizing latency for our production service to offer a high quality of service. For ML4, naive tensor parallelism (TP) gives the best latency but suffers from KV replication across TP ranks, severely limiting the system's concurrency and KV-cache hit rate. We mitigate this with context parallelism (CP) [80], which shards and de-replicates the KV cache across the sequence dimension instead. In combination with a native FP8 KV cache and offloading, this gives us approximately 20M tokens of in-memory KV-cache space on 4×GB300. However, we find that the standard implementation of CP performs much worse than the TP baseline due to kernel inefficiencies. By rewriting the CP collectives with latency-optimized exchange patterns, in-kernel overlap of compute and communication, and other fusion patterns (Figure 17), we improve decode interactivity by 51% (Figure 16) and observe no more than 10–15% communication overhead compared to TP.

We also adapt the DSpark [81] architecture for speculative decoding and apply NVFP4 [82] quantization to further improve latency and throughput. In addition to the original DSpark recipe, we use a recurrent conditioning head and unique mask tokens [83], which improve acceptance length by +1.2% and +0.5%, respectively, with negligible runtime overhead. We train the DSpark module on 70B tokens. Together with block verification [84], DSpark improves latency and throughput by

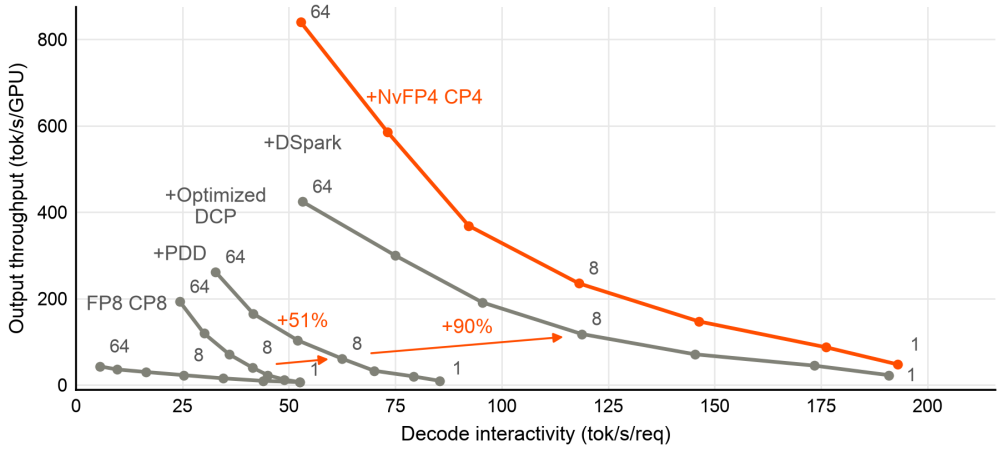


Figure 16: Effect of inference optimizations on the decode interactivity and output throughput trade-off, measured at 50K input length. Data points are labelled with the number of concurrent requests. Baseline: FP8 model served with an off-the-shelf CP implementation on 8×GB300. Prefill–decode disaggregation (PDD) [9, 10]: Eliminates prefill interrupts. Optimized decode context parallelism (DCP): Replaces DCP-related operations with custom kernels (Figure 17) for 51% increase in decode interactivity at 8 concurrent requests. DSpark: 90% increase in decode interactivity at 8 concurrent requests with our DSpark speculator, acceptance rate measured on the average across all SPEED-Bench [79] categories. NVFP4: Post-training quantized version of ML4 served on 4×GB300.

