MAI

MAI-Thinking-1: Building a Hill-Climbing Machine

The Microsoft AI Team [1]

Abstract

Progress in AI is driven not by a single model, but by the ability to continually improve upon the current state of models. Achieving this requires treating model development as a system-level optimization problem, for which the solution is building a hill-climbing machine for rapid improvement. Our process includes a scaling-focused framework for pre-training modeling decisions, as well as a robust reinforcement learning recipe and infrastructure that sustains long, log-linear performance improvement. The first model developed using our process is MAI-Thinking-1, a 35B active / 1T total parameter MoE that stands among the strongest models of similar size on STEM reasoning and coding tasks (*e.g.,* 52.8% on SWE-Bench Pro, 97.0% on AIME 2025, and 87.7% on LiveCodeBench v6). MAI-Thinking-1 is trained from-scratch, exclusively on clean, enterprise-grade data, without distillation from third-party models. In this technical report, we offer a deep dive into the development of MAI-Thinking-1. By sharing our technical details and learnings we hope to cultivate a transparent and science-driven approach to further development in AI.

1 Introduction

We introduce MAI-Thinking-1, a powerful reasoning model developed from scratch by Microsoft AI (MAI) that is competitive with models of similar size on STEM reasoning and coding tasks. Our pre-training is focused on a simple scaling approach which emphasizes empirically-driven iterative improvements to our architecture and data, while our reinforcement learning (RL) framework is optimized for sustained log-linear climbs over many thousands of steps (Figure 1).

We develop our model following three main design principles: First, capabilities should be learned, not inherited. Although faster to acquire, intelligence imitated through distillation lacks the steerability and

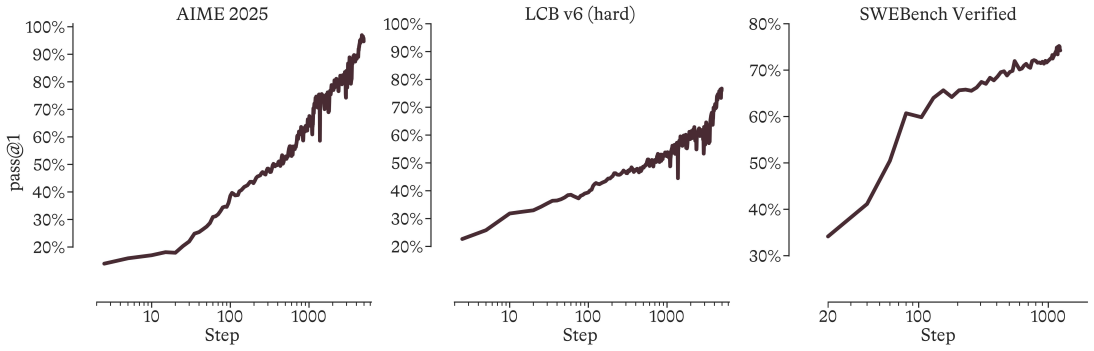

Figure 1. Performance during reinforcement learning of MAI-Thinking-1. **STEM Climb (left and center):** Evaluation performance during our STEM-focused climb with evals on AIME 25 and a hard subset of LiveCodeBench v6 (after Jan. 25). For both curves, we show pass@1 average over 3 adjacent checkpoints. **Agentic Climb (right):** Pass@1 performance on SWEBench Verified during our code-heavy agentic climb.

[1]Correspondence should be sent to mai-technical-report@microsoft.com.

robustness essential to long, enduring climbs. Second, simplicity is sustainable. We favor simple, scalable recipes; clean, trustworthy data; and transparent infrastructure that together support climbing from scratch. Third, scientific rigor avoids shortcuts. Every decision must be testable through data-driven ladders, ablations, and evaluations that expose reliable paths to the top.

MAI-Thinking-1 is pre-trained on 30T tokens extracted from a mixture of publicly available and licensed human-generated data covering covering web data, public GitHub code, books, academic papers, news, multilingual text, and domain-specific materials. Each of these sources are processed in-house from start to finish. We choose to not use any synthetic data generated by language models during pre-training and make an effort to avoid and remove AI-generated content within collected data sources. For pre-training, we do not use any open source training datasets and decontaminate common machine learning databases from our training data. In mid-training we further emphasize STEM, math, and coding abilities to build a strong foundation for reasoning RL climbs. After mid-training, MAI-Thinking-1 achieves a maximum context length of 256K. Pre- and mid-training give our base model broad predictive competence and knowledge, but they do not specify how the model should behave, solve long-horizon tasks, or allocate inference-time computation.

During the reinforcement learning (RL) climb, we teach the model to reason and respond. The model learns to leverage chains of thought (CoTs) against task-specific feedback, use external tools to interact with environments, as well as follow human preference and safety signals. Our RL climb starts from scratch, learning to reason with no prior exposure to reasoning traces. A robust RL recipe, self distillation, and infrastructure improvements enable us to sustain RL runs over thousands of steps. Using this recipe, we train three domain-specific specialist models: one for STEM reasoning, one for agentic coding and tool use, and one for helpfulness and safety.

We compare the performance of MAI-Thinking-1 to a variety of open- and closed-weights frontier models across public benchmarks and human side-by-side evaluations. These evaluations span a broad set of areas, highlighting the model's versatility across different domains: STEM, agentic coding, knowledge, instruction following, long context, safety, health, and tool calling. MAI-Thinking-1 achieves 52.8% on SWE-Bench Pro, 97.0% on AIME 2025, 94.5% on AIME 2026, 87.7% on LiveCodeBench v6, and is competitive with Sonnet 4.6 across a wide-range of benchmarks.

A key technical challenge when developing MAI-Thinking-1 was striking a balance between safety (which asks models to refuse user requests) and helpfulness (which asks models to comply). A safe model needs to provide helpful responses that remain compliant with our safety policies and standards. We develop internal safety benchmarks to ground our progress and incorporate helpfulness and safety training into our RL climbs. As part of responsible deployment, we continuously red-team throughout model development to surface and remediate vulnerabilities before release.

MAI-Thinking-1 is the first model developed using our hill-climbing machine: the integrated process of building data pipelines, training infrastructure, reinforcement learning environments and rewards, evaluation suites, and safety tests that turn model development into an empirical optimization loop on a specified domain. The hill-climbing machine allows us to advance AI while grounding progress around human needs from the ground up.

Contents

2 Pre-training

The base model for MAI-Thinking-1, called MAI-Base-1, is a 35B-active / 1T-total sparse MoE model pre-trained from scratch on 8K GB200 GPUs on a Microsoft-operated cluster within the Azure platform, using an in-house distributed training infrastructure (Sec. 2.8). Our training corpus was built entirely in-house from both publicly available and acquired data sources, free of any distillation data (Sec. 2.4). From this corpus we ran a main pre-training phase of 30 trillion tokens, followed by mid-training phases totaling 3.55 trillion tokens (Sec. 2.6).

Our goal in pre-training is to produce a base model that is capable across a wide range of domains, and whose capabilities are learned from human knowledge, rather than imitations of existing AI models. We establish an ablation protocol to carefully examine the scaling properties of our model decisions by measuring efficiency gains (Sec. 2.2) along a "ladder" of pre-trained models of increasing scale. To support these ablations, we developed a suite of pre-training metrics (Sec. 2.3), consisting of held-out negative log-likelihood (NLL) evaluations covering a broad range of subjects, tasks, and data distributions. We report the performance of MAI-Base-1 and contemporaneous base models to illustrate the use of these evaluations on models without post-training (Sec. 2.7).

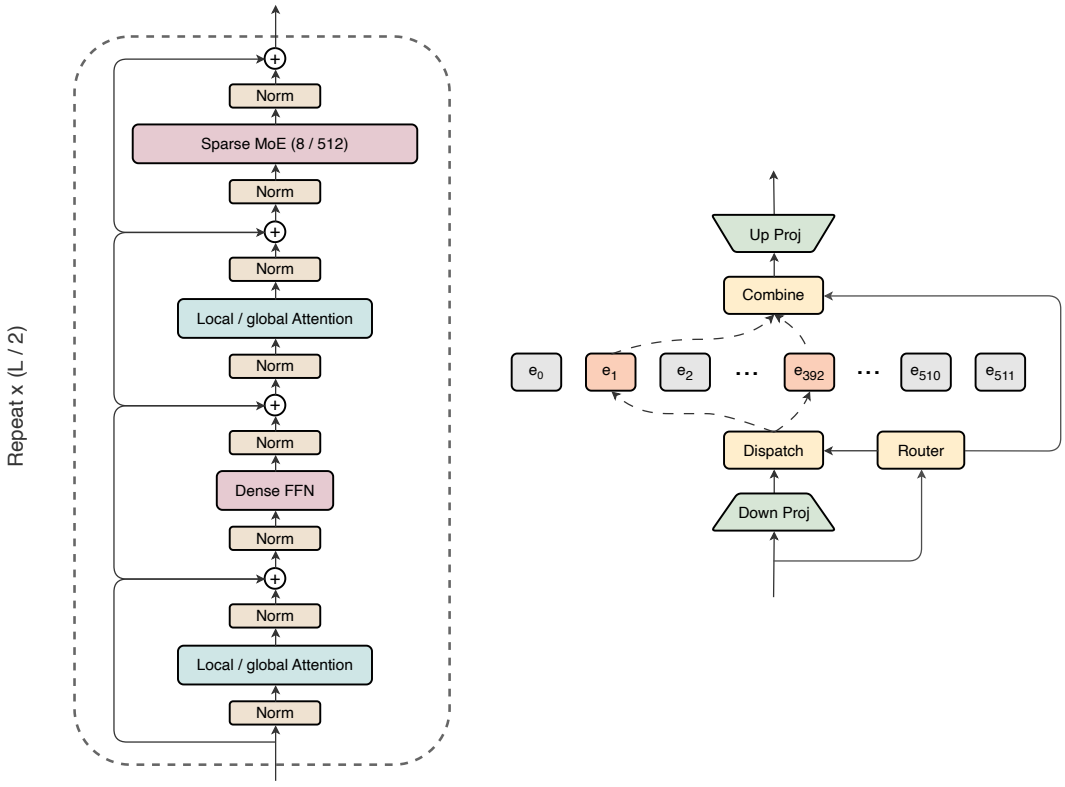

Figure 2. Overview of the MAI-Base-1 architecture. *Left*: the overall layout of the Transformer body, where we interleave high-sparsity MoE layers with small dense FFNs, and global attention with local attention. *Right*: The MoE layer, in which 8 of 512 experts are activated per token in a compressed latent space.

2.1 Model Architecture

The model architecture of MAI-Base-1 was designed to leverage the benefits of scale on our infrastructure and hardware. We use a decoder-only Transformer (Vaswani et al., 2017) with periodic local and global attention layers and alternating dense and MoE feed-forward blocks. A high-level overview of the architecture is shown in Fig. 2. In each layer, we employ RMSNorm (Zhang and Sennrich, 2019) at both the input and output immediately before the residual addition (Gemma Team et al., 2025). We do not use any biases in the model, and the input and output embedding weights are tied. While we have observed some improvement in model performance from tokenizers trained in-house, we chose the `o200k_base` tokenizer with a vocabulary size of 200,019 (OpenAI, 2022; Hurst et al., 2024) to simplify integration with existing in-house tools and workflows. A key consideration in our architecture design was the co-optimization with the underlying GPU infrastructure to ensure efficient scaling for both pre-training and reinforcement learning (RL).

Attention. We follow the periodic attention design of Gemma 3 (Gemma Team et al., 2025), pairing 5 local attention layers (Beltagy et al., 2020) with 1 global attention layer, which significantly reduces the computational cost of attention for training and the KV cache size during inference. The local attention layers use rotary position encoding (RoPE) (Su et al., 2023) with sliding window size of 512 and a base frequency of 10,000. The global attention layers do not use any position encoding (Kazemnejad et al., 2023), which performs comparably to RoPE while being more efficient. We use group-query attention with 8 KV heads (Ainslie et al., 2023) and a per-head dimension of 128. We apply RMSNorms to the queries and the keys (Dehghani et al., 2023; Wortsman et al., 2023). The use of these standard components allows us to leverage FlashAttention-4 (Zadouri et al., 2026) and Ulysses-style context parallelism (Jacobs et al., 2023) for efficient training on long contexts.

Feed-forward and MoE. For the feed-forward layer in each block, we alternate between MoE layers and dense feed-forward networks (FFN). We find that this paradigm of pairing a high-sparsity layer with a zero-sparsity (dense) layer scales comparably to a balanced sparsity allocation—i.e., medium-sparsity MoEs throughout—while being more efficient in wall-clock time under both iso-active and iso-total model parameter settings (Sec. 2.2.3). We also find that the every-layer-MoE layout relies much more heavily on shared experts, whereas adding shared experts to the interleaved layout has little or no benefit. Our first feedforward layer is dense (DeepSeek-AI et al., 2025a; Kimi Team et al., 2026; Dai et al., 2024), and we use SwiGLU (Shazeer, 2020) for both dense and MoE feed-forward layers.

We adopt the LatentMoE design (NVIDIA et al., 2025), where a shared down-projection is applied before the all-to-all dispatch. The latent representations are then projected back to their original dimension after the all-to-all combine. Importantly, routing decisions are based on the original representation, and each compressed representation is routed to 8 out of the 512 experts with softmax gating.

We use global-batch load balancing loss (Qiu et al., 2025; Yang et al., 2025) where the empirical expert frequencies are obtained by aggregating across both the data parallelism workers and micro-batches within the same global batch. We found the aggregation strategy to matter much more than the load balancing loss type. The GShard-style loss (Lepikhin et al., 2020) that we use performs similarly to the loss-free variant (Wang et al., 2024a) as long as global aggregation is ensured.

Notably, we observed that the experimental results regarding the optimal load-balancing methodology can depend on the expert capacity. Under finite capacity, even when the token-dropping strategy is carefully designed to avoid subtle causal leakages and the dropping rate is kept low with a generous capacity budget, the resulting conclusions can still differ from those obtained in settings without dropping. We therefore converged to a fully *dropless* MoE implementation (details in Sec. 2.8.1) which supports variable message size for all-to-all communications and ensures bounded memory usage under high imbalance.

Model	Active	Total	Layers	Hidden	FFN	Down Proj	Expert FFN	Top-k/Experts	KV/Q
L12	365M	3.9B	12	1024	2048	512	1536	8/512	8/16
L18	760M	13B	18	1536	3072	768	2304	8/512	8/16
L24	1.5B	30B	24	2048	4096	1024	3072	8/512	8/24
L30	2.6B	58B	30	2560	5120	1280	3840	8/512	8/32
L36	4.0B	100B	36	3072	6144	1536	4608	8/512	8/32
L42	6.1B	159B	42	3584	7168	1792	5376	8/512	8/40
L66	21.7B	615B	66	5632	11264	2816	8448	8/512	8/64
L78	35.6B	1015B	78	6656	13312	3328	9984	8/512	8/80
MAI-Base-1	34.7B	962B	78	6656	13312	3072	10240	8/512	8/80

Table 1. Configurations for all model sizes in the MAI-Base-1 architecture family. The (Expert-)FFN sizes refer to the output size of the first linear layer, which is 2x of the input size for the second linear layer due to SwiGLU. The configuration for MAI-Base-1 is slightly different from L78 to facilitate training and inference efficiency.

2.2 Model Ablation Methodology

We now describe the experimental setup used to evaluate different architectural and data-related design choices for pre-training. A key consideration is the role of scale: the apparent benefits of model innovations or new datasets often diminish as the compute budget increases. Accordingly, before adopting changes to our baselines, we must gather sufficient supporting evidence across different scales. In the following subsections, we introduce the main components that enable us to carry out this process in a principled and reproducible manner: scaling ladders and efficiency gains.

2.2.1 Scaling Ladder

All ablations are done specifically with scaling properties as the central focus. To achieve this, we utilize a scaling ladder: we train different model sizes at a constant number of tokens per (active) parameter (TPP) for any ablation, and compare the scaling curves to our baseline. The TPP to use depends on the nature of the experiment. For instance, most architecture ablations are conducted near the Chinchilla optimal region (Hoffmann et al., 2022) at 100-200 TPP, while our main run is typically over-trained at much higher 500-1,000 TPPs to produce a relatively more compact model suitable for heavy inference needs. Ablations are performed using the latest ladder and the accepted changes become part of the next baseline ladder configuration, and the process repeats.

For our architecture family described in Sec. 2.1, a model configuration is fully specified by a single parameter: the number of layers L. For the 5:1 local to global attention ratio that we use, L must be a multiple of 6 so that the last attention layer applies global attention. The hidden size D is then determined by $D = L \times \frac{256}{3}$ to maintain the same aspect ratio across the ladder. This choice of aspect ratio is similar to that of some recent models (OpenAI, 2025; Liu et al., 2024; Gemma Team, 2024) and leads to more hardware-friendly values for D in our setup since L is always a multiple of 6. The number of query heads is set to L, rounded up to the nearest multiple of 16 for ease of serving with tensor parallelism. Each dense feed-forward layer expands the hidden dimension by a factor of 2. For latentMoE layers, we use a compression factor of $2\times$ for LatentMoE and expand the compressed representations by a factor of $3\times$ within each expert.

The details of the full scaling ladder are shown in Tab. 1. The final architecture of MAI-Base-1 has 78 layers. While it is based on L78 in the ladder, we revised its expert input size slightly for DeepEP (Zhao et al., 2025a), which requires the hidden dimension for all-to-all collectives to be divisible by 512.

	MoE every layer (8/384)		MoE every layer (7+1 shared/384)	
Eval Category	EG_{FLOPs}	EG_{Time}	EG_{FLOPs}	EG_{Time}
General	0.88	0.69	0.99	0.78
STEM	0.93	0.73	1.04	0.82
Math	0.94	0.74	1.06	0.84
Code	0.94	0.74	1.03	0.82
Multilingual	0.94	0.75	1.05	0.82
Weighted Average	0.94	0.73	1.03	0.82

Table 2. Efficiency gain for two MoE-every-layer variants over the interleaved layout of MAI-Base-1 (8/512 MoE + dense FFN). Both variants are near the baseline when measured by EG_{FLOPs}, but are clearly behind on EG_{Time}.

2.2.2 Efficiency Gain

In our architecture and scaling discussions, *efficiency gain* (EG) measures how much higher a training cost the baseline model would need to match an evaluation metric reached by a given candidate model (Kaplan et al., 2020; Hoffmann et al., 2022). In most cases, the evaluation metric is an aggregated evaluation loss defined using our pre-training evaluation suite (Sec. 2.3). We first fit a scaling law to the baseline ladder,

$$L = f(C) = AC^{-\alpha} + E \tag{1}$$

where C denotes the training cost such as FLOPs or time with the given infrastructure. A is the scaling coefficient that controls the overall magnitude of the reducible loss term, α is the scaling exponent that controls how fast the loss decreases with more compute, and E is the irreducible loss. For a candidate run achieving L' with cost C', we compute the baseline cost budget required to reach the same loss as $f^{-1}(L')$, and define EG as

$$\text{EG} = \frac{f^{-1}(L')}{C'} \tag{2}$$

As an example, $\text{EG} = 1.3$ means that the baseline model would require a 30% higher cost to reach the candidate model's loss (Liu et al., 2025a; Dey et al., 2023). We further perform ablations at different cost budgets to obtain a series of EG vs. cost, which helps us in finding improvements that persist as the scale increases.

We most commonly use FLOPs to define the cost C, which intentionally does not account for the wall-clock efficiency that can be achieved in training. Implementations of established architectures have often accumulated numerous performance optimizations over time, which can make them appear stronger than newer variants. By decoupling the metric from FLOPs utilization (MFU) (Chowdhery et al., 2023), we can look at how architectures may stack up against each other if equal effort is given to optimizing their implementations. However, in cases where we do care about hardware efficiency given our existing stack, we use the time definition for cost C above. In those cases, we refer to efficiency gain as EG_{Time} explicitly.

2.2.3 Model Ablation Examples

We use ladder and EG comparisons extensively in our day-to-day ablations. Below we present two set of experiments that demonstrate how EGs may vary across different evaluation categories, and how EGs can be extended to reflect co-design factors with our infrastructure.

Example 1: Sparsity allocation. Tab. 2 compares our interleaved layout with high-sparsity MoE and dense FFN against the more common design of using medium-sparsity MoE layers throughout. Experiments are conducted in a sparsity-controlled setup, in which all model families share similar total-to-active

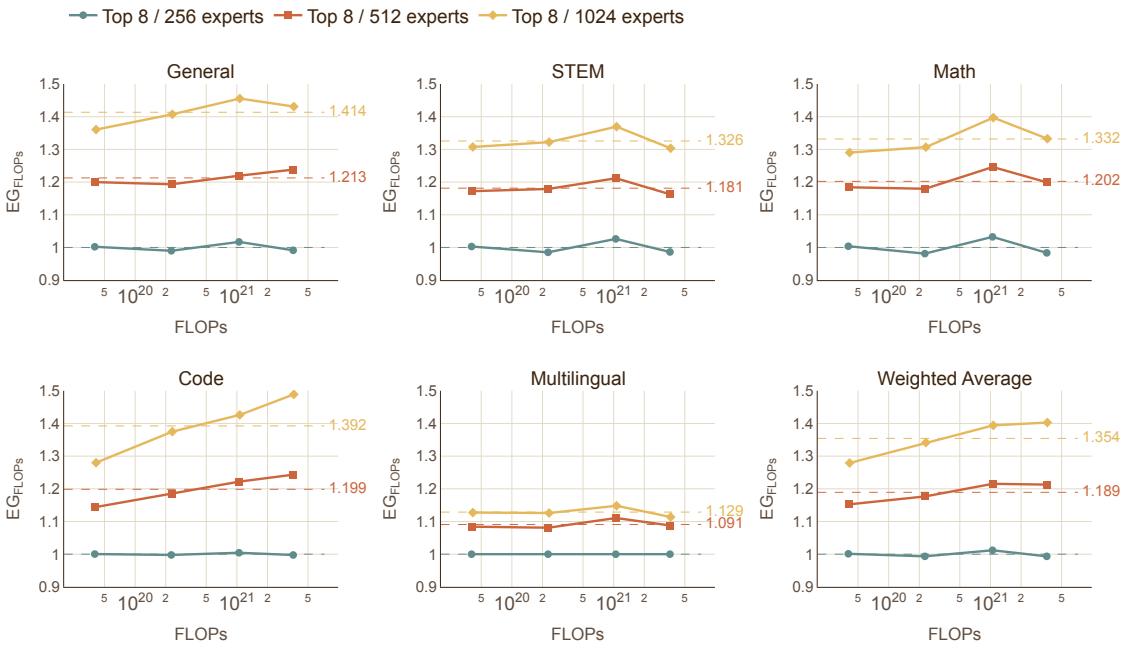

Figure 3. Pre-training efficiency gains (Sec. 2.2.2) across eval task categories with increasing sparsity of model families. Results are obtained by training a model ladder (Sec. 2.2.1) for each sparsity level. Since every data point is compared against the scaling law (Eq. 1), EG for the baseline raw data points may not be exactly one.

parameter ratios; this is achieved by fixing top-k (8, or 7+1 with a shared expert (Dai et al., 2024; Qwen et al., 2025)) while adjusting the expert expansion rate and number of experts for the MoE-every-layer families. We report two versions of EG defined along two cost dimensions: FLOPs and training time, measured on ladder models L12–30 described in Sec. 2.2.1. While MoE-every-layer combined with shared experts achieves an EG of 1.03 in FLOPs, the interleaved paradigm offers the better tradeoff once training MFU is taken into account, as reflected by $\text{EG}_{\text{Time}} < 1$.

Example 2: Sparsity scaling. A scalable architecture design should yield healthy improvements with increased sparsity. Fig. 3 demonstrates that our architecture family carries this property, showing consistent efficiency gains as we increase the number of experts from 256 to 1024 with top-$k = 8$ (MoE sparsity level from 32× to 128×). For MAI-Base-1 we select the top-8/512 configuration to balance model quality with training and inference efficiency, and leave further exploration of sparsity scaling for future work.

2.3 Evaluation Methodology

We use a suite of internal benchmarks focused on NLL tasks for our experimentation and progress tracking. These come from different sources: vendors who have created data fully held out from training, internal sources not present on the web, and web sources that we are careful to remove from our training data. Examples of our benchmarks are listed in Tab. 3, and we discuss our motivation for using NLL evaluation benchmarks in Sec. 2.3.2. We report our evaluation results after pre-training in Sec. 2.7.

In total, we have nearly 40 such benchmarks grouped into 5 different categories: Code, STEM, Math[2], General Knowledge, and Multilingual. When we need to aggregate these into a single number for direct EG

[2]We choose to explicitly up-weight math and treat it differently from the more broad STEM category.

comparisons (Sec. 2.2.2) or for ranking models, we use the following formula:

$$\text{Target} = 0.5 \times \text{Coding} + 0.175 \times \text{STEM} + 0.175 \times \text{Math} + 0.1 \times \text{General knowledge} + 0.05 \times \text{Multilingual} \quad (3)$$

These explicit weights allow us to codify the decisions about the relative priority and importance of each set of benchmarks. Our focus on downstream performance on code and reasoning tasks is reflected in the larger weights we assign to those categories in pre-training.

To compute the individual category scores in Eq. 3, we first normalize the raw NLL results using a fixed in-house reference model as a baseline, then take the uniform average across all the benchmarks in that category. All evaluated models use identical tokenizers, so NLL values are directly comparable. Note that we choose to elevate mathematics to its own category, while STEM focuses on science, technology, and engineering.

Category	Evaluation Data	Source
Coding	Microsoft code and pull requests	Internal private projects
	Human-AI coding sessions	Internal testing of previous models
STEM	Worked solutions to graduate-level STEM problems	Commissioned from vendor
Math	Worked solutions to advanced math problems	Commissioned from vendor
General knowledge	Online Community Discussions	Public web forums
	Human-AI interactions	Internal testing of previous models
	Pyramidal trivia	Publicly available databases deduplicated against training data
	Hard trivia questions	Commissioned from vendor
Multilingual	Multilingual Human-AI interactions	Internal testing of previous models

Table 3. Example types of data within our NLL evaluation suite. All examples are carefully deduplicated from any training data.

2.3.1 Public Evaluation Decontamination

Publicly available evaluation benchmarks have been helpful for guiding the development of many of our models. As the importance of AI models has grown in recent years, there has been an increase in the leakage of evaluation datasets into sources of training data. This is particularly common on GitHub, where many researchers and developers of AI share their work. Many common evaluations are published on GitHub in their original form (Austin et al., 2021), and nearly every benchmark is replicated there many times, typically with solutions, generations from models, etc. Many public evaluations are derived from some of the online sources we use for training, but with additional curation processes (for example (Wei et al., 2024a; Hendrycks et al., 2020)). As our training sets have grown in scale and coverage, the influence of that overlap becomes significant. Similar to prior works (Godey et al., 2025), we've observed in experiments that this contamination can lead to counterintuitive results, such as data that is nominally meant to improve coding performance also leading to an improvement in long-tail general knowledge evaluations.

We have used several methods for detecting and removing leaked evaluation examples in our training data. All data from `huggingface.co` and mirror domains are removed, and a universal 20-gram fuzzy deduplication is applied with a similarity threshold of 80% to all our training sources. We find that these methods accurately remove a significant number of leaked examples, but they are imperfect.

In recognition of this, we have developed private benchmarks for our own use which we are confident are not found elsewhere on the web.

2.3.2 Comparison of Accuracy and NLL Evaluations

We use internal proprietary evaluations for day-to-day development. These evaluations are curated by members of the MAI team, with some commissioned from data vendors who employ domain experts, and others collected from domain experts across the global Microsoft organization.

For pre-training, we choose to focus on evaluations with an NLL scoring method rather than multiple choice, generative, or other formulations. Evaluations that measure how many questions a model can correctly answer are convenient for comparing across models and to human benchmarks, but they have a number of operational challenges outlined below.

Efficiency and cost of evaluations. Many frontier benchmarks require chain-of-thought reasoning (Phan et al., 2025; Rein et al., 2023), tool use, or other generations. This autoregressive generation is expensive and time-consuming. Training frameworks and configurations are often not optimized for the fast autoregressive inference suited to these evaluations. The use of more efficient inference frameworks can accelerate these evaluations, but introduces additional complexity and support for architectural variants. Evaluations with open-ended or natural language answers often require the use of a judge model, adding further cost. In contrast, NLL evaluations are the same fundamental next-token-prediction objective as pre-training and can be run very efficiently. By lowering the cost in time and compute of each evaluation, we can run more exhaustive evaluations and ensure consistency across every experiment.

Sensitivity to confounding factors. Evaluations can be highly sensitive to seemingly minor variations in the pre-trained model's behavior. For example, MMLU (Hendrycks et al., 2020), MMLU-Pro (Wang et al., 2024b), and related benchmarks (Yue et al., 2024, 2025; Singh et al., 2025b) have been valuable for assessing pre-trained knowledge, but their multiple-choice format implicitly assumes that a model has acquired the ability to interpret and answer multiple-choice questions—a capability that typically emerges only at surprisingly large scale when evaluated in pre-training (Du et al., 2024). Although these tasks are sometimes evaluated using an NLL-based scoring scheme over logits for `A`, `B`, `C`, and `D`, many questions require more than factual recall, and free-form generation is often a more faithful evaluation protocol.

Similarly, MATH (Hendrycks et al., 2021) requires strict adherence to formatting conventions such as wrapping answers in `\boxed{}`, and MBPP (Austin et al., 2021) exposes models to inconsistencies between Unix-style newlines (`\n`) in the shared prompt and Windows-style newlines (`\r\n`) in the problem statements. Minor formatting discrepancies in pre-training data, especially when they vary across sources, can significantly influence how models respond to such inconsistencies. We have observed interactions of this kind across many of our experiments, complicating the interpretation of results and increasing the likelihood of misinterpretation by both new and experienced researchers.

NLL evaluations are more robust to such issues. At every prediction step, the model is conditioned on the ground truth prefix. This teacher-forcing setup limits the degree to which a minor error can compound. Our conclusions follow prior work showing that loss- or perplexity-based metrics can provide higher-signal model-development feedback over tasks with formatting choices (Heineman et al., 2025; Du et al., 2024).

Complexity of construction. Building a high-quality Q&A evaluation requires substantial and carefully coordinated effort. Multiple iterations of data collection, difficulty calibration, deduplication, and quality control are typically necessary before an evaluation reaches an acceptable standard, and each stage must be carried out in close collaboration with domain experts. Designing novel questions is itself a nontrivial challenge, particularly at the frontier of domain knowledge.

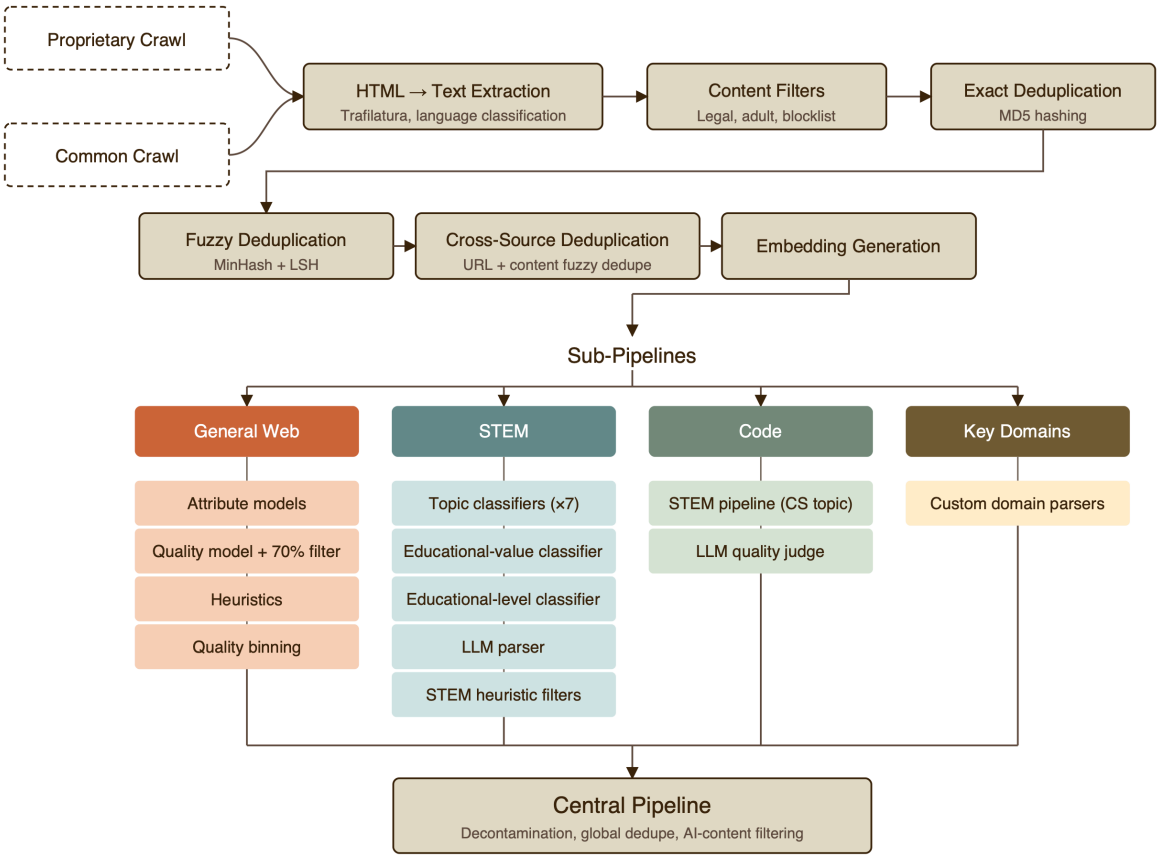

Figure 4. Pipeline for processing HTML pre-training data. We start with both a proprietary crawl and Common Crawl which goes through HTML extraction before filtering based on various heuristics. We then perform exact and fuzzy deduplication, cross source deduplication, and embedding generation. This data is then further processed by different pipelines; see Appendix A for the specifics of these pipelines.

By contrast, the barrier to entry for a NLL-based evaluation is considerably lower. Any topic-relevant content can serve as an initial corpus, and subsequent improvements–such as imposing additional structure, pruning to more representative samples, or incorporating more advanced material–translate directly into incremental gains in evaluation quality.

2.4 Pre-training Data

MAI-Base-1 is trained on a mixture of publicly available and licensed human-generated data. We compile a high-quality, diverse set of pre-training data covering web, public GitHub code, books, academic papers, news, multilingual text, and domain-specific materials. We choose not to use language-model-generated synthetic data for pre-training, and we make an effort to avoid and remove AI-generated content within collected data sources (Appendix A).

All of our pre-training data is processed in-house from one of our base sources: HTML and PDFs from the web, books and journals, and public GitHub code. We do not use any open source training datasets, and we exclude common machine learning repositories from our web data, such as `huggingface.co` and similar sites. We repeatedly found careful iteration on data quality, not just quantity, to be paramount not

Source family	Knowledge cut off date
Web HTML pages	September 2025
Web PDFs	December 2025
Public GitHub Code	June 2025
Books and journals	March 2026

Table 4. Knowledge cut off dates for our training data sources.

only for pre-training results, but also for downstream performance on reasoning benchmarks. The subsections below describe the common techniques we apply when processing each of our major data sources. For specifics on the data pipelines for each individual source, with full technical details, see Appendix A. An overview of our data processing pipeline for web data is shown in Fig. 4.

2.4.1 Data Sources, Licensing, and Governance

The publicly available portion of our data is collected in accordance with applicable terms of use and industry standards for web controls. For web data, we use a proprietary crawler that respects the Robots Exclusion Protocol (`robots.txt`) and related meta-tag and HTML controls, enabling site owners to manage how content on their sites is accessed and used. We also exclude sources that violate Microsoft Responsible AI policies or appear on the Office of the United States Trade Representative (USTR) Notorious Markets list.

Our training corpus also includes datasets acquired from third-party providers through commercial agreements. Third-party datasets are subject to diligence processes to assess data fidelity, ownership, and applicable usage rights, and are governed by confidentiality terms and conditions. Knowledge cut off dates for the various sources are shown in Tab. 4.

We did not use private customer data or data from Microsoft products and services for pre-training, except where users have explicitly opted in or where use is covered under applicable agreements, with respect for user opt-outs. Prior to training, our entire corpus is processed using PII-risk and safety filtering; details on these steps can be found in Appendix A. For privacy, legal, safety, and competitive reasons, we do not disclose the full list of datasets or data providers.

2.4.2 HTML Extraction

A large portion of our data comes from the web, where trainable content needs to be extracted from the raw HTML. There are many ways to do this, and resulting quality differences can often be quite subtle. Some of the content we are most interested in – mathematics, code, etc. – is not represented in a consistent way across the web, and many off-the-shelf methods end up dropping large amounts of this key data, while retaining the surrounding text. Tables and other structured content are also often quite important, but preserving this data requires some care.

We do not use a single extraction method for all sources; instead, we apply different approaches depending on the structure and reliability of each domain:

- **Source-specific structured parsers** are used for standardized formats such as HTML or XML, where schema-aware parsing enables accurate conversion into textual representations.
- **Hand-crafted extractors**, implemented with tools such as BeautifulSoup (Richardson, 2026), are applied to domains that exhibit consistent structure but are not well handled by our general heuristics, or where the value and volume of content justify additional engineering effort.
- **LLM- and agent-based processing** is employed for domains requiring targeted extraction, normalization, or semantic filtering beyond what deterministic rules can provide (Mahabadi et al., 2025). In all cases

where we use this approach, the LLM only ever chooses to keep or remove text from the original source text and is unable to add additional synthetic content to the output.

- **Training on raw content** is used when further processing risks discarding important information. For example, Wikipedia's bespoke markup language, *wikitext* (MediaWiki, 2026), lacks mature tooling compared to HTML, and elements such as infoboxes are not always faithfully preserved by existing parsers. Although wikitext is roughly $3\times$ more verbose, we find that training on the complete raw markup yields better results than relying on stripped-down conversions. For high-value, low-volume sources like Wikipedia, this tradeoff is well justified.

2.4.3 Deduplication

Models at the scale and sparsity of ours possess an exceptional capacity to acquire and recall detailed knowledge. While this capability is central to their effectiveness as generalist systems, it also increases the risk of memorization and overfitting to training data (Lee et al., 2022; Abbas et al., 2023). When memorization occurs, the model fails to generalize the underlying concepts, leading to degraded performance in other domains. A primary driver of such overfitting is repetition within the training corpus.

Large-scale web data contains substantial redundancy: many documents are near-duplicates, differ only by minor formatting changes, or are replicated across multiple sites. Without careful filtering, a model may be exposed to numerous instances of the same content. As model capacity increases, the impact of this redundancy becomes more pronounced. In particular, predictive scaling behavior is sensitive to the effective number of unique tokens: larger models show degraded scaling when trained on corpora with reduced novelty, because they exhaust the supply of new information earlier. Once the dataset's unique content is saturated, additional training tokens provide little benefit, limiting further improvement. Across many experiments, we consistently observe that rigorous deduplication and removal of redundant data improve both pre-training metrics and downstream reasoning benchmarks.

