No Second Run: Does RSI Stop at Frontier Pre-training?

SII & Xinxu AI

Abstract

Pre-training is where every foundation model begins; bringing recursive self-improvement (RSI) there would let AI improve the process that creates AI, and could change how the entire industry builds foundation models. Yet public RSI work has so far targeted only post-training, which starts from an existing base model, runs trials costing GPU-hours, and returns verifiable scores. RSI for pre-training breaks all three conditions at orders-of-magnitude higher cost: a run consumes hundreds of thousands of GPU-hours over months and cannot be repeated; its objective, post-training headroom, cannot be read off any single score; and what must improve is not a model but the whole production & organization system, from data pipelines and training infrastructure to the teams that run them. To our knowledge, **we are the first in the world to apply RSI to an entire frontier pre-training run from scratch**, and to report it in detail. RSI does not stop at frontier pre-training, but it changes form: since no end-to-end loop can close over a single unrepeatable run, we decompose the *pre-training loop* into smaller loops that can, each repairing a step the full run breaks.

The result is 801-30B-A3B-Base, a 30B-A3B MoE model trained from scratch on 11.8T tokens in three months by a newly formed team. We release it as a ***base* model**: full post-training is left to a subsequent release, and the lightweight SFT reported here (6B tokens, no RL) only probes post-training headroom. As a base model, 801-30B-A3B outperforms Qwen3-30B-A3B and Nemotron-3.5-Lightning-30B-A3B, trailing only the larger Qwen3.5-35B-A3B, and keeps this position under identical SFT; with lightweight SFT alone, it achieves the highest average on four agentic benchmarks against fully post-trained models, including Qwen3.6-35B-A3B. Post-training RSI starts where this work ends: it presupposes a strong base model, exactly what pre-training must produce. Bringing RSI into pre-training thus acts on the foundation every later stage builds on. We will open-source 801-30B-A3B-Base and the lightweight SFT data, a ready-made resource for evaluating the post-training headroom of base models, at `https://github.com/GAIR-NLP/801-30B`.

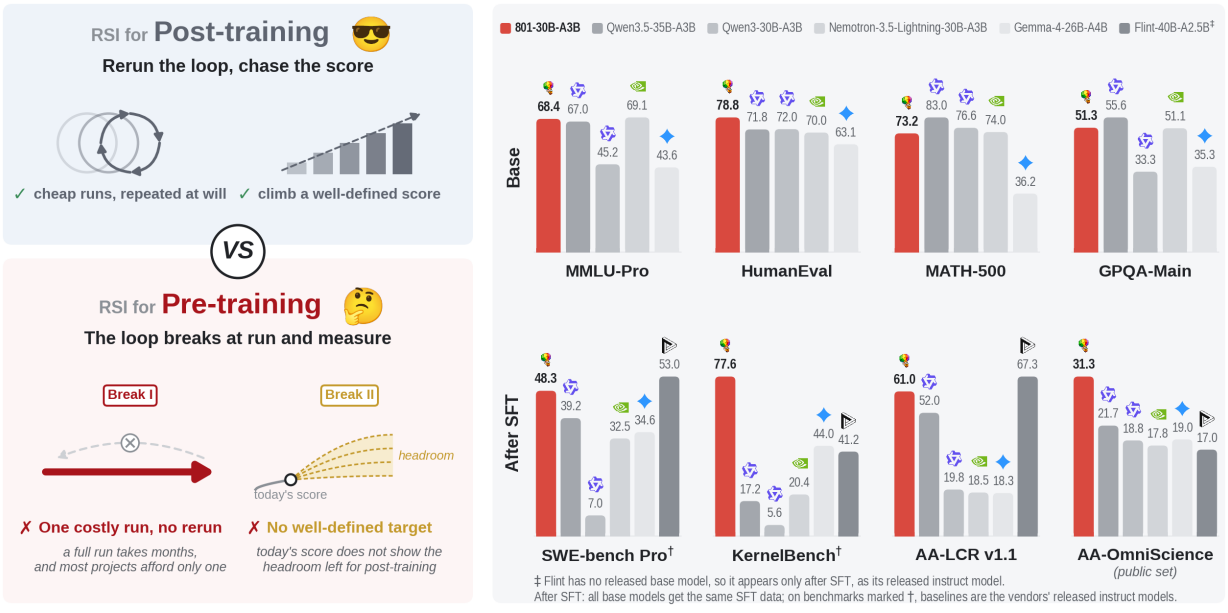


Figure 1: *Left:* recursive self-improvement (RSI) takes a different form in pre-training than in post-training. *Right:* 801-30B-A3B (red) against open baselines on base-model benchmarks (top) and after supervised fine-tuning (bottom).

We present 801 (pronounced eight-oh-one), named after the classroom where it was built.

1 Introduction

Recursive self-improvement (RSI), in which a system turns its own experience and feedback into persistent changes that improve its capability, has moved quickly from idea to practice. Agents now run the loop *propose* → *run* → *measure* → *update* on neural architectures (Xu et al., 2026b), GPU kernels (Yu et al., 2026; Fu et al., 2026a), agent harnesses (Xia et al., 2026), and, closest to model building, on post-training itself (NeoHorse Team et al., 2026; Rank et al., 2026; Huang et al., 2026). Where these loops touch pre-training, they do so through small-scale proxies: data-curation and data-selection strategies validated on small models (Mi et al., 2026; Xu et al., 2026c; Meng et al., 2026), or single-GPU training runs measured in minutes (Karpathy, 2026; Guo et al., 2026). These settings share three conditions: they start from an existing base model or a toy-scale one, each trial costs GPU-hours or less, and each returns a verifiable score. Pre-training is where every foundation model begins, and it satisfies none of the three. Bringing RSI there would let AI improve the very process that creates AI; yet, to our knowledge, no public work has attempted it on a full frontier pre-training run.

RSI for pre-training is not a harder instance of RSI for post-training but a different regime (Table 1). **First**, it breaks *run*. A single run consumes hundreds of thousands of GPU-hours over months, and most projects can afford only one at full length. An ablation that isolates a single cause costs GPU-weeks, so most hypotheses are never tested end to end. **Second**, it breaks *measure*. The goal of pre-training is not the highest score today but headroom for post-training. A run that chases the metric can quietly borrow against post-training, for example by annealing in supervised fine-tuning (SFT) data too early, or by collapsing output diversity at the expense of the headroom that reinforcement learning (RL) later relies on (Yue et al., 2025). Such an objective cannot be read off any single score, and if even a crisply defined benchmark gets overfit by self-improvement loops (Xia et al., 2026), a target this diffuse offers far more ways to be hacked than to be met. **Third**, what must improve is not a model but a production system and the organization that runs it. Before a single token is trained, the data must be built: in our run, six pipelines with more than 30 processing stages turned raw web pages, documents, and code into 1,053 data subsets. The training stack evolved through more than 130 infrastructure code versions and 106 hyperparameter fields. Behind these artifacts stands an organization: data, infrastructure, training, and evaluation teams, each holding a different part of the evidence, with the hardest problems falling between them. A load-balance anomaly, for instance, surfaced in training metrics but traced to an infrastructure change interacting with optimizer settings (Section 3); no single team could have closed that loop alone. Improving pre-training therefore means improving how the organization divides work, shares evidence, and remembers what it has learned. Post-training RSI starts where this system ends: it presupposes a strong base model, which is exactly what the system must produce.

Small-scale proxies and scaling laws seem to offer a fast loop, but a cheap loop cannot answer every expensive question. Seemingly minor fitting choices shift the fitted exponent (Besiroglu et al., 2024; Weng, 2026), small differences at fitting scale grow large at trillion-token scale, and a ladder of proxy runs assumes that scale is the only thing changing (Weng, 2026). Once a question couples architecture, recipe, infrastructure, and multi-source data mixtures, the proxy study needed to settle it becomes as expensive as the run it was meant to de-risk. We therefore use our two pilot runs to surface and eliminate problems, not to extrapolate scores.

Table 1: RSI for Post-training vs. RSI for frontier pre-training operate in different regimes.

	RSI for post-training	**RSI for frontier pre-training (ours)**
Starting point	An existing base model	Random initialization
Cost per trial	GPU-hours (e.g., 10 H100-hours (Rank et al., 2026))	Hundreds of thousands of GPU-hours for one run
Full-length trials	Tens to hundreds	One
Signal at *measure*	Verifiable task score	Post-training headroom, not readable from any single score
Data	A fixed dataset or environment	Built from raw sources: 6 pipelines, >30 stages, 1,053 subsets, 10.5T unique tokens
Infrastructure	A training script	>130 code versions, 106 hyperparameter fields
What improves	A model or a harness	A production system and the organization that runs it
Who is in the loop	One agent or researcher	Data, infrastructure, training, and evaluation teams with their agents

If the end-to-end loop cannot be rerun, where do agents earn their place? The run leaves behind a vast context: internal metrics, hyperparameter changes, data-mixture edits, infrastructure diffs, and dozens of evaluations on every intermediate checkpoint, far more than any team can read, let alone remember. The characteristic waste in this regime is not a bad experiment but re-investigating a question settled weeks earlier, because no one linked the symptom to its class of cause. The bottleneck thus shifts from compute to institutional memory: what was tried, what failed and why, and which signals co-occurred. Agents earn their place by recalling and attributing across evidence no human will finish reading, and, wherever a fast signal can be built, by doing the work itself: building data-pipeline components, crawling the web, and generating agentic training data at a breadth no team could manage by hand.

Our answer to the question in the title is that RSI does not stop at frontier pre-training, but it changes form. Since no end-to-end loop can close over a single unrepeatable run, we decompose pre-training into smaller loops that can close, each repairing a step the full run breaks. *Debugging loops* relocate *run* onto the infrastructure and data stacks, which can be rerun cheaply; the *headroom loop* repairs *measure* for the run as a whole; and *experience-to-metric loops* build a *measure* where none existed. A shared *context infrastructure* serves as the organization's memory: it integrates evidence across teams and preserves negative results and their reasons, which is where the *update* step persists and the recursion lives (Figure 2).

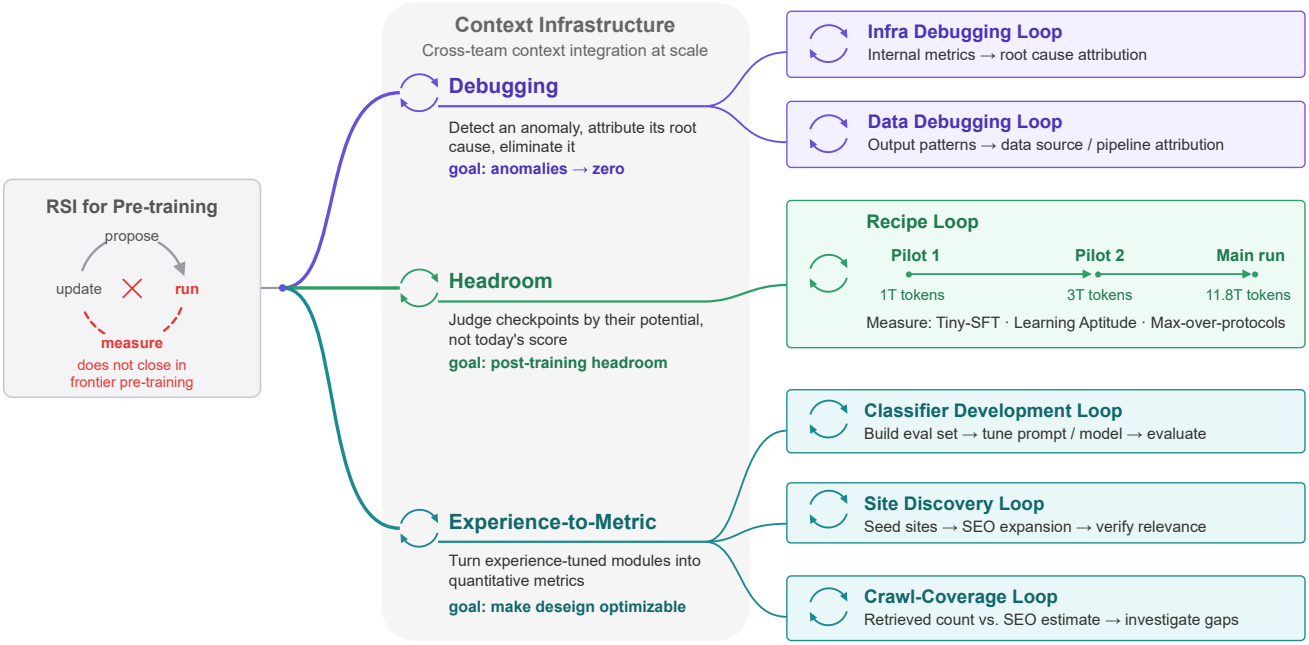


Figure 2: The standard RSI loop (*propose* → *run* → *measure* → *update*) does not close in frontier pre-training: a run takes months and cannot be repeated, and today's score does not reflect the headroom a checkpoint leaves for post-training. We propose three types of loops, instantiated as six practical loops. Debugging loops comprise the infra debugging loop and the data debugging loop, which trace internal-metric and output anomalies to their causes. The headroom loop is instantiated as the recipe loop, using headroom measurements to guide the whole pre-training process. Experience-to-metric loops comprise the classifier development loop, the site discovery loop, and the crawl-coverage loop, turning experience-based judgment into measurable feedback. All six loops share a context infrastructure that integrates evidence across teams at scale.