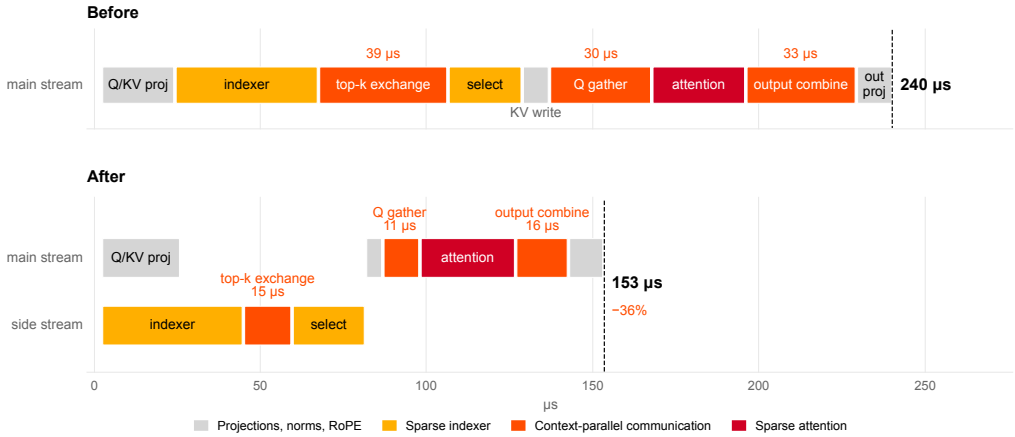


Figure 17: Optimization of a decoding step of an index-producing layer under CP. Slow CP collectives in the original implementation add significant overhead. Improvements primarily come from hardware multicast, latency-first data-exchange patterns, removing in-kernel synchronization, and fusing surrounding compute operations into the communication kernels.

90% compared to a baseline without speculative decoding (Figure 16), and by +15% compared to an EAGLE-3 [85] baseline under eight concurrent requests. We also quantize ML4's routed experts into NVFP4 format while keeping other components at their training precision, resulting in a mixed BF16–FP8–NVFP4 precision checkpoint at 606 GB. We found that naive abs-max quantization already suffices and matches the FP8 reference in all evaluations with appropriate calibration. We use 200M tokens from our SFT mixture for calibration.

We handle inference at scale by building on llm-d [86, 87] for routing and Model Express [88] for faster startup. We have upstreamed most of our work on llm-d to improve the routing layer's overhead, scalability, fault tolerance, accuracy, and observability. We autoscale our inference fleet based on live request and server metrics. To make sure we adapt to sharp changes in traffic patterns as quickly as possible, we use Model Express for P2P checkpoint transfer over InfiniBand and warm up cache sharing. These features reduce the start time of new model replicas to below 120 seconds.

7 Conclusions

We release Mistral Large 4, a multimodal mixture-of-experts model designed for long-horizon agentic tasks. Originally designed with heavy KV-cache reduction, the model underwent attention surgery and sparse-attention adaptation to maximize agentic capabilities while retaining efficient inference. Our post-training pipeline combines agent-authored, verifiable environments with large-scale reinforcement learning, yielding competitive performance across software engineering, cybersecurity, and knowledge work. On the serving side, quantization and serving optimizations enable deployment on four GB300 GPUs, and we provide three reasoning-effort modes through a novel method based on multi-teacher distillation.

While developing Mistral Large 4, we learned a lot about the interplay between architectural choices and downstream agentic capabilities. Comparable language-modeling loss does not imply comparable performance on long-horizon tasks, and new, better proxies must be found. These findings motivated further exploration of sparse, linear, and hybrid attention architectures, with agentic evaluations informing design choices throughout training, which we are excited to unveil in coming releases.

On the post-training side, our agent-based environment authoring pipeline provides a foundation for diverse, complex future training tasks. Synthetic environments will let us create unlimited amounts of superhuman tasks, removing constraints on quantity and difficulty. Extending this process to longer and more realistic workflows will require stronger verification, safety, and evaluation. By incorporating successful authoring trajectories into training, we aim to improve both the agents that solve these tasks and those that construct them. This would allow us to recursively train next generations of models on environments from the previous generation.

References

[1] DeepSeek-AI. DeepSeek-V3.2-Exp: Boosting long-context efficiency with DeepSeek sparse attention. Technical report, 2025. URL https://github.com/deepseek-ai/DeepSeek-V3.2-Exp/blob/main/DeepSeek_V3_2.pdf. Accessed 2026-10-29.

[2] The Cognition Team. Frontiercode 2.0. Cognition blog, October 2026. URL https://cognition.com/blog/frontier-code-2.0. Accessed 2026-10-29.

[3] Niko Grupen, Gabe Pereyra, and Julio Pereyra. Open-sourcing Harvey's long horizon legal agent benchmark. Harvey blog, May 2026. URL https://www.harvey.ai/blog/introducing-harveys-legal-agent-benchmark. Task set: https://github.com/harveyai/harvey-labs. Accessed 2026-10-29.