Given the importance of limiting redundancy, our pipeline incorporates multiple deduplication stages, each targeting a different class of redundancy. All of our data, regardless of source, receives one or more of these deduplication treatments. For the specifics on how each of the sources (web HTML and PDFs, books and journals, public GitHub code) is handled, see the full technical details in Appendix A. Here we describe the variety of techniques that we employ:

- **Boilerplate removal**. Web HTML pages often contain repeated boilerplate text. Using line-occurrence statistics within and across documents, we remove elements such as headers, navigation bars, footers, sidebars, and redundant paragraphs introduced by parsing artifacts.

- **Exact duplicates**. Identical content can appear multiple times due to republication across sites or repeated snapshots caused by system-level faults. We remove all exact byte-level and hash-level duplicates.

- **Fuzzy duplicates**. We apply MinHash Locality-Sensitive Hashing(LSH) based fuzzy deduplication (Broder, 1997) with a similarity threshold of 0.8, following the configuration of Smith et al. (2022).

- **Templated web pages**. Many sites generate pages from a shared template with only minor lexical variation (e.g.,"calculator" web pages with raw arithmetic tables). We skeletonize each page to its most frequent tokens and perform fuzzy deduplication over these templates to eliminate large families of near-identical pages.

- **Semantic duplication**. Documents produced independently can still be highly similar due to shared context or canonical problem formulations. This is especially common in code datasets, where well-known programming exercises recur across homework sets, exams, interviews, and competitions. Although authored independently, solutions often converge (e.g., standard traversals of a binary search tree). Such

clusters are abundant in organic datasets, and removing them improves model robustness and task diversity (Sorscher et al., 2023; Abbas et al., 2023; Tirumala et al., 2023). We use an Qwen3-Embedding-0.6B (Yang et al., 2025) model to identify semantically similar documents and retain only a limited number of representatives per cluster.

We apply these deduplication techniques both at the early stages of each processing pipeline and again across datasets as a final step before training. Historically, we managed overlap among training sources using explicit filters–for example, excluding the Wikipedia domain from all web-crawl data. However, as the diversity and coverage of our sources have increased, and as our processing pipelines have become more complex, opportunities for unintended overlap across datasets have grown.

To address this, we perform cross-dataset deduplication using a global *drop-order* over datasets. When fuzzy or semantic duplicates are detected, the instance is retained only in the highest-ranked dataset in which it appears and removed from all datasets with lower drop-order priority. The choice of drop-order is therefore critical when interpreting experimental results and assessing the contribution or value of individual datasets. Modifications to one dataset can introduce new overlaps, effectively shifting data from one source to another even when no new content has been added.

2.4.4 Filtering and Categorization of Data

A central goal of downstream processing is to convert large, heterogeneous raw corpora into data that is both useful for pre-training and controllable for data mixture optimization (Sec. 2.5). Across sources, we first remove data that is unlikely to contribute positively to training, including spam, restricted or policy-sensitive content, and other source-specific noise. We then categorize the remaining data into interpretable buckets, such as quality tiers, language groups, topical categories, educational value, educational level, source type, and domain-specific subcorpora.

The exact filtering and categorization strategy varies by source, since different corpora expose different signals and failure modes. We therefore apply and combine several classes of techniques:

- **Leveraging metadata signals**, such as domain names, filenames, repository metadata, PDF creator and producer fields, and document-level metadata.
- **Source-specific heuristics**, such as web text-quality filters, OCR-artifact filters, math-aware filters for STEM content, and path- or content-based filters for generated code.
- **Learned classifiers**, including fastText-style classifiers (Joulin et al., 2016) and embedding-based models for language, topic, educational value, educational level, quality, and other semantic attributes.
- **Prompted LLMs**, used selectively for higher-value or more ambiguous decisions such as section-level extraction, quality judging, and nuanced topic labeling.
- **Manual exploration and labeling**, used to identify failure modes, validate filtering precision, audit high-impact source categories, and construct training data for classifiers and LLM judges.

Together, filtering and categorization transform processed source corpora into a structured collection of data buckets. Filtering improves data quality by removing low-value or problematic content, while categorization organizes the remaining corpus into interpretable dimensions that are useful for data mixture optimization. This structure enables controlled ablations across source families, quality tiers, topics, educational levels, languages, and source-specific formats. Further implementation details for each major source are provided in Appendix A.

2.4.5 Data Ablations

Each processed pre-training dataset goes through rigorous ablation. We use two approaches for data ablation: *single-source* ablation and *scaling-ladder* ablation. Single-source ablation is an isolation methodology used to

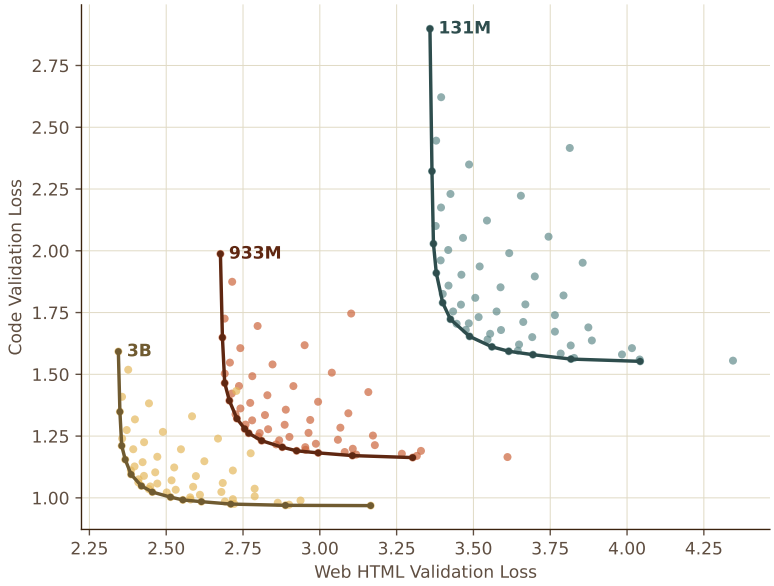

Figure 5. An illustrative example of the frontier of data mixing. 183 models are trained from scratch with equal TPP at 3 different model scales on 61 diverse mixtures of Web HTML, Code, and other data. The solid line indicates the frontier of validation loss performance for Web HTML and Code. Models on this Pareto frontier are the mixtures where these two components dominated, and those off the frontier had a larger fraction of the 'other' subset.

quantify the marginal utility of a specific dataset, and attribute its contribution to NLL-based evaluations. We upweight the target data to 50% in the mixture and train from scratch. For scaling-ladder ablations, we use the scaling ladder described in Sec. 2.2.1 and ablate the data within the full mixture to forecast how a data-curation decision will perform at the final target scale and token horizon, in the context of all other datasets. Data-source downsampling is performed in the ladder ablation to mimic the multi-epoch nature of the target model.

2.5 Selecting a Data Mixture for Training

The data sources sampled during training and their relative proportions significantly affect the resulting model. Given a fixed compute budget and a corpus of hundreds of heterogeneous sources, we must determine how to weigh them.

To achieve this, we first define an objective function. As illustrated in Fig. 5, a given set of target capabilities (e.g., coding or general knowledge) can yield multiple Pareto-efficient mixtures, where it is impossible to improve one capability without degrading another. We must therefore assign relative importance to these different capabilities. Specifically, we choose to minimize Equation 3, an aggregate over a suite of held-out NLL evaluations.

Optimizing the data mixture requires balancing trade-offs between domains of interest, data quality, the volume of unique tokens available, and the maximum number of times a dataset can be repeated before encountering diminishing returns or overfitting.

2.5.1 Challenges to Optimization

The choice of datamix has a significant influence on the resulting model, and so we want to take a principled and structured approach to this problem. An effective technique must deal with several challenges:

- **The definition of utility.** How do we concretely define the goal against which to optimize? Our overarching goal is to develop models that are effective in real-world use, but running a full post-training process for each candidate data mixture would be prohibitively expensive.
- **Vast search space.** As the number of datasets grows to the hundreds, the space of datamixes is so large that an exhaustive search is not computationally feasible.
- **Scale-dependent effects.** One data source may be particularly useful for models at smaller scale, while another is more useful at large scale (Ye et al., 2025).
- **Cross-dataset interactions.** Datasets are both complementary and substitutive for each other in different ways. The utility of a dataset cannot be measured independently of the rest of the mixture.
- **Multi-epoch effects.** Datamixes cannot be evaluated in isolation: training horizons matter (Chen et al., 2026a). While tempting, placing high weight on high quality but small datasets can lead to diminishing returns when the training horizon is large, as the dataset ends up being consumed many times. In the worst case, this can lead to overfitting.
- **Compute cost.** Training many models with various data mixes quickly becomes expensive, especially when trying to explore an extremely high dimensional search space.

2.5.2 Forecasting-Based Approaches

One class of approaches to data selection is to train many models at significantly smaller scale and then build a predictive model over these model's evaluation results. From these fit-functions on the smaller models, one can try to estimate the globally optimal data mixture or, at least, guide the search. This family of approaches includes methods like RegMix (Liu et al., 2025b). One key feature of these techniques is that by relying on small models, they enable rapid iteration.

We experimented with variants of these methods, sampling data mixtures based on a baseline mixture and training several thousand small models, each with between 760M and 4B active parameters (L12-L36 models as per Tab. 1) – examples of the Web / Code data frontier are shown in Fig. 5. For each model, we compute our full NLL suite looking at correlations between different data subsets and different evaluations – both aggregated and individual – an example of which is shown in Figs. 5 and 7. Ultimately, we assign a single score for each mix, as given by Eq. 3 in Sec. 2.3, which we attempt to globally optimize (minimize).

A case of scale dependence in practice. During our data-mixing experiments, we uncovered an empirical finding that challenged one of our core scaling assumptions: the idea that the relative ordering of two datamixes – which one yields better evaluation performance – is preserved as compute increases. This assumption, often referred to informally as the rank invariance hypothesis (Liu et al., 2025b), is appealing because it suggests that inexpensive small-scale experiments can reliably guide decisions for large-scale runs. However, we found this to not always be the case.

In an earlier model generation, we sought to measure the sensitivity of coding and STEM reinforcement learning to pretraining data mixtures. We produced two candidate mixtures with different mixture optimization methods and weightings of our evaluations: a `code-heavy-mix` (approximately 50% code, predicted to be especially strong on our Coding category) and `stem-heavy-mix` (STEM data significantly upweighted, to target the STEM and Math categories). Intermediate scale verifications were run, followed

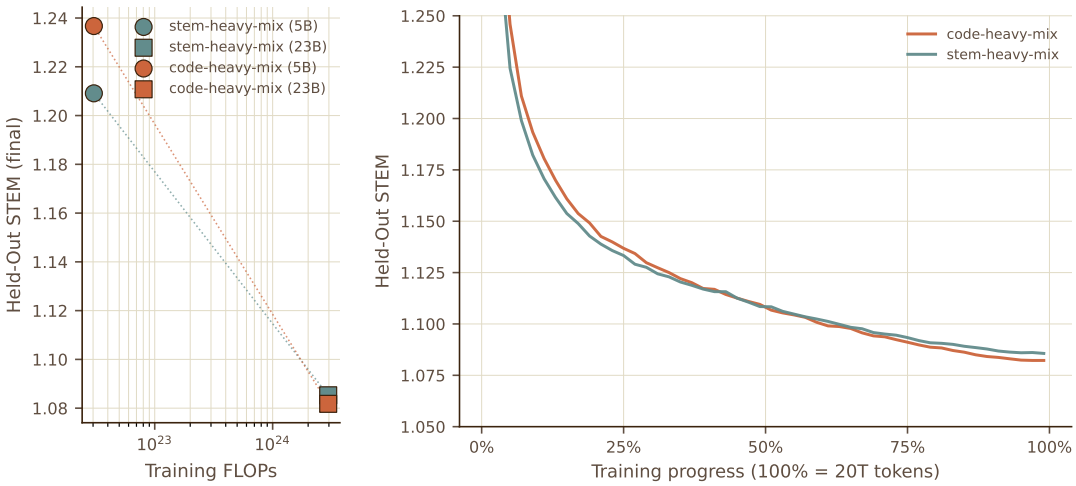

Figure 6. Rank non-invariance in data mixture scaling. **(left):** `stem-heavy-mix` performs better on a held-out STEM NLL evaluation at small scale, but `code-heavy-mix` performs better at larger (23B) scale. **(right):** NLL of the STEM evaluation throughout training for the two 23B active parameter models for different data mixtures.

by 23B active parameter models on each mixture for approximately 20T tokens (roughly equivalent to L66 as per Tab. 1). No changes were made other than the weighting of the training sources.

Results are shown in Fig. 6. As expected from the low-scale results, `stem-heavy-mix` outperformed `code-heavy-mix` on NLL STEM evaluations early in training. However, the STEM evaluation curves crossed midway through training, with `code-heavy-mix` eventually surpassing `stem-heavy-mix`. This contradicts the hypothesis that the relative ranking of datamixes is preserved as training compute is scaled up.

From inspection of training and validation loss trends of individual sources, we identified two data sources that had high quality STEM content, but more fuzzy duplication than our other datasets and less diversity of content. These two sources had 11.8% weight in `stem-heavy-mix`, but just 0.3% in `code-heavy-mix`. Our hypothesis is that these sources were very helpful for the smaller model, but their lack of diversity was inherently less useful at larger scale.

Following this result, we have put more emphasis on the scaling performance of candidate data mixtures in addition to their performance at a fixed experimental scale. Furthermore, in future data mixing studies, we paid extremely careful attention to the scaling properties of data mixes via our ladder approach.

2.5.3 Final Mix Selection

Because of the high number of datasets and our desire to validate them across multiple scales, we adopted a hierarchical approach to improve on our previous best data mix. We divided the data into about 10 categories such as coding, STEM, PDFs, and general web. Then we performed alternating local and global optimizations:

- **Local Search.** We keep all high-level categories fixed, and locally vary the weights within a single subset – for example, varying the weights between code files, PRs, and commits while keeping general web, STEM, etc. weights frozen.
- **Global Search.** We keep the relative make-up of each high-level bin fixed, and vary the relative weight

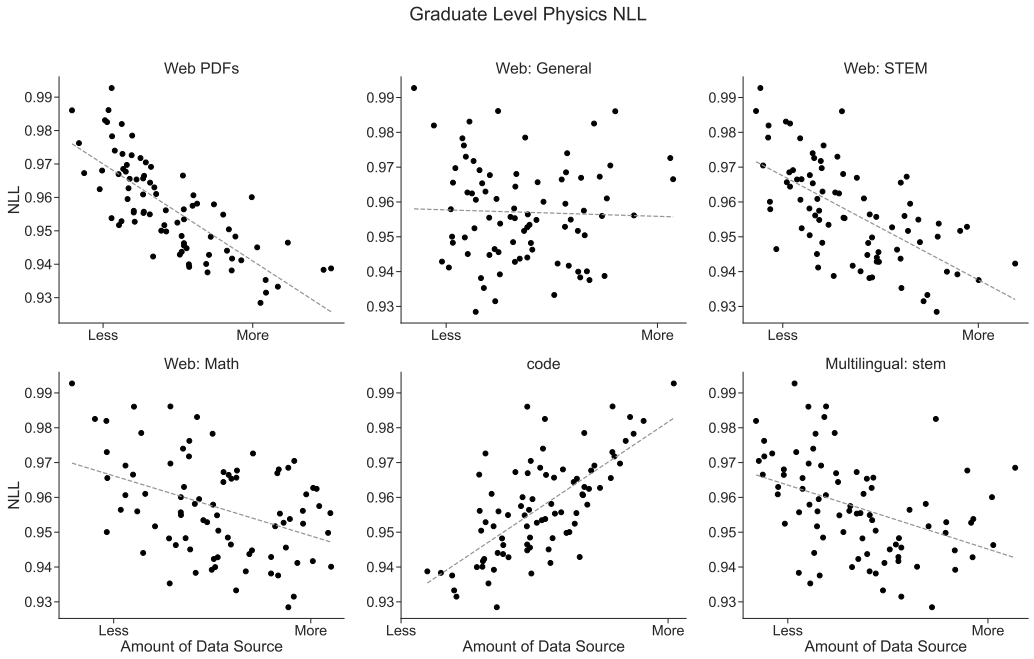

Figure 7. For a vendor-created graduate level physics evaluation set, we show the relative contribution of different datasets (lower NLL is better). We sample different datamixes, train on each of them, and evaluate the resulting models; then, we look at correlations between the final NLLs and the mixes' data composition. We see clear *positive* trends with our PDF datasets, math and STEM data extracted from the web. Our general web corpus is mostly neutral, while adding more code to the mix (instead of others) does not seem to help with Physics.

between different subsets (e.g., up-weighting STEM and PDFs and reducing the weight of all other components). Fig. 7 illustrates how this process works.

Throughout these searches, we cap the maximum number of repetitions of any given dataset at 8, to safeguard against overfitting and the diminishing returns from looping over the same data multiple times (Chen et al., 2026a).

After performing multiple iterative rounds of global and local data optimization, we selected a few promising candidate mixtures that minimized our NLL objective function. We trained these models for approximately 2.8× more compute than that used for global mixing and selected the best candidate. After these adjustments to the data mix procedure, we saw good scaling in this scale-up validation step, where the optimal candidate no longer changed with scale.

Tab. 5 summarizes the resulting composition. Each source family contains several datasets, and the number of epochs per dataset varies. Some datasets could potentially belong to several categories (e.g., a STEM dataset in French), and for the table, we assign each dataset to the first matching row from top to bottom. In our final mixture, we end up with around 300 billion high-quality math tokens. These are sampled 5.28× on average, the most of any source family. The bulk of our training mixture is coding data however, coming in at 16.4T training tokens, with just over 2 epochs on average. Our web text and PDF data are each seen less than once on average (0.55× and 0.53×), meaning the full available corpus of those sources is not exhausted even over a 30T-token run. Note that for both PDFs and Web text, math and stem components are counted in subject-specific categories. Our multilingual corpus is the most aggressively downsampled: 8.1T unique tokens are available, but the final mix consumes only 0.5T (0.06×) – again noting that domain-specific multilingual data is included within different categories.

Source family	Unique tokens (T)	Training tokens (T)	Mix Percentage (%)	Avg. Epochs
Code	7.4	16.4	54.6	2.22×
STEM	2.2	4.7	15.8	2.17×
Math	0.3	1.6	5.4	5.28×
Books and journals	0.6	0.9	3.1	1.65×
PDFs	2.7	1.4	4.7	0.53×
Web text	8.1	4.5	14.9	0.55×
Multilingual (*other*)	8.1	0.5	1.6	0.06×
Total	**29.2**	**30.0**	**100.0**	**1.03×**

Table 5. MAI-Base-1 pre-training data composition. *Unique tokens* is the deduplicated token count per source family. *Training tokens* is the number of tokens consumed from that family over the full run. *Avg. Epochs* is the ratio of training tokens to unique tokens; values above 1× indicate repeated sampling.

We continue to iterate on our data-mixing objective, the NLL-based evaluations that compose it, and the algorithms used to propose new candidate mixtures, as each of these components has a substantial impact on the final datamix. Understanding end-to-end scaling behavior – and identifying the metrics that most reliably predict downstream performance during pre-training – remains a fundamental open challenge, one for which we believe significant headroom still exists.

2.5.4 Mid-training Data Mixture

After pre-training, MAI-Base-1 is mid-trained in two sequential stages as described in 2.6.1. The mid-training data is drawn entirely from the pre-training corpus (Sec. 2.4); no new or synthetic sources are introduced. We instead curate a higher-quality subset by filtering and re-weighting the existing mixture, then re-pack the result at longer sequence lengths for context extension.

Overall, we further bias the mixture towards STEM, math, and code to build a strong foundation for reasoning RL climbs (Wang et al., 2025b). Starting from the pre-training mixture, we increase the proportion of STEM/math data to 35%, keep code at 55%, and assign the remaining 10% to background sources. Within each category, we locally tune weights according to single-source microanneals (Team OLMo et al., 2024), optimizing the same NLL-based objective used during the pre-training mixture search, augmented with long-context NLL tasks (Sec. B.2) for context extension. Additionally, we apply quality filtering to a select set of STEM and code sources as described below.

STEM reasoning data. To improve performance on reasoning tasks, we filter documents primarily from the pre-training PDF corpus using the content quality taxonomy introduced in (Essential AI et al., 2025). We define a *Bloom Analyze* heuristic that keeps documents with high technical correctness, at least intermediate reasoning depth, and a Bloom cognitive processing level at or above *Analyze* (Anderson and Krathwohl, 2001). We also remove documents with extraction artifacts or missing primary content. This filter keeps documents with structured reasoning and analysis, rather than simple factual statements.

Code data. Code data undergoes two additional steps for mid-training. First, we perform file extension filtering on a per-quality-bin basis, where repositories are grouped into three tiers by a repo-level quality metric. For example, common web assets (HTML, CSS, SVG) are retained in top-tier repositories where they typically belong to larger frontend projects, but are removed from lower-tier repositories where they are predominantly low-quality standalone pages. Second, we introduce file-level document formatting alongside the existing repo-level format from pre-training, treating repository understanding and individual file understanding as complementary tasks, which improves generalization on internal file-local NLL benchmarks.

Memorization-aware epoch capping. In addition to filtering for quality, we cap data source exposure during mid-training to avoid overfitting sources that appear highly memorized by the end of pre-training. We estimate this using a per-source validation proxy: the fraction of validation loss improvement between two pre-training checkpoints that comes from tokens predicted with near-certainty (NLL < 0.01). A higher fraction indicates significant NLL reduction due to memorization or highly repeated structure. We assign stricter epoch caps during mid-training for such sources while allowing proportionally more exposure to those with smaller values.

Long context data mixture. Mid-training introduces two stages for extending the context window of MAI-Base-1 to 64k and then 256k as described in Appendix B. In both phases, we re-pack the data at the longer sequence length without modifying mixture weights. Doing so allows us to minimize distribution shift while significantly reducing truncation of existing high quality long context documents (e.g., instructional or academic texts).

2.6 Training Recipe

MAI-Base-1 was trained in three phases with progressively increasing sequence lengths: pre-training, mid-training phase 1, and mid-training phase 2. We describe their training recipes below.

2.6.1 Training Phases

Tab. 6 shows the number of tokens drawn from the data mixes described in Sec. 2.5.3, context length and the number of GPUs used for different phases. We use expert parallelism 64 with ZeRO-2 for pre-training and mid-training 1, and we enable ZeRO-3 (FSDP) for mid-training phase 2. Context parallelism is used for the mid-training phases. The training uses our in-house large-scale training framework (Sec. 2.8).

Phase	Tokens	Context length	GB200 GPUs
Pre-training	30 T	16,384	8,192
Mid-training 1	3.4 T	65,536	8,192
Mid-training 2	150 B	262,144	4,096

Table 6. Training specifications across phases for MAI-Base-1.

2.6.2 Hyperparameters

MAI-Base-1 is optimized with AdamW (Loshchilov and Hutter, 2017) ($\beta_1 = 0.95$, $\beta_2 = 0.925$, $\varepsilon = 10^{-8}$) and a constant weight decay of 0.1, with reduced weight decay on attention weights (0.01) and embedding weights (0.005) to limit regularization on parameters that benefit less from it. The global gradient norm is clipped to 1.0. The learning rate follows a linear warmup of approximately 12 B tokens, then decays along a cosine schedule from a peak of 2×10^{-4} to a minimum of 2×10^{-5}—a final-to-peak ratio of $0.1\times$ rather than the more common $0.01\times$ as we found that decaying the LR less improved post-RL results. We apply dropout 0.15 at each layer's output before the residual add. While high dropout is not standard practice to our knowledge, we find it provides complementary regularization effect alongside weight decay, and observe that it improves performance on evaluation tasks measured by the scaling ladder. By default the model weights are initialized from a zero-mean normal distribution with a standard deviation of 0.02, and the output projection weights preceding residual additions are scaled down by the inverse square root of the total number of residual connections in the model (Radford et al., 2019).

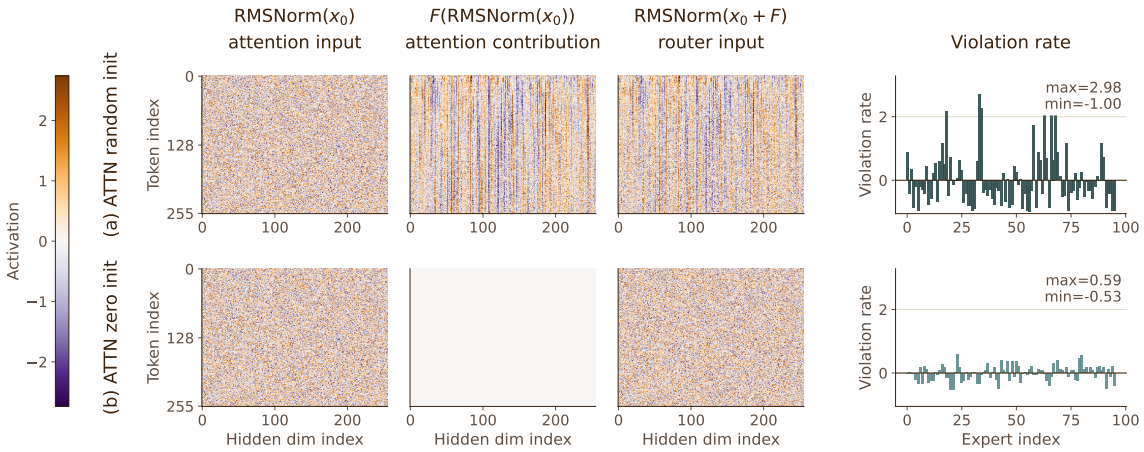

Figure 8. (a) With random initialization, the attention layer initially produces collapsed representation by essentially performing causal mean pooling across the sequence dimension. High correlations between different rows can be observed in attention's contribution, which reduces diversity in token representations and leads to high imbalance of the subsequent MoE layer. (b) With attention layer's output initialized as zero the token distribution in MoE layers is better behaved. The violation rate for an expert is its fractional deviation from the mean token count across experts.

All phases use a global batch size of 134M tokens. The mid-training phases inherit the pre-training hyperparameter setup unless otherwise specified. Mid-training phase 1 uses a cosine decay learning rate schedule with a peak learning rate of 2×10^{-5} and a minimum of 1×10^{-5}, without warmup. Mid-training phase 2 uses a constant learning rate of 1×10^{-6}, also without warmup.

Attention initialization. Upon initialization, the attention softmax is close to uniform, which effectively performs average pooling over tokens subject to the causal constraint, as illustrated in Fig. 8. This behavior reduces the diversity in token representations and can lead to highly imbalanced routing in subsequent MoE layers, increasingly so with depth. To address this, we initialize the attention output to zero, achieved by setting the output RMSNorm gains to zero. This means the model initially behaves like a stack of feedforward layers applied to individual tokens, while the cross-token interactions captured by the attention layers gradually kick in over the course of training. We confirmed that the significant reduction in initial peak imbalance achieved through this method translates to quality gains as measured by EG (see Sec. 2.2.2).

2.6.3 Numerical Precision

For model training our default weight and activation data type is BF16. We use FP8 E4M3 for GEMMs in the forward pass, FP8 E5M2 for the data-gradient, and BF16 for the compute of the weight-gradient with FP32 gradient accumulation (Micikevicius et al., 2017, 2022). All FP8 operations use delayed scaling with a 1024-step history of the absolute maximum value.

We use FP32 in a number of numerically sensitive locations. For activations, this covers any pre-softmax activations (attention scores, MoE router logits, and the final output logits), MoE combine, and the entire residual stream all the way from the input embedding to the output. For parameters, this includes the embedding weights, RMSNorm weights, and router weights. The main parameters and the momentum buffers in the optimizer are kept in FP32, as is all the AdamW computation. The data-parallel all-reduce and the micro-batch gradient accumulation buffers run in FP32. The MoE routing and the final vocabulary weight GeMMs are also both kept in FP32.

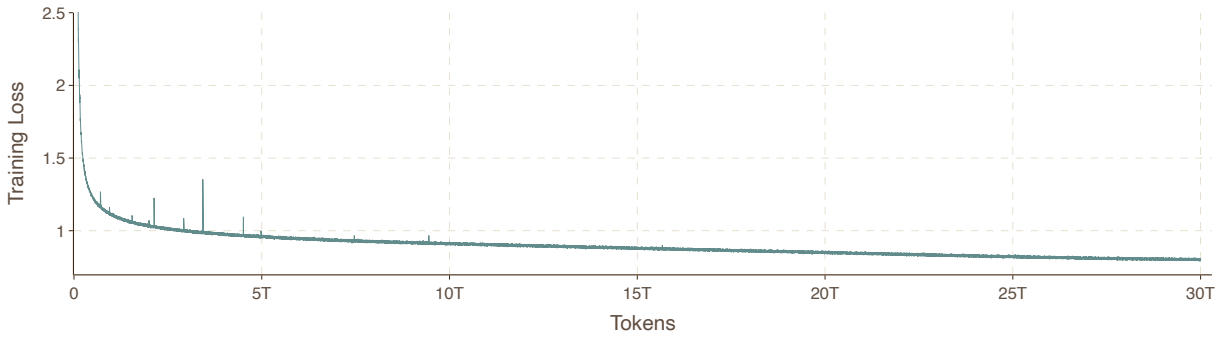

Figure 9. Pre-training loss curve for MAI-Base-1 (35B active / 1T total parameters) over 30T tokens, shown without smoothing. Loss recovered quickly from several early-stage spikes. No training batches were skipped, and no manual interventions were made to the training configuration mid-run.

We fuse casts with adjacent operations (e.g., RMSNorms) where possible. Stochastic rounding (Zamirai et al., 2020; Ozkara et al., 2025) is applied when gradients flow from higher to lower precision. This occurs because the computation precision within a layer is lower than that of the residual stream, so gradient downcasts are required in the backward pass.

2.6.4 Training Loss Curve

The training loss curve of MAI-Base-1 is presented in Fig. 9, without smoothing. Several spikes appeared early in training, but the loss value recovered quickly in each case and we continued the run without interventions. From runtime logging we identified that the spikes predominantly affected coding datasets and correlated with high expert imbalance under dropless routing: a disproportionate concentration of tokens in a few experts.

2.7 Evaluation and Comparison with Contemporaneous Models

In this section we compare MAI-Base-1 with contemporaneous base models available, over a subset of the same NLL evaluations that we use for development, as discussed in Sec. 2.3. One of them is based on the source code for our internal training stack (Sec. 2.8); the others were commissioned from vendors, so we are confident that they were not in the training set of any models. We do not use multilingual benchmarks described on Tab. 3 which are drawn from public sources that we excluded from our own training, but are unable to make a reliable comparison with those.

We focus on pre-training base models in this comparison and report bits-per-byte (BPB) values which are invariant across tokenizers. NLL evaluation is not possible through most model APIs, so we have focused on open-weight models hosted on `huggingface.co`: DeepSeek v3.2 (DeepSeek-AI et al., 2025b)[3], DeepSeek v4 Pro (DeepSeek-AI, 2026)[4], Kimi-K2 (Kimi Team et al., 2026)[5] and Gemma4-31B (Gemma Team, 2026)[6]. In addition, we include one of our previous-generation 23B models that was also trained for 30T, allowing us to quantify our progress.

The results are summarized in Fig. 10. MAI-Base-1 shows a clear advantage over models with a similar number of active parameters, achieving lower bits-per-byte across all four held-out tasks. DeepSeek-v4 achieves the best results, with 1.4× active parameters and 1.6× total parameters relative to MAI-Base-1. Our

[3] deepseek-ai/DeepSeek-V3.2-Exp-Base
[4] deepseek-ai/DeepSeek-V4-Pro
[5] moonshotai/Kimi-K2-Base
[6] google/gemma-4-31B

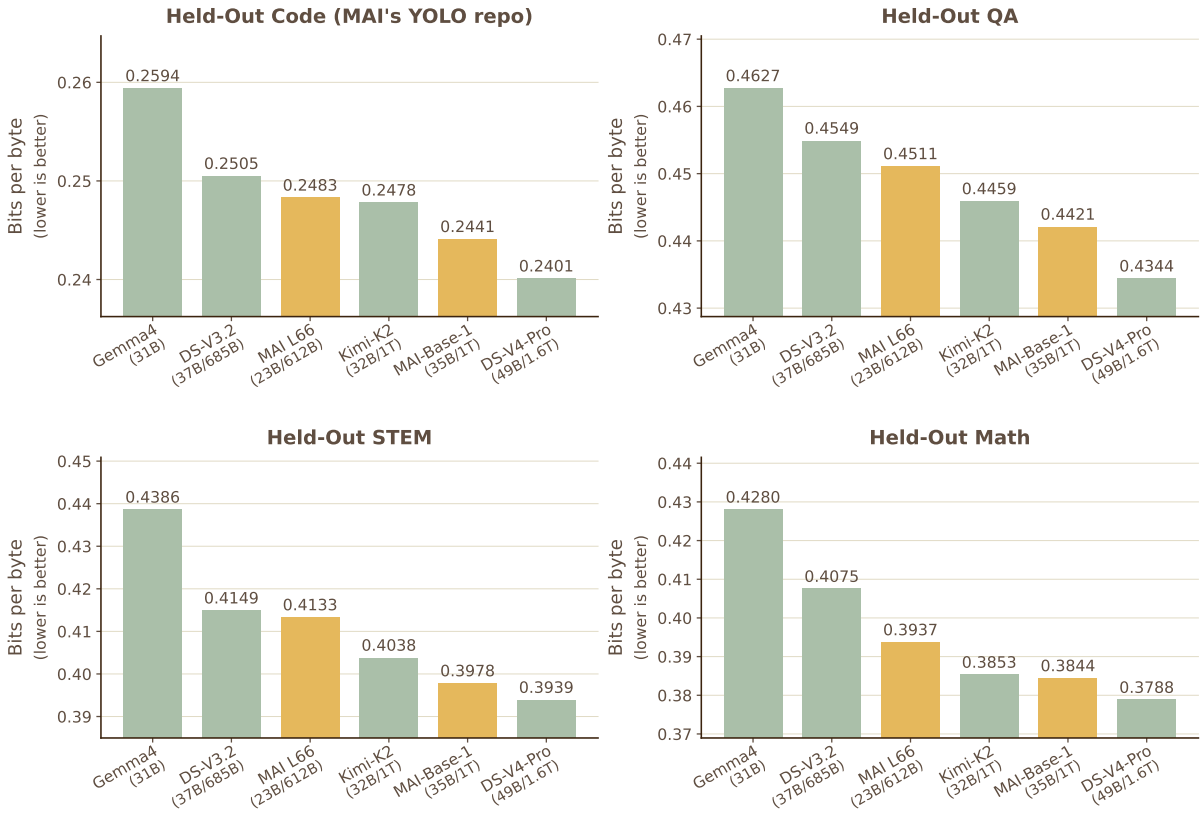

Figure 10. Bits-per-byte (BPB) comparison of base pre-trained models across four held-out evaluation tasks (lower is better). MAI models are illustrated in yellow. We also provide an additional data point with 23B (L66 from an older version of our ladder 2.2.1), a prior run of ours trained over the same number of tokens (30T) with the MAI-Base-1. The internal held-out evaluation tasks used are described in Sec. 2.7.

23B model outperforms the DeepSeek v3.2 base model while using only 62% of its active parameters and a comparable number of total parameters.

2.8 YOLO: Distributed Training at Scale

You **O**nly **L**aunch **O**nce (YOLO) is MAI's in-house framework for large-scale training used throughout the development cycle of MAI-Thinking-1. YOLO is built on top of PyTorch (Paszke et al., 2019) and sits inside the scheduling and runtime stack described in Sec. K.3.

YOLO is designed to support all phases of large-scale model training, including pre-training, mid-training, supervised fine-tuning, and (the training aspect of) reinforcement learning. It implements the core training loop including our model definition, sharding, optimizer, dataloading and checkpointing. We built YOLO from scratch to give us full end-to-end control over the training stack: from kernels to parallelism to scheduling. YOLO draws from the experience of large-scale training frameworks such as Megatron-Core (Yan et al., 2026), DeepSpeed (Rasley et al., 2020), and TorchTitan (Liang et al., 2024) and is co-designed and deeply integrated with the rest of our compute infrastructure and model training. This allows us to equally prioritize training speed, goodput, determinism, flexibility, and developer agility as the compute environment and model architecture evolve. The following section describes YOLO's main

components and how they enable optimization with our design principles.

2.8.1 System Overview

YOLO consists of a set of building blocks that implement the Transformer model, as well as the training loop and any supplementary components such as dataloaders, checkpointing, logging and observability.

Kernels. To efficiently support the numerical recipe at scale (2.6.3), YOLO includes a suite of custom kernels written in Triton, CUDA, CuteDSL, and CUTLASS. These comprise FP8 GEMM kernels optimized for the forward and data-gradient computations with support for FP8 delayed scaling; grouped GEMMs for MoE that improve efficiency when each GPU hosts multiple experts—typically 8 in our pre-training runs (512 experts with expert parallelism size 64)—and support all mixed-precision execution modes, including FP8 delayed scaling; and fused quantization kernels that quantize both the original and transposed tensors needed for the backward pass while applying scale-factor swizzling for memory efficiency, with casts adjacent to RMSNorm layers fused into the normalization kernels where possible to reduce overhead. In addition, the non-grouped quantization kernels support Cluster Launch Control (CLC), a Blackwell-specific feature for dynamic kernel load balancing.

Tensor sharding. YOLO uses combinations of different forms of parallelism, depending on the model size and cluster infrastructure for individual training runs. To enable fine-grained control over how individual tensors in the model are sharded, YOLO implements a set of custom sharding annotations, similar to JAX (Bradbury et al., 2018) or PyTorch's DTensor (Paszke et al., 2019). Sharding annotations describe how tensors are partitioned or replicated across n-dimensional cartesian meshes and can be used to group tensors by their sharding type. The underlying mesh onto which each tensor is partitioned sits on top of a PyTorch communicator that exposes the collective- and point-to-point communication primitives for that tensor. We deliberately keep annotations purely descriptive, meaning they do not automatically insert communication primitives into the computational graph to avoid creating accidental synchronization points.

YOLO's distributed tensors give us the flexibility to define the sharding for each parameter in the model independently. This allows flexible tensor-parallelisms within our model. For example, we use tensor-parallelism for embedding, loss, and attention weights, but not for MLP and MoE weights. Another example is that we can use varying degrees of data parallelism between tensors that belong to an MoE or non-MoE layer (referred to as *parallel folding* (Yan et al., 2026)).

Parallelism. Based on the distributed tensor, YOLO implements all commonly used forms of parallelism: data parallelism, tensor and context parallelism, expert parallelism, and pipeline parallelism (Huang et al., 2019; Qi et al., 2023).