We built these loops while training 801-30B-A3B, a 30B-A3B Mixture-of-Experts (MoE) (Shazeer et al., 2017) model trained from scratch on 11.8T tokens in three months by a newly formed team. To our knowledge, this is the first public account of RSI applied to an entire frontier pre-training run. Our contributions are:

- *(a) Debugging loops (Section 3).* An *infra debugging loop* and a *data debugging loop*, driven by internal-metric and output anomalies, respectively. Across two pilots (stopped at 1T and 3T tokens) and the main run, agents localized causes spanning infrastructure, data, and recipe, from a load-balance penalty interacting with optimizer state to low-diversity synthetic data, and resolved five cross-stack diagnoses in about one month. Together with data and recipe revisions, these fixes raised the mean score over 22 benchmarks (Appendix B) at a matched 1T-token budget from 47.78 (initial pilot) to 51.96 (revised pilot) to 56.83 (main run).
- *(b) A headroom loop (Section 4).* Because base scores do not preserve post-training rankings, we judge checkpoints by what they are *worth* after adaptation, using tiny-SFT, a learning aptitude curve, and a max-over-protocols

estimate of the capability ceiling. These measurements separate apparent stagnation in base evaluation from continued gains after adaptation.

- *(c) Experience-to-metric loops (Section 5).* Classifier development, site discovery, and crawl coverage, in which agents turn judgment once made by feel into recorded, quantitatively evaluated workflows. We built six data pipelines from scratch this way.
- *(d) Agentic data at pre-training scale (Section 6).* Agents cross from consuming training to producing it: 220K executable pull requests rolled out across nine harnesses yielded more than 100B tokens of verified agentic trajectories (134B agentic tokens in total), used in the long-context stage of pre-training.

What we release and how to compare it. We release 801-30B-A3B-Base as a *base* model. Full post-training is left to a subsequent release; the lightweight SFT reported here (6B tokens, no RL) serves only to probe post-training headroom. We therefore compare at three levels, from the fairest to the least favorable to us (Section 8). (1) *Base against base*: 801-30B-A3B outperforms Qwen3-30B-A3B (Yang et al., 2025a) and Nemotron-3.5-Lightning-30B-A3B (NVIDIA et al., 2026), trailing only the larger Qwen3.5-35B-A3B (Qwen Team, 2026a). (2) *Identical SFT*: when every base model is fine-tuned on the same SFT data, it keeps this position. (3) *Lightweight SFT against full post-training*: even so, it achieves the highest average across four agentic benchmarks among the fully post-trained models we compare, including Qwen3.6-35B-A3B (Qwen Team, 2026b). The third comparison pits a few billion SFT tokens against complete vendor post-training pipelines; we report it as a reference point for headroom, not as a like-for-like comparison. We will open-source 801-30B-A3B-Base and the lightweight SFT data, a ready-made resource for evaluating the post-training headroom of base models, at `https://github.com/GAIR-NLP/801-30B`.

2 Adapting RSI to Frontier Pre-Training

Recursive self-improvement (RSI) is the idea that a system turns its own experience and feedback into persistent changes that improve its capability and, in the strongest form, the process of improvement itself. The engine behind it is a closed loop, exemplified by AlphaEvolve (Novikov et al., 2025) and ASI-Evolve (Xu et al., 2026c). We write that loop as *propose* $\rightarrow$ *run* $\rightarrow$ *measure* $\rightarrow$ *update* and use it as the operational definition of RSI throughout this report. In this report, the improving system is the pre-training project itself, comprising its agents, researchers, and infrastructure, rather than the model being trained.

The four steps are worth defining precisely. *Propose* forms a candidate change or hypothesis, the role played by the generator in generate-critique-revise schemes (Madaan et al., 2023) and by the proposer in proposer-solver self-play (Zhao et al., 2025). *Run* realizes that candidate to produce evidence, by training, rolling out, or executing it. *Measure* scores the evidence against a signal that stands in for human judgment or the desired target. Chen et al. (2026) arrange candidate signals into a verification hierarchy, from formal verifiers at the strong end to a model's intrinsic self-assessment at the weak end. *Update* folds the outcome back into a persistent change, keeping what scored higher and discarding the rest, and in the recursive case it also revises the loop machinery that produced the change. This persistence is what makes the loop recursive rather than a one-off search.

Two preconditions decide whether this loop improves anything. The first is cheap, repeatable iteration: gains compound only when a trial is cheap enough that *run* and *update* can cycle many times. The second is a reliable signal at *measure*, aligned with the true goal. Without an external signal there is no reliable improvement ; with a weak or misaligned one, the loop degenerates through self-confirming errors, model and diversity collapse (Shumailov et al., 2024), or frame lock-in on a well-posed but wrong objective (Chen et al., 2026).

As argued in Section 1, frontier pre-training breaks both preconditions: a full-length run cannot be repeated, and its true objective, post-training headroom, is too diffuse to serve as a trustworthy *measure*. The system that must improve is, moreover, an organization rather than a model. The standard end-to-end loop therefore does not transfer to this regime. We keep the loop, but change where it closes.

Three types of loops. Rather than casting pre-training as one score-driven loop, we close three types of smaller loops around it, with six concrete instances (Figure 2). Table 2 shows how each type instantiates the four steps.

- *Debugging loops* relocate *run* onto the infrastructure and data-pipeline stacks, where a diagnosis can be rerun cheaply. They are driven by anomalies rather than by a score to maximize: *measure* asks whether the anomaly disappears, and each ruled-out cause is recorded as a negative result. The *infra debugging loop* traces unusual internal metrics to infrastructure code or training-recipe settings, and the *data debugging loop* traces unusual output patterns to a specific data source or processing stage (Section 3).
- The *headroom loop* repairs *measure* for the run as a whole. Its signal is post-training headroom, estimated through tiny-SFT, learning aptitude, and max-over-protocols evaluation rather than read off a base-model

Table 2: How each loop type instantiates the four RSI steps. Debugging and experience-to-metric loops close quickly on cheap components; the headroom loop closes only a few times per project, because its *run* is the pre-training run itself.

Loop type	Propose	Run	Measure	Update
Debugging	A suspected cause: code version, configuration, or data subset	Targeted diagnosis on the infra or data stack; cheap and repeatable	Does the anomaly disappear?	Apply the fix; record ruled-out causes
Headroom	A recipe change, or whether to continue training	Pilots and main-run checkpoints; costly and not repeatable	Tiny-SFT, learning aptitude, max-over-protocols	Revise the recipe; judge training progress
Experience-to-metric	A new prompt, model, or crawler version	Component-level evaluation; cheap	Task-specific evaluation set; SEO page-count estimate	Deploy, or iterate again

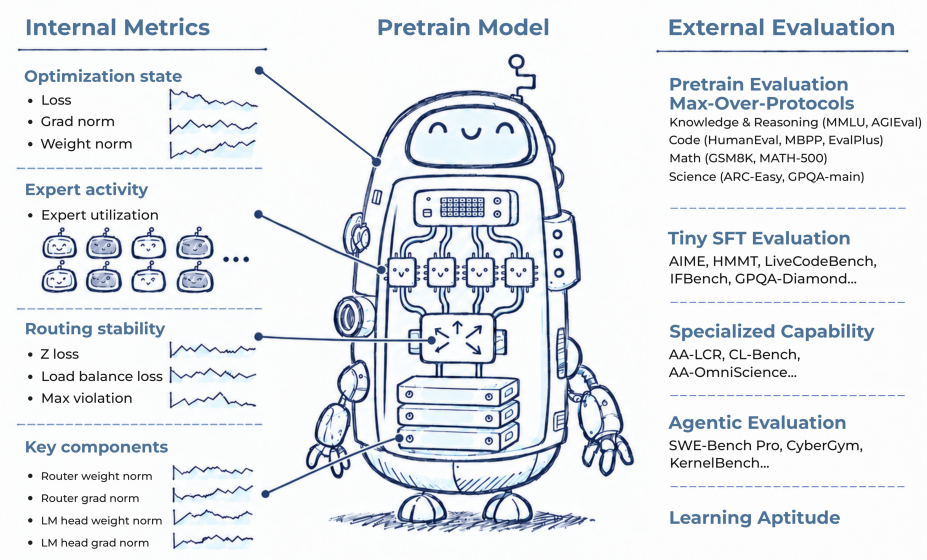


Figure 3: Overview of internal training metrics and external evaluation protocols for pre-training monitoring.

score. It is instantiated as the *recipe loop*, which uses these measurements to revise the recipe from the two pilots, stopped at 1T and 3T tokens, to main pre-training, and to monitor progress within the main run (Section 4). Because its *run* is the costly run itself, it closes only a few times per project.

- *Experience-to-metric loops* build a *measure* where none existed. Steps such as building data-pipeline components and acquiring web data were historically tuned by hand; we give each an explicit evaluation, such as a task-specific evaluation set or an SEO page-count estimate, so that revisions can be compared. This type is instantiated as the *classifier development loop*, the *site discovery loop*, and the *crawl-coverage loop* (Section 5).

Context infrastructure. These loops operate on different parts of pre-training, but all require evidence that spans teams: an anomaly may connect a model's output to a data-processing decision made weeks earlier, headroom measurements must be read alongside recipes and checkpoint histories, and data workflows must retain the evaluations behind each revision. We built a *context infrastructure* to integrate this evidence (internal metrics such as per-layer loss, per-module gradient norms, and expert load-balance statistics, together with hyperparameter changes, data-mixture edits, infrastructure diffs, and checkpoint evaluations), produced by agents and by researchers who separately own the data, infrastructure, training, and evaluation stacks, at a scale no team can read unaided (Table 1). It records not only what was tried but what failed and why, so that negative results become persistent knowledge. This is where the *update* step persists and where the recursion in our loops lives: each new proposal starts from what earlier iterations have ruled out, and agents direct their effort to the next unresolved bottleneck instead of rediscovering old ones.

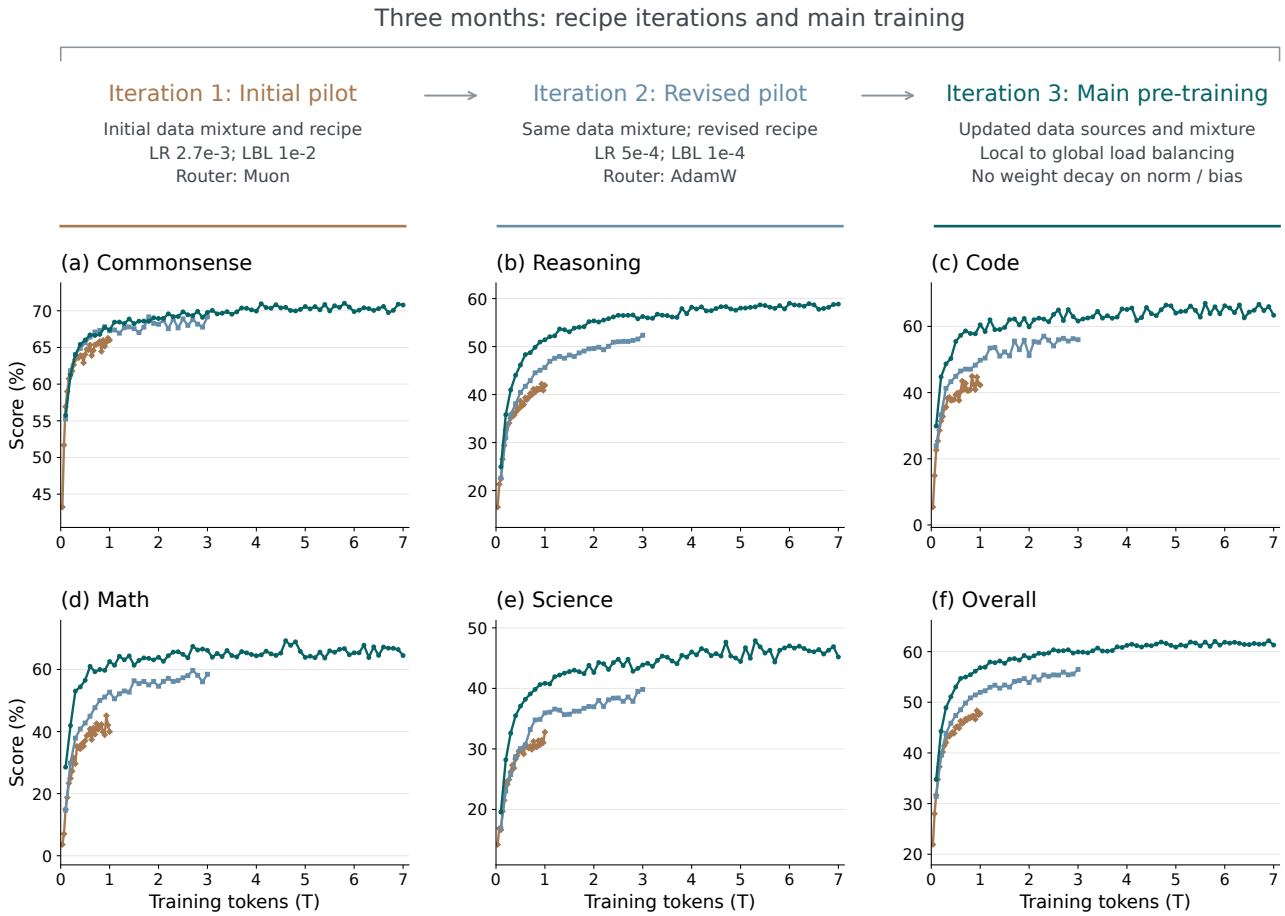


Figure 4: Two pilots and the main pre-training in a three-month period. *Top*: main configuration changes; matching colors link each iteration to its evaluation curves below. *Bottom*: initial pilot (warm brown) and revised pilot (gray-blue) stopped at 1T and 3T tokens, respectively; main pre-training (dark teal) is shown through Stage1-1 at 7T tokens. The five subset scores are means over their constituent benchmarks, and Overall is the mean over all shared benchmarks.