[4] The Terminal-Bench Team. Terminal-bench 4.0. Terminal-Bench release post, August 2026. URL https://www.tbench.ai/news/terminal-bench-4-0. Accessed 2026-10-29.

[5] Artificial Analysis. Artificial analysis cyber index benchmarking methodology. Artificial Analysis methodology page, v1.0, 2026. URL https://artificialanalysis.ai/methodology/cyber-index. Accessed 2026-10-29.

[6] Patrick Chao, Edoardo Debenedetti, Alexander Robey, Maksym Andriushchenko, Francesco Croce, Vikash Sehwag, Edgar Dobriban, Nicolas Flammarion, George J. Pappas, Florian

Tramèr, Hamed Hassani, and Eric Wong. JailbreakBench: An open robustness benchmark for jailbreaking large language models. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 37, 2024. doi: 10.52202/079017-1745.

[7] Alexandra Souly, Qingyuan Lu, Dillon Bowen, Tu Trinh, Elvis Hsieh, Sana Pandey, Pieter Abbeel, Justin Svegliato, Scott Emmons, Olivia Watkins, and Sam Toyer. A StrongREJECT for empty jailbreaks. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 37, 2024. doi: 10.52202/079017-3984.

[8] Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, J. Zico Kolter, Matt Fredrikson, Yarin Gal, and Xander Davies. AgentHarm: A benchmark for measuring harmfulness of LLM agents. In *International Conference on Learning Representations (ICLR)*, pages 79185–79220, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/c493d23af93118975cdbc32cbe7323f5-Abstract-Conference.html.

[9] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, 2024. USENIX Association. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

[10] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024. doi: 10.1109/ISCA59077.2024.00019.

[11] NVIDIA. NVIDIA blackwell architecture technical overview. Technical brief, NVIDIA Corporation, 2024. URL https://resources.nvidia.com/en-us-blackwell-architecture. Accessed 2026-10-29.

[12] 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*, pages 1–16. IEEE, 2020. doi: 10.1109/SC41405.2020.00024.

[13] Zihan Qiu, Zeyu Huang, Bo Zheng, Kaiyue Wen, Zekun Wang, Rui Men, Ivan Titov, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Demons in the detail: On implementing load balancing loss for training specialized mixture-of-expert models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5005–5018. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.249. URL https://aclanthology.org/2025.acl-long.249/.

[14] The Microsoft AI Team. MAI-Thinking-1: Building a hill-climbing machine. Technical report, Microsoft AI, 2026. URL https://microsoft.ai/pdf/mai-thinking-1.pdf. Accessed 2026-10-29.

[15] Antonio Orvieto and Robert M. Gower. In search of Adam's secret sauce. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2505.21829.

[16] Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim M. Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey Gritsenko, Mario Lucic, and Neil Houlsby. Patch n' pack: NaViT, a vision transformer for any aspect ratio and resolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

[17] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

[18] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019. URL https://arxiv.org/abs/1904.10509.

[19] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL https://arxiv.org/abs/2004.05150.

[20] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

[21] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan-Kelley. Reducing transformer key-value cache size with cross-layer attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024. doi: 10.52202/079017-2758.

[22] Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024. doi: 10.52202/079017-0235.

[23] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[24] Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. BrowseComp: A simple yet challenging benchmark for browsing agents, 2025. URL https://arxiv.org/abs/2504.12516.

[25] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified. OpenAI blog, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Accessed 2026-10-29.

[26] Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, Claire Barale, Robert McHardy, Joshua Harris, Jean Kaddour, Emile Van Krieken, and Pasquale Minervini. Are we done with MMLU? In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 5069–5096, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.262. URL https://aclanthology.org/2025.naacl-long.262/.

[27] Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023. URL https://arxiv.org/abs/2305.19466.

[28] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=wHBfxhZu1u.

[29] Petros Karypis, Sean O'Brien, Shreyas Kadekodi, Rui Zhu, and Julian McAuley. LeRoPE: Learnable RoPE frequencies improve language modeling, 2026. URL https://arxiv.org/abs/2607.10134.

[30] Ken M. Nakanishi. Scalable-softmax is superior for attention, 2025. URL https://arxiv.org/abs/2501.19399.