- **Data parallelism.** YOLO uses custom versions of ZeRO stages 1-3 (Rajbhandari et al., 2020) built from scratch to work with our optimizer and performance optimizations. The implementation groups parameters by their unique shardings, data types, and configurable ZeRO-units, and combines all parameters and gradients into contiguous buffers. This minimizes the total number of collectives required for gradient reduction and ZeRO 1-3. Furthermore, our ZeRO implementation breaks parameters into multi-dimensional chunks so that they remain compatible with slicing for distributed checkpointing and to avoid reordering before and after executing collectives. YOLO's ZeRO implementation always stores parameters in sharded form, regardless of the ZeRO stage. This simplifies the implementation of distributed AdamW, as the optimizer state is always automatically sharded as well. The main difference between the ZeRO stages is when and how often the buffers for gathered parameters or gradients are cleared. E.g., for ZeRO-1, parameters are all-gathered during the first forward pass and only cleared after the final micro-batch, whereas for ZeRO-3 (FSDP), parameters are cleared right after their usage.

- **Tensor parallelism** in YOLO is implemented via column-parallel and row-parallel GEMMs, in which both activations and weight tensors are sharded across the same number of ranks (commonly referred to as sequence parallelism) (Shoeybi et al., 2019; Korthikanti et al., 2023). Any additional sharding of activations tensors beyond the degree of the weight sharding is treated as context parallelism in the attention layer and as data-parallelism otherwise.
- **Context parallelism.** YOLO uses Ulysses-style context parallelism (Jacobs et al., 2023), in which activation tensors are re-partitioned via a parallel matrix-transpose (all-to-all) to move the sharding of the sequence dimension to the attention-head (hidden) dimension. For MAI-Base-1, context parallelism was only used for long-context mid-training, whereas tensor/sequence parallelism was used for all training stages.
- **Expert parallelism**. Initially, YOLO's implementation of expert parallelism was similar to GShard (Lepikhin et al., 2020) which uses all-to-all communication with static tensor shapes and a pre-defined capacity factor. In later runs, we switched to a dropless MoE implementation that routes every token to its selected experts using variable-size all-to-all communication. All pre-training runs use expert-parallelism within the NVLink domain (with tensor-parallelism of size one) and data-parallelism across racks.

Activation checkpointing and offloading. When training MAI-Base-1 with ZeRO data parallelism, activations consumed the majority of the available high-bandwidth device memory (HBM). To reduce the memory footprint and enable expert parallelism within a single NVLink domain, YOLO supports both activation checkpointing and offloading activations to host memory. Similar to frameworks such as Megatron-Core, activation checkpointing can be configured at different granularities, ranging from recomputing one or more full transformer layers to fine-grained recomputation of intermediate activations within a layer (e.g., recomputing the output of the activation function). Across all training phases of MAI-Base-1, we use a combination of transformer-layer checkpointing and fine-grained recomputation. Since the memory required to store activations for checkpointing scales linearly with the number of transformer layers, YOLO also implements activation offloading. Activation offloading complements checkpointing and introduces minimal overhead as device-to-host transfers are executed asynchronously on dedicated CUDA streams using pinned host memory.

Mixture of experts. Our MoE layer balances research flexibility with training efficiency. It supports both dropless and capacity-capped modes, multiple load-balancing strategies (including global-batch load balancing loss (Qiu et al., 2025; Yang et al., 2025) and loss-free (Wang et al., 2024a)), configurable top-k scoring, fine-grained telemetry, and router replay for RL rollouts. For expert parallelism, we partition local experts into groups of configurable size and pipeline the `dispatch` → `compute` → `collect` phases across groups, so that all but the first dispatch and last collect overlap with expert computation; the remaining exposed communication is overlapped with global load-balance computation and, optionally, shared experts. To prevent imbalance-induced memory swings, fragmentation, and OOMs in dropless mode, we support a static-memory dropless mode that runs multiple capped `dispatch` → `compute` → `collect` rounds per group, each processing up to a fixed capacity of tokens. In the backward pass, we then perform per-expert-per-round fine-grained recompute in order to avoid storing imbalanced activations. For performance, we use custom Grouped GEMM and quantization kernels supporting all numerics recipes with low overhead, and custom CuTe DSL symmetric-memory kernels for device-initiated, variably-sized, high-throughput all-to-all communication over NVLink.

2.8.2 Determinism and Correctness

We enforce determinism in training for reasons including scientific reproducibility, developer agility, and debugging system health, even at the cost of reduced training efficiency (e.g., MFU). For a fixed hardware topology, model configuration, and software version, we ensure *bitwise* reproducibility in the pre-training

process. In short, two training runs with the same hardware resources and configurations will produce bitwise identical models. Determinism requires careful attention at several levels of the training stack:

- **Data pipelines**. The ordering of training micro-batches is fixed for a given configuration of data-parallel workers. While these workers asynchronously prefetch batches to hide I/O latencies, we ensure that the queue of pending batches is consistent across runs and restarts.
- **Training checkpoints**. We store all stateful data including model weights, optimizer state, FP8 scaling history, dataloader progress, and random-number generators.
- **GPU kernels**. Floating-point accumulation is non-associative, so reproducible reductions require control over the order in which values are combined. Our training stack performs deterministic accumulations in prescribed orders, rather than relying on unordered GPU atomics whose accumulation order is scheduler-dependent. For example, RMSNorm backpropagation accumulates gradients using a two-stage tiled reduction: the first kernel materializes per-tile partial sums, and a finalization kernel reduces those partials in a fixed order. Another example is the top-k selection for MoE routing: our kernel uses a stable sort to avoid non-deterministic tie-breaking.
- **Network collective communications**. Similar to floating-point reductions within GPU kernels, collectives such as all-reduce and reduce-scatter must be deterministic. We disable NVLink SHARP (Graham et al., 2020) to ensure that the collective operations are deterministic at the cost of reduced performance. We also enforce consistent NCCL topologies run-to-run so that intra-rack reductions always use the same accumulation order.

We also leverage determinism to build confidence in our numerical correctness as YOLO evolves. Our tests track "golden" model configurations and compare for exactness against all training metrics (e.g., loss, gradient norms, and weight tensor checksums). This does not mean that we disallow any changes to numerics that result in a different training result; changes are expected when we make changes that affect floating-point behavior (e.g., upgrade system software or GPU kernels). This gives us an explicit history of expected numerical changes in our software history.

2.8.3 Fault Tolerance

Large-scale training must tolerate frequent hardware failures with minimal effect on training efficiency. YOLO's fault-tolerance approach combines distributed checkpointing and fast-failover to hot standbys to minimize lost training progress and downtime during failures.

Distributed checkpoint format. We build on PyTorch's distributed checkpointing library (DCP), which deduplicates replicated state across ranks and supports tensor resharding at load time. We rewrote DCP's metadata creation and serialization paths to lower CPU memory overhead, reduce Python garbage collection pressure, and accelerate serialization/deserialization. Collectively, these optimizations reduced both CPU memory overhead and checkpoint save time by more than $10\times$. In our implementation, only the public APIs and the core logic for mapping tensor shards to storage – the save plan – remain from the original library. We also pre-compute DCP save plans and cache them in persistent storage, removing this expensive computation from the critical path.

Asynchronous checkpointing. To minimize training downtime when saving checkpoints, we first copy checkpoint tensors from device to host directly in the training processes, then offload the remaining work to a separate checkpointing process while training proceeds. The checkpointing process writes data to persistent storage via the frontend network and handles cross-rank coordination to atomically commit the checkpoint. To support our scale, we replaced the standard CPU collective backend with a custom coordination layer built on a shared key-value store. We further employ an opportunistic admission-control mechanism that polls for completion of in-flight checkpoints and initiates the next one as soon as possible, while ensuring at

most one checkpoint is in flight at a time. This approach, combined with careful host memory management, allows us to checkpoint frequently while minimizing impact on training MFU.

Storage infrastructure. Our training data center has local Azure Blob Storage (Calder et al., 2011) that provides low-latency, high-bandwidth access. This enables us to rely directly on checkpointing for fault tolerance without the need for intermediate node-local caches or cross-rank state replication. To avoid storage hotspots at scale during checkpoint load when many ranks require the same replicated state, we load replicated state once from storage and broadcast it via NCCL, eliminating single-blob fan-in hotspots that can overload a single storage server.

Rapid job recovery. We achieve fast recovery through in-job restarts using `Ray` (Moritz et al., 2018) actors with hot standbys, eliminating the overhead of pod recreation. On GB200 systems, we explicitly clear OS caches during actor restarts to prevent crash loops caused by out-of-memory errors when GPUs hold on to residual memory from previous runs. Finally, we validate determinism after restarts by comparing to historical loss metrics for recomputed steps.

2.8.4 Co-optimizing Performance with Model Architecture

The final architecture of MAI-Base-1 was the result of a continuous co-design process spanning five major generations of the model (v1–v5). Each generation pursued two complementary objectives. First, we sought to improve the efficiency gain and modeling capability of the architecture through changes to the model design, routing strategy, and training recipe. Second, we continuously improved training efficiency by identifying performance bottlenecks and incorporating new infrastructure, kernel, communication, and memory-management optimizations. As architectural changes altered the computational characteristics of the model, they frequently introduced new system bottlenecks, requiring corresponding improvements throughout the training stack.

We describe the architecture and infrastructure evolution of our models to highlight our approach towards building large scale models. The two metrics that were tracked across each evolution are:

- **Model FLOP Utilization (MFU)** (Chowdhery et al., 2023), defined as $\text{FLOP}/(t_{\text{step}} \cdot \text{FLOP}_{\text{spec}})$ which quantifies training efficiency in practice. Here FLOP denotes the total number of floating point operations from all leading-order compute kernels, including GEMMs and attention, but excluding memory-bound operations such as RMSNorm and activation functions. t_{step} is the end-to-end training step time, including data loading, forward and backward passes, communication, and optimizer updates. Although training uses a mixture of FP8, BF16, and FP32 operations, we normalize MFU using the FP16/BF16 tensor core throughput reported by NVIDIA ($\text{FLOPsspec} = 2.5 \times 10^{15}$ FLOPS per GB200 GPU for dense operations). Operations introduced by activation checkpointing are not included in the FLOP count.

- **Efficiency Gain:** Defined in Sec. 2.2.2, which captures the theoretical improvement in model efficiency for achieving a given quality.

Tab. 7 summarizes the key architectural changes across model generations. Fig. 11 reports both the initial MFU obtained when running the new architecture with the optimization stack from the previous generation and the final MFU achieved after applying version-specific performance improvements. Across the development of MAI-Base-1, more than twenty infrastructure and kernel optimizations were introduced, enabling each generation to sustain an MFU above 20% despite substantial increases in model scale and architectural complexity. Generations v0 and v1 were early prototypes trained on H100 systems and are therefore excluded from Fig. 11.

Beginning with **Version 2**, all major training runs were conducted on GB200 NVL72 clusters. Version 2 of MAI-Base-1 (23B active parameters) was the first model version trained on a GB200 NVL72 cluster, using 4,096 GPUs in total. To improve system robustness, we used 64 GPUs per rack (NVL64), reserving spare capacity to tolerate node failures and unhealthy devices. Version 2 adopts a deeper and narrower design than earlier versions while maintaining a 23B active parameter budget, leading to relatively smaller hidden

Model version	v2	v3	v4	v5 (MAI-Base-1)
Active/Total params	23B/600B	23B/600B	23B/611B	35B/1T
Layers	54	54	66	78
Hidden	4352	4352	5632	6656
FFN	34816	34816	11264	13312
Down-proj Dim	4352	4352	4096	3072
Expert FFN	17408	17408	9216	10240
Expert capacity factor	2	∞	∞	∞
Top-k/Experts	4/192	4/192	8/512	8/512

Table 7. Architecture evolution during the development of MAI-Base-1. Here we report the target architectures for large-scale infrastructure testing instead of the ladder models.

dimension. To maximize GEMM efficiency we selected an expert parallelism (EP) degree of 64 and tensor parallelism (TP) degree of 1, ensuring that all expert all-to-all (A2A) communication remained within the NVL64 domain. This configuration achieved an initial MFU of 18%. We used cross-rack InfiniBand (IB) for data parallel (DP) communications (e.g., all-gather of parameters and reduce-scatter of gradients). We maintained the same setup across subsequent versions.

Profiling revealed three primary bottlenecks: DP communication, deterministic attention kernels, and expert routing overhead. Enabling GPU Direct RDMA support improved overlap between communication and compute and reduced end-to-end training time by approximately 1.1×. Due to the inefficiency of FlashAttention-2's (FA2) deterministic mode on GB200 and lack of availability of FA4, we developed a custom block-sparse attention backend that improved step time by an additional 1.06×. Given the single-micro-batch training configuration, we adopted ZeRO-2 optimizer sharding, which reduced gradient memory consumption without introducing additional communication relative to ZeRO-1. This allowed us to disable additional activation recomputation and further improve performance by 1.03×. Finally, we replaced a memory-bound PyTorch expert encode kernel, which utilized only 10% of available HBM bandwidth, with a custom Triton implementation that achieved approximately 80% HBM utilization and improved end-to-end performance by an additional 1.03×. Collectively, these optimizations increased MFU from 18% to 22%.

Version 3 retained the same overall model architecture but switched from capacity-constrained routing to dropless MoE routing. From the efficiency gain point of view, this allowed us to use a better MoE load balancing policy. This also had a positive impact on MFU by eliminating expert-capacity padding, thus reducing both communication volume and expert GEMM computation. However, dropless routing introduced new synchronization overheads because expert token counts and tensor shapes must be communicated dynamically during execution. To mitigate these costs, we overlapped token-count communication with other runtime operations and moved synchronization points to dedicated streams. These changes allowed us to realize the efficiency benefits of dropless routing while maintaining an MFU comparable to Version 2.

Version 4 introduced three significant architectural changes: the number of experts increased from 192 to 512, routing expanded from top-4 to top-8 expert selection and use of LatentMoEs. Increasing the number of experts helped increase the learning capacity of the model, while latentMoE helped manage the A2A communication cost with the increased number of experts. At the same time, the training job was scaled from 4,096 to 8,192 GPUs. LatentMoEs reduced expert GEMM sizes, making CPU launch overhead and kernel efficiency substantially more important. As a result, the initial MFU dropped from 22% to approximately 16%. Use of FA4 deterministic kernels, (1.14× over block-sparse attention backend) combined with several improvements aimed at reducing CPU overhead and increasing runtime batching efficiency, boosted the MFU from 16% to 20%.

Version 5 increased the active parameter count from 23B to 35B and total parameters from 600B to 1T. The larger model significantly increased both parameter and activation memory requirements. Initial

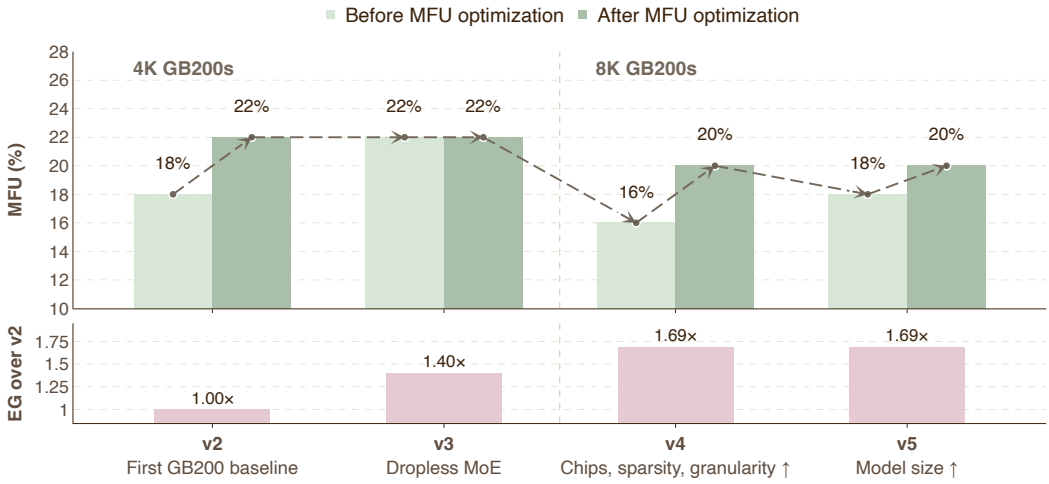

Figure 11. Evolution of MFU and EG across pre-training configurations trained on GB200s. From v2 onward, each model change improves EG but initially degrades MFU when running with the *previous* configuration before deploying optimizations. In total, we added 20+ optimizations to surpass an MFU of 20% for each pre-training run. Note here we only list model changes with notable infrastructure implications—they are not the only changes between the model versions that have contributed to the improved EG.

deployments therefore relied on ZeRO-3 optimizer sharding, but profiling showed that the additional parameter all-gather operations caused the backward pass to become communication bound. To address this issue, we reduced GPU memory pressure through activation offloading, allowing the training system to revert to ZeRO-2 sharding. Eliminating the ZeRO-3 parameter all-gather operations restored overlap between communication and computation and improved overall training efficiency, enabling the final Version 5 configuration to sustain an MFU of approximately 20%.

3 The Reinforcement Learning Climb

Pre- and mid-training give MAI-Base-1 broad predictive competence and knowledge, but they do not specify how the model should behave, solve long-horizon tasks, or allocate inference-time computation. Our reinforcement learning (RL) climb addresses this by optimizing the model against task-specific feedback, allowing it to produce chains of thought (CoTs) before responding, use external tools, interact with environments, and follow preference and safety signals.

As MAI-Thinking-1 is our first in-house reasoning model, our RL climb starts from a checkpoint with no prior exposure to reasoning traces. The model therefore has to develop its reasoning abilities from scratch, making long-term training stability a central challenge. We enable this via three mechanisms: (i) two simple but crucial adjustments to Group Relative Policy Optimization (GRPO, Shao et al., 2024), (ii) *self-distillation* for resuming RL climbs after crashes or updates to the base policy (Sec. 3.1), and (iii) infrastructure improvements that eliminate numerical mismatch between training and inference (Sec. 3.6).

To enable parallel development, we train three domain-specific specialist models: one model for STEM and competitive code (Sec. 3.2), one for agentic coding and tool use (Sec. 3.3), and one for helpfulness and safety (Sec. 3.4). The RL climbs for all specialists follow the same recipe but differ in the distribution of prompts and what the model is rewarded for. As illustrated in Fig. 12, the specialist models are subsequently

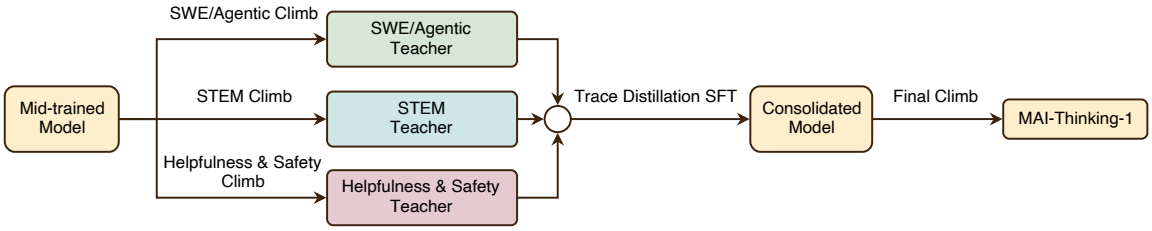

Figure 12. Overview of our RL climbs. We begin with MAI-Base-1 and train three specialist models using reinforcement learning. These specialists are then distilled into a single consolidated model with simple SFT, which undergoes a final RL stage to produce MAI-Thinking-1.

consolidated into a single model using supervised finetuning (SFT). A final lightweight RL climb turns this consolidated model into MAI-Thinking-1, a model that performs strongly across all domains (Sec. 3.5).

3.1 Reinforcement Learning Recipe

3.1.1 RL Objective

Our reinforcement learning climb starts from a policy π_θ (e.g., our mid-trained model). For a prompt q, the rollout policy samples a group of G responses $y_{1:G}$, and each response y_i receives a scalar reward:

$$R_i = R(q, y_i). \tag{4}$$

The reward function R is domain-dependent; typically, it is either based on executing code or feedback from a prompted AI judge or a trained reward model. We derive our training objective from GRPO (Shao et al., 2024) with token-level policy gradient (Yu et al., 2025):

$$\mathcal{J}(\theta) = \mathbb{E}_{q\sim P(Q),\, y_{1:G}\sim\pi_{\text{old}}}\left[\frac{1}{\sum_{i=1}^{G}|y_i|}\sum_{i=1}^{G}\sum_{t=1}^{|y_i|}\min\Big(r_{i,t}(\theta)\,A_i,\ \text{clip}\big(r_{i,t}(\theta),\,1{-}\epsilon,\,1{+}\epsilon\big)\,A_i\Big)\right], \tag{5}$$

where $P(Q)$ is the distribution over all prompts and π_{old} denotes the policy used to generate the rollouts. In practice, the normalization is computed over all tokens in the global training batch (across all data-parallel ranks), so that every token contributes equally regardless of response length. For response y_i and token position t, the importance-sampling ratio is

$$r_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid q,\, y_{i,<t})}{\pi_{\text{old}}(y_{i,t} \mid q,\, y_{i,<t})}. \tag{6}$$

The response-level advantage $A_i = \big(R_i - \text{mean}(R_{1:G})\big)\big/\,\text{std}(R_{1:G})$ is shared across all tokens in response y_i.

We apply two modifications to this objective: *adaptive entropy control*, which dynamically adjusts the upper clip bound to maintain a target policy entropy, and an *outer ratio clip*, which caps the unclipped branches of our objective to prevent gradient-norm explosions.

Adaptive entropy control. Similar to Yu et al. (2025) and Mistral-AI et al. (2025), we use separate lower and upper clip bounds. We parametrize these using a single base hyperparameter ϵ, which controls the base trust-region width, together with an entropy-dependent relaxation k of the clipping upper bound:

$$r^{\text{tr}}_{i,t}(\theta) = \text{clip}\left(r_{i,t}(\theta), 1-\epsilon, (1-\epsilon)^{-1}+k\right).$$

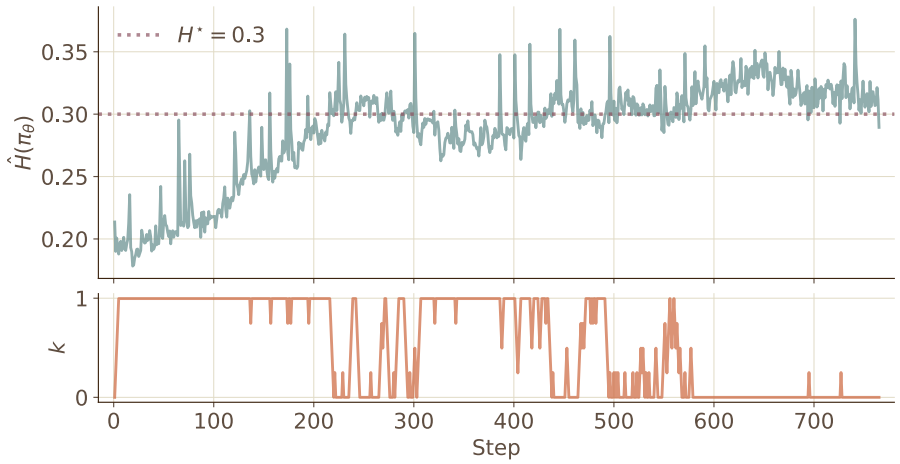

Figure 13. Adaptive entropy control during training. Top: observed entropy $\hat{H}(\pi_\theta)$ over 800 steps of an RL climb with target entropy $H^\star = 0.3$. Bottom: corresponding values of k with $k_{\max} = 1.0$: it is decreased when the observed entropy exceeds the target and increased otherwise. Adjusting k provides an effective mechanism for regulating policy entropy.

We found that the upper bound requires careful tuning to avoid exploding entropy when it is too large, or entropy collapse when it is too low (see also Cui et al., 2025; Xi et al., 2025).

To address this, we *dynamically* adjust k based on the current policy's entropy using a simple integral controller. At each training step, we estimate the target policy's per-token entropy via an importance-weighted estimator:

$$\hat{H}(\pi_\theta) = \frac{1}{|\mathcal{T}|} \sum_{(i,t)\in\mathcal{T}} -\log \pi_\theta(y_{i,t} \mid q,\, y_{i,<t}) \cdot r_{i,t}(\theta)\,, \tag{7}$$

where $\mathcal{T}$ denotes the set of all (response index, token index) pairs in the current training batch. Given a target entropy $H^\star$, the controller updates k with a step size $\delta \in \mathbb{R}^+$ after each step:

$$k \;\leftarrow\; \text{clip}\left(k + \delta \cdot \text{sign}\left(H^\star - \hat{H}(\pi_\theta)\right), 0, k_{\max}\right). \tag{8}$$

Intuitively, when entropy is too low, increasing k widens the upper clipping bound, allowing the policy to increase the probability of alternative tokens more aggressively. When entropy is sufficiently high, k is decreased to tighten the trust region. As illustrated in Fig. 13, this mechanism acts as an automatic entropy regularizer without requiring an explicit entropy bonus term in the loss, which we found to underperform the adaptive entropy mechanism.

We initialize $k = 0$, so that the clipping bounds $1 - \epsilon$ and $(1 - \epsilon)^{-1}$ are multiplicative inverses, making the initial clipping interval symmetric in log-ratio space. We then update k online according to the entropy controller above.

Outer ratio clip. The GRPO objective deliberately leaves two cases *unclipped*: (a) advantage is negative and the new policy assigns higher probability than the old (i.e. $A_i < 0$ and $r_{i,t} > 1$), and (b) advantage is positive and the new policy assigns lower probability ($A_i > 0$ and $r_{i,t} < 1$). The original motivation (Schulman et al., 2017) is to leave the policy unbounded when it corrects itself in the right direction, only bounding moves that exploit the advantage estimate.

In practice, however, we found that these unclipped branches sometimes led to catastrophic gradient-norm spikes. We address this by adding a hard *outer clip* that is applied to all branches:

$$r_{i,t}^{\text{out}}(\theta) = \text{clip}\big(r_{i,t}(\theta),\ r_{\min},\ r_{\max}\big), \tag{9}$$

where $r_{\max}$ is set to a large value and $r_{\min}$ can be left unconstrained; this is similar in spirit to the dual-clip PPO objective proposed by Ye et al. (2020).

This two-level strategy discards cases where there is an extreme discrepancy between old and new probabilities while preserving the standard trust-region behavior for ratios in the normal range. Empirically, we found this to lead to fewer gradient-norm spikes and more stable climbing.

3.1.2 Reward Design

While our task-specific component differs across our domain-specific RL climbs, we use the same reward decomposition throughout:

$$R(q, y_i) = R_{\text{task}}(q, y_i) + w_{\text{lang}} \cdot R_{\text{lang}}(y_i) - w_{\text{len}} \cdot R_{\text{len}}(y_i). \tag{10}$$

Here, R_{task} denotes the task-specific reward, R_{lang} is a language-consistency reward, and R_{len} is a length penalty. The coefficients w_{lang} and w_{len} are scalar hyperparameters.

Language consistency reward. As context lengths increase during RL, we observe that models begin producing foreign-language tokens within their CoTs. These mixed-language CoTs correlate with spikes in log-probability divergence between the training and inference policies, ultimately destabilizing training. To address this, we use a language consistency reward $R_{\text{lang}}(y_i)$ similar to Guo et al. (2025). Since English dominates our training distribution, we define language consistency with respect to English:

$$R_{\text{lang}}(y_i) = \max\Big(1 - \alpha \cdot n_{\text{non-english}}(y_i),\ 0\Big), \tag{11}$$

where $n_{\text{non-english}}(y_i)$ is the number of non-English words in the CoT of response y_i and α is a per-word penalty. In practice, we found that top-p sampling is similarly effective in preventing single, low-probability, foreign language tokens.

Length penalty. Following Xiang et al. (2025), we define our length penalty as:

$$R_{\text{len}}(y_i) = \rho_q \cdot \frac{|y_i|}{\ell_{\max}} \tag{12}$$

where ρ_q is the pass rate of problem q and $\ell_{\max}$ is the maximum rollout length. The penalty depends on both response length and problem difficulty. Hard problems with low pass rates receive weaker penalties, allowing the model to explore longer reasoning traces. Easy problems receive stronger penalties, encouraging concise and cost-efficient reasoning by removing redundant loops and hedging behavior.

3.1.3 Sampling Strategy

We employ several sampling strategies to improve both the efficiency and stability of our RL climb. These operate at two levels: selecting which problems to train on and controlling how rollouts are generated for those problems.

```
A conversation between User and Assistant. The user asks a question, and the
assistant solves it. The assistant first thinks about the reasoning process in the
mind and then provides the user with the answer. The reasoning process and answer are
enclosed within <think> </think> and <answer> </answer> tags, respectively, i.e.,
<think> reasoning process here </think>
<answer> answer here </answer>.

User: {QUERY}
Assistant: <think>
```

Figure 14. Prompt template used for reasoning prior to the first round of self-distillation with `{QUERY}` being a placeholder for the actual user query (from Guo et al. (2025)).

Problem sampling. For each problem q in the training set, we generate a group of rollouts from the current inference model. To reduce inference cost, we use an *early exit* strategy: we first sample $G_{\text{early}} < G$ responses and compute their empirical pass rate, i.e. the fraction of responses with positive reward. If the early pass rate lies in an acceptable range $[\rho_{\min}^{\text{early}}, \rho_{\max}^{\text{early}}]$, we sample the full G responses; otherwise the problem is discarded.

After all G responses are generated, we apply a second pass-rate filter $[\rho_{\min}, \rho_{\max}]$ to the full group. Only problems whose full pass rate falls in this range are used for training. The main motivation for this second filter is to remove very low-variance groups: if almost all responses are correct or incorrect, the group provides little useful relative learning signal.

Rollout sampling. We employ top-p sampling (Holtzman et al., 2019) using π_{old} to sample rollouts $y_{1:G}$. We found that continuing to backpropagate through logits corresponding to tokens outside the sampled nucleus can lead to catastrophic off-policy mismatch, causing divergence within a few training steps. To prevent this, we exclude these tokens during training by reusing the top-p truncation mask from rollout sampling to set the logits of all excluded tokens to $-\infty$ prior to softmax computation, following DeepSeek-AI et al. (2025b). We found that top-p masking substantially reduces policy divergence during RL training, at the cost of additional overhead for mask storage and replay.

To improve training efficiency during the early stages of a climb, we initially cap the maximum rollout length at 8k tokens. As training progresses, we increase this limit in powers of two until reaching the final output length of 128k tokens. We found this length extension curriculum to significantly reduce inference cost during the low-performance regime, where long reasoning traces are rarely required, while still allowing the model to gradually adapt to longer contexts as its capabilities improve.

3.1.4 Self-Distillation

Achieving strong performance from a mid-trained checkpoint requires training over a large number of RL steps. We utilize *self-distillation* (Huang et al., 2022; Zelikman et al., 2022; Singh et al., 2023; Zhao et al., 2026) to make such long-running climbs more practical. For self-distillation, we collect rollouts generated during RL and perform SFT on a midtrained checkpoint using these rollouts. The resulting model serves as the starting point for continuing our RL climb, while preserving capabilities discovered during previous RL stages. We use self-distillation for several purposes:

- For our initial climb, we use domain- or task-specific prompts to elicit the target behavior; for example, Fig. 14 shows the initial raw text prompt used for our STEM climb. Self-distillation allows us to move from prompting to using our native chat format by simply updating the formatting of the SFT data.
- Another primary use case is to recover from occasional run failures. Early in the development of our RL

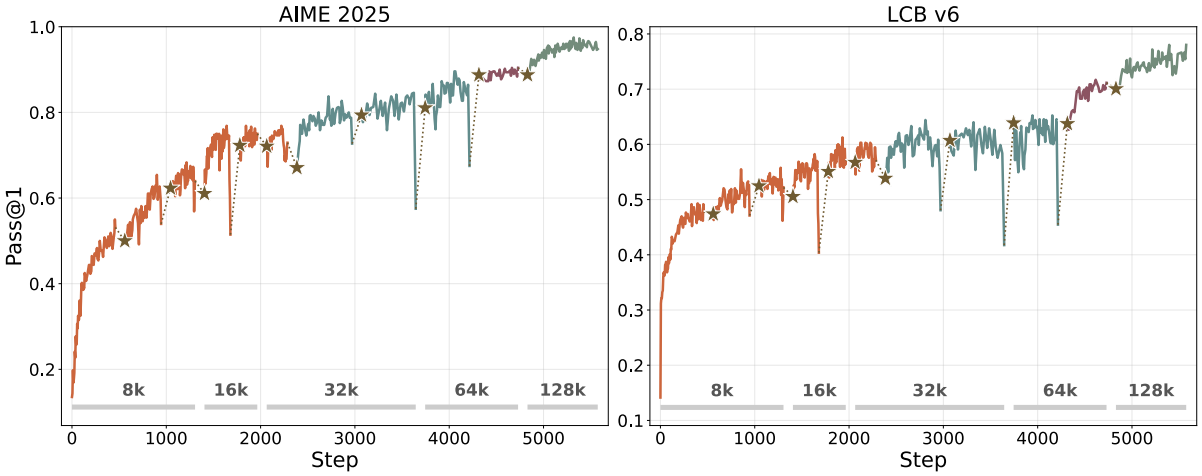

Figure 15. Performance on AIME 2025 (left) and a hard subset of LiveCodeBench v6 (right) during our STEM climb. Self-distillation is indicated through ⋆ markers; different pre- and mid-trained model versions are shown in different colors. Maximum output length throughout training is indicated at the bottom. Self-distillation is an effective way of resetting numerics after a collapse (visible through sudden drops in performance) and changing the base policy during our run. As we made infrastructure and algorithmic improvements throughout our climb, self-distillation because of run collapses became less frequent.

stack – and prior to the introduction of stability improvements described in Sec. 3.6.4 – small numerical discrepancies between inference and training would sometimes accumulate during training and cause a climb to diverge. In such cases, self-distillation provides a simple and effective way to carryover progress from one RL climb to the next. The alternative of resuming from a checkpoint before the collapse is often not viable, as some instabilities are already embedded in the model parameters many steps before the actual collapse occurs.

- As new pre- and mid-trained checkpoints become available, self-distillation allows us to carry forward the progress of previous climbs to the next generation of models.
- During the self-distillation process, filters can be applied to reject any samples that demonstrate undesired forms of reward hacking.

Fig. 15 illustrates how self-distillation was used in our STEM climb both to recover from numeric instabilities and to update the base policy.

Key findings and best practices. We conducted extensive ablations to understand how to best perform self-distillation in practice. Our main findings are:

- $\mathcal{O}(1\text{M})$ reasoning traces are sufficient to match teacher performance while retaining the stability benefits of SFT. Using substantially larger datasets provides diminishing returns and risks over-constraining the policy, narrowing its output distribution and leaving little room for exploration once RL resumes (Zheng et al., 2024; Gao et al., 2025).
- Training on traces including those that lead to incorrect final answers performs similarly to training on successful traces only (Gandhi et al., 2025). Since our RL runs typically produce far more than $\mathcal{O}(1\text{M})$ successful traces, we ultimately restricted training to successful traces.

- Using traces from later stages of the climb is important. Including traces from very early checkpoints leads to noticeable performance degradation and requires many subsequent RL steps to recover. Generating traces from only the final RL checkpoint results in weaker performance after resuming RL. A possible explanation is that traces collected across a range of strong checkpoints provide greater diversity than traces generated from a single final policy, leading to better exploration during RL. Reusing traces already generated during RL also avoids the additional compute and time required for resampling.
- For a fixed token budget, increasing prompt diversity is more valuable than increasing the number of traces per prompt. We also found simple random sampling to outperform several biased selection strategies, including shortest-trace sampling and heuristic filtering approaches similar to those explored by Yang et al. (2025).
- During self-distillation, the model can forget long-context behavior learned from mid-training, especially when reasoning traces are collected from earlier RL runs trained with shorter maximum lengths. To mitigate this, we mix mid-training data with reasoning traces during self-distillation before length extensions.

3.1.5 Hyperparameters

Below, we report the hyperparameters typically used for both our main RL climbs across the three domain-specific models and our self-distillation runs.

RL climb. We train using AdamW with $\beta_1 = \beta_2 = 0.95$, $\epsilon = 10^{-15}$, and no weight decay. We use a constant learning rate of $\eta = 10^{-6}$ with no warmup or decay. The global batch size is 7040 after packing, with the maximum number of total unpacked sequences capped at 12000. Because our RL stack is fully asynchronous, as we increase the generation length the latency of the rollouts increases while the training step times do not increase proportionally due to many short generations. This means the longer rollouts, which are typically the hardest problems, increase in off-policiness. To counter this, we opted to lowering the learning rate at higher lengths to 9×10^{-7} to reduce off-policiness and increase stability. The maximum generation length is 128k though we trained first at 8k, 16k, 32k and 64k.

For GRPO with adaptive entropy control, we set $\epsilon = 0.6$, $k_{\max} = 2.5$, step size $\delta = 0.25$, and target entropy $H^\star = 0.3$. For outer clipping, we set $r_{\max} = 50$ and leave $r_{\min} = 0$ unconstrained.

For our language consistency reward, we set $w_{\text{lang}} = 0.5$ and the per-word penalty to $\alpha = 0.005$. For the length penalty, we fix $w_{\text{len}} = 0.25$ up to the 64k length extension stage. Since the penalty is normalized by $\ell_{\max}$, the effective penalty naturally weakens as the maximum length increases, allowing the model to explore longer reasoning traces at larger token budgets. We remove the length penalty ($w_{\text{len}} = 0$) at the 128k extension stage.

For problem sampling, we set $G = 128$ total rollouts with $G_{\text{early}} = 16$ and $[\rho_{\min}^{\text{early}}, \rho_{\max}^{\text{early}}] = [0.05, 0.8]$. Pass-rate filtering uses $[\rho_{\min}, \rho_{\max}] = [0.1, 0.8]$. For top-$p$ sampling, we use $p = 0.97$. Larger nucleus values improve exploration but increase the infrastructure overhead of transferring sampling masks, and we found $p = 0.97$ to provide a good balance between exploration and training efficiency.

We perform 5 gradient steps between inference model updates and discard any rollout whose generating policy is more than 8 inference updates stale (i.e., 40 gradient steps behind). This slightly relaxes on-policy freshness in exchange for substantially higher throughput. We use dropless MoE training and set the global MoE load balancing coefficient to $1 \cdot 10^{-5}$.

Self-distillation. For self-distillation SFT, we use packed sequences with a global batch size of $2,048$ and a sequence length of 128k tokens. Optimization uses AdamW with weight decay 0.001 and a cosine learning rate schedule. The maximum learning rate is set to $1.7 \cdot 10^{-5}$, the minimum learning rate to $5.2 \cdot 10^{-6}$, and the learning rate warmup ratio to 2%.