From pilots to main pre-training. Together, these loops supported revisions of the data and training recipe across two pilots and main pre-training (Figure 4). The initial pilot stopped at 1T tokens, the revised pilot at 3T, and the main run used approximately 11.8T tokens; the figure shows main pre-training through Stage1-1 at 7T. These revisions brought together debugging, headroom assessment, and quantitatively evaluated data workflows.

At 1T tokens the mean score over the 22 common benchmarks rises from 47.78 to 51.96 to 56.83, and at 3T tokens main pre-training reaches 59.94 against the revised pilot's 56.48. Because data and recipe changed together, these gains reflect the combined iterations supported by the three types of loops rather than any single fix. The following sections describe how each type contributed to this process.

3 Debugging Loops

This section shows the two debugging loops operating on the context infrastructure as we introduced in Section 2. (1) The *infra debugging loop* traces unusual internal metrics to infrastructure code or training-recipe settings. (2) The *data debugging loop* traces unusual output patterns to a specific data source or processing stage (Figure 5; Table 2 maps both loops to the four RSI steps). Unlike conventional RSI approaches, which improve performance by repeatedly training and comparing candidate models, these loops operate within a single pre-training run. Rather than restarting training to evaluate new candidate models, they continuously identify, attribute, and eliminate problems as training progresses, improving the ongoing run itself. We conclude with the five cross-stack diagnoses produced by these loops (Table 3).

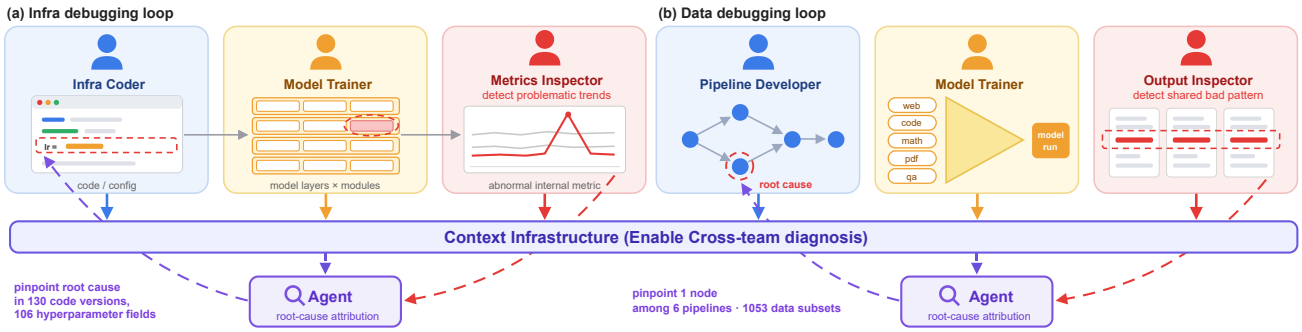


Figure 5: The debugging loops are instantiated as **(a)** the infra debugging loop and **(b)** the data debugging loop.

3.1 Infra Debugging Loop

The infra debugging loop uses internal training metrics as the starting point for identifying and resolving anomalies during pre-training. By linking these signals to infrastructure code and training-recipe configurations through the context infrastructure, agents can trace unexpected model behavior to its underlying causes, propose targeted fixes, and verify their effects within the ongoing training run. We illustrate this process through the anomalies observed in the two pilot runs, following four steps: detect, attribute, fix, and verify.

Detect. During the two pilot runs, agents scanning internal metrics through the context infrastructure identified two contrasting patterns in MoE load-balancing indicators (Figure 6). In the initial pilot, both aggregate and layer-wise MaxVio (Wang et al., 2024a) declined to low levels under a strong balancing penalty, raising concerns about potential over-balancing. The revised pilot revealed a different pattern: layers 15 and 18 developed large, persistent MaxVio violations, while the aggregate remained within a typical range. This discrepancy showed that aggregate monitoring alone could obscure severe layer-specific imbalance. Layer 48, the final layer, was also elevated, but we do not treat this alone as evidence of failure, since its function is closely tied to the output space. Together, these two patterns illustrate why monitoring MoE load balancing requires examining both aggregate trends and layer-specific deviations.

Attribute. To identify the source of these anomalies, agents used the context infrastructure to correlate metric trajectories with infrastructure code commits and optimizer configurations. By searching across more than 130 infrastructure code versions and 106 hyperparameter fields, they narrowed the cause to a change in the load-balance penalty implementation and its interaction with optimizer state updates for certain parameter groups. This process established an attribution chain from training stagnation to internal-metric anomalies, and from those anomalies to specific infrastructure and configuration changes. Rather than treating unusual metrics as isolated symptoms, agents traced them across the training stack to identify their underlying causes.

Fix. Based on this diagnosis, we first revised the pilot configuration by lowering the load-balancing (Fedus et al., 2022) coefficient from 10^{-2} to 10^{-4}, reducing the learning rate from 2.7×10^{-3} to 5×10^{-4}, and switching the optimizer for router parameters from Muon (Jordan et al., 2024; Liu et al., 2025) to AdamW (Loshchilov and Hutter, 2019). These settings were carried into main pre-training, with two further adjustments: computing the load-balance loss over the global batch rather than per micro-batch (Qiu et al., 2025), and excluding normalization and bias parameters from weight decay. Because several factors changed together, the comparison does not isolate the contribution of any single adjustment. Together, these changes constituted the revised training configuration used in main pre-training.

Verify. Subsequent training shows that the revised configuration mitigates layer-wise load imbalance and suggests that internal metrics may reveal potential training problems earlier than downstream evaluation alone. (1) First, the revised configuration substantially mitigated the layer-specific imbalance observed in the pilot. At a matched training budget of 3T tokens, layer-15 MaxVio had already exceeded 14 in the revised pilot, whereas it was around 5 in main pre-training, with pronounced escalation occurring only at a later stage (Figure 7). This indicates that the revised configuration substantially alleviated the earlier anomaly, although it did not eliminate it entirely. (2) Second, the internal metrics provided a more granular view of training dynamics than downstream evaluation alone. During main pre-training, layer-15 MaxVio rose sharply over approximately 2.2–6T tokens (the shaded interval in Figure 7). Over the same interval, downstream evaluation gains gradually diminished. After approximately 6T tokens, layer-15 MaxVio remained at a high plateau, while downstream evaluation gains within Stage1-1 had largely flattened. This temporal correspondence suggests a possible association between layer-wise load imbalance and training saturation. Layer-wise MaxVio rises well before downstream performance plateaus, indicating that internal metrics can flag potential training problems while evaluation scores are still

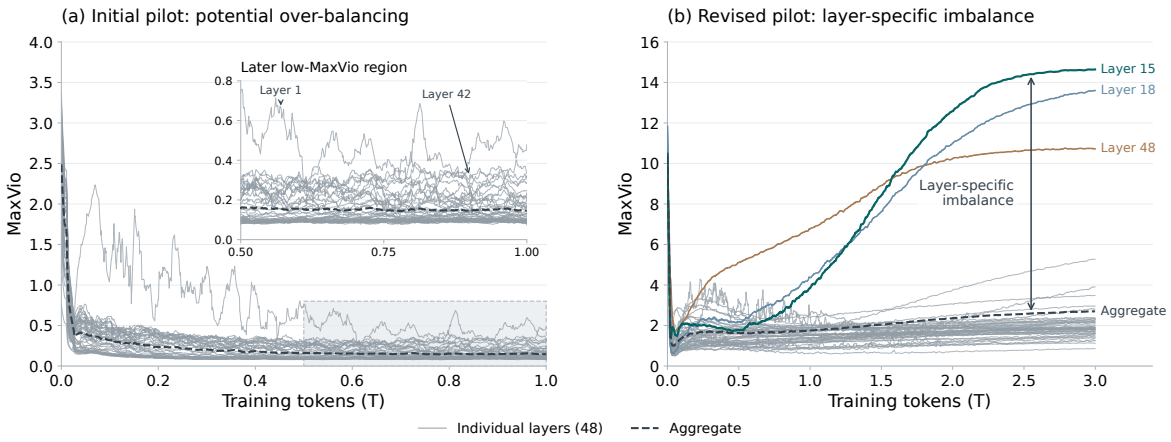
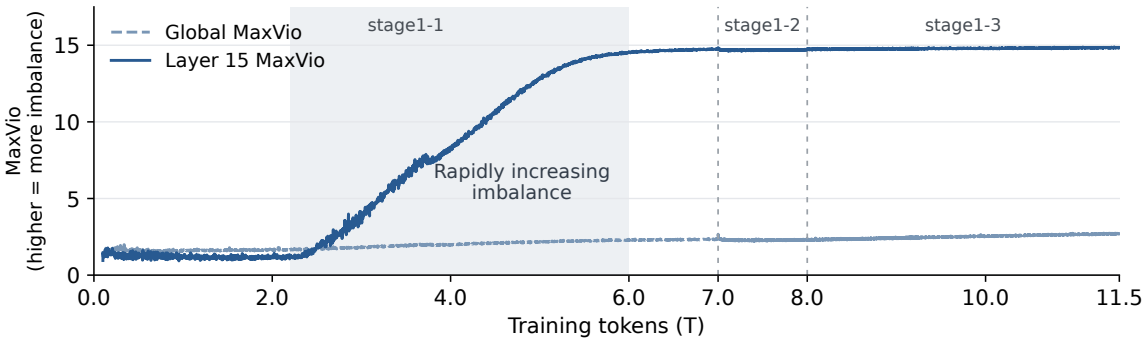
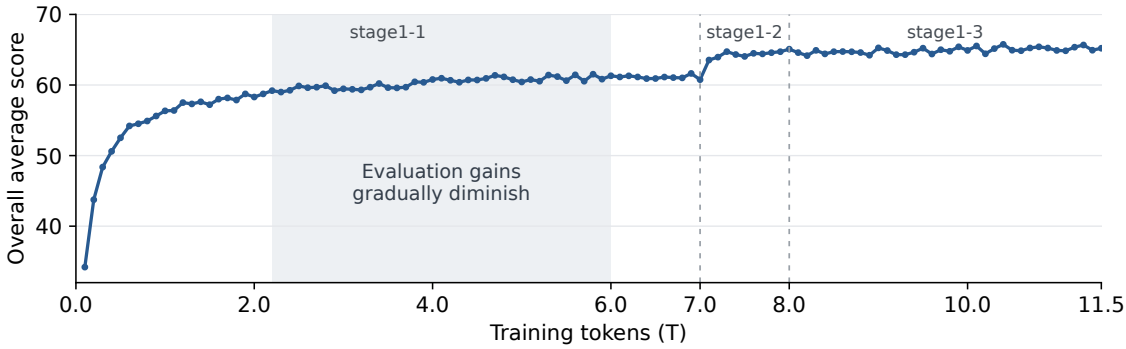


Figure 6: Two contrasting load-balancing anomalies in pilot runs. Gray curves show layer-wise MaxVio across 48 layers; the dashed curve shows aggregate MaxVio. (a) In the initial pilot, both aggregate and layer-wise MaxVio decline under strong balancing, suggesting potential over-balancing. The inset enlarges the low-MaxVio region. (b) In the revised pilot, layers 15, 18, and 48 exhibit elevated MaxVio despite a moderate aggregate, revealing layer-specific imbalance obscured by aggregate monitoring.



(a) Internal signal: layer-wise load imbalance.



(b) External signal: downstream evaluation.

Figure 7: Layer-wise load imbalance rises as downstream evaluation gains diminish during training. The gray shaded region marks the same interval (approximately 2.2–6T tokens) in both panels. (a) Layer-15 MaxVio increases rapidly during this interval, while global MaxVio remains relatively stable. (b) Over the same interval, downstream evaluation gains gradually diminish. Higher MaxVio indicates greater load imbalance. This contrast illustrates how layer-wise internal metrics provide a more granular signal for detecting potential training issues than downstream evaluation alone.

improving. Monitoring these metrics alongside downstream evaluations may therefore enable earlier investigation and intervention.

Finally, the context infrastructure allows these diagnostic experiences to persist across iterations. Because the

previous diagnosis and its supporting evidence were preserved, the recurrence of similar anomalies could be anticipated rather than rediscovered from scratch. This allowed us to use the early MaxVio signal as a basis for further investigation, rather than waiting until downstream evaluation gains had flattened.