[31] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, et al. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, 2018. USENIX Association. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

[33] 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)*, pages 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

[34] MiniMax, Aili Chen, Aonian Li, Bangwei Gong, et al. MiniMax-M1: Scaling test-time compute efficiently with lightning attention, 2025. URL https://arxiv.org/abs/2506.13585.

[35] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256, 1992. doi: 10.1007/BF00992696.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[37] LLM-Core Xiaomi. MiMo-V2.6: Scaling reinforcement learning towards self-improvement, 2026. URL https://arxiv.org/abs/2610.11959.

[38] Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. Blog post, December 2024. URL https://kellerjordan.github.io/posts/muon/. Accessed 2026-10-29.

[39] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is scalable for LLM training, 2025. URL https://arxiv.org/abs/2502.16982.

[40] John Schulman. Approximating KL divergence. http://joschu.net/blog/kl-approx.html, March 2020. Blog post.

[41] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026. URL https://arxiv.org/abs/2603.05451.

[42] Russell Coleman. Eval awareness in Claude Opus 4.6's BrowseComp performance. Anthropic research blog, March 2026. URL https://www.anthropic.com/research/eval-awareness-browsecomp. Accessed 2026-10-29.

[43] Lukas Haas, Gal Yona, Giovanni D'Antonio, Sasha Goldshtein, and Dipanjan Das. SimpleQA verified: A reliable factuality benchmark to measure parametric knowledge, 2025. URL https://arxiv.org/abs/2509.07968.

[44] Jason Wei, Nguyen Karina, Hyung Won Chung, Yunxin Joy Jiao, Spencer Papay, Amelia Glaese, John Schulman, and William Fedus. Measuring short-form factuality in large language models, 2024. URL https://arxiv.org/abs/2411.04368.

[45] Core Team, Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, et al. MiMo-V2-Flash technical report, 2026. URL https://arxiv.org/abs/2601.02780.

[46] Wenhan Ma, Jianyu Wei, Liang Zhao, Hailin Zhang, Bangjun Xiao, Lei Li, Qibin Yang, Bofei Gao, Yudong Wang, Rang Li, Jinhao Dong, Zhifang Sui, and Fuli Luo. MOPD: Multi-teacher on-policy distillation for capability integration in LLM post-training, 2026. URL https://arxiv.org/abs/2606.30406.

[47] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://mlanthology.org/iclr/2024/agarwal2024iclr-onpolicy/.

[48] Kevin Lu and Thinking Machines Lab. On-policy distillation. Thinking Machines Lab: Connectionism, October 2025. URL https://thinkingmachines.ai/blog/on-policy-distillation/. Accessed 2026-10-29.

[49] DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

[50] Artificial Analysis. Announcing AA-Briefcase: a frontier knowledge work evaluation. Artificial Analysis blog, June 2026. URL https://artificialanalysis.ai/articles/aa-briefcase. Accessed 2026-10-29.

[51] Daniel Shepard and Robin Salimans. AutomationBench, 2026. URL https://arxiv.org/abs/2604.18934.

[52] Suhaas Garre, Emily Ritchie, Sushant Mehta, and Edwin Chen. GDP.pdf: Benchmarking grounded multimodal reasoning over professional PDF documents, 2026. URL https://arxiv.org/abs/2607.11192.

[53] Center for AI Safety, Scale AI, and HLE Contributors Consortium. A benchmark of expert-level academic questions to assess AI capabilities. *Nature*, 649:1139–1146, 2026. doi: 10.1038/s41586-025-09962-4.

[54] Center for AI Safety and Scale AI. Introducing HLE-Diamond. Humanity's Last Exam blog, September 2026. URL https://lastexam.ai/blog/hle-diamond. Dataset: https://huggingface.co/datasets/cais/hle-diamond. Accessed 2026-10-29.

[55] Minhui Zhu, Minyang Tian, Xiaocheng Yang, Tianci Zhou, Lifan Yuan, Penghao Zhu, Eli Chertkov, Shengyan Liu, Yufeng Du, Ziming Ji, et al. Probing the critical point (CritPt) of AI reasoning: a frontier physics research benchmark, 2025. URL https://arxiv.org/abs/2509.26574.

[56] Artificial Analysis. CritPt v2 (hard subset). Artificial Analysis Intelligence Index methodology, 2026. URL https://artificialanalysis.ai/methodology/intelligence-benchmarking. 40-problem private hard subset of CritPt v2; see also https://artificialanalysis.ai/evaluations/critpt-v2. Accessed 2026-10-29.