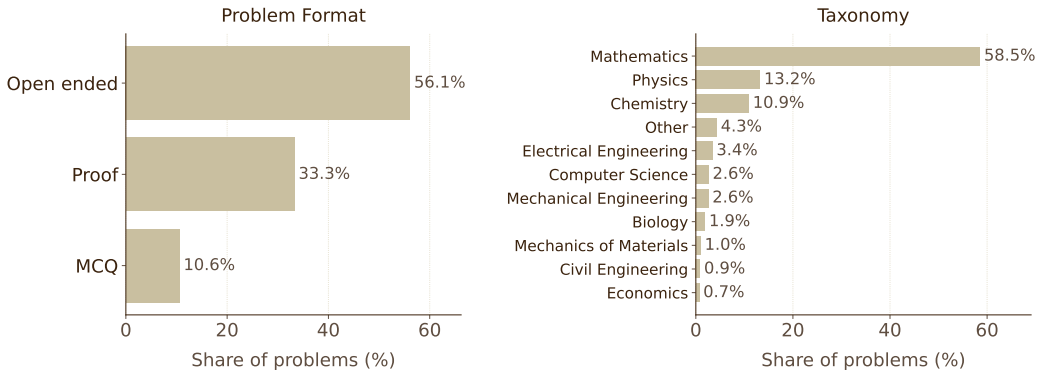

Figure 16. Distribution of our STEM Mix dataset by original problem format (left) and subject taxonomy (right), where *Other* aggregates topics each below 0.5% of the dataset. Multiple-choice and proof problems are converted to open-ended form during ingestion (and dropped where conversion is infeasible); we retain a small fraction of multiple-choice problems so the model remains familiar with the format.

We carefully tune two hyperparameters during self-distillation: dropout (Srivastava et al., 2014) and the MoE load balancing loss coefficient. We use a relatively high dropout rate of 0.15, which increases entropy and helps prevent model collapse, improving performance in subsequent RL stages. As the data distribution in domain-specific RL is substantially narrower than the mixtures used during pre- and mid-training, expert selection during RL can become highly imbalanced. However, using a large load balancing coefficient during RL also harms stable performance improvement. To address this, we use a comparatively large load balancing coefficient $1 \cdot 10^{-2}$ during self-distillation and a much smaller coefficient $1 \cdot 10^{-5}$ during RL. Since the self-distillation contexts are generated from the RL run itself and therefore follow a similar distribution, the expert balancing effect induced during self-distillation is preserved during RL.

3.2 STEM Climb

The STEM climb is the longest of our three domain-specific RL training runs and is designed to strengthen the model's core reasoning abilities in single-turn problem-solving settings. It covers a broad set of STEM domains, including mathematics, physics, chemistry, and competitive programming. We also witness the evolving model behavior reflected in rollouts during the climbing, described in more details in Appendix C.1. Since data quality is central to the success of this climb, we focus on our training data and the pipeline we have built to construct, verify, and filter this data.

Our entire STEM climb operates on pairs of *verifiable* data: To produce the task-specific reward $R_{\text{task}}(q, y_i)$, we extract the model's final answer from y_i and either compare it to a ground truth using a formal verifier such as SymPy (Meurer et al., 2017), an AI judge, or, in the case of competitive coding, we run the model-produced code snippet against a suite of problem-specific test cases. Thus, every RL data instance comprises either a pair (q, a) of a query and a ground truth answer, or a pair $(q, \{t_1, \ldots, t_n\})$ of a query and n test cases. In designing our pipeline to obtain such pairs, we take three main criteria into account: high quality, appropriate difficulty, and topical diversity.

Using our STEM data pipeline, we processed millions of documents, producing the *STEM Mix* dataset with more than 5M samples that is used for our STEM climb; the most challenging portion of this mix contains more than 550k (q, a) pairs. Some properties of this dataset are illustrated in Fig. 16.

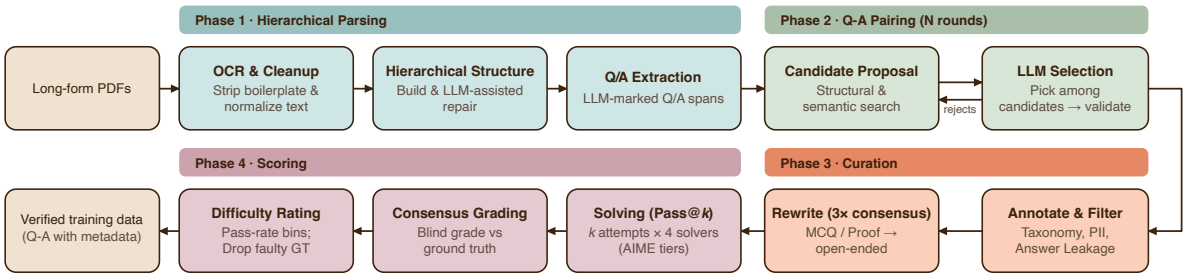

Figure 17. Pipeline for extracting (q, a) pairs from textbooks and academic PDFs for STEM Mix.

3.2.1 Data Pipeline

Our STEM data pipeline transforms heterogeneous raw sources— textbooks, academic PDFs, forum discussions, competition archives, and problems obtained from various vendors—into a dataset of (q, a) pairs. The pipeline is built around a composeable, stage-based architecture: each processing step is implemented as an independent asynchronous stage. Source-specific instantiations of our pipeline select and compose subsets of these stages as needed.

We describe several pipeline stages below, grouped into four phases following the example shown in Fig. 17 for extracting (q, a) pairs from long-form textbooks and academic PDFs. Stages that are sensitive to noise or hallucinations are run multiple times with consensus voting.

Hierarchical parsing. This phase comprises the stages that convert raw documents into initial (q, a) pairs. Depending on the document type, OCR is performed using a vision-language model or OCR services. Documents are chunked, pages without STEM content are discarded, and boilerplate is removed. A separate stage builds a hierarchical representation using structural cues in the OCR output, and resolves broken cross-references, misaligned numbering, and split-across-page artifacts. Finally, in the Question and Answer (QA) extraction stage, an LLM marks question and answer spans within the cleaned document structure, producing candidate (q, a) pairs.

QA pairing. For sources where questions and answers appear in separate locations (*e.g.,* exercises at the end of a chapter, answer keys in appendices), a multi-round pairing stage matches each question to its corresponding answer. A question is first matched against candidate answers using structural signals and semantic similarity; among the retrieved candidates, an LLM then selects the best-matching answer and validates the pairing.

Curation. Stages in the curation phase annotate, filter, and rewrite (q, a) pairs to ensure quality and format consistency. Several of these stages are LLM-based classifications:

- **Verifiability:** Items are classified as verifiable or non-verifiable; non-verifiable items (*e.g.,* open-ended forum discussions with no checkable answer) are dropped.
- **Question type:** We distinguish among question types, including open-ended questions, multiple-choice questions, and proofs.
- **Taxonomy:** A hierarchical topic classifier assigns fine-grained STEM topics, spanning mathematics, physics, chemistry, biology, computer science, and engineering.
- **PII detection:** A dedicated classifier flags personally identifiable information; flagged items are dropped.

- **Answer leakage:** (q, a) pairs where a is trivially contained in q are detected and dropped.

Two further curation stages transform the problems to be more useful for RL climbing. *Conversion* rewrites multiple-choice and proof-style questions into open-ended form; multiple-choice questions can often be solved by guessing, providing an unreliable reward signal, and proofs are hard to verify directly without relying on a stronger AI judge. This stage is run three times followed by a consensus stage; an item that does not reach consensus is treated as not reliably convertible and is dropped. *Cleanup* removes extraneous, non-mathematical text and external references not relevant to the question.

Scoring. This final phase quantifies problem difficulty and filters items whose ground-truth answers are likely incorrect. In the solving stage, each problem is solved k times by each of four model tiers, whose AIME 2025 performance serves as a proxy for ability. Pass rates are used to group items into difficulty brackets.

A blind-grading stage guards against faulty ground truths. For items the strongest tier solves at a low pass rate, we present that model's consensus answer and the ground truth to a judge, in randomized order. If it prefers the consensus answer, we drop the item as having a suspect ground truth; if it prefers the ground truth, we keep it as a genuinely hard problem.

3.2.2 Competitive Coding Data

We use a dedicated pipeline for competitive coding data that deviates from our main STEM pipeline; this is because for competitive coding problems, we require a set of test cases $t_1, \ldots, t_n$ for each prompt. Prompts with comprehensive test cases are unlikely to be found in unstructured sources such as PDF documents. We therefore rely on targeted sources and vendor-acquired data, which removes the need for many of the extraction and filtering steps used in our STEM data pipeline.

For each problem, we obtain reference solutions and verify that they pass all associated test cases. In total, we collect 160k problems from multiple sources, covering a diverse range of topics, including divide-and-conquer, dynamic programming, graph and tree algorithms, and search algorithms. In addition to test cases, each problem includes runtime and memory constraints. Our final dataset supports 17 programming languages, including Python, C++, C#, Java, JavaScript, Rust, and TypeScript.

3.2.3 Deduplication and Decontamination

Both our STEM Mix dataset and our competitive coding datasets are self-deduplicated and deduplicated against benchmarks we report in Sec. G, as well as in-house Olympiad and graduate-level STEM evaluations that we track internally. We deduplicate using a three-stage pipeline:

1. **Exact deduplication.** As a first step, we identify exact duplicates using SHA-256 question hashes.
2. **Lexical fuzzy deduplication.** We use character-level n-gram shingling followed by MinHash locality-sensitive hashing to identify near-duplicate questions. Pairs with Jaccard similarity above a given threshold are marked as duplicates.
3. **Vector deduplication.** Finally, questions are embeddeded using a lightweight embedding model. Pairs with cosine similarity above a given threshold are marked as duplicates.

We carefully tuned the hyperparameters at each stage to retain as much data as possible while ensuring strict exclusion of any benchmark leakage.

3.3 Agentic Climb

The agentic climb trains MAI-Thinking-1 to solve tasks that require interaction with an external environment rather than a single-pass text response. In this setting, the model must decompose a user request,

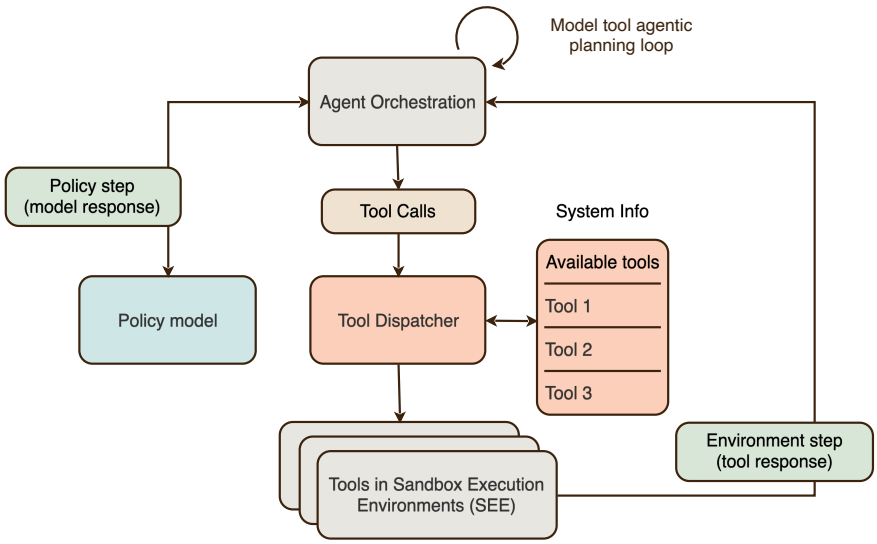

Figure 18. Agentic loop and multi-step orchestration in RL training.

choose tools or code actions, observe the results, and adapt its plan across multiple steps and turns if needed. The training signal combines verifiable rewards, such as tests passing in a software environment or a database reaching a target state, with AI-feedback rewards for aspects of behavior that are harder to specify exactly, such as task interpretation, helpfulness, and trajectory quality.

We focus on two agentic domains: (i) software engineering (SWE) (Sec. 3.3.1), which involves executable software-engineering environments built from real repositories; and (ii) general tool use (Sec. 3.3.2), which involves calling structured tools in multi-step tasks.

In practice, we jointly climb on a mixture of both agentic- and reasoning-focused STEM tasks, including our competitive coding mixture. We found that inclusion of STEM tasks helps to stabilize the RL climb and shows positive transfer to multi-step software engineering and tool-calling performance. Conversely, the agentic tasks transferred neither positively nor negatively to STEM-related single-pass performance. Similar to the STEM climbing, we observe evolving model behaviors throughout climbing, with examples in Appendix C.2. The agentic setting differs from the STEM setting in its use of multi-step rollouts and containerized environments, as described in the following sections.

Multi-step RL framework. Our agentic multi-step RL uses the same core objective as the single-step reasoning RL recipe in Sec. 3.1, but extends rollouts from a single sampled response to a trajectory of policy steps and environment steps (observations). Each RL environment consists of a task specification, a Sandbox Execution Environment (SEE) session for executing tools, and a set of verifiable or judged rewards for evaluating task completion while tracking environment state. At each policy step, the model can emit tool calls or produce a final answer. Tool calls are executed inside the SEE session, and their outputs are appended to the context before the next policy step. The complete trajectory is then graded for correctness and, optionally, given AI feedback, after which credit assignment is applied uniformly across all tokens from all policy steps.

The orchestration harness is a ReAct-style loop (Yao et al., 2023): it parses the model's reasoning and actions, dispatches tool calls to SEE, appends returned observations to the context, and returns control to the policy for the next action. All the previous steps' tokens are preserved in context and they are strict prefixes for the next steps. The same loop supports both software-engineering and general tool-use workloads. In SWE, actions read and edit files, run shell commands, and inspect repository state inside a problem-specific container image. In general tool use, actions call structured tools backed by mutable task state, such as seeded

databases, with the tool outputs returned as observations for the next model action. The loop terminates when the model emits no tool calls, or when the rollout exceeds the step, context-length, or time budget; the rollout and its corresponding SEE are then sent to graders. They combine format checks, rule-based checks, executable tests, verifiable state comparisons, and AI judges. Fig. 18 sketches a summary of our agentic RL framework. The RL environments come from two major domains: software engineering Sec. 3.3.1 and general tool-use Sec. 3.3.2.

Sandbox execution environment. Our Sandbox Execution Environment (SEE) provides on-demand, isolated container environments at scale. SEE is used during RL to provide highly parallel, low latency containers for model training environments. It provisions a fresh container for each agentic task and destroys it upon task completion. This ensures reproducibility, enables safe exploration, and prevents state leakage between tasks.

Containers are network-isolated by default, ensuring that training episodes are deterministic and free from external side effects such as rate limits or transient failures. When an environment does require network access (for example, to install packages), traffic is mediated through caching proxies and domain allowlists, balancing the needs for environment fidelity, training reproducibility, and safety.

3.3.1 Software Engineering

Training a frontier model to perform SWE tasks requires the model to interact with real codebases inside realistic development environments. A large number of high-quality coding environments is essential for training a strong coding model. To achieve this, we source naturally-harvested data from contributions to real codebases. To curate our repository of RL environments, we rely on the following primitives: a large dataset of code repositories (e.g., public GitHub code), a pipeline to create executable software engineering problems from real-world open-source developer pull requests, and our sandbox container service SEE for execution.

SWE RL environments, tasks, and tools. Each SWE RL problem is packaged as a self-contained container image that includes a repository checked out at a specific commit with all dependencies pre-installed, a problem statement, and unit tests for grading. Because the image captures all dependencies of the repository, environments are deterministic and immediately ready for use with no setup required at the start of the rollout. During the rollout, the model interacts with the container through tool calls: reading and editing files, running shell commands, and navigating the repository. When the model signals completion or a turn limit is reached, a grader executes the tests inside the same container and compares the results to expected outcomes, producing a verifiable reward signal.

Two common tools are available for the model to use: *Bash,* a tool that executes bash commands and returns standard output and error streams, and *String replace editor:* a tool for models to edit files via exact string replacements, instead of having to use non-ergonomic bash tools for this purpose (Anthropic, 2025). Full tool descriptions and parameter schema are provided in Appendix D.

In-house scalable SWE environment building. Existing work (Pan et al., 2024; Chen et al., 2026b; Fu et al., 2026) has explored transforming raw code repositories from the Internet into RL environments. We use public GitHub repository issues and pull requests (PRs) as the source of our organic data and develop a scalable pipeline for ingesting real-world verifiable SWE problems, inspired by SWE-ReBench (Badertdinov et al., 2026a,b). To construct high-quality RL environments, we ensure that problem statements provide sufficient information for agents to correctly implement solutions without overspecifying the task. We additionally ensure that graders provide sufficient coverage while avoiding overly specific evaluation criteria. The pipeline consists of the following stages, with additional infrastructure details provided in Appendix F:

- **Public GitHub PR and issue collection.** The pipeline starts with 102 million public GitHub PRs. First, we filter PRs to obtain a suitable subset for environment building. To be included, a PR must be merged into the main branch of its repository and modify fewer than 15 files. Each PR must also contain both code and test changes so that the grader can utilize the test changes as hidden tests to evaluate model's ability to fix the issues. Test and code changes are split based on the file contents in the patch. We further filter PRs based on issue linkage, retaining PRs associated with GitHub, Jira, Bugzilla, YouTrack, Phabricator, Launchpad or Linear issues. After this stage, we obtain approximately 4.87 million PRs with linked issues.
- **Automatic agentic environment building.** We convert the selected PRs into trainable environments using an LLM agent that reads the repository state and creates Docker files to build executable container images. We validate each image by executing the test suite and discarding builds with dependency or environment errors.
- **Reference grading signal extraction.** For each candidate problem, we derive reference test outcomes by executing the repository test suite in two stages against the base commit: first with only the test diff applied (pre-fix),and then with both the test and code diffs applied (post-fix). Tests that transition from fail-to-pass (**F2P**) form the issue-resolution signal, requiring the model patch to flip these tests in order to receive credit for solving the problem. Tests that remain passing across both stages, pass-to-pass (**P2P**) form the regression signal, ensuring that generated patches do not break existing functionality. Problems with no surviving **F2P** tests are discarded.
- **Environment and grader verification.** After obtaining the executable environment, test and code changes, test execution scripts, and expected pre-fix and post-fix outcomes, we re-validate the environments within the same SEE training infrastructure used during RL training. This serves as a final sanity check against potential discrepancies between the environment-building sandbox and the training sandbox. We verify that an empty code patch fails the grader while the golden solution patch succeeds, each across multiple trials. Although the upstream patch is known to be correct, re-validation may still fail due to cluster differences such as network egress policy, CPU/memory limits, execution timeouts. We additionally filter environments that exhibit non-deterministic test behavior across repeated executions to reduce reward noise.
- **Quality filtering and SWE environment rewriting.** Correct execution and airtight tests alone are insufficient, as many environments still contain low-quality or underspecified problem statements. For example, we frequently observe short and vague bug descriptions, incomplete specifications, or statements such as "fix things" where the expected behavior is only implicitly encoded in the hidden tests. If there are large gaps between the problem statement and what the final tests run by the grader are expecting, then the model has to guess what the tests require.

  To improve environment quality, we deploy an agent within the same environment to inspect the problem statement, repository, and tests, and score the task on specification clarity, test quality, leakage risk, and feasibility. For low-quality environments, the agent rewrites the problem statement to better align with the test requirements while avoiding unnecessary information leakage or overspecification.

Out of the initial 4.87 million candidate problems, 2.08 million (42.8%) successfully passed automatic agentic environment building, 745,452 (15.3%) passed reference grading signal extraction, and 265,617 (5.5%) survived environment and grader verification across 94,044 unique repositories.

Synthetic data. While many environments generated by our environment building pipeline did not pass validation, a large fraction were still valid executable environments. In most cases, the failures were due to low-quality problem statements or insufficient test coverage rather than invalid execution environments. To reuse these executable environments, we adopt methods inspired by BugPilot (Sonwane et al., 2025),

SWE-Smith (Yang et al., 2026) and SWE-Mirror (Wang et al., 2025a) to generate new synthetic problems and corresponding tests from PRs associated with environments that were successfully built but failed our quality checks.

Preventing reward hacking during RL training. Even though SWE environments are graded with verifiable, executable test cases, they are still susceptible to reward hacking. We use an LLM monitor to review the rollouts and do manual human reviews of flagged examples. This identifies three main types of reward hacking:

- **Internet search.** Since our RL environments are sourced from public repositories on GitHub, the PR (and therefore the golden solution) can often be searched for and retrieved. We employ network access control to our sandbox execution environments to prevent this by either disabling Internet, when the problems are self-contained, or otherwise only allowing for the bare minimum required by the problem.
- **Local git history search.** Git is an integral part of software repositories. Sometimes the agent will search all the git commits and logs to try to find the solution commit hidden in the local git database.[7] Removing all git commits is also not an option, as git is a valuable skill and can provide legitimate information necessary for solving the problem. We completely sanitize the environment by scrubbing commits, references and branches that come after the base commit of the problem, to create a "time-traveled" version of the repository back to the original state.
- **Tampering with tests.** Following the evaluation protocol used in SWE-Bench (Jimenez et al., 2024), we reset all test files modified by the agent before grading to prevent agents from tampering with the test cases to make them pass the grading. We additionally hide test changes from the agent during inference and only apply them at grading time to further reduce opportunities for cheating. However, there are still vectors for cheating, *e.g.,* through monkey-patching the testing framework or modifying equivalence behavior, that are not easily prevented by simply resetting test files or introducing hidden tests. We use LLMs to monitor agent behaviors and continuously strengthen test file detection, test file reset, and other anti-tampering heuristics.

3.3.2 General Tool Use

In contrast to SWE RL environments, general tool-use RL environments exhibit substantially greater diversity in both available tools and application domains. They cover a broad range of generic tool-calling tasks (*e.g.,* inventory management, scheduling platforms, report creation, customer support, and other enterprise-oriented scenarios), emphasizing reliable and stateful interactions with external systems.

Tool-use RL environments. Each tool-use RL problem is instantiated as an interactive, stateful environment backed by mocked backends that simulate realistic API or MCP (Model Context Protocol, 2026) behavior. Concretely, each problem consists of a query, a set of available tools with schema[8], an initial environment state, and a grader. Unlike SWE settings, which typically involve only a small number of tools, tool-use environments emulate rich real-world service interactions with substantially larger tool sets, often exceeding 50 tools within a single environment. This setting trains the model to select appropriate tools efficiently and improves generalization across diverse tool-use scenarios. We train on both human-curated environments and synthetic environments generated by our tool-calling task generation framework, described below.

[7] For example, SWE-Bench was vulnerable to this problem: https://github.com/SWE-bench/SWE-bench/issues/465. This has since then been fixed.

[8] We use OpenAI's function-calling schema: https://developers.openai.com/api/docs/guides/function-calling.

Synthetic environment. To increase the scale and diversity of our training data, we synthetically generate self-contained closed-world environments with seeded databases, tool definitions, and verifiable tasks. Our pipeline requires only plain-English descriptions of the desired environments to generate complete tool-use environments end-to-end. This approach follows prior work on automatically generating tool-use environments (Sullivan et al., 2025; Zhang et al., 2026; Wang et al., 2026; Mika, 2026), particularly the FunReason-MT pipeline (Xu et al., 2025b; Hao et al., 2025). Overall, we aim to replicate common enterprise and consumer tool-use scenarios, such as travel booking and inventory management.

Our pipeline consists of three main phases: (*i*) *environment bootstrapping*, which generates tool descriptions, implements functions, and seeds databases with relevant entities; (*ii*) *task creation*, which samples likely tool-call trajectories, creates entities relevant to the interaction chain, and formulates user requests; and (*iii*) *verification and refinement*, which executes the actions generated in the previous phase, and removes tasks that are overly similar. Each phase involves multiple LLM calls for tasks such as entity generation and function implementation. We additionally employ iterative critique-and-refinement loops to improve the outputs at each phase (Dhuliawala et al., 2023; Madaan et al., 2023).

We generate environment-specific personas (Ge et al., 2024) to further diversify tasks. To mitigate over-eager tool-calling behavior, we additionally augment our data with tasks that include tool descriptions but do not require tool usage. Overall, our pipeline synthesizes more than 150 environments and 130,000 tasks.

Reward design. During training, we use a mixture of environment-specific and cross-environment graders. Environment-specific graders assign rewards based on the final environment state, tool usage patterns, and final answers. For synthetic environments, we use an LLM judge to decompose tasks into multiple sub-tasks and grade each independently. We additionally employ cross-environment graders that encourage efficient tool usage, including parallel tool calls when possible, avoidance of duplicate calls, and correct tool invocation with valid parameter types and arguments.

3.4 Helpfulness and Safety Climb

The helpfulness and safety RL climb optimizes MAI-Thinking-1's general helpfulness as judged by human preference, instruction following, steerability, safety, honesty, and style. (Ouyang et al., 2022; Radford et al., 2019; Li et al., 2024a) This climb differs from the other climbs in that it focuses on tasks where performance is not as objectively defined and machine-verifiable.

We organize this section by first describing all the reward signals (Sec. 3.4.1) used during RL climbing, followed by each domain's specific data recipe and reward designs.

3.4.1 Rewards

Compared to the other climbs, the helpfulness and safety climb combines a more diverse set of reward types to guide subjective aspects of model behavior. We use a combination of a reward model trained on human preference data (Lambert et al., 2025), AI judge feedback (typically rubric-guided) (Gu et al., 2024), and additional verifiable rewards (Peng et al., 2025; Pyatkin et al., 2026) to form an aggregate reward signal. (Lambert et al., 2024; Wang et al., 2024c)

Reward model. Our reward model is based on a post-trained version of MAI-Base-1, which we fine-tune to predict human preferences expressed as text tokens. It is trained exclusively on human preference data collected with human annotators from several vendors. We employ a robust reward hacking mitigation pipeline as described in Liu et al. (2025c) to counter relevant biases in our training data.

Training. For a context c and k-way side-by-side with responses y_1, ..., y_k, and corresponding scores $s_1, \ldots, s_k \in [1; 5]$, the input to the reward model is

$$c \texttt{<|im\_sep|>} y_1 \texttt{<|im\_sep|>} y_2 \texttt{<|im\_sep|>} \ldots \texttt{<|im\_sep|>} y_k \texttt{<|im\_sep|>}$$

where the training objective is the sequence $s_1 \dots s_k$, trained via SFT.

Inference. The inclusion of multiple candidate responses into a single reward model context allows for better calibration of pointwise quality scores across responses. However, due to the autoregressive nature of the reward model, this also increases noise for scores $y_{i,i>1}$. To counter this, we apply the reward model *cyclically*: For a given context c and k responses $y_1, \dots, y_k$, we prompt the reward model k times on the response permutations $(y_1, \dots, y_k), (y_2, \dots, y_k, y_1), \dots, (y_k, y_1, \dots, y_{k-1})$. For each of these k inference calls, we only decode the first token and look at the full probability distribution for this token, which for the i-th call corresponds to the reward score s_i of the i-th candidate response. The reward signal $R_{\text{RM}}(c, y_i)$ is then set to the probability of y_i being scored as the highest quality ($s_i = 5$). We found this to provide a more stable climbing signal than alternative formulations we considered.

Evaluation. We evaluate our reward models using a combination of actual training rollouts rated by our annotators and a validation split of our reward model training data.

AI judge. In addition to the human preference signal provided by the fine-tuned reward model, we employ AI judges (Gu et al., 2024; Tan et al., 2025) for feedback that can be adapted quickly and customized to any given context. In particular, AI feedback provides a fast lever to shape the behavior of the model in targeted ways without incurring delays for reward model data collection and retraining.

Verifiable rewards. We employ verifiable rewards to improve capabilities in areas such as instruction following where adherence to a constraint can be checked directly. For example, verifiable rewards are included for contexts that say "respond in a single paragraph" or "use fewer than 10 words". Compared to non-verifiable rewards, we find verifiable rewards to be less prone to reward hacking (Zhao et al., 2025b; Wen et al., 2024), less sensitive to multi-epoching, and generally useful for stabilizing training.

Verifiable rewards are also used to mitigate biases in non-verifiable rewards. For example, AI feedback rubrics tend to induce an upwards pressure on length and style elements, which we mitigate using both the reward model and verifiable rewards. For length specifically, we identify acceptable response-length distributions per context offline and penalize responses that fall outside a predefined quantile range. This keeps response length stable through training without overly constraining the response or introducing a reward that can be hacked by collapsing the response into one that is overly terse.

Combining rewards. When optimizing these rewards for helpfulness and safety climbing, two challenges arise. (Wang et al., 2024c) First, the different types of rewards occupy different scales and are not directly comparable. Second, the reward distribution is itself context-dependent, narrow for some prompts and wide for others. Naïvely summing such rewards would result in the largest-magnitude signal dominating irrespective of its importance. In addition, while we want to optimize our rewards jointly, certain criteria are non-negotiable: for example, a well-written response that is unsafe remains unacceptable, regardless of its quality. We address these challenges with two complementary strategies, applied selectively based on the context.

Lexicographic reward shaping. For one set of contexts, lower-importance rewards become active only when all rollouts in a group score equally on higher-importance rewards. This induces a strict priority ordering, where a secondary reward influences the gradient only if the primary reward is tied within the rollout group (Skalse et al., 2022). Because it operates on within-group relative comparisons, this formulation is also invariant to the absolute scale of each reward.

Gated reward application. For other contexts, higher-importance rewards must satisfy a minimum level of performance before lower-importance rewards are applied at all (Achiam et al., 2017). Safety is the canonical case: an unsafe response receives the minimum reward and is never graded on response quality.

Both strategies sidestep cross-scale calibration by replacing additive combination with conditional or rank-based logic, ensuring that high-priority objectives are never traded away for gains on lower-priority ones.

3.4.2 Instruction Following and Steerability

Instruction following (IF) is a core capability for LLMs: models should adhere to constraints specified directly by users during the conversation, by API users through developer instructions, and by platform owners through privileged system instructions. The model should be steerable across such formatting and behavioral constraints according to predefined priorities (Wallace et al., 2024). To build robust instruction following capabilities, we construct a dataset spanning a range of constraints, scenarios, and levels of complexity, drawing from synthetic data and expert human annotations.

Data. We use two distinct sources of data: expert-written contexts and synthetic data. We find that the complex constraints in expert-written prompts help bootstrap capabilities, while synthetic data enables maximum coverage.

Synthetic data generation. Inspired by prior work (Xu et al., 2025a; Dong et al., 2025), we build a flexible multi-stage pipeline that generates realistic instructions and scenarios with evaluation rubrics. It starts with prompted *instruction and model spec generation* using an LLM guided by a manually curated *constraint taxonomy* (see Sec. E for more details) and a set of diverse seeds. This is followed by *generating scenarios* that are multilingual, cover short and extended dialogues, include cases with system, developer, and user messages, and span over 40 domains including coding, writing, analysis, and travel planning. In addition, we include adversarial cases with conflicting system, developer, and user instructions to train the model to respect instruction hierarchy. A final *critique and rewrite* step evaluates and updates each scenario based on naturalness, rubric alignment, and groundedness.

Data filtering. We maintain high quality data through multiple rounds of filtering, combining various quality heuristics, complexity filters, and rejection sampling. In particular, rubrics are validated for self-containment, unambiguity, and alignment with their stated constraints, while prompts are screened against our safety policies. Following An et al. (2025), we additionally control for difficulty level via *pass-rate* analysis. We keep only examples that are challenging but solvable by our model.

Reward design. To prevent degenerate responses and reward hacking during RL, we use hybrid reward signals. For constraints with deterministic verifiers, we use rule-based checks. This provides fast, deterministic, and well-calibrated reward signals.

In addition, an LLM judge (Sec. 3.4.1) independently evaluates atomic rubrics, producing a binary judgment per rubric. Multiple judgment passes are averaged for robustness. Finally, our trained reward model evaluates general response quality complementing the constraint-specific IF judges. Rewards are then combined with the lexicographic grading aggregation scheme from Sec. 3.4.1, with IF-specific rewards serving as the primary signal.

3.4.3 Safety

In line with our commitment to supporting human autonomy, we define safety as the model's ability to provide helpful responses that remain compliant with our policies (Microsoft, 2022).

Data. To keep training aligned with this target, we develop a taxonomy for our data curation. Safety data targets two complementary failure modes: *unsafe compliance*, where the model fulfills a request it should refuse, and *over-refusal*, where it unnecessarily declines a legitimate request. Each candidate example is annotated against the policy taxonomy and assigned to one of two prompt categories:

Category	Pipeline	Source
Harmful	Human Red-Teaming	Vendor-written prompts Internal red-team exercises
	Automated Attacks	Template-based attacks, e.g., PyRIT (Munoz et al., 2024) Non-interactive LLM-generated, e.g., PAP (Zeng et al., 2024a) Interactive LLM-based, e.g., TAP (Mehrotra et al., 2024)
Borderline	Do-Not-Refuse Slice Capability Data	Curated data carried across data generations Annotated via safety pipeline into do-not-refuse slice

Table 8. Sources of harmful and borderline prompts.

- **Harmful prompts:** requests where some or all assistance is disallowed by policy. The response strategy is *full refusal* or *partial refusal* (decline the unsafe portion while offering safe alternatives).
- **Borderline prompts:** requests that touch sensitive domains but remain answerable within policy. The response strategy is *do-not-refuse*: provide a bounded, helpful answer rather than hedging or refusing.

Data sources. Tab. 8 summarizes how prompts in each category are sourced. Harmful prompts are collected through both human red-teaming and automated adversarial generation. Borderline prompts are drawn from an existing *do-not-refuse* slice carried forward across data generations, as well as from capability data that is routed through the safety annotation pipeline and selected into the do-not-refuse slice, exposing the model to policy-adjacent requests that should remain answerable.

Reward design. Each model response is scored by safety judges along three dimensions:

- **Policy compliance** measures whether the response violates our safety policies.
- **Response engagement** compares a response's engagement level to the expected target (*refuse*, *partial-refuse*, or *do-not-refuse*), penalizing both over-refusals and compliance on severe requests.
- **Response style** measures whether the response is following the expected tone and guiding principles, e.g., acknowledging difficulty without moralizing or recounting traumatic events back to the user for sensitive self harm requests.

The judge scores are combined with the reward signal from our reward model and, depending on the data source, additional AI judge signal. A simple weighted average is insufficient: scalar rewards can remain positive on policy-non-compliant responses.[9] We therefore use a safety-gated aggregator: the policy-compliance field gates the reward as described in Sec. 3.4.1. If the safety judge marks a response non-compliant, the rollout receives the minimum reward regardless of other scores; otherwise the normal weighted mixture is used.

3.4.4 Honesty

We define honesty as the model's ability to produce correct responses when it knows the answer, and to hedge appropriately when it does not. At the same time, the model should not over-hedge, since minimizing factual errors alone can simply push the model to make fewer claims (Min et al., 2023). The objective must therefore balance factual precision with informativeness.

[9] In a paired rollout audit, expected safety Likert had only moderate agreement with the policy-compliance field (Pearson 0.293, Spearman 0.344), and 87.8% of policy-non-compliant responses received reward model scores ≥ 3.

Focus Area	Example Guidance
Emojis	Minimal usage - only in clearly appropriate, casual contexts. Do not use emojis for list headers or as a sign-off.
Tables & Markdown	Bullet-lists and tables are used for making multiple points, giving examples, or presenting data. Headings break up answers that strictly need multiple distinct parts to answer.
Tone & Formality	Default to professional tone, adopting a playful or engaging tone only when directly asked by the user.
Layout	Minimal preamble or wrap-up phrases, no sycophantic introductions. Clear information hierarchy. Answer is easy to quickly scan through and important information is presented up front.
Density	No wall-of-text sections. Bold text is used strategically. Spacing between paragraphs. Sentences stay short enough to be readable.

Table 9. Examples of target behavior descriptions from our style guide.

Data. We compile a diverse set of manually curated data from vendors, PII-filtered consumer Copilot logs[10], and synthetically generated data. Inspired by prior work (Zhang et al., 2024; Wei et al., 2024b,c), our data spans a spectrum of difficulty: (a) *established factual queries,* both short-form and long-form, where responses can be precisely verified against reference answers; (b) *challenging factual queries,* targeting long-tail or obscure topics where model coverage is expected to be inconsistent and reference labels are generated via search-augmented verification, and (c) *false-premise queries,* where the question contains an incorrect presupposition and no correct affirmative answer exists. Covering these boundary cases pushes the model to maintain factual integrity and hedge only when it lacks knowledge.

Reward design. For factuality grading, reference labels are generated offline for every RL example via retrieval-augmented generation and verification. Each model response is then scored by an LLM judge along two dimensions—factuality and confidence—producing one of five categories: CONFIDENT_CORRECT, UNCONFIDENT_CORRECT, NOT_ATTEMPTED, UNCONFIDENT_INCORRECT, and CONFIDENT_INCORRECT. These are combined into a scalar reward via a weighted sum: confident-correct responses receive the highest reward, confident hallucinations the steepest penalty, abstentions a neutral reward, and unconfident-but-correct responses a reduced reward to discourage over-hedging.

3.4.5 Style

We define a style guide specifying what good outputs look like: warmth without sycophancy, scannable structure, and tone calibrated to context rather than simply mirroring the user. The style guide also covers when and how to use emojis, formality level, notation for math and code, and general text density. Tab. 9 gives high-level examples of the guidance we train our model to follow.

Data. Style is graded on a broad set of PII-filtered Microsoft consumer Copilot logs, vendor-written contexts (both static and interactive), and Arena conversations. This data covers low-to-medium difficulty prompts, excluding complex instruction following, coding, math, and STEM. Prompts are classified by user intent (e.g., creative writing, practical guidance, information-seeking, and chit-chat), with active collection targeting areas where the model is weak.

[10]Consumer Copilot log data is only collected for users who have not opted out of having their data used for model training and excludes certain users as set out in Privacy FAQ for Microsoft Copilot | Microsoft Support.

Capability	Sample weight	Token weight
STEM and Coding	56%	89%
Agentic Capability	11%	9%
General Helpfulness and Safety	33%	2%

Table 10. Consolidation SFT data mixture composition by capability.

Reward design. In addition to the reward model introduced in Sec. 3.4.1, we use prompted LLM judges to grade outputs along specific axes and penalize undesirable behaviors. These prompted judges grade responses on an integer scale of 0, 1, or 2, for major, minor, or no style issues. We found that coarse graders outperform more granular graders or prompt-specific rubrics because they give the LLM judge more flexibility to interpret the rubric appropriately for the given prompt and response, making it harder for models to hack the grader. Style graders are applied only after verifiable rewards and safety constraints are satisfied, and are weighted alongside other graders depending on the domain.