3.2 Data Debugging Loop

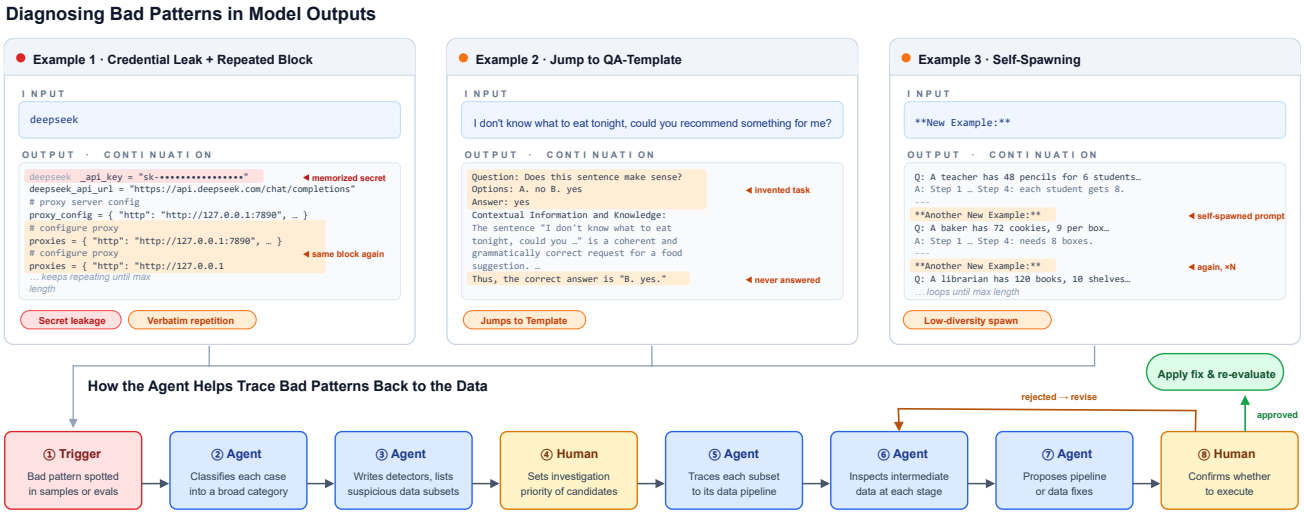


Figure 8: Diagnosing bad patterns in model outputs. *Top*: three representative output patterns, a credential leak with a verbatim-repeated config block, a jump into a fixed "Does this sentence make sense?" QA template, and a self-spawning chain of near-identical word problems. *Bottom*: the eight-step loop that traces each pattern back to a data-pipeline defect. Agents handle detection, tracing, and inspection at scale, humans set triage priority and approve the fix.

During the two pilots, we continuously monitored the output behavior of the trained model. Agents identified recurring patterns in the model's responses to greetings and common questions, which are indicative of repetitive, low-diversity, or insufficiently cleaned training data. They extracted these anomalous patterns and searched the data pipeline for matching signatures at scale.

Anomalous patterns. Figure 8 illustrates three representative patterns observed in model outputs. (1) First, *credential leakage and repetition*: given the prompt `deepseek`, the model emits a secret resembling an API key, followed by repeated copies of an identical proxy-configuration block, continuing until the generation length limit is reached. (2) Second, *jumping to a fixed QA template*: when given a natural request for a dinner recommendation, the model fails to answer the question and instead switches to a fixed "Does this sentence make sense?" multiple-choice template, proceeding to solve this invented task. (3) Third, *self-spawning repetitions*: given only the marker `**New Example:**`, the model generates an unbounded sequence of nearly identical division word problems, each following the same four-step solution scaffold.

Automated diagnosis process. The bottom of Figure 8 illustrates how agents trace these anomalous patterns back to specific data-pipeline defects. The eight-step process can be organized into three stages. (1) The first stage is *detection and pattern extraction*. Agents identify anomalies in model-generated samples or evaluation results and classify the corresponding cases. Based on the identified patterns, agents write detectors and perform large-scale matching across all 1,053 data subsets to identify those potentially containing the corresponding anomalies. (2) The second stage is *attribution and localization*. After candidate data subsets have been identified, humans determine the priority of investigation. Agents then trace these subsets to their data pipelines (six in total) and inspect the intermediate data produced at more than 30 processing stages across these pipelines, ultimately locating the specific data sources and processing stages responsible for the anomalies. (3) The final stage is *resolution and verification*. Once the source of an anomaly has been identified, agents propose fixes to the data or pipeline. Humans decide whether to approve their execution. Approved fixes are applied to the relevant processing stages, followed by model re-evaluation to verify their effectiveness; rejected proposals are returned for revision.

Throughout this process, agents handle large-scale anomaly detection, data tracing, and stage-by-stage inspection, while humans retain control over two key decision points: investigation prioritization and fix approval. This division of labor makes large-scale investigations systematic and practical, beyond what could reasonably be accomplished through manual inspection alone.

Root cause attribution and fixes. Through this process, the three anomalous patterns were traced to three distinct data-pipeline defects. (1) First, the credential-leakage and repetition case was traced to insufficiently cleaned configuration files in the code corpus. These files contained sensitive information and repetitive configuration content that had entered the training data and subsequently surfaced in model generations. We addressed this issue by fixing the corresponding data-processing stage to remove such inadequately cleaned content. (2) Second, the fixed-QA-template case was traced to insufficient filtering of synthetic data in a subset of an open-source QA corpus. This subset contained a large volume of "Does this sentence make sense?" questions. We strengthened synthetic-data filtering at the corresponding stage to reduce the presence of such fixed-template examples in the training data. (3) Finally, the self-spawning case was traced to a synthetic-data pipeline whose prompt design lacked sufficient diversity, resulting in highly similar generated examples. We revised the corresponding prompt design to improve the diversity of the synthetic data. For all three cases, we modified the corresponding data-pipeline stages and re-evaluated the model after applying the fixes to verify their effectiveness.

3.3 Accumulated Diagnoses

The two debugging loops described above illustrate how agents identify anomalies in model behavior and trace them back to underlying issues in the infrastructure, training recipe, and data pipeline. However, these are only representative examples of a broader effort to diagnose and resolve problems throughout the training process.

Across the two pilots and main pre-training, we applied the same approach to systematically investigate issues across different parts of the training system. Many of these issues could not be attributed to a single component. Diagnosing them required connecting evidence from internal metrics, infrastructure code, training configurations, and data pipelines, and tracing the interactions among them. Table 3 summarizes five cross-stack issues identified and resolved through this process, covering load balancing, optimizer configurations, data quality, weight decay, and data loading. These findings demonstrate how agents, supported by the context infrastructure, can identify and resolve problems that span multiple parts of the training system.

Table 3: The five cross-stack diagnoses resolved with the context infrastructure across the two pilots and main pre-training.

Area	Finding	Fix
Load balancing	Expert load was either over-balanced across all layers or severely imbalanced in a few individual layers (Figure 6).	Adopt global-batch load balancing (Qiu et al., 2025), which relaxes the per-micro-batch constraint and allows experts to specialize.
Optimizer	Weight-norm and gradient-norm trends diverged across modules.	Reassign which modules use Muon and which stay on AdamW (Table 5).
Data quality	Pilot models generated advertisements, which were traced to advertisements and low-quality web pages remaining in the training corpus.	Filter them further with quality classifiers.
Weight decay	Router logits showed abnormal trends during training.	Exclude normalization and bias parameters from weight decay.
Data loader	When resuming a training task with the same global batch size but a different DP size, the sampler did not preserve the per-step data order of the previous task.	Restore the per-step sample order of the previous task when resuming with a different DP size.

Beyond the fixes themselves, the time required to complete these diagnoses demonstrates the value of the context infrastructure. In our experience, diagnosing issues of this kind can take months without such infrastructure. With the support of agents and the ability to connect evidence across the training stack, we completed all five diagnoses in roughly one month, substantially reducing the manual effort required to identify and trace the problems.

More importantly, the value of these diagnoses extends beyond the immediate fixes. The context infrastructure preserves the observations, attempted fixes, and negative results from each investigation, allowing subsequent iterations to build on previous findings rather than start from scratch. Each investigation thus contributes not only to improving the ongoing training run, but also to a growing body of knowledge that can be reused in future iterations. Together, the two debugging loops turn individual problem-solving efforts into a cumulative process of continuous improvement, with concrete data and training-recipe improvements carried forward into the successive iterations shown in Figure 4.

4 Headroom Loop

The goal of pre-training is not to maximize today's score but to leave the greatest headroom for post-training. Measuring this headroom can supply the signal for the headroom loop. We first describe three complementary measurements (Section 4.1), then show what they revealed that base evaluation did not (Section 4.1), and finally how the recipe loop used them (Section 4.2).

4.1 Measuring Post-Training Headroom

We use three measures to assess post-training headroom: Tiny-SFT, learning aptitude, and max-over-protocols evaluation. Together, they capture three aspects of a model's potential: its performance after limited post-training, its efficiency in learning from additional data, and the capabilities it can demonstrate under evaluation protocols.

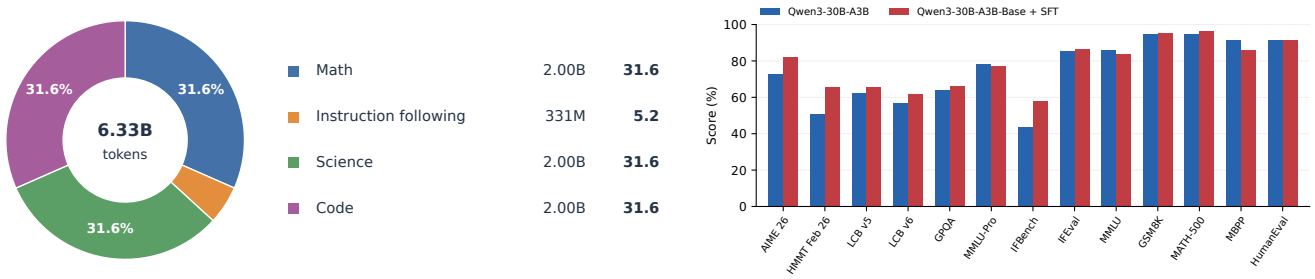


Figure 9: Tiny-SFT data composition and validation. *Left:* Token composition of the 6.33B-token mix. Math, Science (Xu et al., 2026a), and Code (Ahmad et al., 2025) each contribute approximately 2.00B tokens (31.6%), while instruction following (Pyatkin et al., 2025) contributes 0.331B tokens (5.2%). *Right:* Qwen3-30B-A3B versus its base model (Qwen3-30B-A3B-Base) after one epoch of Tiny-SFT. Scores are percentages across 13 benchmarks; the fine-tuned model scores higher on 10 of them. This validation supports the effectiveness of Tiny-SFT as a proxy for monitoring model performance after limited post-training.

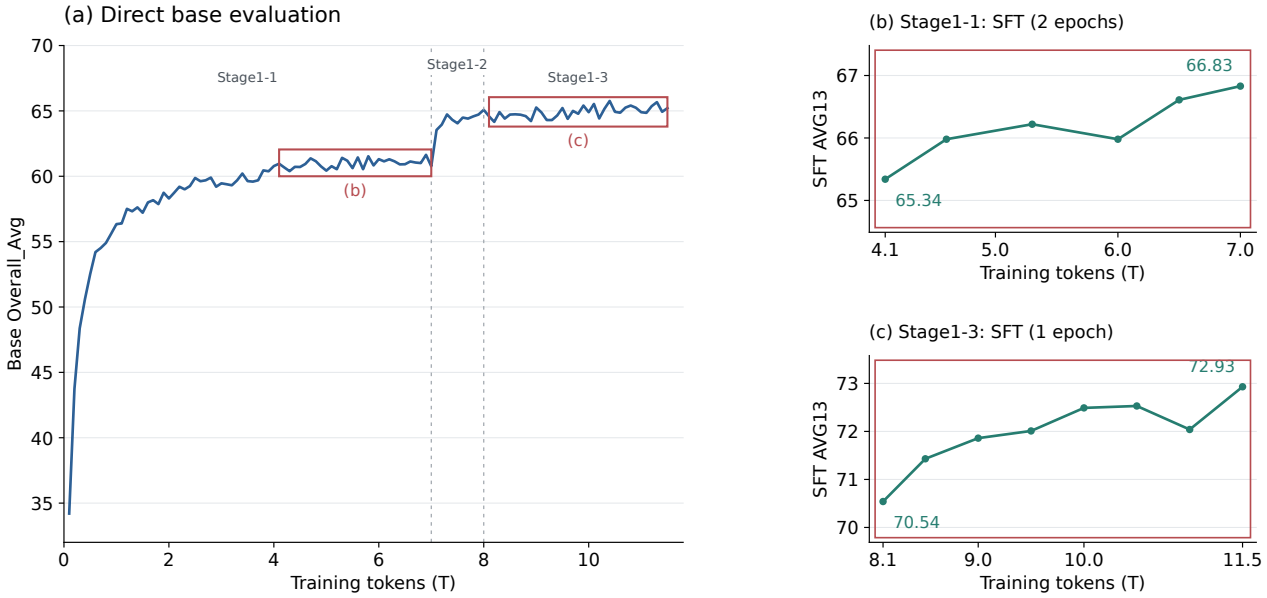
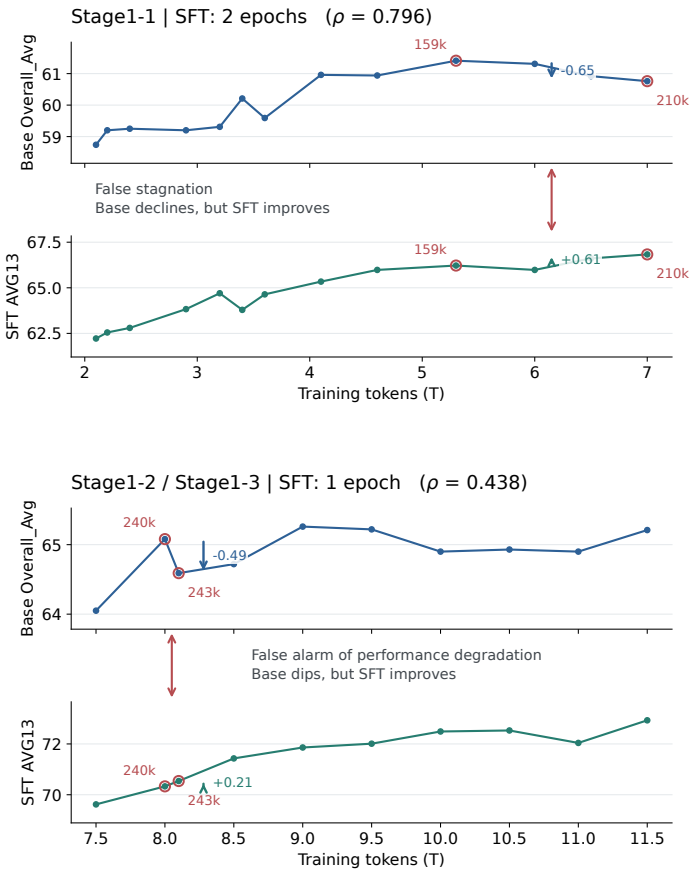


Figure 10: Apparent stagnation in base evaluation can coexist with improving post-SFT performance. (a) Direct base-model Overall_Avg over pre-training; red boxes mark the checkpoint intervals examined in (b) and (c). Each point in (b) and (c) reports SFT AVG13 after applying the same SFT procedure independently to a checkpoint from (a). (b) Stage1-1, steps 123,000–210,000 (4.1T-7T tokens): post-SFT scores improve by 1.49 points despite the near-flat base trend. (c) Stage1-3, steps 243,000–345,000 (8.1T-11.5T tokens): post-SFT scores improve by 2.39 points while base scores fluctuate around a plateau.