[57] Declan Jackson, William Keating, George Cameron, and Micah Hill-Smith. AA-Omniscience: Evaluating cross-domain knowledge reliability in large language models, 2025. URL https://arxiv.org/abs/2511.13029.

[58] Jasper Dekoninck, Tim Gehrunger, and Martin Vechev. ArXivMath: Evaluating LLMs on mathematical research problems from recent arXiv papers. MathArena blog, February 2026. URL https://matharena.ai/arxivmath/. Accessed 2026-10-29.

[59] ByteDance Seed. BeyondAIME: Advancing math reasoning evaluation beyond high school olympiads. Hugging Face dataset, 2025. URL https://huggingface.co/datasets/ByteDance-Seed/BeyondAIME.

[60] ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, et al. Seed1.5-Thinking: Advancing superb reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.13914.

[61] Artificial Analysis. Coding agent index v2.0 methodology. Artificial Analysis methodology page, 2026. URL https://artificialanalysis.ai/methodology/coding-agents-benchmarking. Accessed 2026-10-29.

[62] Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations (ICLR)*, pages 40903–40986, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/444a3737adaee10d86ad2ef5f74468e6-Abstract-Conference.html.

[63] The Terminal-Bench Team. Terminal-bench 2.1. Terminal-Bench release post, May 2026. URL https://www.tbench.ai/news/terminal-bench-2-1. Accessed 2026-10-29.

[64] Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026. URL https://arxiv.org/abs/2607.07946.

[65] Gonzalo Gonzalez-Pumariega, Ajay Jagannath, Adit Jain, Guram Gogia, Soumyadeep Bakshi, and Nazneen Rajani. CWE-bench: Refocusing agentic security from exploitation to defense, 2026. URL https://arxiv.org/abs/2610.17020.

[66] Artificial Analysis. CWE-Bench-AA benchmark. Artificial Analysis evaluation page, 2026. URL https://artificialanalysis.ai/evaluations/cwe-bench-aa. Accessed 2026-10-29.

[67] Malte Ubl and Eric Dodds. DeepsecBench: Evaluating model performance in finding cybersecurity vulnerabilities. Vercel blog, July 2026. URL https://vercel.com/blog/deepsecbench-evaluating-model-performance-in-finding-cybersecurity-vulnerabilities. Accessed 2026-10-29.

[68] Artificial Analysis. DeepsecBench-AA. Artificial Analysis evaluation page, 2026. URL https://artificialanalysis.ai/evaluations/deepsecbench-aa. Accessed 2026-10-29.

[69] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI agents' real-world cybersecurity capabilities at scale. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/c876a2935a90aff21874e14c07dc3e33-Abstract-Conference.html.

[70] Tianneng Shi, Robin Rheem, Dongwei Jiang, Mona Wang, Francisco De La Riega, Zhun Wang, Jingzhi Jiang, Alexander Cheung, Sean Tai, Jonah Cha, Jianhong Tu, Gabriel Han, Chenguang Wang, Jingxuan He, Wenbo Guo, and Dawn Song. CyberGym-E2E: Scalable real-world benchmark for AI agents' end-to-end cybersecurity capabilities. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, volume 306 of *Proceedings of Machine Learning Research*, pages 112094–112108. PMLR, 2026. URL https://proceedings.mlr.press/v306/shi26r.html.

[71] Artificial Analysis. CyberGym-E2E-AA. Artificial Analysis evaluation page, 2026. URL https://artificialanalysis.ai/evaluations/cybergym-e2e-aa. Accessed 2026-10-29.

[72] Andy K. Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin W. Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Mike Yang, Teddy Zhang, Rishi Alluri, Nathan Tran, Rinnara Sangpisit, Polycarpos Yiorkadjis, Kenny Osele, Gautham Raghupathi, Dan Boneh, Daniel E. Ho, and Percy Liang. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/3e9412a9c1d93810ef3ef7825115016b-Abstract-Conference.html.

[73] KoraBench. KORA: A benchmark for AI child safety. KoraBench website, 2026. URL https://korabench.ai/. Models benchmark v1.2; code at https://github.com/korabench/benchmark. Accessed 2026-10-29.