3.5 Consolidating Capabilities into a Single Model

The preceding sections describe three teacher models, trained independently for STEM and competitive coding (Sec. 3.2), agentic capabilities (Sec. 3.3), and helpfulness and safety (Sec. 3.4). We consolidate them into a single model in two stages, shown in Fig. 12. The SFT stage reuses the self-distillation pipeline of Sec. 3.1.4, applied to each of the specialist teachers, though the three teachers require different filtering and rejection sampling strategies. For the STEM and agentic teachers, we sample rollouts across multiple checkpoints of each climb, following the diversity finding of Sec. 3.1.4, prioritizing later checkpoints. We keep multiple correct rollouts per context and apply only light filtering to remove degenerate CoTs. For the helpfulness and safety teacher, we use LLM judges and heuristic filters to score traces on style, structure, and known defects in addition to correctness.

Consolidation SFT. The SFT stage distills the three teachers into a single model. We iterated on the data mixture and hyperparameters to balance reasoning, agentic, and general helpfulness abilities, evaluated on public benchmarks, internal benchmarks, and human evaluation. Tab. 10 reports the resulting mixture by sample and token weight. We find that balancing the mixture by sample weight is important; the token distribution is correspondingly dominated by STEM and coding, owing to their longer traces, which in practice does not harm helpfulness and safety capabilities. Deviating from our standard self-distillation recipe, consolidation SFT is done for 4 epochs, decaying the learning rate by $2\times$ from a maximum of $1 \cdot 10^{-5}$.

Consolidation RL. The final stage of lightweight RL further improves safety, over-refusals, and style. The recipe is based on the Helpfulness and Safety Climb (Sec. 3.4), with a few changes to maintain reasoning performance. We train at a maximum sequence length of 128k tokens and retain a small proportion of STEM and coding data in the RL mixture; we found both to be important, as reasoning performance on complex tasks otherwise degrades slowly over the climb.

3.6 Reinforcement Learning Infrastructure

Our RL climb relies on *Rocket*, an in-house framework for large-scale asynchronous distributed reinforcement learning, using YOLO for the learner (Sec. 2.8) and SGLang (Zheng et al., 2024) for model inference. We built Rocket because our requirements, namely the ability to support asynchronous RL across several thousand GPUs at the scale MAI-Thinking-1 demanded, were not served by available open-source RL frameworks (von Werra et al., 2020; Hu et al., 2024; Sheng et al., 2024). Rocket's core RL data flow is organized

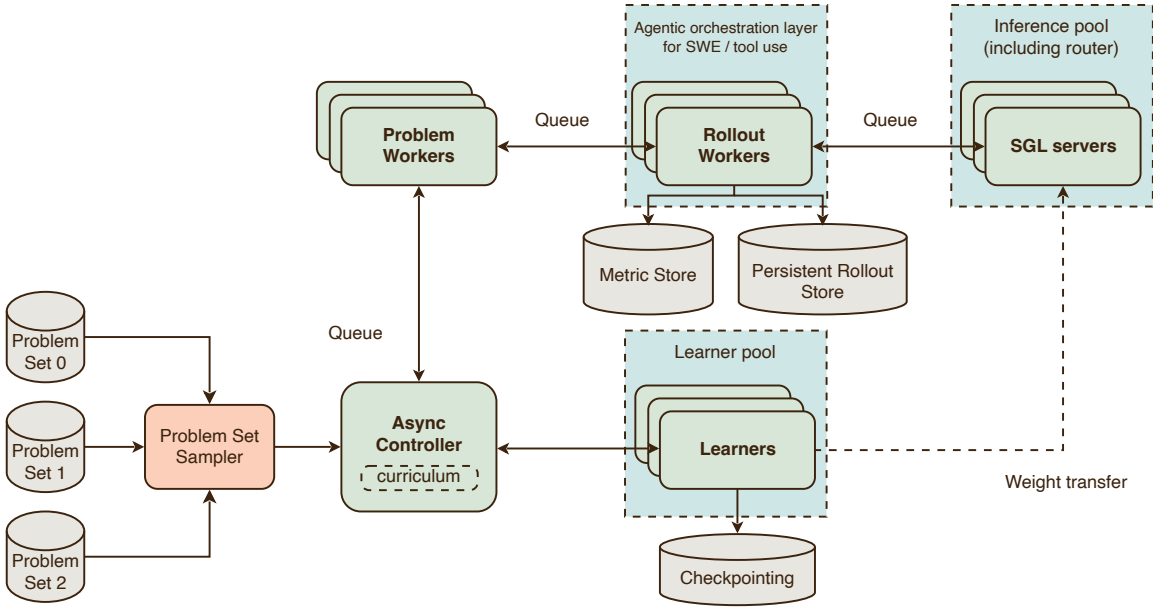

Figure 19. An overview of Rocket, our large scale Reinforcement Learning framework.

around a single controller, a pool of problem and rollout workers, and the router and inference servers that produce model generations. The controller, problem workers, and rollout workers are each implemented as a single Python process / Ray actor, while routing and inference are provided by the SGLang-based serving stack. An overview of the Rocket architecture is shown in Fig. 19.

3.6.1 Controller

The controller loads RL tasks and sends them to the problem workers for processing. For each task, the controller receives a set of completed rollouts with associated grading metadata (pass/fail decisions, rewards, normalized advantages, etc.). The completed rollouts are filtered by the controller and sent as batches to the learner. The controller abstraction allows Rocket to switch between different controller implementations for on-policy and off-policy RL. In practice, we primarily use off-policy RL for large runs and reserve on-policy RL for smaller experiments and debugging.

3.6.2 Problem Worker

When a problem worker receives an RL task from the controller, it generates a set of rollouts, computes a normalized advantage for each rollout, and sends the results back to the controller. The problem worker does not handle rollout generation itself. Instead, it sends a set of requests to the rollout workers—one for each rollout. The problem worker implements fault tolerance for large scale asynchronous RL such as retrying rollouts for a task if a rollout worker fails.

The typical flow for a single request to the problem worker is:

1. **Early-exit rollouts.** Following the early-exit strategy described in Sec. 3.1.3, the problem worker receives an RL task from the controller and sends out 16 requests to the rollout workers, retrying failed requests as needed. Each rollout is graded, and this information is aggregated into an overall pass rate for task. If the overall pass rate is within a predetermined interval, the problem worker proceeds to the full rollout stage. Otherwise, the RL task is aborted.

2. **Full rollouts.** The problem worker sends out an additional 128 requests to the rollout workers, again retrying failed requests as needed. Each rollout is graded, and this information is aggregated into an overall pass rate for task. If the overall pass rate is within a predetermined interval, the problem worker proceeds to postprocessing. Otherwise, the RL task is aborted.

3. **Postprocessing.** The problem worker computes a normalized advantage for each rollout to use in the RL objective (Sec. 3.1.1). Depending on the problem in question, this step may include applying length penalties for long rollouts, applying GRPO reward normalization, or other heuristics.

3.6.3 Rollout Worker

When the rollout worker receives a request from a problem worker, it generates a single rollout, optionally grades it, and then sends the results back to the caller. In a typical flow, the rollout worker starts by generating an initial model prompt and sending it to a model inference server. When the response comes back, the rollout worker parses it, performs any tool calls, and incorporates the results into a follow-up request to the inference server. This process continues until the model stops making tool calls and outputs a final answer, or exceeds some pre-determined limit on the number of steps or the amount of time the rollout can take.

After a rollout finishes, the next step is to grade the model's output. The grader's output contains a numeric reward and an optional pass/fail verdict. Depending on the problem, grading can either happen on the rollout worker or on the problem worker. If individual rollouts can be evaluated in isolation, grading is performed by the rollout worker. For example, on a math problem, the grader might output $(1, \texttt{pass})$ if the model's output is mathematically equivalent to the problem's groundtruth answer and $(0, \texttt{fail})$ otherwise. On the other hand, some graders need to look at the answers for multiple rollouts simultaneously, e.g., so that they can ask an LLM to perform pairwise comparisons of different model answers. In that setting, grading happens on the problem worker. A single problem can have multiple grades, in which case a user-defined policy aggregates the individual grades into a single overall reward.

3.6.4 Router and Inference

Inference is one of the most important components of the system: depending on the scale and characteristics of the job, the ratio of inference to learner GPUs can be as large as 5:1. In our largest RL job using 4864 GB300 chips, 4096 chips are dedicated to inference while the remaining 768 chips are dedicated to the learner. Inference performance and system stability are therefore first-order concerns. The numerics gap between the learner and the inference engine is critical to control for RL training stability.

Inference setup. Our RL inference stack is built on top of SGLang and the SGLang router (Zheng et al., 2024). We chose SGLang because it met our needs of a rich set of RL serving features, fast experimentation against open-source models, and strong ecosystem velocity. On top of the individual SGLang workers, we use the SGLang router to handle load balancing, traffic control, and prefix caching.

Inference performance. Because most of a job's GPU usage comes from inference, optimizing inference throughput is critical for efficient RL. Minimizing request latency is also critical to control staleness. We specialize our optimizations depending on whether rollouts occur over single turns or multiple turns. For single-turn workloads, prompts are typically short but generations (including thinking) can reach up to 128k tokens, so the main bottleneck is KV cache memory. We reduce KV cache and weight footprints via expert parallelism and data parallelism for attention. We also disable prefix caching so that sliding-window-attention tokens can be fully evicted during long generations (our model architecture relies heavily on sliding window attention). We further reduce communication overhead via data parallelism on MLP layers, DeepEP (DeepEveryParallel) (Zhao et al., 2025a), and EPLB (Expert Parallelism Load Balancer) (DeepSeek-AI, 2025). For

multi-turn workloads, prompts can grow very long while generations are often short, making these workloads largely prefill-heavy. Here we lean heavily on prefix caching: our prefix cache hit rate reaches 97–98% on production RL runs.

Inference system stability. At the scale of RL used to train MAI-Thinking-1 with thousands of inference chips, individual replicas crash, hang, slow down, or fail due to node failure on a regular basis; the inference layer must degrade and restart gracefully rather than fail the entire job when this happens. We rely on defense-in-depth at three layers. At the *replica* level, each SGLang server runs a self-watchdog that probes its own generation endpoint and monitors scheduler memory; an unhealthy replica triggers its own graceful restart so the orchestrator can bring up a clean replacement. At the *router* level, the SGLang router acts as a circuit breaker: when backend replicas become unhealthy, it stops admitting new requests, runs a multi-stage probe before reopening, and periodically rediscovers replicas that silently dropped out of its active set. Per-replica flow control prevents a slow replica from accumulating a backlog that would degrade the whole pool. At the *job* level, a liveness monitor tracks the number of live actors in each class (inference replicas, routers, rollout workers, learner ranks) and fails the job for a clean restart if any class falls below its allowed threshold. An orthogonal step-progress watchdog catches the harder case where everything appears alive but training has stopped advancing. Finally, when a rollout worker's request to an inference server fails mid-flight, it is retried against another replica, so individual replica failures degrade pool throughput but do not lose rollouts.

Numerics gap between inference and learner. A critical aspect of RL is the numerics gap between the inference engine and the learner. YOLO (the learner) and SGLang (the inference engine) use different kernels, scheduling, and parallelism strategies. Even small per-token logprob discrepancies compound across long rollouts and can destabilize the importance-sampling correction in off-policy RL. To mitigate these issues, we use bf16 for both the learner and the inference engine in our RL runs; this yields substantially smaller numerics gaps and more stable training than the lower-precision alternatives we evaluated. On top of bf16, we apply MoE routing replay (Ma et al., 2025) and top-p mask replay (Sec. 3.1.3).

3.6.5 Weight Transfer

Moving from synchronous to asynchronous RL turns weight transfer into a recurring distributed synchronization problem: fresh learner weights must reach the inference fleet every k steps, and transfer time creates inference idleness and additional staleness. The core difficulty is that learner and inference allocations shard the same tensors differently: Each side may use different combinations or degrees of FSDP, pipeline parallelism, data-parallel attention, and tensor layouts, while parameters may also differ in precision, quantization state, or matrix layout. Every transfer must therefore both reshard bytes and perform per-tensor transformations.

Transfer plan. We compile resharding, data movement, and per-tensor transformation into a single *transfer plan* computed once at job start and reused thereafter. For each parameter, the planner intersects the learner's and inference server's sharding layouts and emits one entry per non-empty sub-shard overlap, recording source rank, destination rank, byte extent, and required operations such as `dtype` casts or layout permutations. Resharding is implicit in this intersection; each learner rank sends only the slices that its paired inference rank needs, eliminating separate materialization of full tensors. Transforms run on whichever side minimizes their cost, and the runtime pipelines packing, transfer, and unpacking so successive sub-shards overlap in time. To decouple planning from fleet size, the plan targets an idealized topology of one learner and one inference server; at runtime each source–destination pair expands into a *transfer group* covering all live replicas of the destination rank, so the plan remains valid as replicas join, leave, or are replaced. Because data parallelism replicates rather than shards parameters, one DP group already holds

Benchmark	MAI-Thinking-1	Sonnet 4.6	Opus 4.6	GPT 5.4	Kimi K2.6	DeepSeek V3.2	DeepSeek V4	GLM-5.1
AIME 2025	97.0	95.6	99.8	—	—	93.1	—	—
AIME 2026	94.5	—	—	—	96.4	—	—	95.3
HMMT Feb 2026	84.9	—	—	—	92.7	—	95.2	82.6
GPQA Diamond	84.2	89.9	91.3	92.8	90.5	82.4	90.1	86.2
LCB v6	87.7	—	—	—	89.6	83.3	93.5	—
Terminal-Bench 2.0	46.0	59.1	65.4	75.1	66.7	46.4	67.9	69.0
SWE-bench Verified	73.5	79.6	80.8	—	80.2	73.1	80.6	—
SWE-Bench Pro	52.8	—	53.4	57.7	58.6	—	55.4	58.4

Table 11. Post-trained model evaluation results on STEM and agentic coding public benchmarks. Other model numbers are taken from respective official model cards. Models listed with "—" scores do not report numbers for the respective benchmark. For agentic coding evaluations, we use a total context length of 256k. For all other evals above we use maximum output tokens of 256k.

every model parameter—the plan need not enlist all D learner ranks. We restrict it to a subset of DP groups, each paired with a disjoint slice of inference replicas and running independently. A 36-server fleet split four ways performs four 9-server transfers in parallel, improving throughput while containing the failure blast radius.

4 Evaluations

4.1 Benchmark Evaluations

We compare the performance of MAI-Thinking-1 to a variety of open- and closed-weights frontier models on public benchmarks and with human side-by-side evaluations. These evaluations span a broad set of areas, highlighting the model's versatility across different domains: STEM, agentic coding, knowledge, instruction following, long context, safety, health, honesty, and tool calling. All benchmark evaluations for MAI-Thinking-1 are reported as the average across 4 runs using uniform inference settings with temperature $T = 1$ and top-p sampling with $p = 0.97$ unless otherwise stated.

Tab. 11 reports our results for STEM and agentic coding benchmarks. We draw other model's numbers from their official model cards and release announcements. Taken together, the results place MAI-Thinking-1 in the competitive range of other popular LLMs: it does not lead the field, but it delivers consistently strong performance across a broad set of benchmark categories. Of note, MAI-Thinking-1 exceeds Claude Sonnet 4.6 on AIME 2025 and approaches Claude Opus 4.6 performance on SWE-Bench Pro.

Additionally, all of our SWE agentic training data uses only bash and string-replace as tools, and does not include targeted terminal-interaction environments. The model's current Terminal-Bench performance therefore reflects generalization from broader agentic training rather than direct training on Terminal-Bench-like environments.

Math, science, and competitive coding. For Math, we evaluate MAI-Thinking-1 on the 2025 and 2026 editions of AIME, as well as the February 2026 HMMT benchmark from MathArena (Dekoninck et al., 2026). For Science, we evaluate on Graduate-Level Google-Proof Q&A benchmark (GPQA) (Rein et al., 2023) which comprises knowledge-intensive, graduate- and research-level questions, primarily in STEM domains. For competitive coding, we evaluate on LiveCodeBench v6 (LCB v6) (Jain et al., 2024) which contains up-to-date competitive coding problems. More evaluation details can be found in Appendix G.

	Knowledge		Instruction Following			Long Context	Tool Calling
Model	MMLU Pro	SimpleQA Verified	IF Bench	Adv. IF	Multi-Challenge	GraphWalks (≤128k)	BFCL v3
MAI-Thinking-1	85	31	69	85	53	90	72
Sonnet 4.6	87	29	50	86	57	96	76

	Safety			Honesty		Health	
Model	AIR-Bench	CyberSec Instruct	CyberSec Auto	Long Fact	Truthful QA	HealthBench Prof.	MedXpert QA
MAI-Thinking-1	88	63	63	98	88	35	43
Sonnet 4.6	88	62	56	98	88	38	49

Table 12. Post-trained model evaluation results on various public benchmarks. All results for Sonnet 4.6 were generated via our own evaluation suite. For AdvancedIF, we report rubric-level scores.

Agentic coding and tool calling. For agentic coding, we evaluate MAI-Thinking-1 on SWE-bench Verified (Chowdhury et al., 2024), SWE-Bench Pro (Deng et al., 2025), and Terminal-Bench 2.0 (Merrill et al., 2026). For tool calling we evaluate MAI-Thinking-1 on BFCL v3 (Patil et al., 2025). Unlike the STEM evaluations, these evaluations are multi-turn and require models to interact with environments. For all three benchmarks, we evaluate our model with a very simple ReAct-style (Yao et al., 2023) loop, where we always append to the agent loop depicted in Figure 18. For SWE-bench Verified and SWE-Bench Pro, we enable both bash and string replacement tools. For Terminal-Bench 2.0, we only enable bash tool to mimic the bare minimal terminal interface. To eliminate the inference speed and infrastructure confounding factor (Segato, 2026), we ignore predefined timeouts for Terminal-Bench 2.0. More details about the evaluation setup are in Appendix H.

General capabilities. In Tab. 12, we report our results on benchmarks covering knowledge, instruction following, long context, safety, honesty, health, and tool calling. For benchmarks in these areas, we found that not all labs report official results in model cards or model announcements. In order to provide a baseline against which to compare MAI-Thinking-1, we evaluated Sonnet 4.6 using max reasoning effort and max sequence length on these benchmarks and also report those results in Tab. 12. Across most benchmarks we found that our model is comparable with Sonnet 4.6.

Specifically, we report on SimpleQA Verified (Haas et al., 2026) and MMLU-Pro (Wang et al., 2024b) for knowledge and reasoning capabilities, IFBench (Pyatkin et al., 2026), AdvancedIF (He et al., 2025) and MultiChallenge (Deshpande et al., 2025) for Instruction following, GraphWalks for long context capabilities, AIR-Bench (Zeng et al., 2024b) and CyberSecEval 4 (Wan et al., 2024; Meta, 2024) for safety, TruthfulQA (Lin et al., 2022) and LongFact (Wei et al., 2024c) for honesty, HealthBench (Arora et al., 2025) and MedXpertQA (Zuo et al., 2025) for health knowledge tasks. For a description of each evaluation setup, see Appendix J.

4.2 Human Side-by-Side Evaluations

To supplement the capability-focused public benchmarks described above, which focus on narrow and objectively-defined quality criteria, we conducted human evaluations across a breadth of real-world tasks. These evaluations compare two models side by side holistically with a focus on overall helpfulness. Side-by-side evaluations help surface quality differences that aren't apparent when reviewing responses in isolation.

Task category	Share of prompts
Open QA, brainstorming and advising, content authoring	13–14% each
Structured problem-solving, information extraction, academic help, insight generation, content summarization	6–7% each
Task planning, context-based QA, other text analysis	5% each
Personal support, entertainment, chit-chat, role-play	3–4% each

Table 13. Task distribution for human side-by-side evaluation.

Evaluation task selection. Evaluation tasks are derived from complementary sources to ensure comprehensive coverage of genuine user needs and strong discriminative power for comparing models. The final set contains 1276 tasks, all in English, 30% of which are multi-turn. See Table 13 for details on the task distribution.

The first source of tasks is expert-authored prompts following a structured taxonomy spanning real-world use cases of varying complexity, including both single- and multi-turn conversations. The second source is logs from Microsoft's consumer Copilot product, carefully filtered to exclude prompts containing personally identifiable information (PII), prompts that are incomplete or lack necessary conversational context, adversarial prompts, and prompts requiring custom configurations such as coding environments, image generation capabilities, or access to external tools. We used stratified sampling to ensure use case coverage and balance difficulty across dimensions such as specificity and constraint diversity.

Rater pool and evaluation process. To conduct these model evaluations, we worked with evaluators managed by Surge AI, a reputable data labeling vendor. These evaluators are native English speakers from a range of generalist and professional backgrounds. Evaluators are screened through a multi-stage qualification process that assesses their ability to evaluate core LLM capabilities and failure modes, including fact-checking, reading comprehension, and instruction following. Training materials include rating instructions and examples of common failure modes.

For each prompt, a rater is first asked to carefully assess the MAI and other model responses individually across several dimensions: *instruction following* (both explicit and implicit), *factuality* (using a search engine to help with fact-checking), *conciseness and relevance*, *completeness*, and *style and tone*. For each dimension, the rater determines whether the response had no, minor, or major issues. As a final step, the rater decides on an overall preference rating between the two responses on a 7-point Likert scale, ranging from "much worse than" (-1.5) to "much better than" (1.5). We observed strong inter-annotator agreement across the rater pool, validating that ratings are consistent and reproducible within acceptable noise thresholds.

Results. Table 14 shows the human evaluation results on overall pairwise preference on a scale of $[-1.5, 1.5]$ and the delta on individual quality dimensions on a scale of $[-1, 1]$. In aggregate we found that raters preferred MAI-Thinking-1 to Sonnet 4.6, but preferred Opus 4.6 to MAI-Thinking-1. Versus Sonnet 4.6, MAI-Thinking-1 won 49% of comparisons, tied on 6%, and lost on 45%. Versus Opus 4.6, MAI-Thinking-1 won on 43%, tied on 5%, and lost on 52%. On targeted dimensions, raters found MAI-Thinking-1 superior to Sonnet 4.6 on *conciseness and relevance* and *style and tone*, and roughly equivalent (within noise) on *instruction following*, *factuality*, and *completeness*.

4.3 Internal Safety Evaluation

Safety and over-refusal. We construct an internal benchmark to measure over-refusal on prompts identified as low-risk requests that the model should answer. A refusal judge then scores each response against the prescribed strategy, flagging refusals, hedging, or unwarranted partial refusals. The over-refusal rate

Model Performance	vs Sonnet 4.6	vs Opus 4.6
Overall side-by-side preference	0.07 ± 0.06	-0.07 ± 0.06
Instruction following Δ	-0.01 ± 0.02	-0.04 ± 0.02
Factuality Δ	-0.02 ± 0.02	-0.03 ± 0.02
Conciseness and relevance Δ	0.11 ± 0.02	0.07 ± 0.02
Completeness Δ	-0.01 ± 0.02	-0.02 ± 0.02
Style and tone Δ	0.08 ± 0.02	0.05 ± 0.02

Table 14. Human evaluation results for MAI-Thinking-1 versus Sonnet 4.6 and Opus 4.6. Positive values indicate a preference for MAI-Thinking-1, while negative values indicate a preference for Sonnet or Opus.

is the fraction of prompts where the response fails to comply, and helpfulness is reported as one minus this rate. Paired with the safety pass rate (judge safety score > 3 on a 1–5 Likert scale) on high-sensitivity items, this surfaces the ideal model behavior that is safer on higher-risk, harmful requests and more helpful on lower-risk, benign ones. See Appendix I for a detailed explanation of the evaluation methodology and dataset construction.

Fig. 20 plots this safety-helpfulness balance for MAI-Thinking-1 against Sonnet 4.6. Across five of the eight categories, MAI-Thinking-1 sits above and/or to the right of Sonnet 4.6 indicating positive performance, with the largest gains on Chemical, Biological, Radiological, and Nuclear (CBRN), Self Harm, and Elections & Politics.

Jailbreaks. We source 2.5K unique seed scenarios from vendors, internal red-teaming, and open-source benchmarks including HarmBench (Mazeika et al., 2024) and StrongREJECT (Souly et al., 2024) to construct an internal jailbreak evaluation suite. We augment the sourced prompts to produce a final evaluation set of approximately 9.5K jailbreak prompts. We group these into three buckets by degree of transformation and attacker adaptivity: *Foundational*, *Compositional*, and *Adaptive* Techniques (defined in Fig. 21). Foundational Techniques are single-step transformations that preserve harmful intent through simple modifications such as jailbreak wrappers or prompt templates. Compositional Techniques combine multiple transformations or structured rewrites, including template-based attacks such as PyRIT (Munoz et al., 2024), PAP-style transformations (Zeng et al., 2024a), and non-English or mixed-language variants. Adaptive Techniques introduce interaction, search, or multi-turn structure, including TAP (Mehrotra et al., 2024) and multi-turn attacks (Russinovich et al., 2024).

Fig. 21 reports attack success rate (ASR) across the three buckets; lower values indicate stronger safety. Across these prompt transformation types, MAI-Thinking-1 achieves a low ASR comparable with Sonnet 4.6 and Opus 4.6.

5 Safety Red Teaming

To evaluate safety properties beyond automated benchmarks, we conducted red-teaming in parallel with the model development cycle. The goal was to surface adversarial vulnerabilities, novel attack vectors, and harm category gaps that automated evaluations do not reliably detect, and to feed findings back into training data collection and policy refinements on a rolling basis.

5.1 Internal Red Teaming

Methodology. Internal red-teaming was conducted by MAI red teamers (safety researchers and recruited external annotators), and was performed on different model versions throughout the development cycle.

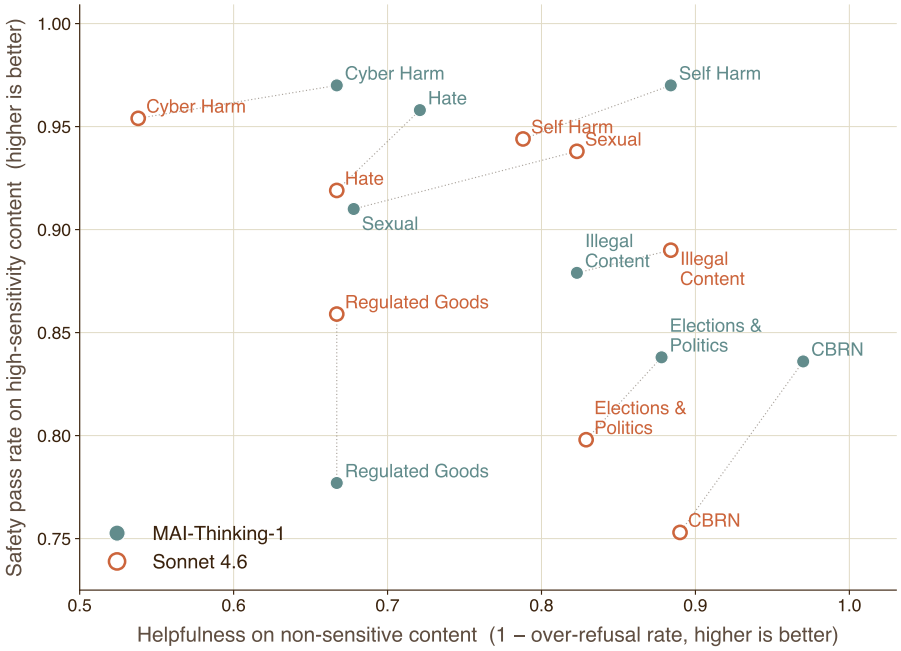

Figure 20. *Safety–helpfulness tradeoff by harm category*. Metrics are averaged over two generations per prompt. For a given harm category, the model with better helpfulness-safety tradeoff lies to the top-right of the other model. Both models are connected with a dotted line.

Across fifteen engagements spanning the early, mid, and late stages during MAI-Thinking-1's development, red teamers executed over 2,170 goal-based adversarial scenarios across 25 policy categories. Each scenario ran over 5–10 conversational turns to allow escalation beyond first-turn refusals. We increased the scope and difficulty of evaluations as successive model versions became more capable.

Annotators were screened for their adversarial-prompting ability and given access to internal model deployments. They received a pre-populated tracker of goal-based scenarios, each with supporting context and a target policy category, and were instructed to elicit policy violations over multiple conversational turns. Violations were annotated with the outcome, the severity, the turn of the first violation, the text from the worst violation, and other metadata. The program had several scope constraints: testing was primarily in English, with non-English inputs used in limited cases as a jailbreak vector rather than systematically evaluated; and agentic tool-use and multimodal inputs were out of scope.

Findings and mitigations. A central output of the program is a taxonomy of attack patterns that recurred independently across red teamers and model checkpoints, which we take as evidence that the patterns (not the individual prompts) are the durable adversarial surface. Six patterns dominated: multi-turn escalation under a benign pretext; fictional or novelistic framing; credentialed-persona pretexts; gradual recursion or formatting drift (repeated requests to expand, reformat, or operationalize a previously hedged answer); in-context age-indicator bypass; and authoritative-document fabrication. Across categories, the same patterns drove the majority of red-team successes, and the taxonomy now functions as a coverage checklist for prompt collection and judge-rubric development in subsequent cycles.

Findings fed back into training on a rolling cadence. A prominent recurring pattern was fiction-framing bypasses that surface self-harm content, including multilingual variants. This pattern seeded a curated set that was scaled into the safety evalulation and into seed-expansion for adversarial prompt collection in the

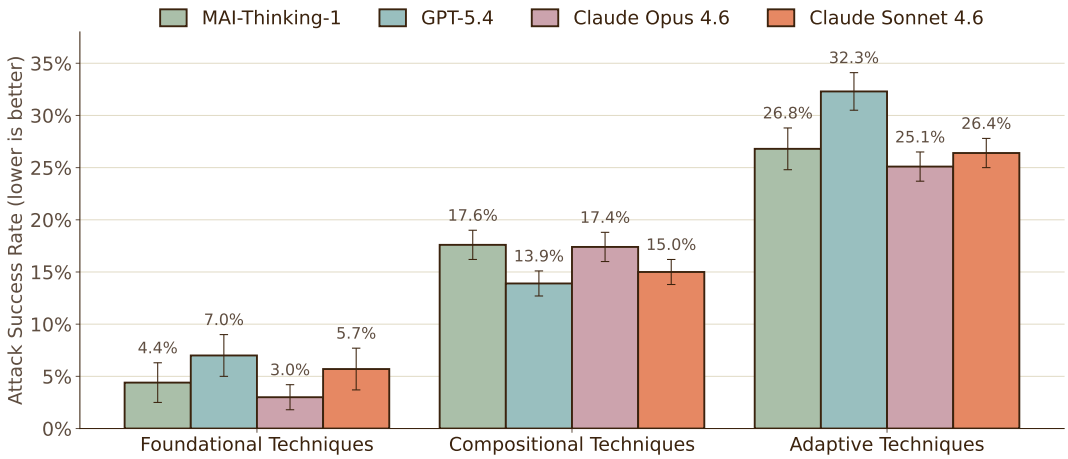

Figure 21. *Attack success rate on jailbreak evaluations grouped by transformation type.* **Foundational Techniques** are single-step transformations that preserve harmful intent through simple modifications such as jailbreak wrappers or prompt templates. **Compositional Techniques** combine multiple transformations or structured rewrites, including template-based attacks such as PyRIT (Munoz et al., 2024), PAP-style transformations (Zeng et al., 2024a), and non-English or mixed-language variants. **Adaptive Techniques** introduce interaction, search, or multi-turn structure, including TAP (Mehrotra et al., 2024) and multi-turn attacks (Russinovich et al., 2024). Lower attack success rate indicates stronger safety. Error bars denote 95% confidence intervals. Third-party model results include provider-side safety filtering.

SFT and RL safety mixes.

Across the top priority remediation categories identified during red teaming, aggregate attack success fell by roughly 22% from pre-mitigation to the final candidate. We reduced jailbreak success by ∼44%, hate & fairness by ∼43%, child safety issues by ∼30% and mental health attacks by ∼20%.

5.2 Independent Red Teaming

Additional red-teaming was conducted independently by Microsoft's AI Red Team (AIRT) and third-party vendors. Engagements are scoped to risk areas where static evaluations are weakest — automated adversarial attack methods, code and cyber-misuse safety, psychosocial and mental-health harms, and multilingual coverage. Structured dangerous-capability / uplift evaluations (Phuong et al., 2024; Li et al., 2024b) are out of scope for this release.

Adaptive Tree of Attacks with Pruning (TAP) attacks (Mehrotra et al., 2024) were surfaced as a robustness gap. In response, we built a targeted adversarial data pipeline that begins with broad generation of realistic harmful scenarios, expands them through a diverse set of attack-transformation templates, and then uses TAP-style adaptive refinement to optimize the resulting prompts against the current model until they produce reliable model-specific failures. This closed-loop process turned an external red-team finding into targeted remediation data, producing a large reduction in TAP jailbreak susceptibility and bringing the model comparable to state-of-the-art models on the same attack vectors.

Low-resource language framing (Deng et al., 2024) was flagged as another vulnerability. Content reliably refused in English was elicited in Yoruba, Telugu, Amharic, Burmese, Khmer, and Malay. We responded by expanding the safety training data mixes with multilingual adversarial seeds, translating and re-targeting high-yield English attack patterns into the affected languages. This closed a significant portion of the English/non-English gap on the targeted languages, though multilingual robustness in the long tail of lower-resource languages remains an area of continued investment.

6 Cluster Environment

MAI-Thinking-1 required a cluster environment that is composable enough to support rapid iteration across data, model, training, evaluation, and serving components; scalable enough to test improvements at frontier size; and reliable enough that measured gains in model quality remain the focus rather than infrastructure noise.

We therefore treat the cluster environment as an active part of model development. In training, the objective is to maximize useful FLOPs per wall-clock day while preserving numerical correctness, deterministic recovery, high MFU, and high goodput. During inference, the objective is to maximize useful tokens per second and per watt while preserving quality, long-context correctness, predictable latency, and deployment efficiency. The same systems principles apply in both regimes: topology matters, memory movement matters, silent correctness failures are unacceptable, and aggregate throughput is only useful when it translates into reliable model progress or reliable serving capacity.

This section describes three parts of that environment. Sec. 6.1 describes how the training cluster converted raw accelerator capacity into a stable experimental substrate. Sec. 6.2 describes the stability, determinism, and goodput metrics used to measure whether allocated GPUs were producing useful learning progress. Sec. 6.3 briefly describes inference results from running MAI-Thinking-1 on Microsoft's MAIA-200 hardware (Microsoft, 2026).

6.1 Training Cluster

The cluster was designed as a system-level optimization problem. The required compute envelope was derived from scaling-law estimates, data quality targets, token budget, and observed training efficiency. Infrastructure planning then translated that envelope into usable FLOPs across datacenter power and cooling, GPUs, CPUs, scale-up NVLink domains, scale-out InfiniBand fabric, storage, system software, diagnostics, scheduling, checkpointing, and observability.

The planning target was not theoretical peak FLOPs alone. Long-running training jobs lose useful capacity to MFU loss, checkpoint overhead, recomputation, hardware attrition, maintenance, validation workloads, and recovery buffers. We therefore optimized for usable training capacity: the fraction of delivered hardware that could be scheduled, kept healthy, driven at high MFU, and recovered quickly when failures occurred.

MAI-Base-1 was trained on a single logical cluster at one site with 8K GB200 GPUs. This reduced experimental variance by keeping the run on a homogeneous accelerator generation, within known-good rack health boundaries, under stable scheduler behavior, and close to predictable storage paths. The broader lab environment included H100, GB200, and GB300 systems for development, validation, comparative profiling, and next-generation bring-up, but the main run prioritized locality, topology stability, and operational homogeneity.

More details about our cluster environment are in Appendix K.

6.2 Training Stability, Determinism and Goodput Metrics

Training stability was measured by the cluster's ability to keep the job stepping productively, recover quickly from failures, and preserve numerical correctness across restart paths. We tracked both visible failures and silent efficiency losses. Visible failures included crashloops, node failures, InfiniBand and NVLink link flaps and link downs, out-of-memory errors, pod terminations, checkpoint stalls, and manual requeues. Silent efficiency losses included MFU degradation, recomputation, long startup paths, slow actor scheduling, checkpoint-induced stalls, degraded memory behavior, and fabric conditions that did not immediately crash the job but reduced useful throughput.

Determinism was treated as a first-class infrastructure property. At frontier scale, deterministic training

is not only a model-code concern. It depends on the correctness and stability of the full execution substrate. The cluster must eliminate silent data corruption (Hochschild et al., 2021; Dixit et al., 2021), keep communication topology stable, prevent unhealthy devices and links from entering the job, and preserve the order of floating-point reductions and accumulations across checkpoint and restart boundaries.

Goodput was the primary production KPI. We define goodput as the ratio of ideal training duration to actual wall-clock duration (Jiang et al., 2024; Grattafiori et al., 2024), with the gap decomposed into overhead categories. This framing made the cost of failure explicit. A failure did not only cost the time until restart. It could also force recomputation since the last durable checkpoint, trigger startup and actor-scheduling delays, perturb placement, block checkpoint progress, or cause lower MFU after recovery. Similarly, an MFU drop was treated as a production incident even when the job continued running, because the cluster was still consuming allocated GPU-hours while producing less training progress than expected.

6.2.1 MFU and Goodput

The broader pattern was a transition from reliability to efficiency. The first layer of improvement came from reducing interruptions like crashloops, node failures, link flaps, requeues, out-of-memory (OOM) errors, and checkpoint stalls. The second and harder layer came from eliminating silent goodput loss like MFU degradation, recomputation, long startup paths, slow process scheduling, checkpoint-induced stalls, and degraded network or memory behavior. At frontier scale, both layers mattered because every hour of overhead compounded across thousands of GPUs. The operating goal was to treat goodput as the primary production KPI: every failure mode, runtime slowdown, checkpoint event, and MFU regression had an owner, a detection signal, a prevention path, and a quantified impact on usable FLOPs.

MAI-Base-1 pre-training run reached 90.0% goodput at 8K GPUs, despite being larger than earlier pre-training runs. Total overhead dropped to 51 hours. Recomputation, the time spent reproducing previously-computed steps after falling back to a checkpoint, fell to 6.5 hours, only 15% of overhead. Non-stepping time dropped to 14 hours, or 27% indicating that the system became much better at staying alive, avoiding repeated rework, and recovering without long manual intervention. However, the final run also showed the next bottleneck clearly. MFU drop overhead became the largest single remaining category, at 18 hours and 35% of overhead. This was driven by checkpointing, network degradation, memory pressure, and hardware health transitions. The failure trends also improved but did not disappear.

6.3 Inference Efficiency and Model Deployment

As reasoning models become increasingly important, inference cost, latency, and serving scalability emerged as key constraints on deployment. We therefore treated inference efficiency as a first-class objective throughout the development of the model, pursuing opportunities across the stack: model architecture, serving engine, and even choice of hardware for deployment.

To improve our performance per watt, we implemented MAI-Thinking-1 on Microsoft's MAIA-200 hardware. Compared to a GB200-based deployment, MAIA-200 delivers over 40% higher token generation throughput under the same rack power budget. This improvement in performance per watt enables more efficient utilization of datacenter power and supports serving larger inference workloads at scale.