Tiny-SFT. Base-model evaluation alone does not fully capture a model's potential after post-training. We



Checkpoint	Base score	Base rank	SFT score	SFT rank
159,000	61.41	1	66.22	3
180,000	61.31	2	65.98	4.5
123,000	60.96	3	65.34	6
138,000	60.94	4	65.98	4.5
195,000	60.92	5	66.61	2
210,000	60.76	6	66.83	1
102,000	60.21	7	63.79	10
108,000	59.59	8	64.64	8
96,000	59.31	9	64.70	7
72,000	59.25	10	62.80	11
66,000	59.20	11.5	62.55	12
87,000	59.20	11.5	63.83	9
63,000	58.74	13	62.22	13

Checkpoint	Base score	Base rank	SFT score	SFT rank
270,000	65.26	1	71.86	6
285,000	65.22	2	72.01	5
345,000	65.21	3	72.93	1
240,000	65.08	4	70.33	9
315,000	64.93	5	72.53	2
300,000	64.90	6.5	72.49	3
330,000	64.90	6.5	72.04	4
255,000	64.72	8	71.43	7
243,000	64.59	9	70.54	8
225,000	64.05	10	69.62	10

Figure 11: Base and post-SFT checkpoint rankings are not interchangeable. *Left*: base Overall_Avg and SFT AVG13 for the same checkpoints. *Right*: all paired checkpoints, sorted by base rank within each group, with their corresponding SFT ranks. Rank 1 is best, ties receive average ranks, and ρ denotes Spearman correlation computed within each group from the reported scores. Base and SFT scores use different benchmark suites.

therefore apply a small SFT budget to each evaluated pre-training checkpoint and directly measure its performance on benchmarks designed for post-trained models. The resulting scores approximate how the checkpoint would perform after a representative post-training phase, providing a more realistic measure of its downstream potential. The SFT corpus spans multiple task categories to approximate the diversity of actual post-training workflows (Figure 9). We note that SFT was not repeated with multiple random seeds, so small differences between adjacent checkpoints should be interpreted with caution. In our experiments, Tiny-SFT helped us identify two interesting patterns that are not readily apparent from base-model evaluation alone:

(1) ***False stagnation.*** We observed that base-model scores can appear to plateau, even as the model continues to improve after SFT. Figure 10 shows two intervals in which average base-model scores fluctuate around a near-plateau, while average scores after SFT continue to improve. In Stage1-1, from step 123,000 to 210,000 (4.1T–7T tokens), the average base-model score remains near 61, while the average score after two epochs of SFT increases from 65.34 to 66.83. In Stage1-3, from step 243,000 to 345,000 (8.1T–11.5T tokens), the average base-model score stays around 64–66, while the average score after one epoch of SFT rises from 70.54 to 72.93. It is worth noting that the base-model and post-SFT scores are computed over different benchmark collections. We therefore compare their trends within each evaluation setting rather than their absolute values. These results show that relying only on base-model evaluation could lead to premature decisions to stop training or unnecessary changes to the training recipe.

(2) ***Ranking disagreement.*** We also found that the relative ranking of checkpoints can change after SFT, compared with their ranking under base-model evaluation. To examine this difference, we paired 23 pre-training checkpoints with their available SFT results and compared their average base-model scores across the 22 public and internal benchmarks listed in Appendix B with their average scores after SFT

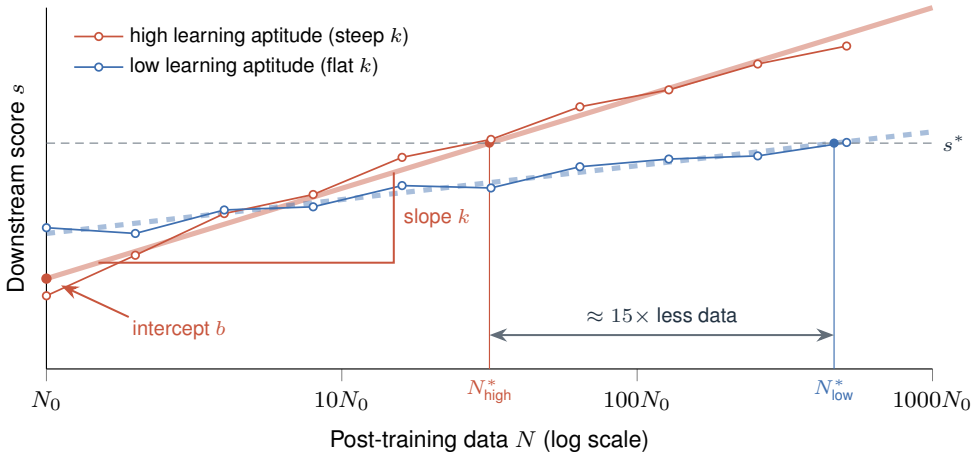


Figure 12: A toy illustration of learning aptitude. Thin lines with hollow markers are raw scores at SFT checkpoints, and each straight line is their least-squares fit of Eq. (1): the intercept b is the score at the first checkpoint N_0, and the slope k is the gain per tenfold increase in post-training data. The model with high learning aptitude starts lowest, yet reaches the target score s^* with about $15\times$ less data than the baseline (Eq. (2)). The high-start, flat model only reaches it at the end of the plotted budget.

(Figure 11). Since checkpoints early in training have weaker instruction-following ability, we used the same Tiny-SFT data but assigned different SFT budgets across training stages. Specifically, we computed rankings separately for 13 Stage1-1 checkpoints evaluated after two SFT epochs and 10 Stage1-2/Stage1-3 checkpoints evaluated after one epoch. The corresponding Spearman rank correlations are 0.796 and 0.438. The rankings are positively associated, particularly in Stage1-1, but they do not provide the same assessment of checkpoint headroom. For example, within the Stage1-1 group, step 159,000 ranks first under base-model evaluation but third after SFT, whereas step 210,000 moves from sixth to first. Between these checkpoints, the average base-model score decreases from 61.41 to 60.76, while the average score after SFT increases from 66.22 to 66.83. These results show that base-model scores alone cannot fully capture differences in post-training potential across checkpoints.

Learning aptitude. A model's initial performance does not necessarily reflect how effectively it can learn from additional post-training data. We therefore measure its *learning aptitude*, which captures how efficiently a model improves as the post-training data budget increases.

To measure learning aptitude, we select a specific domain or task family and fine-tune different base models using the same domain-specific SFT data. We evaluate a sequence of intermediate checkpoints throughout training to track how each model's performance changes as it sees more data. This allows us to fit a learning curve and measure how efficiently the model learns from additional data. For most models, scores grow approximately linearly with the logarithm of the post-training data budget. We therefore fit a log-linear learning curve for each base model m (Figure 12):

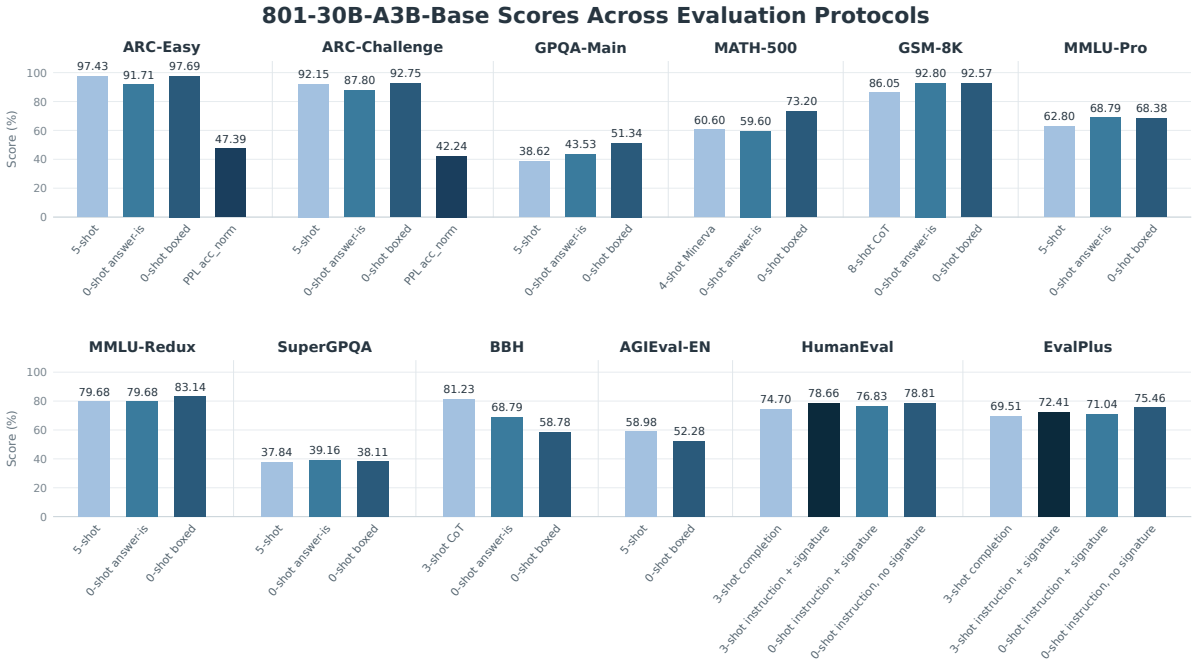
$$s_m(N) = b_m + k_m \log_{10} \frac{N}{N_0}, \tag{1}$$

where $s_m(N)$ is the downstream score after N units of post-training data (in practice, SFT steps at a fixed batch size), and N_0 is the data budget of the first evaluated SFT checkpoint. The intercept b_m is the score at this first checkpoint, representing the model's initial performance in the domain. The slope k_m measures the score gain for every tenfold increase in post-training data, capturing how efficiently the model learns from additional data. We estimate both parameters by ordinary least squares over all SFT checkpoints, excluding the untuned base model.

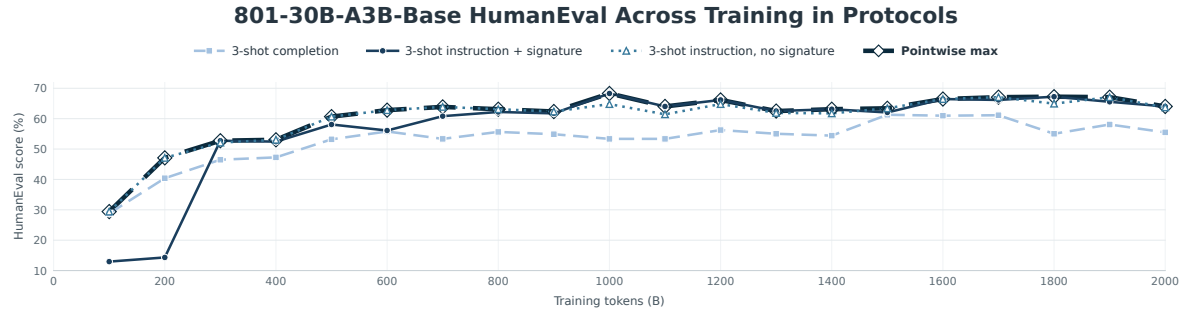
Neither parameter alone captures a model's learning aptitude. A model with a high initial score but a shallow learning curve starts strong but gains little from additional data. In contrast, a model with a lower initial score but a steeper learning curve may eventually surpass it as the data budget increases. We therefore consider both parameters together and estimate the amount of post-training data required for model m to reach a target score s^*:

$$N^*_m(s^*) = N_0 \cdot 10^{(s^* - b_m)/k_m}, \tag{2}$$

We then compare this data requirement with that of a baseline model, using the ratio N^*_{base}/N^*_m to measure data savings. A larger ratio means that the model requires less post-training data to reach the same target performance.