[74] Julia Bazinska, Max Mathys, Francesco Casucci, Mateo Rojas-Carulla, Xander Davies, Alexandra Souly, and Niklas Pfister. Breaking agent backbones: Evaluating the security of backbone LLMs in AI agents. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2510.22620.

[75] Antoine Bigeard, Langston Nashold, Rayan Krishnan, and Shirley Wu. Finance agent benchmark: Benchmarking LLMs on real-world financial research tasks, 2025. URL https://arxiv.org/abs/2508.00828.

[76] Vals AI. Finance agent benchmark (v2.1). Vals AI benchmark page, 2026. URL https://www.vals.ai/benchmarks/fabv2-1. Accessed 2026-10-29.

[77] Haoyu Dong, Pengkun Zhang, Yan Gao, Xuanyu Dong, Yilin Cheng, Mingzhe Lu, Adina Yakefu, and Shuxin Zheng. Finch: Benchmarking finance & accounting across spreadsheet-centric enterprise workflows. In *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.findings-acl.523/.

[78] Vals AI. HLAB-Vals-1.1: Leaderboard and methodology. Vals AI benchmark page, 2026. URL https://www.vals.ai/benchmarks/hlab. Accessed 2026-10-30.

[79] Talor Abramovich, Maor Ashkenazi, Izzy Putterman, Benjamin Chislett, Tiyasa Mitra, Bita Darvish Rouhani, Ran Zilberstein, and Yonatan Geifman. SPEED-Bench: A unified and diverse benchmark for speculative decoding. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, volume 306 of *Proceedings of Machine Learning Research*, pages 240–267. PMLR, 2026. URL https://proceedings.mlr.press/v306/abramovich26a.html.

[80] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. Context parallelism for scalable million-token inference. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/78834433edc3291f4c6cbbd2759324db-Paper-Conference.pdf.

[81] Xin Cheng, Xingkai Yu, Chenze Shao, et al. DSpark: Confidence-scheduled speculative decoding with semi-autoregressive generation, 2026. URL https://arxiv.org/abs/2607.05147.

[82] Eduardo Alvarez, Omri Almog, Eric Chung, Simon Layton, Dusan Stosic, Ronny Krashinsky, and Kyle Aubrey. Introducing NVFP4 for efficient and accurate low-precision inference. NVIDIA Technical Blog, June 2025. URL https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/. Accessed 2026-10-29.

[83] Ali Tehrani. DSpark for GLM-5.2: Let's draft longer. Makora blog, July 2026. URL https://www.makora.com/blog/dspark-for-glm52. Accessed 2026-10-29.

[84] Ziteng Sun, Uri Mendlovic, Yaniv Leviathan, Asaf Aharoni, Jae Hun Ro, Ahmad Beirami, and Ananda Theertha Suresh. Block verification accelerates speculative decoding. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/3e710b42b1a9ed898f607ec0f4fcc971-Abstract-Conference.html.

[85] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pages 136737–136756, 2025.

[86] llm-d Project. llm-d: Kubernetes-native distributed inference serving stack, 2026. URL https://github.com/llm-d/llm-d. Apache-2.0. Accessed 2026-10-29.

[87] Robert Shaw, Clayton Coleman, and Carlos Costa. llm-d: Kubernetes-native distributed inferencing. Red Hat Developer, May 2025. URL https://developers.redhat.com/articles/2025/05/20/llm-d-kubernetes-native-distributed-inferencing. Accessed 2026-10-29.

[88] NVIDIA. Dynamo ModelExpress: Model weight distribution for LLM inference, 2026. URL https://github.com/ai-dynamo/modelexpress. Apache-2.0. Accessed 2026-10-29.