6.4 Sustainability and Community-First AI Infrastructure Initiatives

It is important to us that the development of AI is sustainable socially, environmentally, and within the communities that our infrastructure is built and operated. Microsoft is committed to being carbon negative, water positive, zero waste, and to protecting ecosystems. In 2025, the company met a milestone on this journey by matching 100% of our annual global electricity consumption with renewable energy.

MAI-Thinking-1 was primarily trained on Microsoft-owned infrastructure in Phoenix, AZ, which has been built to meet LEED Gold Certification standards, recognizing excellence in environmental sustainabil-

ity and energy efficiency. As one example, our backup generators here use a renewable diesel that reduces net carbon emissions compared to a traditional petroleum-based diesel.

Microsoft's Community-First AI Infrastructure initiative commits to minimizing environmental impact and investing in the communities where it builds and operates. This approach is grounded in close partnership with Phoenix-area utilities, water authorities, and conservation organizations to strengthen shared infrastructure while reducing overall system demand.

Microsoft has committed more than $50M to growing public water assets, including a municipal water storage facility with the City of Goodyear, alongside funding for water and wastewater pipeline extensions and broader system upgrades that benefit the community. These investments are paired with conservation efforts in collaboration with partners such as the Gila River Water Storage system and The Nature Conservancy, helping recharge groundwater and improve basin-level efficiency.

At the same time, Microsoft is investing in long-term workforce development through its Datacenter Academy hosted at Estrella Mountain Community College and Glendale Community College, ensuring that infrastructure growth in Phoenix translates into durable economic opportunity for the community.

7 Conclusion and Future Directions

We introduced our hill-climbing machine, our approach to model development that optimizes every component of the pipeline, from data and infrastructure to RL recipes and evaluation. MAI-Thinking-1 is the first model produced by this machine: a 35B active / 1T total parameter MoE trained without distillation from third-party models. MAI-Thinking-1 stands among the strongest models in its weight class on STEM reasoning and software engineering tasks.

MAI-Thinking-1 is a starting point, not a destination. Looking forward, we plan to extend our hill-climbing to more modalities, larger scales, and refined capabilities. Progress in AI is not the product of any single model; it is the product of pipelines that can be reliably improved. We will continue refining ours, and we look forward to sharing our future climbing progress.

References

Amro Abbas, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S. Morcos. Semdedup: Data-efficient learning at web-scale through semantic deduplication, 2023. URL https://arxiv.org/abs/2303.09540.

Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *International Conference on Machine Learning*, volume 70, pages 22–31. PMLR, 2017.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *ICLR*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. Polaris: A post-training recipe for scaling reinforcement learning on advanced reasoning models, 2025. URL https://hkunlp.github.io/blog/2025/Polaris.

L.W. Anderson and D.R. Krathwohl. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2001.

Anthropic. Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet. https://www.anthropic.com/engineering/swe-bench-sonnet, January 2025.

Rahul K Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, et al. Healthbench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*, 2025.

Anthony C. Atkinson, Alexander N. Donev, and Randall D. Tobias. *Optimum Experimental Designs, with SAS*. Oxford University Press, 2007. ISBN 978-0-19-929659-0.

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

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026a.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026b.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*, 2024.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025.

Adrien Barbaresi. Trafilatura: A Web Scraping Library and Command-Line Tool for Text Discovery and Extraction. In *Proceedings of the ACL-IJCNLP 2021 System Demonstrations*, pages 122–131, 2021. doi: 10.18653/v1/2021.acl-demo.15.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of compression and complexity of sequences 1997*, pages 21–29. IEEE, 1997.

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

Mayee F. Chen, Tyler Murray, David Heineman, Matt Jordan, Hannaneh Hajishirzi, Christopher Ré, Luca Soldaini, and Kyle Lo. Olmix: A framework for data mixing throughout LM development. *arXiv preprint arXiv:2602.12237*, 2026a. URL https://arxiv.org/abs/2602.12237.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, et al. Swe-universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*, 2026b.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023. URL http://jmlr.org/papers/v24/22-1144.html. arXiv:2204.02311.

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, et al. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, Zhiyuan Liu, Hao Peng, Lei Bai, Wanli Ouyang, Yu Cheng, Bowen Zhou, and Ning Ding. The entropy mechanism of reinforcement learning for reasoning language models, 2025. URL https://arxiv.org/abs/2505.22617.

Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Zhao, Xiaodong Sun, Aixin Liu, and Wenfeng Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024.

DeepSeek-AI. Expert Parallelism Load Balancer (EPLB). https://github.com/deepseek-ai/EPLB, 2025. GitHub repository.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. Technical report, https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang,

Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025a. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models, 2025b. URL https://arxiv.org/abs/2512.02556.

Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, An-

dreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, Rodolphe Jenatton, Lucas Beyer, Michael Tschannen, Anurag Arnab, Xiao Wang, Carlos Riquelme, Matthias Minderer, Joan Puigcerver, Utku Evci, Manoj Kumar, Sjoerd van Steenkiste, Gamaleldin F. Elsayed, Aravindh Mahendran, Fisher Yu, Avital Oliver, Fantine Huot, Jasmijn Bastings, Mark Patrick Collier, Alexey Gritsenko, Vighnesh Birodkar, Cristina Vasconcelos, Yi Tay, Thomas Mensink, Alexander Kolesnikov, Filip Pavetić, Dustin Tran, Thomas Kipf, Mario Lučić, Xiaohua Zhai, Daniel Keysers, Jeremiah Harmsen, and Neil Houlsby. Scaling vision transformers to 22 billion parameters, 2023. URL https://arxiv.org/abs/2302.05442.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with llms. 2026. URL https://arxiv.org/abs/2605.00674.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025.

Yue Deng, Wenxuan Zhang, Sinno Jialin Pan, and Lidong Bing. Multilingual jailbreak challenges in large language models, 2024. URL https://arxiv.org/abs/2310.06474.

Kaustubh Deshpande, Ved Sirdeshmukh, Johannes Baptist Mols, Lifeng Jin, Ed-Yeremai Hernandez-Cardona, Dean Lee, Jeremy Kritz, Willow E Primack, Summer Yue, and Chen Xing. Multichallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier llms. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 18632–18702, 2025.

Nolan Dey, Daria Soboleva, Faisal Al-Khateeb, Bowen Yang, Ribhu Pathria, Hemant Khachane, Shaheer Muhammad, Zhiming, Chen, Robert Myers, Jacob Robert Steeves, Natalia Vassilieva, Marvin Tom, and Joel Hestness. BTLM-3B-8K: 7b parameter performance in a 3b parameter model, 2023. URL https://arxiv.org/abs/2309.11568.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. *arXiv [cs.CL]*, September 2023. doi: 10.48550/arXiv.2309.11495.

Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021. URL https://arxiv.org/abs/2102.11245.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models. In *International Conference on Learning Representations*, volume 2025, pages 39286–39313, 2025.

Zhengxiao Du, Aohan Zeng, Yuxiao Dong, and Jie Tang. Understanding emergent abilities of language models from the loss perspective. In *Advances in Neural Information Processing Systems*, 2024.

Essential AI, :, Andrew Hojel, Michael Pust, Tim Romanski, Yash Vanjani, Ritvik Kapila, Mohit Parmar, Adarsh Chaluvaraju, Alok Tripathy, Anil Thomas, Ashish Tanwer, Darsh J Shah, Ishaan Shah, Karl Stratos, Khoi Nguyen, Kurt Smith, Michael Callahan, Peter Rushton, Philip Monk, Platon Mazarakis, Saad Jamal, Saurabh Srivastava, Somanshu Singla, and Ashish Vaswani. Essential-web v1.0: 24t tokens of organized web data, 2025. URL https://arxiv.org/abs/2506.14111.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, et al. davinci-env: Open swe environment synthesis at scale. *arXiv preprint arXiv:2603.13023*, 2026.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *arXiv preprint arXiv:2503.01307*, 2025.

Bofei Gao, Yejie Wang, Yibo Miao, Ruoyu Wu, Feifan Song, Longhui Yu, Tianyu Liu, and Baobao Chang. Towards a better initial policy model for scalable long-CoT reinforcement learning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 7652–7665, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.397. URL https://aclanthology.org/2025.findings-acl.397/.

Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv [cs.CL]*, June 2024. doi: 10.48550/arXiv.2406.20094.

Gemma Team. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Gemma Team. Gemma 4 model card. https://ai.google.dev/gemma/docs/core/model_card_4, 2026.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray

Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Nathan Godey, Wissam Antoun, Rian Touchent, Rachel Bawden, Éric de la Clergerie, Benoît Sagot, and Djamé Seddah. Gaperon: A peppered English-French generative language model suite. *arXiv preprint arXiv:2510.25771*, 2025. doi: 10.48550/arXiv.2510.25771. URL https://arxiv.org/abs/2510.25771.

Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. https://skylion007.github.io/OpenWebTextCorpus, 2019.

Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *International Conference on High Performance Computing*, pages 41–59. Springer, 2020.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *The Innovation*, 2024.

Melody Y Guan, Miles Wang, Micah Carroll, Zehao Dou, Annie Y Wei, Marcus Williams, Benjamin Arnav, Joost Huizinga, Ian Kivlichan, Mia Glaese, et al. Monitoring monitorability. *arXiv preprint arXiv:2512.18311*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning

in llms through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Lukas Haas, Gal Yona, Giovanni D'Antonio, Sasha Goldshtein, and Dipanjan Das. SimpleQA Verified: A reliable factuality benchmark to measure parametric knowledge, 2026. URL https://arxiv.org/abs/2509.07968.

Bingguang Hao, Zengzhuang Xu, Maolin Wang, Yuntao Wen, Yicheng Chen, Cunyin Peng, Long Chen, Dong Wang, Xiangyu Zhao, Jinjie Gu, Chenyi Zhuang, and Ji Zhang. FunReason: Enhancing large language models' function calling via self-refinement multiscale loss and automated data refinement. *arXiv [cs.LG]*, May 2025. doi: 10.48550/arXiv.2505.20192.

Yun He, Wenzhe Li, Hejia Zhang, Songlin Li, Karishma Mandyam, Sopan Khosla, Yuanhao Xiong, Nanshu Wang, Xiaoliang Peng, Beibin Li, et al. Advancedif: Rubric-based benchmarking and reinforcement learning for advancing llm instruction following. *arXiv preprint arXiv:2511.10507*, 2025.

David Heineman, Valentin Hofmann, Ian Magnusson, Yuling Gu, Noah Smith, Hanna Hajishirzi, Kyle Lo, and Jesse Dodge. Signal and noise: A framework for reducing uncertainty in language model evaluation. *arXiv preprint arXiv:2508.13144*, 2025.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Rebecca Soskin Hicks, Mikhail Trofimov, Dominick Lim, Rahul K Arora, Foivos Tsimpourlas, Preston Bowman, Michael Sharman, Chi Tong, Kavin Karthik, Arnav Dugar, et al. Healthbench professional: Evaluating large language models on real clinician chats. *arXiv preprint arXiv:2604.27470*, 2026.

Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, pages 9–16, Ann Arbor, MI, USA, 2021. ACM. doi: 10.1145/3458336.3465297.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, DDL Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 10, 2022.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *arXiv preprint arXiv:2405.11143*, 2024. URL https://arxiv.org/abs/2405.11143.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve, 2022. URL https://arxiv.org/abs/2210.11610.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. DeepSpeed Ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023. URL https://arxiv.org/abs/2309.14509.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, 2024. URL https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng. arXiv:2402.15627.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers, 2023. URL https://arxiv.org/abs/2305.19466.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H.

Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi K2: Open agentic intelligence, 2026. URL https://arxiv.org/abs/2507.20534.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, et al. Rewardbench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797, 2025.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, 2022.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding, 2020. URL https://arxiv.org/abs/2006.16668.

Junyi Li, Charith Peris, Ninareh Mehrabi, Palash Goyal, Kai-Wei Chang, Aram Galstyan, Richard Zemel, and Rahul Gupta. The steerability of large language models toward data-driven personas. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7290–7305, 2024a.

Nathaniel Li, Alexander Pan, Anjali Gopal, Summer Yue, Daniel Berrios, Alice Gatti, Justin D. Li, Ann-Kathrin Dombrowski, Shashwat Goel, et al. The WMDP benchmark: Measuring and reducing malicious use with unlearning. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024b. URL https://arxiv.org/abs/2403.03218.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods, 2022. URL https://arxiv.org/pdf/2109.07958.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

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, 2025a. URL https://arxiv.org/abs/2502.16982.

Qian Liu, Xiaosen Zheng, Niklas Muennighoff, Guangtao Zeng, Longxu Dou, Tianyu Pang, Jing Jiang, and Min Lin. Regmix: Data mixture as regression for language model pre-training. In *International Conference on Learning Representations*, volume 2025, pages 38305–38339, 2025b.

Tianqi Liu, Wei Xiong, Jie Ren, Lichang Chen, Junru Wu, Rishabh Joshi, Yang Gao, Jiaming Shen, Zhen Qin, Tianhe Yu, Daniel Sohn, Anastasiia Makarova, Jeremiah Liu, Yuan Liu, Bilal Piot, Abe Ittycheriah, Aviral Kumar, and Mohammad Saleh. Rrm: Robust reward model training mitigates reward hacking, 2025c. URL https://arxiv.org/abs/2409.13156.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

Zhiyuan Lu, Chenliang Li, Yingcheng Shi, Weizhou Shen, Ming Yan, and Fei Huang. CorpusQA: A 10 million token benchmark for corpus-level analysis and reasoning. 2025.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Zhifang Sui, and Fuli Luo. Stabilizing MoE reinforcement learning by aligning training and inference routers, 2025. URL https://arxiv.org/abs/2510.11370.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv [cs.CL]*, March 2023. doi: 10.48550/arXiv.2303.17651.

Rabeeh Karimi Mahabadi, Sanjeev Satheesh, Shrimai Prabhumoye, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. Nemotron-CC-Math: A 133 billion-token-scale high quality math pretraining dataset, 2025. URL https://arxiv.org/abs/2508.15096.

Mantas Mazeika, Long Phan, Xuwang Yin, Andy Zou, Zifan Wang, Norman Mu, Elham Sakhaee, Nathaniel Li, Steven Basart, Bo Li, et al. Harmbench: a standardized evaluation framework for automated red teaming and robust refusal. In *Proceedings of the 41st International Conference on Machine Learning*, pages 35181–35224, 2024.

MediaWiki. Wikitext 1.0.0 Specification. https://www.mediawiki.org/wiki/Specs/wikitext/1.0.0, 2026. Accessed: 2026-05-31.

Anay Mehrotra, Manolis Zampetakis, Paul Kassianik, Blaine Nelson, Hyrum Anderson, Yaron Singer, and Amin Karbasi. Tree of attacks: Jailbreaking black-box llms automatically, 2024. URL https://arxiv.org/abs/2312.02119.

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

Meta. PurpleLlama CyberSecEval 4. https://github.com/meta-llama/PurpleLlama, 2024.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022. URL https://arxiv.org/abs/2209.05433.

Microsoft. Microsoft responsible AI standard, v2. Technical report, Microsoft Corporation, 2022. URL https://go.microsoft.com/fwlink/?linkid=2311742. Accessed: 2026-06-01.

Microsoft. Maia 200: The AI accelerator built for inference. https://blogs.microsoft.com/blog/2026/01/26/maia-200-the-ai-accelerator-built-for-inference/, 2026.

Mika. General agent: A self-evolving, synthetic agent environment. https://www.primeintellect.ai/blog/general-agent, May 2026. Prime Intellect Blog.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. Factscore: Fine-grained atomic evaluation of factual precision in long form text generation, 2023. URL https://arxiv.org/pdf/2305.14251.

Mistral-AI, :, Abhinav Rastogi, Albert Q. Jiang, Andy Lo, Gabrielle Berrada, Guillaume Lample, Jason Rute, Joep Barmentlo, Karmesh Yadav, Kartik Khandelwal, Khyathi Raghavi Chandu, Léonard Blier, Lucile

Saulnier, Matthieu Dinot, Maxime Darrin, Neha Gupta, Roman Soletskyi, Sagar Vaze, Teven Le Scao, Yihan Wang, Adam Yang, Alexander H. Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Andy Ehrenberg, Anmol Agarwal, Antoine Roux, Arthur Darcet, Arthur Mensch, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Chris Bamford, Christian Wallenwein, Christophe Renaudin, Clémence Lanfranchi, Darius Dabert, Devon Mizelle, Diego de las Casas, Elliot Chane-Sane, Emilien Fugier, Emma Bou Hanna, Gauthier Delerce, Gauthier Guinet, Georgii Novikov, Guillaume Martin, Himanshu Jaju, Jan Ludziejewski, Jean-Hadrien Chabran, Jean-Malo Delignon, Joachim Studnia, Jonas Amar, Josselin Somerville Roberts, Julien Denize, Karan Saxena, Kush Jain, Lingxiao Zhao, Louis Martin, Luyu Gao, Lélio Renard Lavaud, Marie Pellat, Mathilde Guillaumin, Mathis Felardos, Maximilian Augustin, Mickaël Seznec, Nikhil Raghuraman, Olivier Duchenne, Patricia Wang, Patrick von Platen, Patryk Saffer, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Romain Sauvestre, Rémi Delacourt, Sanchit Gandhi, Sandeep Subramanian, Shashwat Dalal, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Thibault Schueller, Thibaut Lavril, Thomas Robert, Thomas Wang, Timothée Lacroix, Valeriia Nemychnikova, Victor Paltz, Virgile Richard, Wen-Ding Li, William Marshall, Xuanyu Zhang, and Yunhao Tang. Magistral, 2025. URL https://arxiv.org/abs/2506.10910.

Model Context Protocol. What is the Model Context Protocol (MCP)?, 2026. URL https://modelcontextprotocol.io/docs/getting-started/intro.

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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577, Carlsbad, CA, USA, 2018.

Gary D. Lopez Munoz, Amanda J. Minnich, Roman Lutz, Richard Lundeen, Raja Sekhar Rao Dheekonda, Nina Chikanov, Bolor-Erdene Jagdagdorj, Martin Pouliot, Shiven Chawla, Whitney Maxwell, Blake Bullwinkel, Katherine Pratt, Joris de Gruyter, Charlotte Siska, Pete Bryan, Tori Westerhoff, Chang Kawaguchi, Christian Seifert, Ram Shankar Siva Kumar, and Yonatan Zunger. Pyrit: A framework for security risk identification and red teaming in generative ai system, 2024. URL https://arxiv.org/abs/2410.02828.

Muse Spark Team. Muse Spark Eval Methodology. https://ai.meta.com/static-resource/muse-spark-eval-methodology, 2026.

NVIDIA, :, Aaron Blakeman, Aaron Grattafiori, Aarti Basant, Abhibha Gupta, Abhinav Khattar, Adi Renduchintala, Aditya Vavre, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, Aleksandr Shaposhnikov, Alex Kondratenko, Alexander Bukharin, Alexandre Milesi, Ali Taghibakhshi, Alisa Liu, Amelia Barton, Ameya Sunil Mahabaleshwarkar, Amir Klein, Amit Zuker, Amnon Geifman, Amy Shen, Anahita Bhiwandiwalla, Andrew Tao, Anjulie Agrusa, Ankur Verma, Ann Guan, Anubhav Mandarwal, Arham Mehta, Ashwath Aithal, Ashwin Poojary, Asif Ahamed, Asit Mishra, Asma Kuriparambil Thekkumpate, Ayush Dattagupta, Banghua Zhu, Bardiya Sadeghi, Barnaby Simkin, Ben Lanir, Benedikt Schifferer, Besmira Nushi, Bilal Kartal, Bita Darvish Rouhani, Boris Ginsburg, Brandon Norick, Brandon Soubasis, Branislav Kisacanin, Brian Yu, Bryan Catanzaro, Carlo del Mundo, Chantal Hwang, Charles Wang, Cheng-Ping Hsieh, Chenghao Zhang, Chenhan Yu, Chetan Mungekar, Chintan Patel, Chris Alexiuk, Christopher Parisien, Collin Neale, Cyril Meurillon, Damon Mosk-Aoyama, Dan Su, Dane Corneil, Daniel Afrimi, Daniel Lo, Daniel Rohrer, Daniel Serebrenik, Daria Gitman, Daria Levy, Darko Stosic, David Mosallanezhad, Deepak Narayanan, Dhruv Nathawani, Dima Rekesh, Dina Yared, Divyanshu Kakwani, Dong Ahn, Duncan Riach, Dusan Stosic, Edgar Minasyan, Edward Lin, Eileen Long, Eileen Peters Long, Elad Segal, Elena Lantz, Ellie Evans, Elliott Ning, Eric Chung, Eric Harper, Eric Tramel, Erick Galinkin, Erik Pounds, Evan Briones, Evelina Bakhturina, Evgeny Tsykunov, Faisal Ladhak, Fay Wang, Fei Jia, Felipe

Soares, Feng Chen, Ferenc Galko, Frank Sun, Frankie Siino, Gal Hubara Agam, Ganesh Ajjanagadde, Gantavya Bhatt, Gargi Prasad, George Armstrong, Gerald Shen, Gorkem Batmaz, Grigor Nalbandyan, Haifeng Qian, Harsh Sharma, Hayley Ross, Helen Ngo, Herbert Hum, Herman Sahota, Hexin Wang, Himanshu Soni, Hiren Upadhyay, Huizi Mao, Huy C Nguyen, Huy Q Nguyen, Iain Cunningham, Ido Galil, Ido Shahaf, Igor Gitman, Ilya Loshchilov, Itamar Schen, Itay Levy, Ivan Moshkov, Izik Golan, Izzy Putterman, Jan Kautz, Jane Polak Scowcroft, Jared Casper, Jatin Mitra, Jeffrey Glick, Jenny Chen, Jesse Oliver, Jian Zhang, Jiaqi Zeng, Jie Lou, Jimmy Zhang, Jinhang Choi, Jining Huang, Joey Conway, Joey Guman, John Kamalu, Johnny Greco, Jonathan Cohen, Joseph Jennings, Joyjit Daw, Julien Veron Vialard, Junkeun Yi, Jupinder Parmar, Kai Xu, Kan Zhu, Kari Briski, Katherine Cheung, Katherine Luna, Keith Wyss, Keshav Santhanam, Kevin Shih, Kezhi Kong, Khushi Bhardwaj, Kirthi Shankar, Krishna C. Puvvada, Krzysztof Pawelec, Kumar Anik, Lawrence McAfee, Laya Sleiman, Leon Derczynski, Li Ding, Lizzie Wei, Lucas Liebenwein, Luis Vega, Maanu Grover, Maarten Van Segbroeck, Maer Rodrigues de Melo, Mahdi Nazemi, Makesh Narsimhan Sreedhar, Manoj Kilaru, Maor Ashkenazi, Marc Romeijn, Marcin Chochowski, Mark Cai, Markus Kliegl, Maryam Moosaei, Matt Kulka, Matvei Novikov, Mehrzad Samadi, Melissa Corpuz, Mengru Wang, Meredith Price, Michael Andersch, Michael Boone, Michael Evans, Miguel Martinez, Mikail Khona, Mike Chrzanowski, Minseok Lee, Mohammad Dabbah, Mohammad Shoeybi, Mostofa Patwary, Nabin Mulepati, Najeeb Nabwani, Natalie Hereth, Nave Assaf, Negar Habibi, Neta Zmora, Netanel Haber, Nicola Sessions, Nidhi Bhatia, Nikhil Jukar, Nikki Pope, Nikolai Ludwig, Nima Tajbakhsh, Nir Ailon, Nirmal Juluru, Nishant Sharma, Oleksii Hrinchuk, Oleksii Kuchaiev, Olivier Delalleau, Oluwatobi Olabiyi, Omer Ullman Argov, Omri Puny, Oren Tropp, Ouye Xie, Parth Chadha, Pasha Shamis, Paul Gibbons, Pavlo Molchanov, Pawel Morkisz, Peter Dykas, Peter Jin, Pinky Xu, Piotr Januszewski, Pranav Prashant Thombre, Prasoon Varshney, Pritam Gundecha, Przemek Tredak, Qing Miao, Qiyu Wan, Rabeeh Karimi Mahabadi, Rachit Garg, Ran El-Yaniv, Ran Zilberstein, Rasoul Shafipour, Rich Harang, Rick Izzo, Rima Shahbazyan, Rishabh Garg, Ritika Borkar, Ritu Gala, Riyad Islam, Robert Hesse, Roger Waleffe, Rohit Watve, Roi Koren, Ruoxi Zhang, Russell Hewett, Russell J. Hewett, Ryan Prenger, Ryan Timbrook, Sadegh Mahdavi, Sahil Modi, Samuel Kriman, Sangkug Lim, Sanjay Kariyappa, Sanjeev Satheesh, Saori Kaji, Satish Pasumarthi, Saurav Muralidharan, Sean Narentharen, Sean Narenthiran, Seonmyeong Bak, Sergey Kashirsky, Seth Poulos, Shahar Mor, Shanmugam Ramasamy, Shantanu Acharya, Shaona Ghosh, Sharath Turuvekere Sreenivas, Shelby Thomas, Shiqing Fan, Shreya Gopal, Shrimai Prabhumoye, Shubham Pachori, Shubham Toshniwal, Shuoyang Ding, Siddharth Singh, Simeng Sun, Smita Ithape, Somshubra Majumdar, Soumye Singhal, Stas Sergienko, Stefania Alborghetti, Stephen Ge, Sugam Dipak Devare, Sumeet Kumar Barua, Suseella Panguluri, Suyog Gupta, Sweta Priyadarshi, Syeda Nahida Akter, Tan Bui, Teodor-Dumitru Ene, Terry Kong, Thanh Do, Tijmen Blankevoort, Tim Moon, Tom Balough, Tomer Asida, Tomer Bar Natan, Tomer Ronen, Tugrul Konuk, Twinkle Vashishth, Udi Karpas, Ushnish De, Vahid Noorozi, Vahid Noroozi, Venkat Srinivasan, Venmugil Elango, Victor Cui, Vijay Korthikanti, Vinay Rao, Vitaly Kurin, Vitaly Lavrukhin, Vladimir Anisimov, Wanli Jiang, Wasi Uddin Ahmad, Wei Du, Wei Ping, Wenfei Zhou, Will Jennings, William Zhang, Wojciech Prazuch, Xiaowei Ren, Yashaswi Karnati, Yejin Choi, Yev Meyer, Yi-Fu Wu, Yian Zhang, Yigong Qin, Ying Lin, Yonatan Geifman, Yonggan Fu, Yoshi Subara, Yoshi Suhara, Yubo Gao, Zach Moshe, Zhen Dong, Zhongbo Zhu, Zihan Liu, Zijia Chen, and Zijie Yan. NVIDIA Nemotron 3: Efficient and open intelligence, 2025. URL https://arxiv.org/abs/2512.20856.

OpenAI. tiktoken: A fast bpe tokeniser for use with openai's models, 2022. URL https://github.com/openai/tiktoken.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. *CoRR*, abs/2508.10925, 2025. doi: 10.48550/ARXIV.2508.10925. URL https://doi.org/10.48550/arXiv.2508.10925.

OpenAI. simple-evals: Lightweight library for evaluating language models. https://github.com/openai/simple-evals, 2025. GitHub repository. Accessed: 2026-05-31.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Gray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=TG8KACxEON.

Kaan Ozkara, Tao Yu, and Youngsuk Park. Stochastic rounding for llm training: Theory and practice, 2025. URL https://arxiv.org/abs/2502.20566.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of Forty-second International Conference on Machine Learning*, 2025.

Hao Peng, Yunjia Qi, Xiaozhi Wang, Bin Xu, Lei Hou, and Juanzi Li. Verif: Verification engineering for reinforcement learning in instruction following. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 30312–30327, 2025.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

Mary Phuong, Matthew Aitchison, Elliot Catt, Sarah Cogan, Alexandre Kaskasoli, Victoria Krakovna, David Lindner, Matthew Rahtz, Yannis Assael, et al. Evaluating frontier models for dangerous capabilities, 2024. URL https://arxiv.org/abs/2403.13793.

Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinyBenchmarks: evaluating LLMs with fewer examples, 2024. URL https://arxiv.org/abs/2402.14992.

Fabrice Prigent. UT1 URL Blacklist, 2026. URL http://dsi.ut-capitole.fr/blacklists/. Accessed 2026.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hanna Hajishirzi. Generalizing verifiable instruction following. *Advances in Neural Information Processing Systems*, 38, 2026.

Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism, 2023.

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, 2025. URL https://arxiv.org/abs/2501.11873.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. Accessed: 2024-11-15.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher, 2022. URL https://arxiv.org/abs/2112.11446.

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.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506, 2020.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Leonard Richardson. Beautiful soup documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2026. Beautiful Soup version 4.14.3; accessed 31 May 2026.

Mark Russinovich, Ahmed Salem, and Ronen Eldan. Great, now write an article about that: The crescendo multi-turn LLM jailbreak attack, 2024. URL https://arxiv.org/abs/2404.01833.

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

Gian Segato. Quantifying infrastructure noise in agentic coding evals. Engineering at Anthropic. https://www.anthropic.com/engineering/infrastructure-noise, February 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025a.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

Shivalika Singh, Angelika Romanou, Clémentine Fourrier, David Ifeoluwa Adelani, Jian Gang Ngui, Daniel Vila-Suero, Peerat Limkonchotiwat, Kelly Marchisio, Wei Qi Leong, Yosephine Susanto, et al. Global mmlu: Understanding and addressing cultural and linguistic biases in multilingual evaluation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18761–18799, 2025b.

Joar Skalse, Lewis Hammond, Charlie Griffin, and Alessandro Abate. Lexicographic multi-objective reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 3430–3436, 2022.

Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, a large-scale generative language model, 2022. URL https://arxiv.org/abs/2201.11990.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, et al. Bugpilot: Complex bug generation for efficient learning of swe skills. *arXiv preprint arXiv:2510.19898*, 2025.

Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari S. Morcos. Beyond neural scaling laws: beating power law scaling via data pruning, 2023. URL https://arxiv.org/abs/2206.14486.

Alexandra Souly, Qingyuan Lu, Dillon Bowen, Tu Trinh, Elvis Hsieh, Sana Pandey, Pieter Abbeel, Justin Svegliato, Scott Emmons, Olivia Watkins, et al. A strongreject for empty jailbreaks. *Advances in Neural Information Processing Systems*, 37:125416–125440, 2024.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Michael Sullivan, Mareike Hartmann, and Alexander Koller. Procedural environment generation for tool-use agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18555–18573, 2025.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Tang, Alejandro Cuadron, Chenguang Wang, Raluca Popa, and Ion Stoica. Judgebench: A benchmark for evaluating llm-based judges. In *International Conference on Learning Representations*, volume 2025, pages 63277–63303, 2025.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 OLMo 2 Furious. 2024. URL https://arxiv.org/abs/2501.00656.

Kushal Tirumala, Daniel Simig, Armen Aghajanyan, and Ari S. Morcos. D4: Improving LLM pretraining via document de-duplication and diversification, 2023. URL https://arxiv.org/abs/2308.12284.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Kiran Vodrahalli, Santiago Ontanon, Nilesh Tripuraneni, Kelvin Xu, Sanil Jain, Rakesh Shivanna, Jeffrey Hui, Nishanth Dikkala, Mehran Kazemi, Bahare Fatemi, Rohan Anil, Ethan Dyer, Siamak Shakeri, Roopali Vij, Harsh Mehta, Vinay Ramasesh, Quoc Le, Ed Chi, Yifeng Lu, Orhan Firat, Angeliki Lazaridou, Jean-Baptiste Lespiau, Nithya Attaluri, and Kate Olszewska. Michelangelo: Long context evaluations beyond haystacks via latent structure queries, 2024. URL https://arxiv.org/abs/2409.12640.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020. URL https://github.com/huggingface/trl.

Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The instruction hierarchy: Training llms to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208*, 2024.

Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, Vlad Ionescu, Yue Li, and Joshua Saxe. Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models, 2024. URL https://arxiv.org/abs/2408.01605.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. Swe-mirror: Scaling issue-resolving datasets by mirroring issues across repositories. *arXiv preprint arXiv:2509.08724*, 2025a.

Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. Auxiliary-loss-free load balancing strategy for mixture-of-experts, 2024a. URL https://arxiv.org/abs/2408.15664.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024b.

Zengzhi Wang, Fan Zhou, Xuefeng Li, and Pengfei Liu. Octothinker: Mid-training incentivizes reinforcement learning scaling. *arXiv preprint arXiv:2506.20512*, 2025b. URL https://arxiv.org/abs/2506.20512.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

Zihao Wang, Chirag Nagpal, Jonathan Berant, Jacob Eisenstein, Alex D'Amour, Sanmi Koyejo, and Victor Veitch. Transforming and combining rewards for aligning large language models. *arXiv preprint arXiv:2402.00742*, 2024c.

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. *arXiv preprint arXiv:2411.04368*, 2024a.

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, 2024b. URL https://arxiv.org/pdf/2411.04368.

Jerry Wei, Chengrun Yang, Xinying Song, Yifeng Lu, Nathan Hu, Jie Huang, Dustin Tran, Daiyi Peng, Ruibo Liu, Da Huang, Cosmo Du, and Quoc V. Le. Long-form factuality in large language models, 2024c. URL https://arxiv.org/pdf/2403.18802.

Bosi Wen, Pei Ke, Xiaotao Gu, Lindong Wu, Hao Huang, Jinfeng Zhou, Wenchuang Li, Binxin Hu, Wendy Gao, Jiaxin Xu, et al. Benchmarking complex instruction-following with multiple constraints composition. *Advances in Neural Information Processing Systems*, 37:137610–137645, 2024.

Mitchell Wortsman, Peter J. Liu, Lechao Xiao, Katie Everett, Alex Alemi, Ben Adlam, John D. Co-Reyes, Izzeddin Gur, Abhishek Kumar, Roman Novak, Jeffrey Pennington, Jascha Sohl-dickstein, Kelvin Xu, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Small-scale proxies for large-scale transformer training instabilities, 2023. URL https://arxiv.org/abs/2309.14322.

Zhiheng Xi, Xin Guo, Yang Nan, Enyu Zhou, Junrui Shen, Wenxiang Chen, Jiaqi Liu, Jixuan Huang, Zhihao Zhang, Honglin Guo, Xun Deng, Zhikai Lei, Miao Zheng, Guoteng Wang, Shuo Zhang, Peng Sun, Rui Zheng, Hang Yan, Tao Gui, Qi Zhang, and Xuanjing Huang. BAPO: Stabilizing off-policy reinforcement learning for llms via balanced policy optimization with adaptive clipping, 2025. URL https://arxiv.org/abs/2510.18927.

Violet Xiang, Chase Blagden, Rafael Rafailov, Nathan Lile, Sang Truong, Chelsea Finn, and Nick Haber. Just enough thinking: Efficient reasoning with adaptive length penalties reinforcement learning, 2025. URL https://arxiv.org/abs/2506.05256.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. WizardLM: Empowering large pre-trained language models to follow complex instructions, 2025a. URL https://arxiv.org/abs/2304.12244.

Zengzhuang Xu, Bingguang Hao, Zechuan Wang, Yuntao Wen, Xinyi Xu, Yang Liu, Long Chen, Dong Wang, Maolin Wang, Tong Zhao, Yicheng Chen, Cunyin Peng, Jinjie Gu, Leilei Gan, Xiangyu Zhao, Chenyi Zhuang, and Shi Gu. Funreason-mt technical report: Advanced data synthesis solution for real-world multi-turn tool-use, 2025b. URL https://arxiv.org/abs/2510.24645.

Zijie Yan, Hongxiao Bai, Xin Yao, Dennis Liu, Tong Liu, Hongbin Liu, Pingtian Li, Evan Wu, Shiqing Fan, Li Tao, et al. Scalable training of mixture-of-experts models with megatron core. *arXiv preprint arXiv:2603.07685*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li,

Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, Qiaobo Chen, Yinyuting Yin, Hao Zhang, Tengfei Shi, Liang Wang, Qiang Fu, Wei Yang, and Lanxiao Huang. Mastering complex control in moba games with deep reinforcement learning, 2020. URL https://arxiv.org/abs/1912.09729.

Jiasheng Ye, Peiju Liu, Tianxiang Sun, Yunhua Zhou, Jun Zhan, and Xipeng Qiu. Data Mixing Laws: Optimizing data mixtures by predicting language modeling performance. In *International Conference on Learning Representations*, 2025. doi: 10.48550/arXiv.2403.16952. URL https://arxiv.org/abs/2403.16952.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, et al. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9556–9567, 2024.

Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, et al. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15134–15186, 2025.

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.

Pedram Zamirai, Jian Zhang, Christopher R. Aberger, and Christopher De Sa. Revisiting bfloat16 training. *CoRR*, abs/2010.06192, 2020. URL https://arxiv.org/abs/2010.06192.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022. URL https://arxiv.org/abs/2203.14465.

Yi Zeng, Hongpeng Lin, Jingwen Zhang, Diyi Yang, Ruoxi Jia, and Weiyan Shi. How Johnny can persuade LLMs to jailbreak them: Rethinking persuasion to challenge AI safety by humanizing llms, 2024a. URL https://arxiv.org/abs/2401.06373.

Yi Zeng, Yu Yang, Andy Zhou, Jeffrey Ziwei Tan, Yuheng Tu, Yifan Mai, Kevin Klyman, Minzhou Pan, Ruoxi Jia, Dawn Song, Percy Liang, and Bo Li. AIR-Bench 2024: A safety benchmark based on risk categories from regulations and policies, 2024b. URL https://arxiv.org/abs/2407.17436.

Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019. URL https://arxiv.org/abs/1910.07467.

Hanning Zhang, Shizhe Diao, Yong Lin, Yi Fung, Qing Lian, Xingyao Wang, Yangyi Chen, Heng Ji, and Tong Zhang. R-tuning: Instructing large language models to say 'i don't know'. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7113–7139, 2024.

Puzhen Zhang, Weijie Bai, Wendong Fan, and Guohao Li. Toolathlon-GYM: Large-Scale Long-Horizon Environments for Tool-Use Agents, 2026. URL https://github.com/eigent-ai/toolathlon_gym.

Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. DeepEP: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025a.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models, 2026. URL https://arxiv.org/abs/2601.18734.

Yulai Zhao, Haolin Liu, Dian Yu, Sunyuan Kung, Meijia Chen, Haitao Mi, and Dong Yu. One token to fool llm-as-a-judge. *arXiv preprint arXiv:2507.08794*, 2025b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

Yuxin Zuo, Shang Qu, Yifei Li, Zhangren Chen, Xuekai Zhu, Ermo Hua, Kaiyan Zhang, Ning Ding, and Bowen Zhou. Medxpertqa: Benchmarking expert-level medical reasoning and understanding. *arXiv preprint arXiv:2501.18362*, 2025.

A Pre-training Data Pipeline Details

This appendix provides further details for the processing of our major pre-training data sources described in Sec. 2.4.