(a) 801-30B-A3B Base scores across benchmarks under different evaluation protocols. The same benchmark can yield different scores depending on the protocol, showing that a single evaluation setting may not capture the capabilities a model has acquired.



(b) HumanEval scores throughout pre-training under different evaluation protocols. Both the scores and the relative ranking of protocols change across checkpoints. We therefore take the pointwise maximum across available protocols at each checkpoint to obtain a more complete measure of the model's demonstrated capability.

Figure 13: Assessing base-model capabilities with max-over-protocols evaluation.

In this work, we use learning aptitude to compare different models rather than different checkpoints within a single training run. As shown in Section 8, a model with a lower initial score but higher learning aptitude can eventually surpass a model with a higher initial score after sufficient fine-tuning. Applying this measure to checkpoint selection within a single training run is a natural direction for future work.

Max-over-protocols evaluation. A model's evaluation score does not always fully reflect its acquired capabilities, as the extent to which these capabilities are demonstrated can depend on the evaluation protocol. The same benchmark may yield very different scores under perplexity scoring, few-shot prompting, and zero-shot prompting (Figure 13a; protocol definitions in Appendix C). Moreover, the relative performance of these protocols can change across pre-training checkpoints, as illustrated by HumanEval in Figure 13b.

A single evaluation protocol may therefore underestimate model's capability. To obtain a more complete picture, we evaluate each checkpoint under all available protocols and take the highest score for each benchmark. Our goal is to measure the capabilities a model can demonstrate, rather than how well it adapts to a particular evaluation format. Taking the maximum across protocols provides a more complete estimate of the model's capability ceiling. For cross-model comparisons in Section 8, we follow the evaluation protocols specified in that section.

4.2 The Recipe Loop

The *recipe loop* brings post-training headroom measurements into the pre-training process, using them as feedback to assess training progress, guide recipe adjustments, and inform subsequent iterations. The two pilots, stopped at 1T and 3T tokens, and the subsequent main pre-training run form the sequence of iterations in this loop (Figure 4).

Throughout this process, we use headroom measurements to track how the model's post-training potential evolves across training stages and feed the results back into the pre-training recipe. Rather than focusing solely on current base-model scores, we consider post-training performance, learning efficiency, and the capabilities the model can demonstrate under different evaluation protocols. This helps us judge whether pre-training is still accumulating useful capabilities and whether the current recipe needs further adjustment.

Based on the issues identified through earlier debugging, we continue to refine the infrastructure and training configurations while developing the training data. In particular, main pre-training introduced new QA data and revised the proportions and sampling weights of web, code, mathematics, books, and papers.

The central idea is that pre-training progress should not be judged solely by current base-model scores, but also by whether the model continues to accumulate capabilities that can be further developed through post-training. Even when base-model evaluation appears to plateau, pre-training may still be making meaningful progress if post-training headroom continues to grow.

5 Experience-to-Metric Loops

Many data-processing decisions in pre-training have traditionally relied on expert experience, such as how to clean and filter data, and how to judge whether a processing strategy works well. We turn these experience-driven tasks into standardized workflows with explicit evaluation metrics. This allows agents to analyze errors, revise their approaches, compare alternatives, and improve their performance through repeated iterations.

We call this the *Experience-to-Metric Loop*, and apply it to three tasks: classifier development, site discovery, and crawl-coverage optimization. All three follow the same principle: turn human experience into measurable tasks, let agents carry out and improve the work, and use evaluation results to guide the next iteration.

5.1 Classifier Development Loop

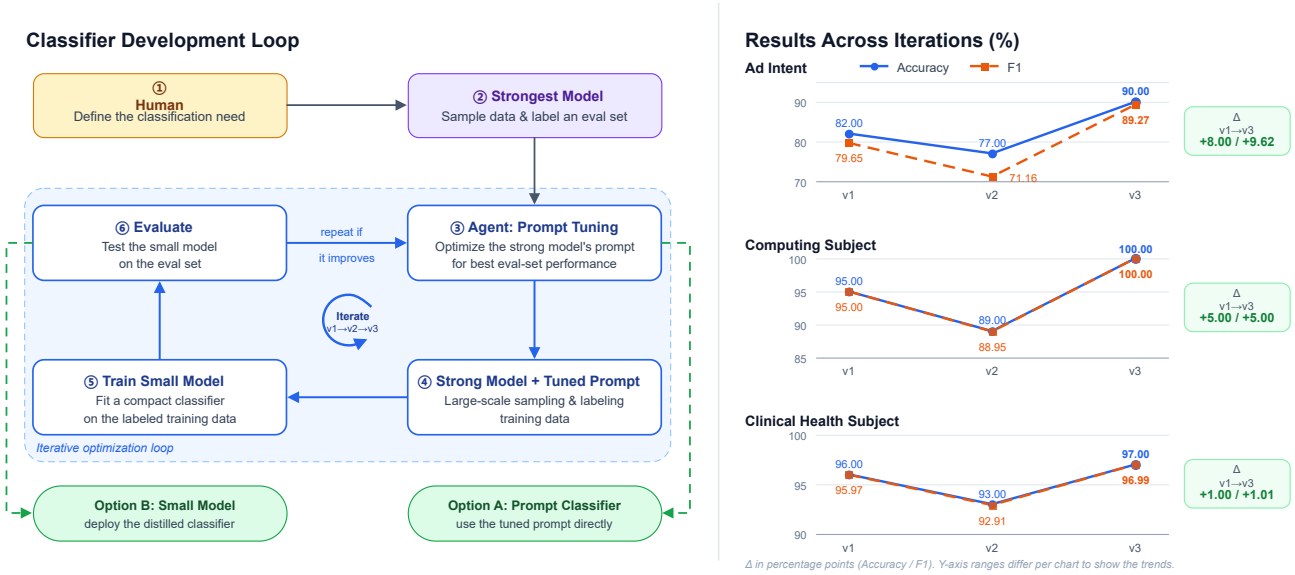


Figure 14: Iterative development of a data classifier. *Left*: a human defines the classification task (1); a strong model samples data and annotates an evaluation set (2); an agent analyzes cases and tunes the labeling prompt (3); the strong model uses that prompt to label training data (4); a compact classifier is distilled from these labels (5) and evaluated (6). The component can be deployed as a tuned prompt (Option A) or a distilled model (Option B). *Right*: accuracy and F1 over three iterations (v1–v3) for classifiers of ad intent, computing subject, and clinical health subject.

Earlier work on *Data Darwinism* provides a framework for the first question. It introduces a ten-level taxonomy (L0–L9) that organizes data processing operations, from data acquisition and format normalization to filtering,

model-driven refinement, and synthesis (Qin et al., 2026a). Following its use in daVinci-LLM (Qin et al., 2026b), we use this framework to guide pipeline design. For example, OCR belongs to format normalization (L1), rule-based and model-based filtering correspond to L2–L3, and refinement for cleaning or repairing content falls under L4. This taxonomy helps us define what each component should do before choosing its model, prompt, or implementation.

DataEvolve explores a complementary question: how data-curation strategies can improve through repeated evaluation and diagnostic feedback (Mi et al., 2026). Inspired by this approach, we develop classifiers, filters, OCR models, and rewriters through a shared evaluation workflow.Data Darwinism guides the choice of processing operations, while the evaluation loop helps determine whether a particular implementation performs its intended task well. Figure 14 illustrates this process for classifiers, with three stages (six steps in the figure).

Step 1: Define the task and construct an evaluation set. A researcher specifies what the classifier should do. A strong model then annotates representative samples to create a reference evaluation set. This establishes quality criteria and provides a common basis for comparing prompts, models, and processing rules. Since the reference labels come from a strong model, the evaluation measures agreement with that model rather than with human judgment.

Step 2: Analyze errors and improve the component. Agents analyze hundreds to thousands of examples in parallel, identify recurring error patterns, and use these findings to revise prompts, compare models, or adjust processing rules. Each revision is evaluated against the same task definition and evaluation set, allowing us to determine whether the changes actually improve performance.

Step 3: Validate and deploy. The revised component is evaluated before deployment. For a classifier, the result can be used as a tuned labeling prompt or distilled into a smaller model using training data labeled by the strong model. The distilled classifier is evaluated and the loop continues revisions improve its performance. The evaluation catches regressions: in Figure 14, all three classifiers lost accuracy at v2 before improving at v3.

In under three months, we built six data pipelines, with agents contributing 36 model-selection decisions as well as prompt optimization and classifier construction.

5.2 Web Acquisition: Site Discovery and Crawl-Coverage Loops

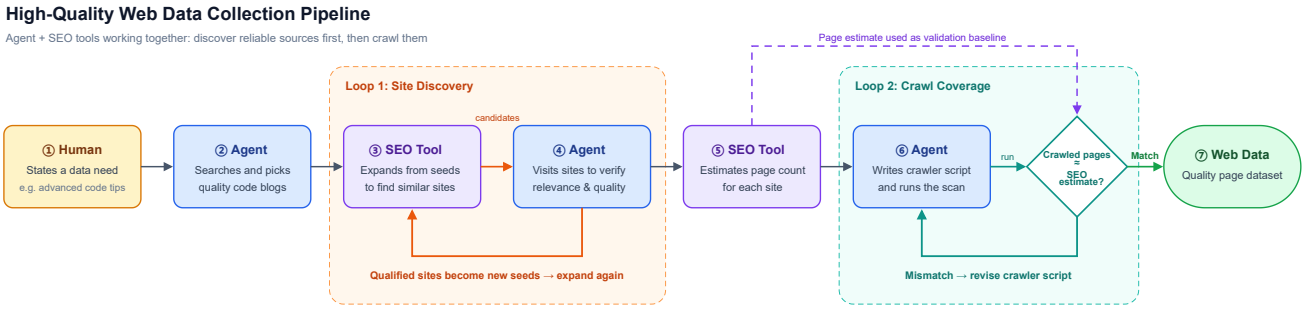


Figure 15: Structure-aware web data acquisition. A human requirement seeds an agent-driven pipeline with two loops. Loop 1 (site discovery) expands and verifies related sites through the web's link structure. Loop 2 (crawl coverage) crawls each confirmed site to a coverage target informed by its SEO page estimate. Links, search indexes, and robots directives are treated as structured signals rather than discarded.

The public web is a major source of pre-training data, but systematically finding high-quality sources and collecting their content as completely as possible remain challenging. A common approach starts from seed documents, uses a classifier to retrieve relevant pages, and identifies the domains where these pages are concentrated (Shao et al., 2024). However, two questions remain: how can we continuously discover more valuable sites, and how can we tell whether we have collected enough data from each site?

We address these questions with two connected agent-driven loops: the *site discovery loop*, which finds high-quality data sources, and the *crawl-coverage loop*, which aims to collect their content as completely as possible. Both use information from search-engine-optimization (SEO) tools, which provide related-site information and estimates of indexed page counts, as signals to guide discovery and measure crawl coverage. Figure 15 shows the pipeline.

Loop1: Site Discovery Loop. The goal of site discovery is to expand a small set of high-quality seed sites into a broader collection of relevant data sources. We use SEO tools to identify candidate sites related to the seeds through the web's link structure. An agent then visits each candidate to check its relevance and quality, adding verified sites to the seed set for the next round of discovery. We measure the effectiveness of each round by its *verified yield*, the number of discovered sites that pass verification. Each round's verified sites become seeds for the next round, allowing discovery to expand iteratively. The process stops when a round produces no new qualified sites. This

turns the experience-based decision of where to look next into a measurable and repeatable discovery process.

Loop2: Crawl-Coverage Loop. Once valuable sites have been identified, the next task is to collect their content as completely as possible. For each confirmed site, an agent writes and runs a crawler, then compares the number of retrieved pages with the page-count estimate provided by SEO tools. We use the *coverage ratio* as the evaluation metric, defined as the number of retrieved pages divided by the estimated total number of pages:

$$\text{Coverage Ratio} = \frac{\text{Retrieved Pages}}{\text{Estimated Pages}}$$

Although SEO page counts are estimates, they provide a quantitative reference for assessing crawl coverage. The agent compares the retrieved count with this estimate, investigates potential gaps, revises the crawler, and runs it again to measure the resulting coverage. In this way, crawling becomes an iterative process driven by measurable coverage rather than an informal judgment of when a site is sufficiently crawled.

Closing the loops. The site discovery and crawl-coverage loops address two complementary challenges in web data acquisition: where to find more valuable data sources and how to collect their content as completely as possible. Together with the classifier development loop, they follow the same principle: turn experience-driven tasks into workflows with explicit evaluation metrics, so that agents can execute the work, assess the results, and improve their approaches through repeated iterations. Specifically, classifier development uses task-specific evaluation scores, site discovery uses the verified yield of each round, and web crawling uses the coverage ratio. The key is to give tasks that once relied on expert judgment explicit objectives and evaluation procedures. Agents can then use feedback to determine how well a task has been completed and what to improve next. This turns experience-based decisions into systematic and repeatable processes, making pre-training more scientific one component at a time.

6 Scaling Agentic Data into Pre-Training

Agentic data, consisting of trajectories generated by agents solving real-world tasks with verifiable outcomes, has proven effective in post-training (Pan et al., 2025; Yang et al., 2025b). Real-world software development provides a rich source of such tasks and offers a new opportunity for pre-training. Our goal is to scale this data source to hundreds of billions of tokens while preserving task diversity and data quality.