A Contributors

Abdelaziz Bounhar, Abhav Mehrotra, Abhijeet Somani, Aditi Kabra, Adrian Valente, Adrien Petralia, Adrien Sadé, Ahmed Djermani, Alan Jeffares, Albert Jiang, Alberto Bietti, Aleksandr Timashov, Alexandre Cahill, Alexandre Gavaudan, Alexandre Laval, Alexandre Sablayrolles, Alexandros Sanchez Vassopoulos, Amélie Héliou, Amos You, André Jonasson, Andreas Fürst, Andrew Bai, Andrew Ehrenberg, Andrew Zhao, Andy Lo, Angele Lenglemetz, Anmol Agarwal, Antonia Calvi, Antonin Faure, Arata Suzuki, Arjun Majumdar, Arshia Afzal, Arthur Fournier, Artjom Joosen, Ashwani Kumar, Augustin Lombard, Avinash Sooriyarachchi, Aylin Guliz Akkus, Aysenur Karaduman, Baptiste Bout, Baptiste Rozière, Barna Pásztor, Baudouin De Monicault, Benjamin Holzschuh, Benjamin Tibi, Benno Stein, Bernhard Stadlbauer, Bhargavi Bhaskar, Binbing Hou, Błażej Osiński, Boris van Breugel, Camille Le Scao, Chaoran Yu, Charlotte Cronjäger, Chen-Yo Sun, Chris Bamford, Christian Wallenwein, Christophe Renaudin, Clémence Lanfranchi, Colin Bonatti, Connor Chen, Corentin Barreau, Corentin Sautier, Cristiana-Diana Diaconu, Cyprien Courtot, Daniel Lopes, Daniel Marczak, Darius Dabert, Dhia Garbaya, Diego de las Casas, Dominik Nuss, Dylan Rubini, Dzmitry Soupel, Elizaveta Demyanenko, Elliot Chane-Sane, Emilien Fugier, Emmanuel Gottlob, Enguerrand Paquin, Erik Aas, Etienne Duchesne, Etienne Goffinet, Fabian Paischer, Fabian Schlager, Faruk Ahmed, Federico Baldassarre, Félix Rosseeuw, Filip Szatkowski, Filippo Vicentini, Florian Wiesner, Gabrielle Berrada, Gaëtan Ecrepont, Gaspard Beaudouin, Gaspard Blanchet, Gaspard Donada-Vidal, Gaurav Mahajan, Gauthier Guinet, Genevieve Hayes, Georgii Novikov, Georgios Christoglou, Giada Pistilli, Gonzague de Carpentier, Graeme Nail, Guillaume Breton, Guillaume Kunsch, Guillaume Lample, Guillaume Martin, Guillaume Raille, Guillaume Richard, Guillem Simeon, Gunjan Dhanuka, Gunshi Gupta, Hamza Senhaji Rhazi, Han Zhou, Harsh Poonia, Hédi Hadiji, Himanshu Jaju, Hope McGovern, Hubert de Parseval, Hugo Cisneros, Hugo Germain, Hugo Thimonier, Hugo Venturini, Hyeonju Lee, Hyunwoo Oh, Iliass Lasri, Ilyas Oulkadda, Jacques Sun, Jan Ludziejewski, Jason Rute, Jean de Becdelièvre, Jean Quentin, Jean-Malo Delignon, Jérémie Dentan, Jessica Chudnovsky, Jianchao Li, Jie Zhang, Jin Bin Ou, Jinjun Peng, Joachim Studnia, Joep Barmentlo, Johannes Brandstetter, John Harvill, Jonas Amar, Jonas Schweizer, Jonathan Roberts, Joséphine Delas, Josselin Somerville, Jules Depersin, Jules Royer, Julien Denize, Julien Tauran, Julien Verlaguet, Karan Bania, Kartik Khandelwal, Khyathi Raghavi Chandu, Kilian Tep, Konstantin Mishchenko, Kris Szczepaniak, Kush Jain, Larissa Laich, Laura Calem, Laurence Aitchison, Laurent Callot, Laurent Fainsin, Léo Cotteleer, Léonard Blier, Lingxiao Zhao, Loïc Chadoutaud, Loïck Chambon, Lorenzo Proietti, Louis Faucon, Louis Martin, Louis Serrano, Lu Li, Luc Boudier, Lucas Fernandes Martins, Lucas Mebille, Lucile Saulnier, Ludovic Ho Fuh, Luis Montero, Łukasz Borchmann, Maarten Buyl, Malik Hassan Raza, Manon Chossegros, Marcin