A.1 Web HTML

The majority of our web HTML corpus comes from a proprietary crawl. After initial page discovery and selection, approximately 1.2 trillion pages are crawled and parsed. HTML is converted to text using Trafilatura (Barbaresi, 2021), producing extracted text together with language ID and a corresponding confidence score. During extraction, we infer document encodings rather than assuming `UTF-8` prior to processing. This approach improves robustness when handling multilingual web data with legacy, missing, or incorrectly declared encodings.

After parsing, all pages go through filters for policy compliance, adult content, overlap with the web PDF corpus, and blocklisted domains. In addition to Microsoft standard policy Sec. 2.4, we apply UT1 block list (Prigent, 2026) to remove adult content and piracy-related domains. In all, this filtering reduces the corpus from 1.2 trillion pages to 794 billion pages. Given the prevalence of AI-generated content on the web, we also score pages with a proprietary AI-content detection model and use manual inspection to identify domains with extensive AI-generated content; those domains are filtered out of the training corpus.

The filtered web corpus then undergoes exact and fuzzy deduplication. We normalize documents by lowercasing and trimming whitespace, then remove exact duplicates using MD5 hashes. This reduces the corpus from 794 billion to 423 billion documents. Our fuzzy duplication follows the LSH method and parameters described in Sec. 2.4.3. After fuzzy deduplication, the proprietary crawl contains 73.4 billion English documents and 116.5 billion non-English documents.

We process Common Crawl with the same pipeline. Starting from roughly 300 billion pages, we keep the latest retrieved content for each URL, reducing the corpus to about 100 billion pages. After filtering, deduplication, merging with the proprietary web corpus, and a final round of exact-URL and content-level fuzzy deduplication, the Common Crawl portion contains 24.2 billion pages.

For downstream processing, we compute text embeddings for all web pages using Qwen3-Embedding-0.6B (Yang et al., 2025). The resulting corpus is then processed through pipelines tailored to general web pages, STEM pages, code pages, and key domains.

General web pages. For general web pages, we use a combination of filtering and quality binning to enable finer-grained data mixing of this large corpus. First, we apply attribute models to the text embedding of each document. These attribute models score pre-training-relevant properties such as educational value, factual accuracy and reliability, writing quality, information density, reasoning content, and general-knowledge value. On top of these attributes, we train a quality model and filter out the bottom 70% of English documents. We then apply Gopher-style (Rae et al., 2022) heuristic filters based on word statistics, punctuation patterns, common n-grams, and related document-quality features. This retains roughly 4.6 billion English documents from our proprietary crawl and 2.8 billion English Common Crawl documents. For the purpose of data mixing, these documents are binned using a combination of metadata and embedding-based quality scoring. Multilingual web data is prepared similarly to English general web data, but with language-aware filtering and quality scoring.

In addition to the broad general-web pipeline, we construct a high-quality subset of human-curated web content to better capture engaging, well-written long-form prose. We source candidate domains from human-curated surfaces, such as magazine and long-form reading aggregators, as well as external reference links from social platforms, and retrieve the corresponding pages from our web corpus.

STEM pages. Broad coverage of high-quality STEM content during pre-training is important for downstream mathematical and scientific reasoning capabilities. However, such content is relatively sparse on the web and often contains structured elements such as equations, code snippets, and technical markup that are not well handled by generic web processing pipelines. We therefore construct a dedicated pipeline to identify and extract STEM content from the web.

We identify candidate STEM pages using embedding-based classifiers along three dimensions: topic, educational value, and educational level. Topic is modeled using seven binary classifiers corresponding to Mathematics, Physics, Statistics, Chemistry, Biology, Engineering, and Computer Science. A shared educational-value classifier is used to separate educational content, such as research papers, Q&A forums, tutorials, lecture notes, and blogs, from low-value pages such as product listings, citation indices, commercial content, and boilerplate. Finally, an educational-level classifier assigns each document to high school, undergraduate, or graduate level for data mixing. To improve corpus quality, we additionally apply a set of heuristics tailored for STEM content, as generic web filters often over-penalize documents containing equations and numerical expressions. These heuristics remove PDF URLs, calculator domains, patent domains, known AI-generated-content domains, and domains identified as low quality, and filter documents based on word count, digit ratio, alphabetic character ratio, duplicate-line fraction, stop-word density, and the fraction of lines ending with ellipses.

Because Trafilatura does not always preserve mathematical equations and code well, we build a custom extraction pipeline for STEM documents. Starting from the raw HTML, we normalize MathML and LaTeX content and convert the document to Markdown. We then split the document into sections, preserving each math and code block as separate units, and use a language model to classify each section as *keep* or *remove*. The model is restricted to binary inclusion decisions and cannot generate synthetic content. Finally, we clean and normalize the extracted content using OpenWebMath-style post-processing (Gokaslan and Cohen, 2019), so that equations are consistently delimited.

This pipeline yields an English STEM corpus of 680B tokens, of which 76B are mathematics-related. For multilingual data, we obtain an additional 760B STEM tokens, including 58B high-quality mathematics-related tokens.

Code pages. For code-specific web pages, we reuse the STEM pages pipeline, including topic classification, educational-value classification, and LLM-based extraction, filtering on the Computer Science topic. To further improve quality, we score each candidate document using Qwen3-30B (Yang et al., 2025). The judge prompt is optimized with GEPA / DSPy (Agrawal et al., 2026) with approximately 2,000 human labels. After filtering out low-quality documents with additional heuristics, we obtain a dataset of approximately 233B tokens.

Key domains. For selected high-value domains, we use modified extraction pipelines to better preserve domain-specific structure and content. For example, for Wikipedia, we err on the side of inclusion to avoid dropping useful information when standard extraction would otherwise discard important page elements.

A.2 Web PDFs

We collect a web-crawled PDF corpus of approximately 10B documents spanning educational, lifestyle, technical, and professional documents. Using heuristics and classifiers on text extracted from PDF bytes, we filter this down to 620M documents for further processing. To convert the documents into trainable text, we use Azure Document Intelligence (Azure DI).

We apply several post-processing steps to the OCR outputs:

- equation and table normalization
- removal of boilerplate such as headers, footers, and text detected in figures

- removal of OCR artifacts such as glyph errors
- line unwrapping by removing line breaks and hyphenation
- removal of reference sections from academic documents

We remove SEO and spam content using text-level classifiers and PDF metadata heuristics. In particular, the PDF metadata fields "creator" and "producer" are useful high-precision signals for filtering. We further filter documents using heuristics such as text length, gzip compressibility, alphanumeric ratio, and table ratio, as well as quality classifiers.

After fuzzy deduplication, the PDF dataset consists of 1.8T English tokens and 1.85T multilingual tokens. To support data mixing, we classify documents as educational or non-educational. Educational documents are further categorized into mathematics, computer science, STEM, and non-STEM, and assigned an educational level among high school, undergraduate, and graduate.

A.3 Books and Journals

We acquire books and journals from various providers, including through direct agreements with publishers.

For each provider, we build a dedicated ingestion pipeline suited to the underlying data format and consistent with the usage limitations attached to the content. Each pipeline transforms the raw content into a canonical text format. The standardized text is then processed through OCR artifact cleaning, heuristic filtering, and content selection. The cleaned text goes through exact deduplication, title deduplication, and fuzzy deduplication to reduce unwanted repetition. Finally, using an LLM for annotation, each document in the corpus is given topic and quality labels to assess each work's usefulness for LLM pre-training and enable better control over data mixing.

A.4 Public GitHub

We source most of our 7.4T-token code pre-training corpus from public GitHub repositories and organize it into three datasets: **files**, **commits**, and **pull requests** (**PRs**). All three datasets undergo shared filtering, deduplication, decontamination, and quality scoring pipelines.

Shared processing. We first apply heuristic file-level filters to remove non-code and low-quality data. These filters include:

- Removal of files in junk folders such as `node_modules`, `build`, `__pycache__`, and `.vscode`.
- Detection of generated code via path patterns such as protobuf outputs like `_pb2.py` and TypeScript declaration files such as `.d.ts`.
- Exclusion of non-code file types such as binaries, images, archives, documents, and fonts.
- Rejection of files exceeding 30K characters or containing excessively long lines.
- Character-composition checks to filter out data dumps.

We then perform exact deduplication via SHA-512 hashing, followed by MinHashLSH-based fuzzy deduplication and semantic deduplication using cosine similarity over 1024-dimensional embeddings from Qwen3-Embedding-0.6B (Yang et al., 2025). We also decontaminate against coding problems used in reasoning training and evaluations. Finally, each example is assigned a quality score and placed into discrete quality bins for quality-aware sampling.

After dataset-specific processing, examples in each dataset are greedily packed into fixed-length sequences, yielding approximately 1.26T, 4.5T, and 1.19T tokens for files, commits, and PRs, respectively.

Files. The raw files dataset consists of the latest version of each file across all repositories. We apply additional heuristic filtering using filters similar to those described in StarCoder2 (Lozhkov et al., 2024), and specifically discard non-programming files from repositories with large token counts. Files are grouped by repository, sorted in depth-first traversal order of the repository structure, and concatenated into repository-level sequences.

Commits. The raw commits dataset consists of the most recent 10K commits extracted from each repository. Each row contains the pre-commit state of all modified files together with the patch applied by the commit. Commits that appear as constituents of a PR are removed to avoid redundancy across datasets. The pre-commit state of modified files serves as a loss-masked prefix, while the trainable text consists solely of the commit patches, rendered in either git-diff or search-and-replace format.

Pull requests. The raw PR dataset is constructed by extracting all constituent commits from cloned repositories and joining relevant metadata to each PR. Each row contains the PR title, body, linked issues, issue comments, reviews, threads, review comments, pre-commit file states, and the ordered list of constituent commits. The PR dataset is additionally decontaminated against SWE-bench Verified by removing all PRs used in the benchmark. The pre-commit state of modified files serve as a loss-masked prefix; the trainable text comprises the remaining fields, including the PR title, body, linked issues, reviews, comments, and the patches from the constituent commits. When metadata can be mapped to a specific commit (e.g., review comments and threads), it is placed immediately after the corresponding commit patch rather than at the PR level. We also apply prefix compression for excessively long prefixes by stochastically selecting lines surrounding each git patch location to fit within a fixed token budget. Sequences that still exceed 256K tokens after compression are discarded.

B Long Context Extension

Our pre-training target is a 256K token context window. Rather than training at the full 256K context length throughout pre-training or mid-training—which would be impractical due to the low MFU at long sequence lengths—we adopt a staged approach: the model is pre-trained at 16K, mid-trained at 64K (Stage 1), and then extended to 256K in a short dedicated phase (Stage 2). This section presents the ablations and evaluations that informed this recipe. Unless otherwise mentioned, all ablations are done on small scale models

B.1 Data

We experimented with a number of data mixture variants for the context extension phase, including up-weighting long-context documents and adjusting mixture ratios across domains. Ultimately, none of these modifications yielded meaningful improvements. The simplest approach—using the same data mixture as the preceding mid-training stage, re-packed at the target 256K sequence length—works well and is what we adopt for the final model.

B.2 Evaluation Setup

We evaluate long-context quality on a variety of evaluation benchmarks, some of which we document below.

Code NLL. We serialize an internal code repository into a linear token stream and extract 256K-token chunks. For each chunk, the final 16K tokens serve as the suffix on which we compute NLL, and we progressively increase the prefix from the immediately preceding tokens such that the total context (prefix + suffix) is 16K, 32K, 64K, 128K, or 256K. An ideal long-context model should leverage the additional prefix—which

contains related code from the same repository—to further reduce the suffix loss. As shown in Fig. 22(b), the context-extended checkpoint exhibits exactly this behavior: its NLL decreases monotonically from 16K to 256K, demonstrating effective use of the full context window. In contrast, models mid-trained at shorter context lengths without extension see their NLL increase once the contexts reach > 2x of the training distribution.

Retrieval NLL. To test whether the model can retrieve and utilize relevant information regardless of where in the context it is located, we design a controlled experiment: we select a 32K-token document (daily standup notes from post-training) and split it into two 16K-token blocks, A (suffix) and B (prefix), where the prefix B provides useful context for predicting the suffix A (i.e., $\text{NLL}(A \mid B) \ll \text{NLL}(A)$). We then progressively insert irrelevant chunks (pre-training related design documents) between A and B, increasing the total context length up to 256K tokens. Without dedicated long-context training, the model's NLL on the suffix degrades substantially as the related prefix moves further away indicating that the model is increasingly distracted by the intervening irrelevant content (22(a)). In contrast, the long-context checkpoint maintains a nearly flat NLL regardless of the distance between the related blocks, demonstrating robust information retrieval across the full 256K context window.

Generative QA. We construct a simple question-answering benchmark over internal repository documents, placing questions at varying positions within the context and measuring answer accuracy as a function of total context length and evidence position. On this benchmark, the model exhibits up to $4\times$ length extrapolation capability: a model trained at 32K context can answer questions correctly in contexts up to 128K tokens (Fig. 22(d)). We observe that as the total context length increases beyond the training distribution, the model has disproportionate difficulty retrieving information from *recent* (end-of-context) positions rather than distant ones (Fig. 22(e)). This could be explained by the distributional shift—when evaluating a model trained at 32K on a 64K input, the first 32K tokens fall within the training distribution, while the final tokens occupy positions the model has not been trained on. Long-context training resolves this asymmetry.

B.3 Progressive Context Length Scaling

A natural question is how much quality we sacrifice by not mid-training at our maximum desired context length. Fig. 22(b) addresses this by comparing five configurations on code NLL : four mid-training runs of 1T tokens at context lengths 16K, 32K, 64K, and 128K (all following 16K pre-training), and a progressive approach that mid-trains at 32K for 1T tokens then extends to 256K with approximately 100B additional tokens.

At 128K context, the progressive checkpoint matches the NLL of the full 1T-token 128K mid-training run, despite using far fewer tokens at long context lengths. Furthermore, all five configurations converge to roughly the same NLL at short context lengths (16K and 32K), confirming that training at longer context lengths during mid-training does not meaningfully impact short-context quality. This means there is no reason to pay for expensive long-context mid-training if a short extension phase at the end can achieve the same result. The figure supports a general recipe: mid-train at a short, MFU-friendly context length and extend at the end.

B.4 Speed of Adaptation

Fig. 22(c) shows the 256K code NLL trajectory during the context extension phase of an ablation. The adaptation is remarkably fast: the vast majority of the NLL improvement occurs within the first 1–10% of training iterations, after which the curve plateaus. This rapid convergence suggests the model is not acquiring fundamentally new capabilities during context extension, but rather calibrating its positional and attention mechanisms for sequence lengths not seen during earlier training. The representations needed to leverage

long context are already present from pre-training and mid-training; the extension phase simply adjusts them for out-of-distribution positions.

B.5 Final Recipe

For the final model, we opted for 64K mid-training followed by 140B tokens of 256K extension. While the ablation analysis (conducted primarily on NLL metrics) suggested that shorter mid-training context lengths and shorter token horizon for length extension would suffice, the interaction between context length and post-training was less well characterized, and we chose the more conservative option. The overall context extension phase remains fast and cheap relative to the rest of training. Looking ahead, the same staged approach naturally extends to even longer context windows—the short final extension phase could target 1M tokens or beyond in place of 256K, at modest compute cost.

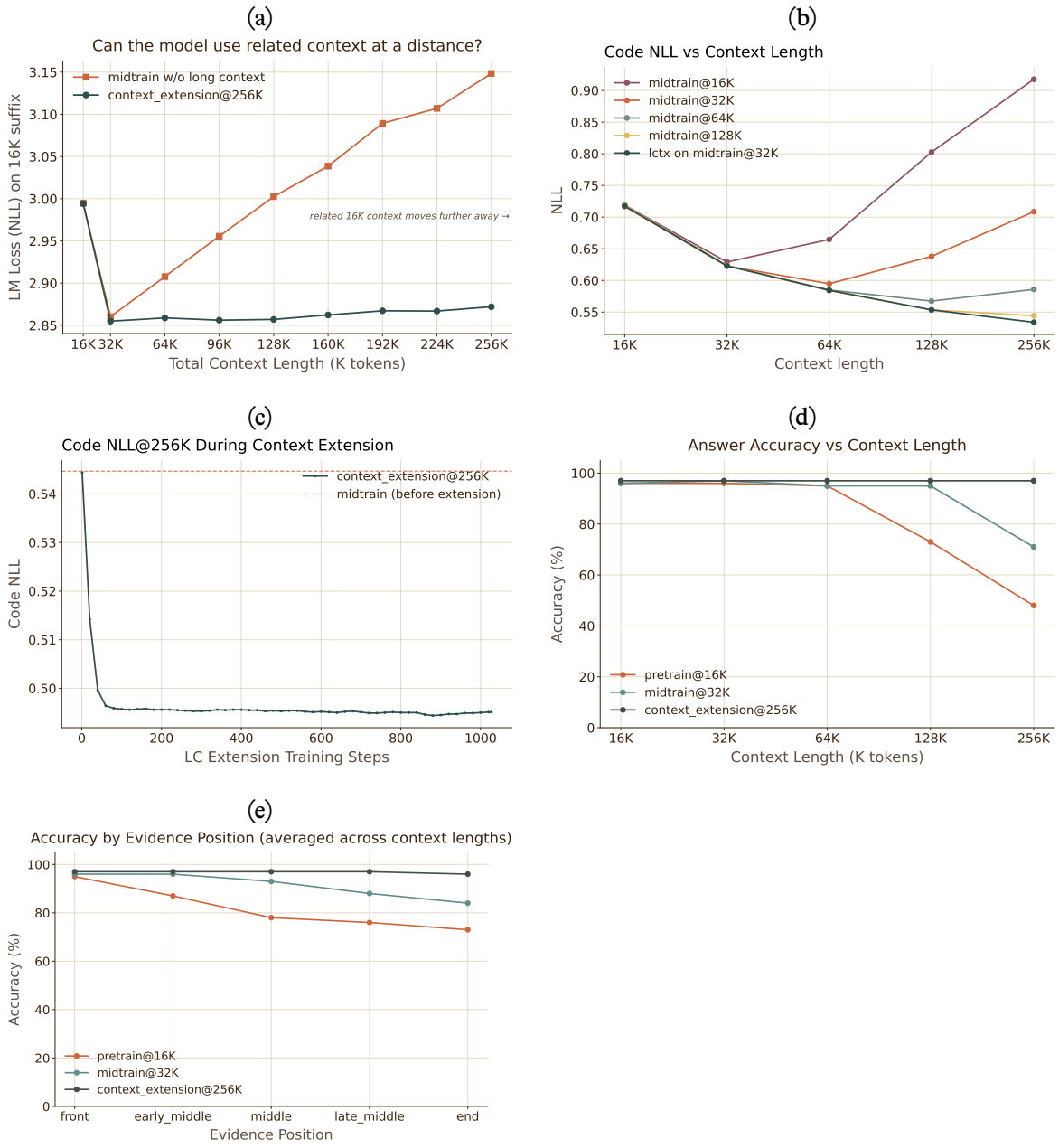

Figure 22. Long-context extension evaluation. (a) Retrieval NLL: suffix NLL as related 16K prefix is pushed further away by irrelevant context. The long-context checkpoint maintains flat NLL regardless of distance; without long-context training, NLL degrades substantially. (b) Code NLL vs. context length for five configurations: mid-training at 16K, 32K, 64K, and 128K (1T tokens each), plus a progressive approach (32K mid-training + short 256K extension). The progressive checkpoint matches or exceeds mid-training at 128K. (c) 256K code NLL during the context extension phase. Most of the improvement occurs within the first 1–10% of training iterations. (d) Answer accuracy vs. context length on the generative QA benchmark. The pre-trained base and mid-trained (32K) models degrade sharply beyond their training context lengths, while the long-context checkpoint maintains near-perfect accuracy up to 256K. (e) Answer accuracy by evidence position (averaged across context lengths). Without long-context training, accuracy drops as evidence moves toward the end of context—the out-of-distribution region. Long-context training resolves this asymmetry.

C Evolution of Reasoning Traces

Our models climb from scratch, without distilling on any third-party chain-of-thoughts (CoTs). This gives us a clean vantage point for studying how CoTs evolve as the model strengthens. In this section, we study some of the behaviors that emerge in our model's CoTs during RL training. We analyze both our STEM climb Sec. 3.2 and our agentic climb Sec. 3.3's rollouts for change of behavior archetypes, refleced in CoTs. (Guan et al., 2025; Baker et al., 2025)

C.1 STEM CoTs

Weak models guess, strong models work hard. Consider the following problem from AIME 2024 (answer: 240).

```
There are exactly three positive real numbers $k$ such that the function $f(x)=\frac{(x-18)(x
    -72)(x-98)(x-k)}{x}$, defined over the positive real numbers, achieves its minimum value at
     exactly two positive real numbers $x$. Find the sum of these three values of $k$.
```

The key distinction is not in the initial algebraic setup, but in the final verification step. The strong model derives the four algebraic candidates $k \in \{8, 32, 200, 512\}$ and filters them by the domain condition $x > 0$, excluding 512. The weak model instead guesses candidate minimizers from the visible roots 18, 72, 98, producing spurious values such as 40 and 152.

Weak model excerpt: guesses the minimizers, fabricating roots

```
Or maybe the other two k's are when x1, x2 are related to 18, 72, 98 in another way, maybe when
     k is such that x1 = 18, x2 = 72, or similar. If x1, x2 are 18, 72, then from (1) x1+x2 =
    90 = 94 + k/2 -> k/2 = 90 - 94 = -4 -> k = -8, which is not positive. So not. Or if x1, x2
    are 18, 98: 116 = 94 + k/2 -> k/2 = 20 -> k = 40. Or if x1, x2 are 72, 98: 170 = 94 + k/2
    -> k/2 = 76 -> k = 152. So possible k's are 40, 152, 512. Let's verify k = 40: if k = 40,
    then x1+x2 = 94 + 40/2 = 114. x1x2 = sqrt(18*72*98*40).

[...] Are 18, 98, 72, 40 related? [...] Not sure, but maybe this is possible. So these may be
    the three k's. So the three k's are 40, 152, 512. Then the sum is 40 + 152 + 549 = 704?
    Wait, 40 + 152 = 192, 192 + 512 = 704.

So the sum is 704.
```

Strong model excerpt: derives each candidate and checks the domain

```
[...] Now check each:

$k=8$: quadratic $x^2 - 98 x + 1008$ has roots $x1\approx11.68>0$, x2\approx86.33>0$. Yes.
    Therefore, for $k=8$, $f(x;8)= m + \frac{(x-x1)^2 (x-x2)^2}{x}$ with $m= -10368$, and the
    global minimum over $x>0$ is $m$ attained exactly at $x1,x2$. Yes, matches (these are in
    the two negative intervals $(8,18)$ and $(72,98)$).

$k=32$: quadratic $x^2 - 110 x + 2016$ has roots $x1\approx23.24>0$, x2\approx86.76>0$. Yes.
    Global minimum $m= -7200$ attained exactly at these two points (intervals $(18,32)$ and $
    (72,98)$).

$k=200$: quadratic $x^2 - 194 x + 5040$ has roots $x1\approx30.90>0$, x2\approx163.10>0$. Yes.
    Global minimum $m= -194688$ attained exactly at these two points (intervals $(18,72)$ and $
```

```
    (98,200)$).

$k=512$: quadratic $x^2 - 350 x - 8064$ has roots $x1\approx371.70>0, x2\approx -21.70 < 0$.
    Therefore, for $k=512$, $f(x;512)= m + \frac{(x-x1)^2 (x-x2)^2}{x}$ with $x2<0 =>$ $f(x)>=
    m$ for all $x>0$ but equality only at $x1>0$. So global minimum attained at exactly one
    positive real number $x$ (and not two). Also we saw there is another local minimum but its
    value is higher than $m$.

Thus $k=512$ is excluded.

Now we have three values: $k=8, 32, 200$.

Also 'lets quickly ensure that for these $k$ there are no other points where $f(x)=m$ (the
    problem states "achieves its minimum value at exactly two positive real numbers "$x$).
    Since $f(x)-m = \frac{(x-x1)^2 (x-x2)^2}{x}$ and $x1!=x2$, equality only when $x=x1$ or $x=
    x2$. Yes, and both > 0.

Also check the domain is $x>0$.

Now we have three values: $k=8, 32, 200$, so the sum is 240. [...]
```

Weak models brute force, strong models find invariants. Consider the following AIME25 problem.

```
Let N denote the number of ordered triples of positive integers (a, b, c) such that a, b, c <=
    3^6 and a^3 + b^3 + c^3 is a multiple of 3^7. Find the remainder when N is divided by 1000.
```

The weak model attempts to simplify the congruence by assuming that cubing is a bijection on units modulo 3^7. This would turn the cubic condition into a linear one, but the assumption is false: $\gcd(3, \varphi(3^7)) = 3$, not 1. By contrast, the strong model identifies the relevant invariant: for powers of 3, unit cubes are not arbitrary units. They form an index-3 subgroup, characterized by congruence to $\pm 1 \pmod 9$.

Weak model excerpt: incorrect linearization

```
Alternatively, observe the condition mod 3: if 3 'doesnt divide some of them.

### Case 1: 3 does not divide a, 3 does not divide b, 3 does not divide c (all units mod3)

Then a,b,c are in $(\mathbb{Z}/3^7\mathbb{Z})^*$. And since x->x^3 is bijection, 'lets set $x \
    equiv u^3 \pmod{3^7}$, $y \equiv v^3 \pmod{3^7}$, $z \equiv w^3 \pmod{3^7}$ with u,v,w in $
    (\mathbb{Z}/3^7\mathbb{Z})^*$.
[..]
Actually, more carefully, since cubing is bijection on units, the map (a,b,c) mod3^7 -> (a^3,b
    ^3,c^3) mod3^7 is bijection on $(\mathbb{Z}/3^7\mathbb{Z})^* \times (\mathbb{Z}/3^7\mathbb{
    Z})^* \times (\mathbb{Z}/3^7\mathbb{Z})^*$. Therefore, the number of solutions with 3 \nmid
     a,b,c and a^3 + b^3 + c^3 \equiv 0 \pmod{3^7} is equal to the number of solutions with 3 \
    nmid a,b,c and a + b + c \equiv 0 \pmod{3^7}.
```

Strong model excerpt: identifying the invariant

```
Also recall the lifting the exponent / structure of cubes of units mod powers of 3: for $x \not
    \equiv 0 \pmod{3}$, 'lets see $x \pmod{3^s}$.
[...]
Note that for $x \not\equiv 0 \pmod{3}$, by 'Eulers theorem $x^{\phi(3^e)} = x^{2\cdot 3^{e-1}}
     \equiv 1 \pmod{3^e}$, and the multiplicative group mod $3^e$, $e\ge2$, is cyclic of order
    $2\cdot 3^{e-1}$. The map $x \mapsto x^3$ has image the unique subgroup of index 3.
[...]
```

```
Also, for $3 \nmid y$, 'lets see $y^3 \pmod{3^m}$.
[...]
Note that for $e\ge2$, $(\mathbb{Z}/3^e\mathbb{Z})^* = \langle -1 \rangle \times \langle 4 \
    rangle \cong C_2 \times C_{3^{e-1}}$. The element 4 has order $3^{e-1}$ mod $3^e$ (a
    generator of the 1-units).
[...]
Then $y \equiv (-1)^s 4^t \pmod{3^e}$, $s\in\{0,1\}, t\in\{0,\ldots,3^{e-1}-1\}$, and $y^3 \
    equiv (-1)^{3s} 4^{3t} \equiv (-1)^s (4^3)^t \pmod{3^e}$.
[...]
Also, the image $\{y^3 \pmod{3^e} : 3\nmid y\}$ is a subgroup of index 3 in $(\mathbb{Z}/3^e\
    mathbb{Z})^*$: it consists of elements $z$ with $z \equiv \pm 1 \pmod{9}$.
```

Strong models are skeptics. Another problem from AIME25. Here the strong model pauses to re-examine its own solution when in doubt ("Wait, let's re-examine"), catching the corner case that lets it answer correctly.

Weak model excerpt: labels the empty grid "maximal" and never rechecks

```
Can a placement with zero chips be maximal?
[...] So for an empty grid, can you add a chip? Yes, you can add a white chip in any cell. But
    in our case, if all rows are W and all columns are B, can you add a white chip? Let's see:
    to add a white chip in cell (i,j), the cell becomes white. So row i must be white, which is
     ok (row i is W). Column j must be white, but column j is B, so you can't add a white chip.
     Can you add a black chip? row i must be black, but row i is W, so no. So actually, if all
    rows are W and all columns are B, the empty grid is maximal (you can't add any chip). So
    placing zero chips is allowed.
[...] So total placements would be Case 1 + Case 2 + Case 3 = 960 + 32 + 32 = 1024.
```

Strong model excerpt: distrusts its own converse and tests it on a small case

```
Take the example: $s_i=W$ for all i, $t_j=B$ for all j => $S=\emptyset$, $c$ is empty function.
     Now take this $(S,c)=(\emptyset, \emptyset)$ (the empty placement). Can we place an
    additional chip in cell (1,1)? [...] choose $x=W$. Then row 1 has a white chip: ok. Column
    1 has a white chip: ok. Conditions 1,2 satisfied. Therefore, the placement $(\emptyset,\
    emptyset)$ does NOT satisfy condition 3.

But according to our general argument above we wrote: "Conversely, if we take any $s_i,t_j \in
    \{W,B\}$, define $S=\{s_i==t_j\}, c(i,j)=s_i$, then $(S,c)$ satisfies 3."
That's wrong in logic? Wait, let's re-examine that argument for the example.
[...]
### Small test case

Let's try a smaller case to check the logic, say $1\times 1$ grid, 1 white chip, 1 black chip.
    Find the number.
[...]
Now using our formula logic: grid $1\times 1 =>$ 1 row, 1 column.
Find $|Y| = $ number of $(s_1, t_1) \in \{W,B\}^2$ with $\{s_1\} = \{t_1\}$.
[...]
So pairs (W,W) and (B,B): 2 pairs.

Yes, that matches exactly
```

C.2 Agentic CoTs

Strong models write and run unit tests while weak models only do basic sanity checks. One behavior that emerges over the course of agentic RL training on SWE tasks is regularly running unit tests to verify work.

Weak model excerpt: does basic sanity checks to confirm the proposed solution addresses the requirements in the problem statement, but does not actually run unit tests.

```
Looking at the changes I've made, I've successfully implemented all the requirements from the
    problem statement. Let me review what I've accomplished:

For model.py:

- Added a SeedDict TypedDict class with a "key" field of type str
- Added proper type imports and type definitions (SeedSubjectString, SeedSubjectType, Seed)
- Created a _normalize_seed() helper method to consistently handle all seed formats
- Added explicit return types and input argument types to all public methods in List and Seed
    classes
- Refactored add_seed(), remove_seed(), and _index_of_seed() to support all seed formats and
    use normalized string keys for duplicate detection
- Fixed get_export_list() to return a dictionary with three keys ("authors", "works", "editions
    ") each containing fully loaded Thing instances
- Updated get_seeds() …
- Added type annotations to all Seed class methods …
- Enhanced the Seed class to handle "author" and "redirect" type strings
For lists.py:

- Added subject_key_to_seed() function …
- Added is_seed_subject_string() function …
The implementation improves type safety, code readability, and maintains backward compatibility.
     The seed handling logic is now more robust with consistent normalization, and the type
    annotations provide better static analysis support. All the requirements from the problem
    statement have been addressed.
```

Strong model excerpt: reviews problems requirements *and* runs unit tests to check work.

```
The tests still pass. So I have all the core functionality working. Let me look at the full
    picture. I need to consider if I've addressed all the requirements from the problem.

For the requirements:

 - All public methods in List and Seed have type annotations — I've added them to the key
    methods
 - SeedDict TypedDict defined in model.py
 - is_seed_subject_string function — I created it
 - subject_key_to_seed function handles normalization correctly … — let me verify
 - get_export_list returns correctly structured dict with lists of dicts
 - add_seed and remove_seed handle all formats and have better duplicate detection …
Overall, I think I've addressed the core requirements. All existing tests continue to pass,
    confirming backward compatibility.
```

Strong models do evidence archaeology vs weak models fixate edit-mechanics. Strong checkpoints during the climb usually have CoTs seeking repository evidence (reverted commit, payloads, tests) before patching. Weak CoT identifies the right target but spends later turns on exact edit mechanics and corruption recovery, with small scope and certain degrees of fixation.

Weak model excerpt: focus on a lot of targeted edits and editing ergonomic, taking small steps only.

```
- I need to start by exploring the codebase structure to understand how Leaflet is organized.
    Since the task involves adding `toGeoJSON()` methods to various layer types, I should first
    : 1. Understand the overall project structure 2. Locate the source files for these layer
    types (Marker, Polyline, Polygon, LayerGroup, etc.) 3. Understand the existing code
    patterns and conventions used in Leaflet Let me get a bird's-...
- Now I have a good understanding of the codebase. Let me plan the implementation: **
    Understanding the coordinate conversion:** - Leaflet uses `[lat, lng]` internally - GeoJSON
```

```
     requires `[lng, lat]` order - For Marker: `{type: 'Point', coordinates: [lng, lat]}` - For
     Polyline: `{type: 'LineString', coordinates: [[lng, lat], ...]}` - For Polygon: `{type: '
    Polygon', coordinates: [[[lng, lat], ...repeat first...], [[ho...
- I need to make sure my old_str exactly matches the content in the file. Let me view more of
    the end of the Marker.js file to see the exact structure and line endings.
- Now I see the exact structure. The file has mixed indentation (tabs are used in some places,
    spaces in others - I need to preserve the style). Let me add the `toGeoJSON()` method to
    Marker, following the same formatting style.
- Looking at the file again, I see it uses tabs for indentation rather than spaces. I need to
    match the exact tab indentation used in the file.
- Let me view the exact range from line 265 to the end to see the precise formatting.
- Now I can see the exact formatting with tabs (shown as spaces but they're likely tabs). Let
    me add the toGeoJSON method.
```

Strong model excerpt: fully explores the repository and collect extensive evidence without taking actions prematurely.

```
- Let me start by exploring the codebase. I need to understand how the check enforcer currently
     works - how it handles CI checks, webhooks, and processes GitHub events. This will help me
     understand what needs to be modified to support GitHub Actions as a first-class citizen
    alongside Azure Pipelines.
- Let me read the main code files - main.go, types.go, and other key files. I need to
    understand the architecture. The problem mentions Azure Pipelines check-suite handling,
    webhooks, and CI evaluation logic. Let me see how it's currently implemented.
- I see the structure. The code currently handles: 1. Issue comments that contain `/check-
    enforcer` commands 2. Check suite events from Azure Pipelines (app named "Azure Pipelines")
     But the problem says we need to handle: 1. GitHub Actions as a first-class signal (not
    just Azure Pipelines) 2. Multiple relevant suites may exist - currently GetCheckSuiteStatus
     in github_client.go I need to see 3. Workflow-run events -...
- I see in GetCheckSuiteStatus (line 105-138 in github_client.go) - it gets all check suites,
    then finds the first one that matches AppTarget (which is "Azure Pipelines" in the current
    code). But it returns just the first matching suite's status/conclusion. But the problem
    says "evaluate the overall commit status from the relevant suites rather than assuming
    'theres only one app or one event shape involved" and "mul...
- Looking at the codebase, I see the current implementation. Let me read more of the test files
     and understand what's expected. I see in main_test.go there are tests. Also I need to
    understand what the requirements are more precisely. Let me look at all the tests to
    understand what behavior is expected. I see in main_test.go, github_client_test.go,
    types_test.go. Let me also look at the README and docs for context.
- Let me continue reading the tests to understand the full expected behavior.
- I see the tests. Now let me look at the other test files to get a fuller picture. Also let me
     search for terms like "workflow_run", "GitHub Actions", to understand what the
    requirements are in the full problem context. Looking at the problem more carefully - I
    need to understand what "workflow-run events" are. And I need to understand how GitHub
    Actions integrates. Let me look at the full problem. Actually let me ...
```

Strong models seek source of truth, weak models speculate adjacent paths. Strong CoT locates the source data as source of truth. Weak CoT understands visibility and code paths on surface but reasons around adjacent generated-code paths and leaves an edge case.