However, achieving this scale requires more than conventional data-processing pipelines. Real-world software tasks are diverse and involve complex task construction, execution environments, testing, and integration across agent harnesses. We argue that *extensive use of agents is a necessary condition for scaling agentic data into pre-training*. The main bottleneck is not generating a handful of high-quality examples, but systematically transforming diverse real-world software development activities into executable, verifiable, and scalable training data.

6.1 Why Agentic Data Requires Agents

Complex task construction and validation. Real software development offers a rich source of tasks for agentic data. Recent work has explored several directions: making development tasks executable through automated environment and evaluation-script construction (Fu et al., 2026b), preserving development context and execution feedback in mid-training trajectories (Zeng et al., 2026), and connecting related PRs into longer task chains (Jiang et al., 2026).

Scaling this data source to pre-training requires handling the full diversity of real-world development activity. This includes both PRs with clear task definitions and well-maintained tests (Jimenez et al., 2024; Pan et al., 2025; Badertdinov et al., 2025), and those with incomplete descriptions, implicit requirements, or multiple work items bundled together. For each PR, the pipeline must construct three mutually consistent components: a reproducible execution environment, an instruction specifying what the agent is expected to do, and tests and evaluation scripts that verify whether the task requirements are met.

These components must be developed together. For example, a repository's CI may inject environment variables that were omitted from the constructed environment, causing the evaluation to fail for reasons unrelated to the code change. In another case, a test may require a function with a specific name, while the instruction describes only the desired behavior, causing a valid implementation to be rejected. A single PR may also bundle multiple work items, leaving the instruction, environment, and tests with inconsistent task boundaries.

Task construction therefore involves more than simply extracting a task description from a PR. The environment, instruction, and evaluation must agree on what the task requires. Agents need to inspect repository context, identify inconsistencies across these components, and revise them accordingly. At scale, this process must handle the long tail of repository-specific conventions and development practices.

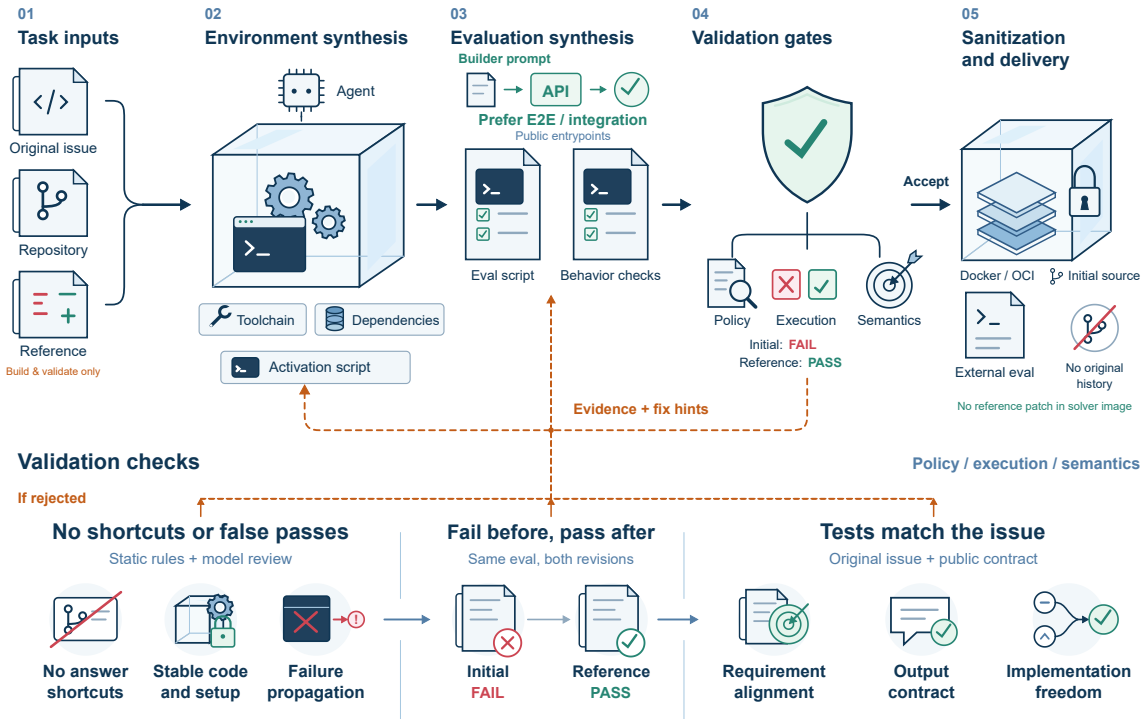


Figure 16: Overview of agentic-data pipeline. The key component is the Validation gates, which admits a task only when it clears policy, execution, and semantic checks and the same evaluation fails on the initial code but passes on the reference, and sanitization into a Docker/OCI image that keeps the external eval but strips the reference patch and original history. Rejected tasks return with evidence and fix hints (dashed path).

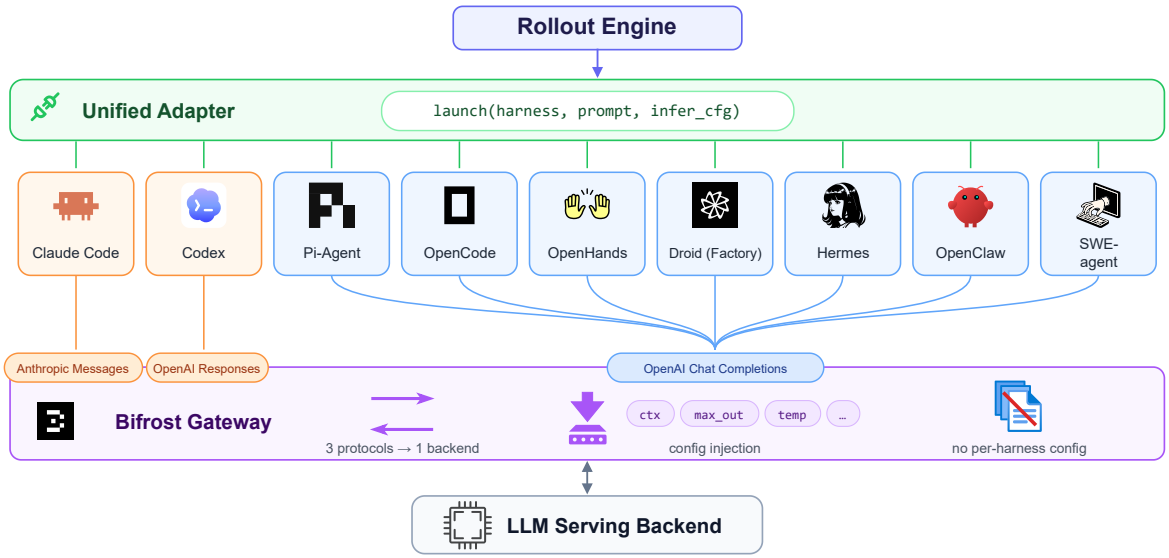


Figure 17: The multi-harness rollout layer. A single `launch(harness, prompt, infer_cfg)` call in the unified adapter drives all nine integrated harnesses, which among them speak three API protocols (Anthropic Messages (Anthropic, 2026), OpenAI Responses (OpenAI, 2026b), and OpenAI Chat Completions (OpenAI, 2026a)). The Bifrost gateway (Maxim AI, 2025) normalizes the three protocols onto one LLM serving backend and injects inference settings (context length, maximum output, temperature) centrally.

Multi-harness integration and maintenance. Even after constructing a task, generating diverse trajectories at scale requires supporting multiple harnesses during execution. We integrated nine harnesses into our agentic data pipeline: Claude Code (Anthropic, 2025), Codex (OpenAI, 2025), Pi-Agent (Earendil Works, 2025), Open-

Code (Anomaly, 2025), OpenHands (Wang et al., 2025), Droid (Factory AI, 2025), Hermes (Nous Research, 2025), OpenClaw (OpenClaw contributors, 2026), and SWE-agent (Yang et al., 2024). Each harness has its own interfaces, dependencies, and failure modes. Even the same model can behave differently across harnesses. In addition, harness updates and new plugins introduce further configuration changes.

As a result, the combinations of models, harnesses, and versions require ongoing integration and maintenance. We use a unified adapter to launch rollouts across different harnesses, with agents helping diagnose execution failures, revise integrations, and validate updated configurations. As these combinations grow and change, manual maintenance becomes unsustainable. Agents are therefore necessary for sustaining large-scale trajectory generation.

Table 4: Agentic anti-hack patterns, grouped by category. Each is a concrete way a model can pass a synthesized task without solving it. Agents scan trajectories for these behaviors and flag candidates for review or re-synthesis.

Category	Specific behavior
Pull the reference fix from version control	Fetch its own merged PR diff or the fixing commit patch Recover the fix from unreachable Git objects (`git fsck`, `cat-file`)
Fetch upstream code over the network	Download files from `raw.githubusercontent`, the GitHub API, or PyPI JSON Clone, fetch, or add a remote pointing at the upstream repository
Install an already-fixed version	`pip download` or `install --target` with a pinned version range Pull packages through conda, apt, or an external mirror (aliyun, tuna, ustc)
Find a solved copy already on disk	Read installed, cached, or `site-packages` source copies Scan from root with `find`, `grep`, or `rg`, or walk the conda tree
Tamper with the evaluation	Write or rewrite test files Deselect or skip the failing tests (`pytest --deselect`)
Reason then exfiltrate	Plan in chain-of-thought to leave the sandbox, then issue a network call

Trajectory quality control and anti-hacking. Anti-hacking is critical to producing high-quality agentic data. Models can exploit evaluation shortcuts in many ways (Zhong et al., 2025; jacobkahn, 2025), and agents are essential for detecting such behavior at scale. In a data synthesis pipeline, agents must distinguish between two types of problems: *hacks*, where a model bypasses the task rather than genuinely solving it, and *synthesis defects*, where the execution environment, tests, or task instructions themselves are flawed. This distinction is particularly difficult because a model's cheating behavior can mask defects in the task, making an invalid task appear solvable. To address this, agents scan execution trajectories for anomalous behaviors, such as accessing Git repositories and logs, downloading related packages, searching for code commits, or using shell commands that bypass the actual task logic. Suspicious cases are flagged for human review or re-synthesis (Table 4).

Thus, producing agentic data at scale requires not only making tasks executable, but also verifying that models genuinely solve them and that the resulting trajectories are trustworthy.

6.2 Scaling Agentic Data for Pre-Training

Through the task-construction, multi-harness execution, and quality-control processes described above, we established a pipeline for producing agentic data at scale.

More than 84% of the raw pull requests we collected were successfully converted into executable tasks. For these PRs, we constructed execution environments, extracted task requirements, and synthesized corresponding tests (Figure 18). We then rolled out the resulting 220K executable PRs across nine harnesses. The trajectories were generated using small and medium-sized open-weight models, including DeepSeek-Flash and GPT-OSS. After layered filtering, we obtained 1.1M agent trajectories that met our pre-training quality standards, totaling more than 100B tokens, with approximately 91K tokens per trajectory on average. Together with trajectories from other diverse sources, the total volume of agentic data available for pre-training reached **134B** tokens. Importantly, these trajectories are not merely an auxiliary ingredient for post-training, but a *first-class data source for pre-training*.

Agent-driven data pipelines represent a shift in how training data is produced. As models become more capable, they can solve harder tasks and generate richer training signals. At the same time, as harnesses and tools evolve, agents can adapt the underlying integration layer. Beyond adding more harnesses, the plugin ecosystems within each harness are also growing rapidly, further expanding the space of configurations that must be supported.

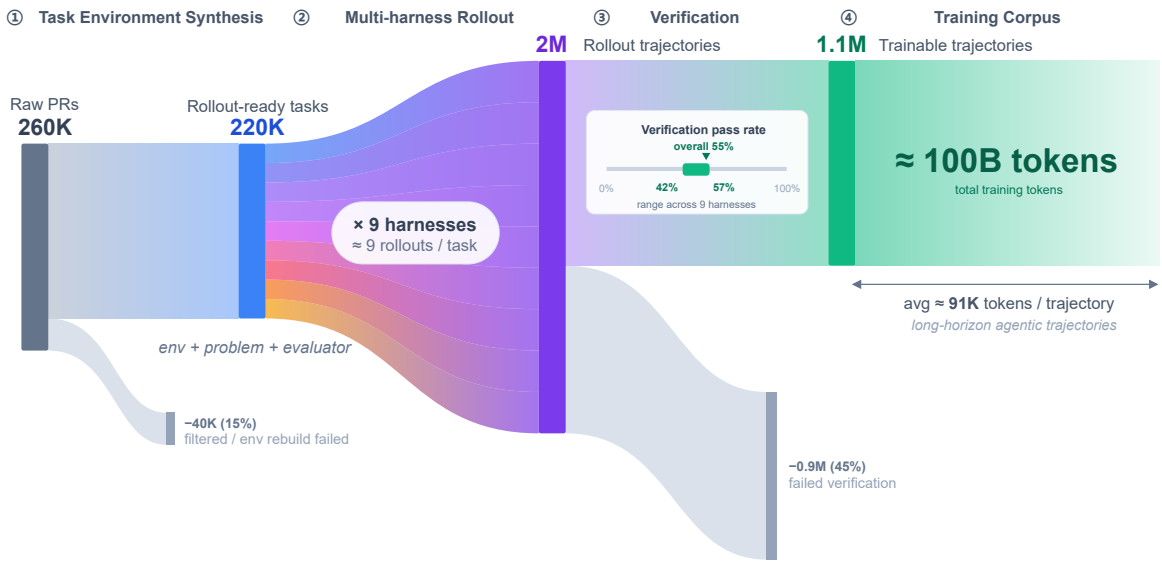


Figure 18: Funnel chart of agentic data pipeline. 260K raw PRs are filtered into 220K executable tasks. Rolling each task out under 9 harnesses gives about 2M trajectories. After verification, 1.1M trajectories remain, totaling more than 100B training tokens (about 91K tokens per trajectory on average). Tasks and trajectories are drawn on separate scales.