Możejko, Margaret Jennings, Markus Hennerbichler, Martin Alexandre, Martin Drieux, Mathieu Poirée, Mathieu Schmitt, Mathilde Guillaumin, Mathis Felardos, Matthieu André, Matthieu Dinot, Matthieu Futeral, Maurits Bleeker, Mauro Comi, Max Mynter, Maxim Berman, Maxime Darrin, Maxime Kunsch, Maxime Langelier, Maxime Louis, Maximilian Augustin, Maximilian Müller, Mehdi Bouaziz, Melina Jingting Laimon, Mert Unsal, Mia Chiquier, Michael Beukman, Michael Pilcer, Michał Pietruszka, Michał Zając, Mickaël Seznec, Mikhail Biriuchinskii, Minh-Quang Pham, Minwoo Kang, Mircea Lica, Morgane Rivière, Namit Katariya, Natasha Ørregaard Klingenbrunn, Nathan Bout, Nathan De Lara, Nathan Grinsztajn, Nathan Simpson, Neeraj Aggarwal, Neil Ashton, Neil Mallinar, Nick Bührer, Nicolas Moreau, Nicolò Lucchesi, Nikhil Chandak, Ola Mysiak, Oliver Leicht, Olivier Bousquet, Olivier Koch, Parag Jain, Patricia Wang, Patrick Blies, Patrick von Platen, Patryk Saffer, Paul Jacob, Paul Teiletche, Paul Wambergue, Paula Kurylowicz, Pavan Kumar Reddy, Pavel Kuksa, Philippe Formont, Philippe Pinel, Philomène Chagniot, Pierre Glandon, Pierre Stock, Pierre-André Savalle, Piotr Miłoś, Poojan Shah, Prateek Gupta, Pravesh Agrawal, Pui Ki Chan, Quentin Desreumaux, Quentin Selle, Quentin Torroba, Quentin Wargnier, Quercus Hernandez, Ram Ramrakhya, Randall Isenhour, Ranjit Parva, Raphaël Balouka, Raphaël Rialland, Raul Perez Pelaez, Reinhard Sonnleitner, Remi Delacourt, Richard Kurle, Rishi Shah, Rob Romijnders, Rohan Shah, Rohin Arora, Romain Sauvestre, Roman Soletskyi, Ronan Riochet, Rosalie Millner, Rupert Menneer, Sagar Vaze, Sahil Dharod, Samuel Barry, Samuel Belkadi, Samuel Humeau, Samuele Papa, Sanchit Gandhi, Sandeep Cranganore, Sandeep Subramanian, Sarthak Mittal, Saskia Adaime, Scott Matthews, Sean Cha, Sebastian Kaltenbach, Seungjae Jung, Shashwat Verma, Sheel Shah, Sherif Waly, Shrimai Prabhumoye, Siddarth Venkatraman, Siddhant Waghjale, Siddharth Gandhi, Simon Lepage, Simon Sorg, Soham Ghosh, Sophie Marbach, Srijan Mishra, Stanislas Lange, Steve Hong, Sumukh Aithal, Szymon Antoniak, Tarun Kumar Vangani, Teddy Turquet, Teven Le Scao, Théo Cachet, Thibaut Lavril, Thomas Chabal, Thomas Coste, Thomas Defard, Thomas Foubert, Thomas Loux, Thomas Robert, Thomas Wang,

Tianyu Zhang, Tim Lawson, Timothée Lacroix, Timothy Wallis, Tobias Kronlachner, Tom Bewley, Tom Edwards, Tomas Hodan, Triantafyllos Afouras, Tuhin Das, Tyler Wang, Udit Saxena, Ulrick Blé, Umar Jamil, Umberto Tomasini, Valentin Macé, Van Phung, Vedant Nanda, Venkata Reddy, Vibha Masti, Victor Jouault, Victor Letzelter, Victor Paltz, Victor Poucheret, Vincent Maladière, Vincent Pfister, Virgile Richard, Vladislav Bataev, Wassim Bouaziz, Wen Ding Li, William Flanagan, William Havard, William Marshall, Witold Jędrzejewski, Xinghui Li, Xingran Guo, Xinyu Yang, Xuanyu Zhang, Yann Dreze, Yann Viegas, Yannic Neuhaus, Yassine El Ouahidi, Yassine Turki, Yassir Bendou, Yessin Moakher, Yi He, Yihan Wang, Yimu Pan, Yuying Zhu, Yves Martin des Taillades, Zaccharie Ramzi, Zhenlin Xu, Zsófia Csákány.