Weak model excerpt: tracing step by step around the adjacent paths but mostly by trial and error.

```
- I see TypeExtendsReader collects all superclasses and interfaces. But I notice it's not
    checking visibility (public/package-private). Let me look more closely at the issue. The
    key is in understanding what "package protected class and interface" means. In Java, if a
    class is not public, it's package-private (accessible only within the package). Similarly
    ...
- Hmm, let me search more specifically for isAddBeanFor in test files.
```

```
- No direct references in tests. Let me look at how isAddBeanFor is defined in the API. Let me
    examine DBuilder and Builder interfaces.
- I see isAddBeanFor in the API. But I need to understand the full picture. Let me look at
    Builder interface.
- Now I understand the API side. But what about the generator side? Let me look at more
    examples of generated code to see the actual output pattern for isAddBeanFor. I need to see
     what TypeAppender produces. Let me look at test data or look at actual generated DI
    classes from test cases.
- Those are generator classes. Let me look for actual generated DI files in test targets.
- So the issue might be clearer if I understand when package-private types cause problems. Let
    me look at the issue more carefully. Let me search for any issues or more context about "
    package protected". Wait - let me look at the full TypeExtendsReader and understand better.
     Because when we collect types to register (for isAddBeanFor), we collect: 1. The ba...
```

Strong model excerpt: ties behavior to the data source that feeds generation and trace from there.

```
- I see the references. Let me look at how isAddBeanFor is generated. I need to understand the
    full picture.
- I see they're checking Modifiers. So there is a pattern. Let me look at how it's done. Let me
     search more comprehensively.
- I see isAddBeanFor is defined in Builder interface and implemented in DBuilder. Let me look
    at how it's used. Let me read DBuilder and understand the semantics. The isAddBeanFor
    determines if a bean should be added - I think it's used for "types" - as in, when
    registering, what types is this bean a candidate for? So only public types are relevant
    from con...
- I see how isAddBeanFor works in the runtime - it decides whether to add/register a bean based
     on if parent scope already has it. But that doesn't help me understand the issue. The
    issue is about generated code. Let me look at how the generated code actually looks. Let me
     find examples of generated D classes with isAddBeanFor. Let me search for actual gene...
- Those are runtime classes. Let me look at generated test code. Let me find generated code in
    test resources or build outputs. But this is source code - let me look at blackbox tests or
     inject-test. Let me search for generated code that has isAddBeanFor.
- So no generated code in target (no build yet). Let me look in test resources or at existing
    test cases. Let me search for 'isAddBeanFor' in test sources - perhaps I can understand
    from tests what the expected behavior is.
- I see MethodReader's buildAddFor also uses typeReader.getTypesRegister() when dealing with
    factory methods that return types. So the logic for determining what types go into
    isAddBeanFor comes from TypeReader. But back to the main issue - "Bug isAddBeanFor()
    generated code includes package protected class and interface. The fix is that only public
    classes...
```

D SWE Agent Tool Schema

We describe the tools used for our SWE agent RL in OpenAI-compatible function calling schema format in Figure 23 and Figure 24.

```
{
  "type": "function",
  "function": {
    "name": "bash",
    "description": "Execute bash commands in the context of the repository.\nDirectory or environment variable
     changes are not persistent. Every action is executed in a new subshell.\nHowever, you can prefix any
     action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir
&& ...` or write/load environment variables from files.\nYou have a full Linux shell environment.\nAlways use
     non-interactive flags (-y, -f) for commands.\nAvoid interactive tools like vi, nano, or any that require
     user input.\nYou can use bash commands or invoke any
tool that is available in the environment.\nYou can also create new tools or scripts to help you with the task
     .\nIf a tool isn't available, you can also install it.\n",
    "parameters": {
      "type": "object",
      "additionalProperties": false,
      "required": ["command"],
      "properties": {
        "command": {
          "type": "string",
          "description": "The bash command to execute. Can include pipes, redirects, and other shell features.
     Commands are executed in the repository working directory."
        }
      }
    }
  }
}
```

Figure 23. Bash Tool Schema.

```
{
  "type": "function",
  "function": {
    "name": "str_replace_editor",
    "description": "Custom editing tool for viewing, creating and editing files.\n* State is persistent across
     command calls and discussions with the user\n* If `path` is a file, `view` displays the result of applying
      `cat -n`. If `path` is a directory, `view` lists
non-hidden files and directories up to 2 levels deep\n* The `create` command cannot be used if the specified `
     path` already exists as a file\n* If a `command` generates a long output, it will be truncated and marked
     with `<response clipped>`\n* The `undo_edit` command
will revert the last edit made to the file at `path`\n\nNotes for using the `str_replace` command:\n* The `
     old_str` parameter should match EXACTLY one or more consecutive lines from the original file. Be mindful
     of whitespaces!\n* If the `old_str` parameter is not
unique in the file, the replacement will not be performed. Make sure to include enough context in `old_str` to
     make it unique\n* The `new_str` parameter should contain the edited lines that should replace the `old_str
     `",
    "parameters": {
      "type": "object",
      "additionalProperties": false,
      "required": ["command", "path"],
      "properties": {
        "command": {
          "type": "string",
          "description": "The commands to run. Allowed options are: `view`, `create`, `str_replace`, `insert`,
     `undo_edit`.",
          "enum": ["view", "create", "str_replace", "insert", "undo_edit"]
        },
        "path": {
          "type": "string",
          "description": "Absolute path to file or directory, e.g. `/testbed/file.py` or `/testbed`."
        },
        "file_text": {
          "type": ["string", "null"],
          "default": null,
          "description": "Required parameter of `create` command, with the content of the file to be created."
        },
        "view_range": {
          "type": ["array", "null"],
          "default": null,
          "items": { "type": "integer" },
          "minItems": 2,
          "maxItems": 2,
          "description": "Optional parameter of `view` command when `path` points to a file. If none is given,
     the full file is shown. If provided, the file will be shown in the indicated line number range, e.g. [11,
     12] will show lines 11 and 12. Indexing at 1 to
start. Setting [start_line, -1] shows all lines from start_line to the end of the file."
        },
        "old_str": {
          "type": ["string", "null"],
          "default": null,
          "description": "Required parameter of `str_replace` command containing the string in `path` to
     replace."
        },
        "new_str": {
          "type": ["string", "null"],
          "default": null,
          "description": "Optional parameter of `str_replace` command containing the new string (if not given,
     no string will be added). Required parameter of `insert` command containing the string to insert."
        },
        "insert_line": {
          "type": ["integer", "null"],
          "default": null,
          "description": "Required parameter of `insert` command. The `new_str` will be inserted AFTER the line
      `insert_line` of `path`."
        }
      }
    }
  }
}
```


Figure 24. String Replace Editor Tool Schema.

E Constraint Taxonomy for Instruction Following Data

We build a constraint taxonomy to organize the instructions a model may be asked to follow. It serves two purposes: guiding data collection toward broad coverage, including rare constraint combinations, and allowing to analyze model performance at a fine-grained level. We split constraints by how they are checked, matching the reward types from Sec. 3.4.1.

Objective (hard) constraints. These are rule-based constraints that trigger Python functions from a catalog of custom and open-source checkers (Pyatkin et al., 2026; Zhou et al., 2023). Examples include numerical constraints (e.g., "respond in exactly 3 sentences"), format constraints (e.g., "output as a JSON object with keys `name` and `age`"), and linguistic constraints (e.g., "respond in French").

Subjective (soft) constraints. These need judgment to evaluate and are scored by the reward model or AI judges. Examples include tone (e.g., "be encouraging but not patronizing"), persona (e.g., "respond as a history scholar"), and interaction behavior (e.g., "if unclear, ask clarifying questions before answering"). Tab. 15 shows example subcategories.

Multi-turn scenario taxonomy. Inspired by Deshpande et al. (2025), we break down dialogue use cases to various scenarios that test model's IF ability in complex conversations involving multiple turns as shown in Tab. 16.

F Infrastructure for Building SWE Environments

The infrastructure is designed primarily to address the scale, stability, and safety challenges. The build infrastructure is organized as a two-pool Ray cluster running on approximately 30,000 CPU cores. The main pool handles work dispatching, Lance I/O, and pipeline state tracking. The builder pool runs the actual container builds; all builder nodes are tagged with a Ray custom resource flag, and the build step uses Ray's logical resource scheduling to ensure its actors land exclusively on builder-capable nodes.

The two pools are separated because container builds are resource-intensive and failure-prone (disk exhaustion, OOM, build hangs), while coordination is lightweight and stateless; isolating them ensures that builder pod failures do not disrupt orchestration and state management. The separation also enforces credential isolation: builder pods execute untrusted Dockerfiles from open-source repositories and do not hold credentials for the pipeline's data storage or internal services. Pod failures—whether from disk pressure, hardware faults, or node evictions—are handled by Kubernetes restart; the restarted pod reconnects to the Ray cluster and is automatically picked up when the next main pool worker schedules a new task.

Each builder pod contains two containers sharing NVMe-backed local storage, co-located for build and grading efficiency. The main container runs the Ray worker with rootless podman for loading built images and running grading containers. The sidecar container runs a rootless BuildKit daemon for OCI image builds via `buildctl`. The two containers communicate through the shared NVMe volume—BuildKit writes OCI tar archives that podman loads—and via localhost for the BuildKit control channel. This co-location avoids network transfer of large container images between build and grading; the entire build-load-run cycle happens on local NVMe. Init containers handle one-time setup including authentication for container registries via federated identity.

The pipeline also exploits BuildKit layer caching to reduce redundant work across same-repo problems. Problems are grouped by repository and dispatched to the same persistent Ray actor on a builder node. Since the actor persists across same-repo problems, shared base layers—`apt-get install`, `pip install`, dependency compilation—are built once and reused, as dependency installation often dominates Docker build

Constraint Type	Subcategories & Examples
Objective (Hard) Constraints	
Numerical	Explicit length limits, sentence/paragraph counts, item counts, range constraints, and ratios.
Format	Document structure, lists, tables, code formatting, data serialization (JSON/YAML), markup/styling, templates, and citation formats.
Content	Required/prohibited terms and symbols, required elements or concepts, source/citation requirements, and temporal/geographic scope.
Linguistic	Language selection, dialect, register, sentence structure, parts of speech, morphological constraints, and grammar rules.
Temporal & sequential	Chronological ordering, procedural steps, and alphabetical order.
Negative constraints	instructions to exclude certain behaviors or formatting pattern
Subjective (Soft) Constraints	
Stylistic	Tone, voice, vocabulary level, audience adaptation, emotional valence, and brand voice.
Persona & self-presentation	Role adoption, personality traits, values, and emotional state.
Interaction & response policy	Conversation navigation, certainty/hedging, clarification strategies, solicitation behaviors, and prosocial behaviors.
Structure & reasoning	Reasoning process requirements, argumentation structure, conditional branching, and scope constraints.
Instruction handling	Conditionality, multi-step instruction chains, conflict handling.

Table 15. Examples of high-level constraints from our taxonomy, divided into objective (hard) and subjective (soft) categories with corresponding subcategories and examples.

time for large repositories.

The primary throughput bottleneck is LLM token consumption rather than CPU or disk. The pipeline runs on 10,000 CPU cores with 12 million tokens per minute of LLM capacity, achieving an 83% prompt cache hit rate. Under this configuration, the pipeline produces approximately 20 grading-passed environments per minute.

G STEM Evaluations Setup

For all STEM benchmarks, we report pass@1 performance using a maximum output token budget of 256k tokens. We show how our model performs at 128k maximum output tokens in Tab. 17 in addition to the already presented numbers at 256k.

G.1 Math

The American Invitational Mathematics Examination (AIME) and the Harvard–MIT Mathematics Tournament (HMMT) are prestigious high school mathematics competitions. These competition-math evaluations serve as useful proxies for a model's reasoning ability. We evaluate MAI-Thinking-1 on the 2025 and 2026 editions of AIME, as well as the February 2026 HMMT benchmark from MathArena (Dekoninck et al.,

Capability	Description
Context switching	Model's ability to track when the user switches topics or returns to a previous topic.
Instruction retention	Ability to track, update and retrieve instructions and facts across long dialogues.
Instruction layering	Ability to maintain and retain a set of active constraints with instructions introduced over multiple previous turns.
Memory & recall	Proper retrieval and use of facts and statements from previous turns.
Self coherence	Avoid unintended contradiction of previous statements.
Version editing	Ability to revert to an earlier version, selectively undo edits, or reproduce exact text from a prior turn.

Table 16. Example of IF capabilities specific to multi-turn.

Benchmark	MAI-Thinking-1		Sonnet 4.6	Opus 4.6	GPT 5.4	Kimi K2.6	DeepSeek V3.2	DeepSeek V4	GLM-5.1
	128k	256k							
AIME 2025	95.0	97.0	95.6	99.8	—	—	93.1	—	—
AIME 2026	93.6	94.5	—	—	—	96.4	—	—	95.3
HMMT Feb 2026	84.3	84.9	—	—	—	92.7	—	95.2	82.6
GPQA Diamond	84.2	84.2	89.9	91.3	92.8	90.5	82.4	90.1	86.2
LCB v6	87.3	87.7	—	—	—	89.6	83.3	93.5	—

Table 17. Post-trained model evaluation results on STEM benchmarks, comparing MAI-Thinking-1 evaluated at 128k vs. 256k maximum output tokens. Other model numbers are taken from respective official model cards. Models listed with "—" scores do not report numbers for the respective benchmark.

2026). The expected answers for these competitions are simple numbers or expressions. We require the model to output its final answer in boxed format, which enables regex-based extraction followed by SymPy verification, with an AI judge used as a fallback.

G.2 Science

Graduate-Level Google-Proof Q&A benchmark (GPQA) (Rein et al., 2023) comprise knowledge-intensive, graduate- and research-level questions, primarily in STEM domains. For GPQA, we evaluate on the Diamond subset, which contains 198 multiple-choice questions. We adopt the instruction prompt from OpenAI's simple-evals package (OpenAI, 2025) to specify the required answer format, and use a custom regex-based extraction procedure to compare model outputs against the ground-truth answers. We report pass@1 performance averaged over 16 rollouts. We use GPT-5-mini as the default judge for AI grading with the prompt described in Fig. 25 to grade model responses on STEM benchmarks. GPT-5mini is used as the default judge.

G.3 Competitive Coding

LiveCodeBench (LCB) (Jain et al., 2024) contains up-to-date competitive coding problems. We evaluated on the v6 split, which contains 1,055 problems released between May 2023 and Apr 2025. Unlike the agentic coding evaluations below which are multi-turn, LCB is a one-shot test setting. The solution is generated in

```
SYSTEM_PROMPT = """Compare two responses, [RESPONSE] and [CORRECT_ANSWER] to a question. Follow the rubric
     below in [NOTES]. Then output [FINAL JUDGMENT].

Some general guidelines to follow in determining alignment:
- Units: If the question does NOT explicitly require units/formatting, then if one response provides units (
     like km/s) and the other does not, but both have the correct numerical value, you can consider both
     responses equally correct. If the question explicitly requires units, missing/incorrect units is NOT
     acceptable.
- Formatting: If the question does NOT explicitly require a specific format, then Latex vs plain text or
     markdown does not matter as long as the answer is equivalent. If the question explicitly requires a
     specific format/structure, deviations are NOT acceptable.
- Rounding: If the question does NOT explicitly require an exact value or specific rounding/precision, small
     rounding differences are acceptable when clearly equivalent (e.g., 9.42 vs 9.43 with the same intended
     precision). If the question explicitly requires exactness or a specified precision, you must enforce it.

Judgment guidance:
- If [RESPONSE] is exactly identical to [CORRECT_ANSWER] or differs only by harmless formatting, you should
     indicate that (alignment) is met in the [NOTES].
- If [RESPONSE] is not exactly identical to but is equivalent to, or can be straightforwardly reduced/
     simplified to, [CORRECT_ANSWER] (without adding new assumptions or changing meaning), you should still
     indicate that (alignment) is met in the [NOTES].
- If [RESPONSE] clearly does not align with the [CORRECT_ANSWER], you should indicate that (alignment) is NOT
     met in the [NOTES] and output [FINAL JUDGMENT] = no.

Examples:
1) [RESPONSE]
Answer: 10
[CORRECT_ANSWER]
10
-> in [NOTES], (alignment) is met, and [FINAL JUDGMENT] = yes.
2) [RESPONSE]
X=\frac{\pi^2}{3}
[CORRECT_ANSWER]
\frac{\pi^2}{3}
-> in [NOTES], (alignment) is met, and [FINAL JUDGMENT] = yes.

[NOTES]
(alignment) How well do the CONTENTS of [RESPONSE] align with the CONTENTS of [CORRECT_ANSWER]? Note any true
     mathematical/numerical differences.
(final) Briefly summarize whether [RESPONSE] should be treated as correct overall, based on alignment.

[FINAL JUDGMENT]
Should [RESPONSE] be considered as correct as [CORRECT_ANSWER] (alignment satisfied)? Answer in yes/no."""

USER_PROMPT = """[QUESTION]
{question}

[RESPONSE]
{response}

[CORRECT_ANSWER]
{correct_answer}"""
```

Figure 25. AI judge prompt for STEM problems. We include additional few-shot examples in the conversation history as prior user and assistant turns to improve grading reliability and reduce susceptibility to judge hacking.

code fences for parsing and executed against the test harness for grading. Grading has pre-defined memory and runtime limits, similar to the constraints of real coding competitions.

H Agentic Coding Evaluations

Agentic coding benchmark results for SWE-bench Verified (Chowdhury et al., 2024), SWE-Bench Pro (Deng et al., 2025), and Terminal-Bench 2.0 (Merrill et al., 2026) are shown in the second section of Tab. 11. Unlike the STEM evaluations from the previous section, agentic coding evaluations are multi-turn and require models to interact with environments. For all three benchmarks, we evaluate our model with a very simple, ReAct-style (Yao et al., 2023), always append agent loop described in Figure 18. At each turn, the model can generate tool calls where we execute in the environment and append the output to the trajectory before the model makes the next tool call. We evaluate with a total context length of 256k tokens, a maximum output length of 8k tokens for SWE-bench Verified and SWE-Bench Pro, 32k tokens for Terminal-Bench 2.0, and a maximum of 1,000 steps. The agent loop terminates when the model outputs no tool calls, reaches the context length limit, or reaches the step limit. After the loop terminates, the respective graders are run inside the SEE used by the agent. For SWE-bench Verified and SWE-Bench Pro, we evaluate with the same two tools used during climbing, bash and string replace editor, as described in Sec. 3.3.1. For Terminal-Bench 2.0 we evaluate with only the bash tool.

SWE-bench Verified. SWE-bench Verified is a 500-task human-curated subset of SWE-Bench (Jimenez et al., 2024) selected to increase reliability. Each task is a real public GitHub issue from a popular Python repository, and the model must generate a patch that resolves the issue and passes the repository's hidden test suite. Tasks have been reviewed to ensure problem statements are solvable and tests fairly assess correctness.

SWE-Bench Pro. SWE-Bench Pro is a more challenging, 731-task successor to SWE-bench Verified, with same unit-test based grading mechanisms, designed to evaluate LLMs on professional-grade software engineering tasks drawn from real-world open-source repositories. It emphasizes harder, multi-file edits, longer-horizon reasoning, and complex bug fixes or feature implementations that better reflect the work of practicing engineers. More importantly, SWE-Bench Pro contains repositories in languages other than Python unlike SWE-bench Verified.

Terminal-Bench 2.0 Terminal-Bench 2.0 is a human-verified, substantially harder successor to Terminal-Bench (Merrill et al., 2026), comprising 89 realistic tasks across software engineering, debugging, data science, machine learning, security, and system administration. Each task runs in a sandboxed terminal environment, and is graded programatically based on output or environment final state. Tasks are designed to require long-horizon planning, error recovery, and genuine command-line fluency rather than pattern-matching.

I Safety Evaluations

I.1 Internal Evaluation Details

I.1.1 Methodology

Evaluation proceeds in two stages: a structured characterization of the user request, followed by the derivation of a response specification against which the model's behavior is graded. This decomposition allows us

Dimension	Description
Harm category & policy violation	Identifies which policy area the request engages and whether it violates a specific rule, grounding the judgment in a concrete policy clause.
Sensitivity level	Records topical sensitivity on an ordinal scale based on scope (individual vs. broad) and severity (worst-case misuse outcome), independent of whether the request is disallowed.
User intent	Binary signal: *good intent possible* vs. *malicious intent*. Dangerous topics alone are not evidence of malice.
User scenario	Captures the user's pragmatic goal (e.g., information-seeking, guidance, creative generation, scenario play).
Jailbreak detection	Dual role: screens for adversarial wrapping and provides a signal for whether the request was adversarially framed.

Table 18. User request classification dimensions.

to distinguish genuine safety defects from over-cautious behavior on benign-but-sensitive topics and exposes per-policy and per-modality measurements rather than a single aggregate score.

Both evaluation stages are carried out by LLM judges, with classification steps executed multiple times and aggregated by majority vote to control judge variance. Each user request also goes through jailbreak detection which screens for adversarial wrapping using a hierarchical taxonomy of attack families and technique variants. When a wrapper is detected, an extraction step recovers the underlying request, and subsequent classification operates on that recovered request. This ensures evaluation reflects model behavior on the actual underlying intent rather than on the surface attack template, preventing wrapper-specific overfitting from inflating safety scores.

Stage 1: user request classification. Each user request is characterized along five complementary dimensions as described in Tab. 18. They jointly capture both the policy-relevant content of the request and the context in which the request is made. For multi-turn dialogues, the entire dialogue history is treated as contextual input.

Stage 2: response strategy generation. The classification from Stage 1 is consumed by a rubric judge that emits a per-request response specification covering the relevant policy constraints, the appropriate refusal posture (which may be compliance, partial refusal, or refusal depending on intent and severity), and the prescribed tone and approach for the response. For requests classified as harmless or unclear, this LLM-generated rubric is replaced by a deterministic default that explicitly penalizes hedging and unwarranted refusal.

I.1.2 Data Collection & Sampling

Desiderata: Evaluation results are only as informative as the prompts they are computed on: a narrow, mis-weighted, or saturated evaluation set yields tight confidence intervals around the wrong quantity, and any model-selection decision made on top of it inherits that error. We treat construction of the evaluation set as a methodological problem and the set must satisfy three properties (Atkinson et al., 2007).

- **Coverage** across the policy areas, sensitivity levels, user scenarios, and adversarial framings introduced above, so that no behaviorally distinct slice of inputs is silently underrepresented.

- **Statistical efficiency** under a fixed evaluation budget, since judging is expensive and confidence intervals on per-category metrics tighten slowly with sample size.
- **Discriminative power** when comparing against other frontier models, so that aggregate scores actually move when a model improves rather than saturating against trivially-easy or trivially-hard prompts.

Source pool and annotation: The candidate prompt pool is assembled from three complementary sources: public safety benchmarks, consumer Copilot logs with PII filtered, and prompts collected by external vendors against our policy taxonomy. Every candidate prompt is passed through the annotation framework described above, attaching labels for the 5 strata that will be used for sampling: harm category, policy violation, sensitivity, user scenario, and jailbreaks.

Informativeness-weighted experiment design. From this pool of candidate prompts, we seek to select the most informative prompts to evaluate on. We approach this in the framing of an automated test assembly problem. This aligns with recent efforts to drastically compress LLM evaluation costs via psychometric modeling, such as *tinyBenchmarks* (Polo et al., 2024), which utilizes Item Response Theory to identify static, universally representative evaluation subsets. As an extension to this framework, we upweight prompts where the model is not always failing or always passing. This helps us to proactively identify areas where we're most likely to be able to make improvements to the model.

Threshold setting. The evaluation set described above is the primary mechanism by which safety considerations enter MAI-Thinking-1's hill-climbing process. Every internal release candidate must pass an acceptance threshold on a fixed bundle of safety evaluations and must do so without regressing against the previous accepted release. The bundle covers both targeted policy-area performance and aggregate behavior on an unbiased leaderboard set, so that a regression localized to a single harm category cannot be averaged away by improvements elsewhere. Thresholds enter the process at two points: as gates at release-candidate sign-off, where a candidate falling below threshold on any bundle metric is not eligible for promotion into release recipe, and also to trigger advance warning alerts if experimental models fall below a lower threshold.

The thresholds themselves are derived from a Pareto analysis over the achievable trade-off surface between safety, over-refusal, and downstream model quality: for each release cycle we plot the frontier of recipes evaluated during that cycle and identify the elbow at which marginal safety gains begin to extract a disproportionate cost in over-refusal or quality. The threshold is then placed at a fixed percentile along this frontier rather than at a static absolute number, so the bar tracks what is currently achievable on the architecture and data mix rather than a stale historical floor. This makes the gating signal stable in meaning across release cycles — passing the bar provides the same evidence of safety quality, relative to the current frontier, regardless of how that frontier has shifted — while leaving room for the absolute numbers to rise as the underlying recipes improve.

J General Capabilities Evaluations

Below we discuss benchmarks covering knowledge, instruction following, long context, safety, honesty, health, and tool calling are shown in Tab. 12.

J.1 Knowledge

We evaluate MAI-Thinking-1 on SimpleQA Verified (Haas et al., 2026) and MMLU-Pro (Wang et al., 2024b) to assess its knowledge and reasoning capabilities. The results are reported in Tab. 12, alongside comparisons with other models.

MMLU-Pro. MMLU-Pro extends MMLU (Hendrycks et al., 2021) by incorporating more challenging, reasoning-focused questions and increasing the number of answer choices from 4 to 10. It evaluates models across a broad set of difficult knowledge and reasoning tasks. We adopt the instruction prompt from OpenAI's simple-evals package (OpenAI, 2025) to specify the required answer format, and use a custom regex-based extraction procedure to compare model outputs against the ground-truth answers.

SimpleQA Verified. SimpleQA (Wei et al., 2024b) evaluates short-form factual knowledge on challenging questions with short, unambiguous answers covering a variety of topics. We report model accuracy on SimpleQA Verified (Haas et al., 2026), a curated subset that addresses label noise and redundancy.

J.2 Instruction Following

A key aspect of model usability is how well it responds to user instructions. We have a suite of internal benchmarks, though we report numbers for three common public benchmarks below.

- **IFBench** (Pyatkin et al., 2026) evaluates the ability to follow precise instructions and covers 58 diverse and verifiable constraints.
- **AdvancedIF** (He et al., 2025) measures complex, multi-turn, and system-prompted instruction following abilities using rubrics and a calibrated LLM judge.
- **MultiChallenge** (Deshpande et al., 2025) targets multi-turn conversations and requires accurate instruction following, context allocation, and in-context reasoning.[11] We verified that for public models, we match the scores reported on the official leaderboard with our implementation.

J.3 Long Context

Long context performance for MAI-Thinking-1 is evaluated across four benchmarks shown in Table 12 and Table 19. We omitted MRCR (Vodrahalli et al., 2024) for reasons discussed below.

GraphWalks. GraphWalks is a synthetic multi-hop reasoning benchmark where the model is given an edge list representation of a graph and is asked to traverse the graph to find either neighboring nodes (via Breadth-First-Search) or parent nodes for a given starting node. The overall score report in Table 12 is the F1 score of model-predicted answer versus the ground truth. We report GraphWalks results on the $\leq$128k subset as measured by `o200k_base` tokeniser.

LongBenchV2. LongBenchV2 (Bai et al., 2024) consists of 4-way multiple-choice questions for long context understanding tasks. We use the same eval setup as the official implementation, with a caveat that the input context length is limited to 256k tokens, resulting in 408 unique questions.

CorpusQA. CorpusQA (Lu et al., 2025) is a multi-document freeform QA data where a model response is scored by an AI judge. Instead of using the default `deepseek-v3` (DeepSeek-AI et al., 2025a) as judge, we use GPT-5.4 (high) which offers higher precision.

[11] We followed the official benchmark's judge model, Gemini 2.5 Pro as the judge. https://labs.scale.com/leaderboard/multichallenge

Model	LongBenchV2	CorpusQA
MAI-Thinking-1	61	82
Sonnet 4.6	66	79

Table 19. Additional Long Context Benchmarks. Sonnet 4.6 results were generated via our own independent evaluation. All models were evaluated using maximum 128k output tokens with maximum reasoning effort (when applicable). Results were averaged from 4 independent evaluations.

MRCR. OpenAI's MRCR (Vodrahalli et al., 2024) is an artificially constructed benchmark which tests a model's capability to perform counting and copying over long contexts. While an ideal model should score well, the nature of the tasks differs substantially from natural user queries. We found that, without targeted training, MAI-Thinking-1 performs poorly on this task compared to existing models, achieving 60% on `avg_similarity@256K`, compared with a state-of-the-art result of 95%. In contrast, a small set of 1,000 synthetically generated in-distribution training examples boosts MRCR performance to 90%+ on a much smaller model in the MAI-Base-1 family, suggesting the benchmark's susceptibility to overfitting. As we do not have evidence that training with such targeted data leads to more general model improvements, we decided to drop MRCR from the set of long-context benchmarks we prioritized.

J.4 Safety

AIR-Bench. AIR-Bench (Zeng et al., 2024b) gives an aggregate read on policy-grounded refusal across regulatory and policy-derived harm taxonomies. Evaluation uses category-specific LLM judge prompts customized to each risk category and the scoring is designed to reward safe engagement rather than only penalizing unsafe responses. MAI-Thinking-1 achieves the same performance as Sonnet 4.6 on the benchmark.

CyberSecEval. CyberSecEval 4 (Wan et al., 2024; Meta, 2024) is an evaluation suite covering a variety of cybersecurity-related capabilities. Here, we focus on insecure code generation. The Instruct benchmark presents coding requests designed to elicit known insecure patterns, while the Autocomplete benchmark prompts the model with code context leading up to a known insecure pattern for the model to complete. Both measure whether the model produces vulnerable code in response using static analysis rules. MAI-Thinking-1 outperforms Sonnet 4.6 on the Autocomplete benchmark, and achieves similar performance on Instruct.

J.5 Honesty

To measure the honesty of MAI-Thinking-1, we selected benchmarks covering two distinct failure modes: repeating plausible-sounding falsehoods in short-form questions, and introducing unsupported claims over extended generations. MAI-Thinking-1 performs on par with Sonnet 4.6 in both benchmarks.

TruthfulQA. TruthfulQA (Lin et al., 2022) evaluates resistance to popular misconceptions, where each question is designed to elicit plausible yet false answers. For reproducible results, we report model accuracy over the dataset in the recommended multiple-choice setting.

LongFact. LongFact (Wei et al., 2024c) evaluates factual precision over longer generations containing multiple claims; we adopt the simplified claim extraction and LLM judge protocol from OpenAI (Singh et al., 2025a) in place of the original SAFE pipeline, retaining the LongFact prompt set and reporting claim-level precision.

J.6 Health

HealthBench Professional. HealthBench (Arora et al., 2025) is a suite of benchmarks developed to assess LLMs' capabilities in continuing realistic health conversations with individuals and health professionals. Language models are prompted with a multi-turn conversation, and asked to generate a response as a continuation of the existing conversation thread. The response is assessed with a calibrated model-based judge against a conversation-specific rubric created by a panel of physicians, covering clinical correctness, guideline adherence, and safety. We report performance on the most recent HealthBench variant, HealthBench Professional, which focuses on 525 conversations between medical experts and language models (Hicks et al., 2026). HealthBench Professional introduces a length penalty for the primary metric, to correct for a well-observed correlation between lengthy responses and artificially increased LLM-grader scores. For all reported scores, we use the standard GPT-5.4 grader and rubrics provided by OpenAI.

MedXpertQA. MedXpertQA Zuo et al. (2025) is a challenging, unsaturated multiple-choice benchmark which evaluates specialist-level medical knowledge via 2,450 expert-curated multiple-choice questions. These questions are designed to evaluate advanced clinical knowledge and multi-step reasoning. Each multiple choice question has 10 answer choices (A-J). We follow the evaluation methodology introduced in Muse Spark Team (2026), which uses a secondary LLM (GPT-5.4) to parse the predicted answer letter from free-form text.

J.7 Tool Calling

BFCL v3 (Patil et al., 2025) benchmarks LLMs' tool-use and function-calling abilities across single-turn, parallel, and multi-function scenarios, as well as multi-turn and multi-step agent interactions. It evaluates function selection and argument generation using abstract syntax tree and execution result matching, while stressing long-horizon reasoning, cross-turn state tracking, and robustness to noisy or missing functions. For better determinism, we use $T = 0.001$ and $p = 0.97$ for inference as recommended by the official leaderboard.

K Cluster Environment Details

This appendix describes the infrastructure mechanisms that made the training environment usable at frontier scale. The central design choice was to expose physical topology and hardware health as first-class scheduling state. Nodes and NVLink rack domains were represented through topology labels, reservation objects, rack-level services, and certification gates. A node was not considered useful simply because it had been provisioned; it became useful only when it was healthy, topologically valid, observable, and recoverable.

K.1 Hardware and Compute Clusters

MAI-Thinking-1 was developed on a heterogeneous accelerator fleet spanning NVIDIA H100, GB200, and GB300 systems. The main pre-training run, however, was placed on a single GB200 cluster at one site. The GB200 and GB300 clusters are hosted in Microsoft first-party datacenters and exposed to MAI through custom images co-developed with Azure teams. These systems are provisioned as rack-scale NVL72 units, with each rack providing a 72-GPU NVLink domain for high-bandwidth scale-up communication, while inter-rack RDMA used InfiniBand. H100 systems also remained part of the lab environment, using 8-GPU nodes with node-local NVLink/NVSwitch and InfiniBand for scale-out communication.

Logical cluster provisioning and management. Each site is partitioned into Kubernetes clusters, typically one per datacenter building. Logical clusters include GPU nodes for training and CPU nodes for support services. Custom controllers reconcile desired logical-cluster state with the underlying Azure resources, maintain node topology labels, and integrate external health databases into scheduling state.

Topology labels are assigned when a node starts, using the physical host identity. These labels represent rack, host, and locality information and are consumed by scheduling, reservation, and rack-level services. This makes physical placement visible to the control plane, allowing jobs to be placed on capacity that is not only available, but also topologically appropriate.

Large jobs are designed to span beyond the boundary of a single Kubernetes cluster when required. Nodes are universally routable across the compute environment, and workload pods use host networking to avoid unnecessary overlay overhead.

K.2 Cluster Readiness and Certification

At thousands of GPUs, failures are expected. Certification therefore serves as the first reliability boundary. Its purpose is to prevent bad nodes, degraded links, marginal storage, and silent-corruption risks from entering the production training pool. The certification framework is automated, Kubernetes-native, and hierarchical. Hardware must pass progressively broader tests before becoming schedulable: single-node diagnostics, rack-level multi-node collectives, and selected cross-rack InfiniBand validation. New nodes and remediated nodes follow the same path. Nodes returning from repair must pass certification again before re-entering production, preventing repeat offenders from cycling back into large training jobs.

Certification. Our diagnostics suite validates each node in isolation. Unlike generic health checks, these tests stress individual components for extended periods: GPUs, CPU cores, HCAs, NVLink links, and main memory which isolate high ECC rates, irregular throttle behavior, GPU clock violations, link flaps, and other marginal conditions before the node is admitted to production.

The suite also uses NCCL collectives to stress both intra-node and inter-node communication paths per rack and identify faulty NVLink behavior. These automated tests were instrumental in stabilizing the GB200 and GB300 racks where we identified multiple racks with fewer than 16 healthy nodes; those racks were excluded from large training runs.

The final stage of certification included cross-rack InfiniBand tests exercising fabric paths that single-node and rack-level tests cannot cover. These tests group nodes across racks, rails and leaf groups to validate spine-layer behavior, path diversity, and cross-rack RDMA performance. Because they consume a larger blast radius, they are run on demand during bring-up, incident investigation, and fabric validation.

Node lifecycle. Certification is one stage in a closed-loop node lifecycle as illustrated in Fig. 26. A new or remediated node enters **Init**, where it is tainted, initialized, labeled, and configured. The certification controller then runs the tiered test suite. Passing nodes become **Available**, while failing nodes become **Impaired**. Runtime monitoring can also move an **Available** node to **Impaired** when NPD conditions, XID errors, ECC thresholds, NVLink degradation, InfiniBand link flaps, or storage faults are detected. Transient issues enter **Auto Remediation**, such as reboot or soft drain. Persistent or hardware-attributed issues are routed through **Guest Health Reporting** to vendor or datacenter maintenance. Repaired nodes return to **Init** and repeated certification. Unrecoverable nodes are decommissioned.

K.3 Scheduling, Orchestration, and Control Plane

The scheduling problem is the placement of heterogeneous workloads across heterogeneous capacity without sacrificing locality, quota isolation, or recovery. Workloads include long pre-training jobs, post-training and

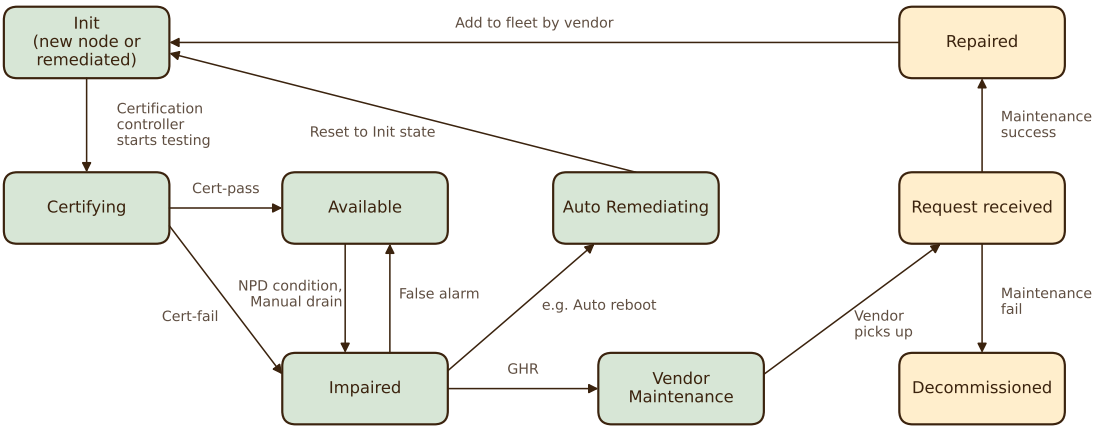

Figure 26. Node lifecycle from initialization through certification, production use, remediation, maintenance, and decommissioning.

reinforcement learning runs, inference and evaluation jobs, and CPU-heavy data pipelines. The stack is therefore layered: Kubernetes maintains cluster state, Kueue makes admission and quota decisions, MAI controllers maintain reservations and topology readiness, Ray executes distributed jobs, and a fleet-wide control plane unifies visibility across clusters and scheduler backends.

Cluster-local control plane. The cluster-local MAI control plane manages reservations, rack topology, quota coherence, and scheduling-readiness gates. It does not replace the scheduler; instead, it maintains the state that the scheduler needs. Kueue handles priority, quota, admission, preemption, and topology-aware placement. When a workload is admitted, Kueue attempts to preserve locality over high-bandwidth domains so that large jobs land in compact topology regions rather than scattered capacity. Rack fragmentation is a recurring scheduling risk. Small jobs from many queues can fill racks in a way that prevents later large jobs from acquiring contiguous capacity. The cluster-local control plane addresses this with soft rack reservations. Queues are assigned preferred racks, and workloads landed on those racks first. Borrowing remains possible when capacity was idle, but `reclaimWithinCohort` allows a queue to reclaim its reserved racks when needed. This improves bin packing under load while preserving utilization during low demand.

Ray runtime. Ray provides the distributed runtime inside admitted jobs. MAI drivers translate the admitted topology into actor placement, communication groups, and NCCL clique configuration. Pre-training jobs require strict learner availability, while RL jobs manage multiple actor types, including learners, inference servers, rollout workers, and routers, each with distinct placement and fault-tolerance needs. The drivers monitor actor liveness, coordinated training loops, and maintain checkpoint consistency across asynchronous components.

K.4 Observability, Telemetry, and Fleet Monitoring

We combined low-level hardware telemetry, scheduler state, job metadata, and fleet-efficiency signals into a single operating view. GPU health was tracked through XIDs, ECC, thermals, power, clock throttling, NVLink state, NVLink bit-error rate, chip-to-chip links, InfiniBand device state, local NVMe health, PCIe errors, and driver state. These signals were converted into Kubernetes node conditions and then into scheduling and remediation actions through custom triage and drain controllers. This made health observable not only to operators, but also to the scheduler.

Job observability was assembled across Kueue, Kubernetes, Ray, training logs, and experiment metadata. Operators could inspect queue, priority, admission state, node placement, worker readiness, restart count, training configuration, step progress, and scoped logs by namespace, pod, job, and restart index. This made it possible to distinguish scheduling delay, runtime failure, node failure, storage degradation, and application-level stalls without manually reconstructing state across systems.

Telemetry was stored and queried across systems optimized for different time horizons. Datadog provided near-real-time metrics and log search. Azure Managed Prometheus provided in-cluster and cross-cluster time-series collection. Azure Data Explorer provided long-retention analytical storage for logs, metrics, storage telemetry, and cluster state. Alerting used the same layered model: local Prometheus rules for low-latency component failures, Datadog monitors for service and log-derived signals, KQL-based checks for long-retention analysis, and Azure Monitor for resource and Prometheus alerts. Critical alerts were routed to incident management, while lower-severity signals fed dashboards and operational reviews.

The unique design point was that observability was part of the control loop. Hardware telemetry, fabric health, storage behavior, scheduling state, and job progress were not passive dashboards; they determined whether capacity was admitted, drained, remediated, or returned to service. This allowed the fleet to be managed in terms of usable training capacity and goodput rather than provisioned GPUs alone.