We expect agent-driven data pipelines to become an increasingly important source of high-quality, diverse, and verifiable training data for future frontier pre-training runs.

7 Training Recipe of 801-30B-A3B

We adopt the Qwen3-30B-A3B (Yang et al., 2025a) Mixture-of-Experts architecture, with approximately 30 billion total parameters and 3 billion active parameters per token. Figure 21a summarizes the selected training recipe: three pre-training sub-stages followed by two context-extension stages. All stages use a global batch of 33,554,432 tokens and Muon (Jordan et al., 2024; Liu et al., 2025), with AdamW (Loshchilov and Hutter, 2019) for the excluded parameter groups, including embeddings, the output head, normalization parameters, biases, and routers.

Stage1 maintains a sequence length of 4,096 tokens. Stage1-1 trains from scratch through step 210,000 (approximately 7T tokens); Stage1-2 continues to step 240,000 (approximately 8T), and Stage1-3 supplies the checkpoint at step 345,000 (approximately 11.5T). At these transitions, the data mixture is updated and the RoPE (Su et al., 2024) base increases from 10^4 to 5×10^5 and 5×10^6, following the base-frequency adjustment for longer contexts (Xiong et al., 2024). The learning rate remains 5×10^{-4} after an 2,000-step warmup; Stage1-2 and Stage1-3 resume the optimizer and training state without a warmup. Stage1 uses global-batch load balancing (Fedus et al., 2022; Qiu et al., 2025) with coefficient 10^{-4} and router z-loss (Zoph et al., 2022) with coefficient 10^{-3}.

Stage2 extends the sequence length to 32,768 tokens and is configured for 3,000 steps (approximately 100B tokens). Stage3 further extends it to 262,144 tokens and is configured for 6,000 steps (approximately 200B tokens). Both retain the RoPE base of 5×10^6, initialize from the preceding stage's weights with fresh optimizer state, use a 1,000-step warmup, and disable both load-balancing and router z-loss penalties. Stage2 keeps the learning rate at 5×10^{-4} after warmup, whereas Stage3 applies cosine decay to 5×10^{-5}. Stage3 also enables length-aware packing. The three stages together correspond to approximately 11.8T tokens.

The training corpus draws from multiple data types (shown in Figure 20): web text, PDF documents, question-answer pairs, code repositories, and agentic trajectories. Beyond the agentic data pipeline detailed in Section 6, each major category branches into sub-pipelines (shown in Figure 19). *Web data* is cross-classified by advertisement presence, content type tags, domain category, and language, creating a Cartesian product of fine-grained cells that serve as the atomic units for review and mixture tuning. *Math data* undergoes model-driven rewriting to increase diversity and refine equation and symbols. *PDF data* passes through OCR extraction, vision-language model (VLM) refinement to correct OCR errors, and multi-stage quality filters. *QA data* pipelines classify whether the problem's context is complete or not and rollout the CoT with monitoring pass rate to probe their difficulty. *Code data* includes static code files and pull-request records.

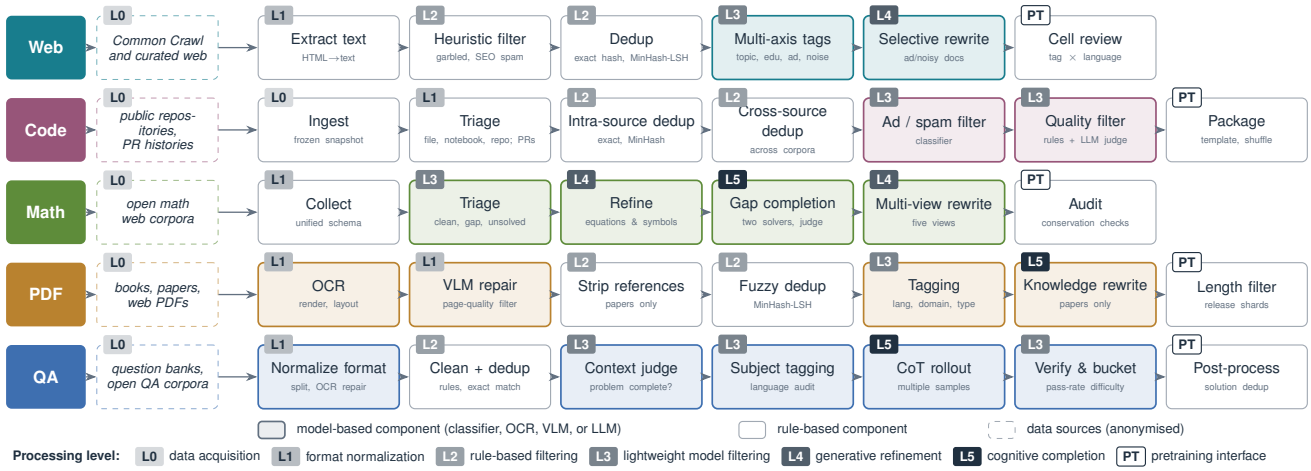


Figure 19: Pretrain data pipelines except the agentic data pipeline. Filled boxes mark model-based components such as classifiers, OCR, VLMs, and LLM rewriters, which we build and tune with the workflow in Figure 14 (Qin et al., 2026a).

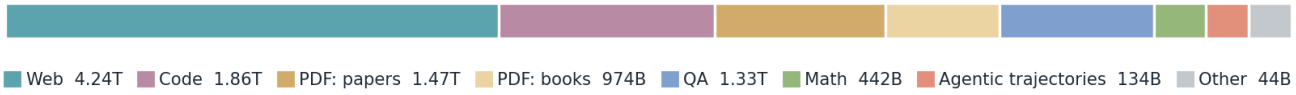
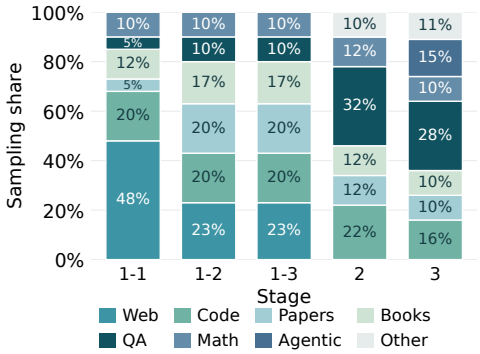


Figure 20: Composition of the pre-training corpus, measured in unique tokens that entered training (10.5T in total).

	Stage1 (11.5T)			Ctx. extension (300B)	
Setup	**1-1**	**1-2**	**1-3**	**Stage2**	**Stage3**
Tokens added	7T	1T	3.5T	100B	200B
Seq. length	4K	4K	4K	32K	256K
RoPE θ	10^4	5×10^5	5×10^6	5×10^6	5×10^6
Learning rate	5×10^{-4}	5×10^{-4}	5×10^{-4}	5×10^{-4}	5×10^{-4} $\rightarrow 5 \times 10^{-5}$
Warmup steps	2,000	No	No	1,000	1,000
LB loss coef.	10^{-4}	10^{-4}	10^{-4}	Off	Off
Router z-loss	10^{-3}	10^{-3}	10^{-3}	Off	Off

(a) Training stages and key recipe changes.



(b) Configured data sampling proportions.

Figure 21: Training recipe of 801-30B-A3B across the five training stages. *(a)* Key recipe changes, including the progression from 4K to 32K and 256K sequence lengths and the RoPE-base updates during Stage1. *(b)* Configured data sampling proportions. Stage1-2 and Stage1-3 retain the same category-level mixture despite updates to their data sources. Other denotes capability-targeted data.

Figure 21b shows how the configured sampling mixture changes across training stages. These proportions describe sampling weights, whereas Figure 20 reports unique corpus tokens. Stage1-1 assigns 48% to web text and 20% to code. In Stage1-2 and Stage1-3, the web share falls to 23%, while papers, books, and QA increase to 20%, 17%, and 10%, respectively. These two sub-stages share the same category-level proportions, although their underlying data sources differ. Stage2 removes the dedicated web component, raises QA to 32%, and assigns 10% to capability-targeted data. Stage3 introduces a 15% agentic component and assigns 11% to capability-targeted data, including instruction-following and long-context examples. The capability-targeted category also includes retrieved web, book, and paper material, so the absence of a dedicated web component does not imply the absence of all web-derived content.

7.1 Attention-Balanced Data Sampling

Mixing long sequences into training data can cause throughput degradation far beyond what their proportion would suggest. In theory, the expected training time should be the proportion-weighted sum of the time spent on

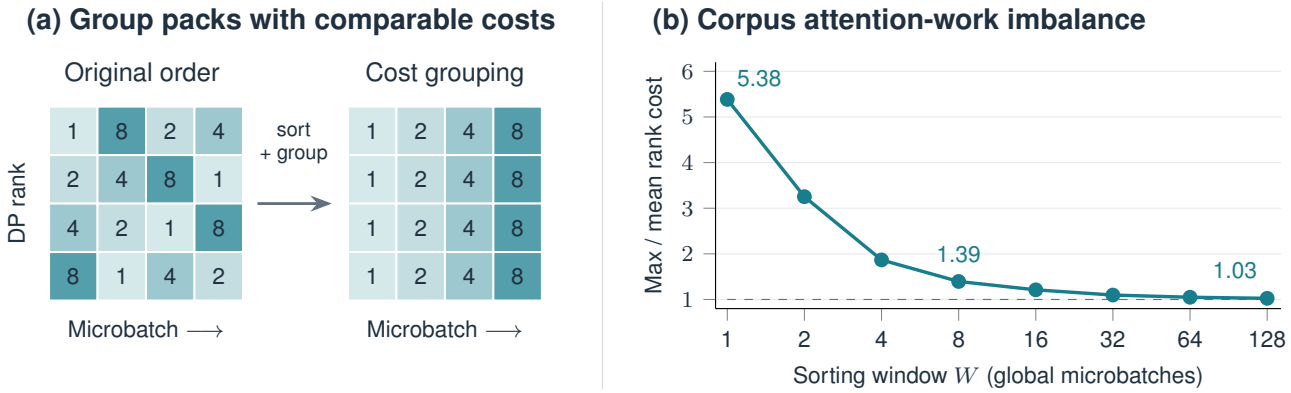


Figure 22: (a) Grouping packs by attention cost reduces imbalance; values are illustrative. (b) Imbalance in our 256K corpus across sorting windows, using 128 DP ranks and one pack per rank. The ratio is maximum to mean rank cost, averaged across microbatches; 1 denotes perfect balance.

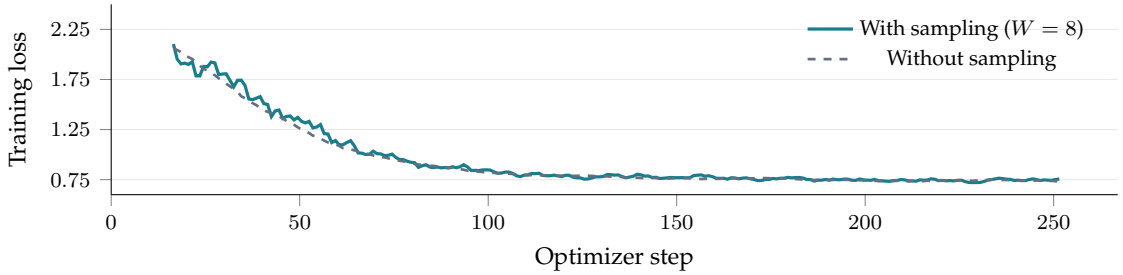


Figure 23: Training loss with and without attention-balanced data sampling (32-step moving average).

long and short samples. In practice, adding only 10% long samples can halve the training speed.

Using the context infrastructure, the agent traces this unexpected slowdown to computation bubbles caused by imbalanced attention workloads across data-parallel (DP) ranks. In synchronous data-parallel training, ranks with heavier attention workloads become stragglers, leaving other ranks idle at synchronization points. This imbalance is particularly severe for packed sequences, where attention cost depends on individual document lengths rather than total token count. We estimate each pack's cost as where m_p counts document segments and $\ell_{p,j}$ is the length of segment j.

$$\widehat{C}(p) = \sum_{j=1}^{m_p} \ell_{p,j}^2, \tag{3}$$

To mitigate this imbalance, the dataloader sort packs by estimated attention cost within a bounded window and group similarly costly packs into global microbatches. When each DP rank receives multiple packs, we further assign packs in descending cost order to the least-loaded rank with remaining capacity (Figure 22a).

The sorting window limits sample reordering. Larger windows improve load balance (Figure 22b), but document length may be associated with data source and difficulty. Aggressive sorting can therefore bias the per-step mixture and substantially increase loss fluctuations early in long-context training. We used a limited sorting window with a buffer of roughly twice the distributed world size to balance computational efficiency and sample diversity.

To evaluate this strategy, we compared two 256K-context pretraining runs on 256 GPUs with matching configurations. We evaluate the proposed dataloader on 256 GPUs with context parallelism of 4 (CP=4), yielding 64 data-parallel (DP) replicas. With one pack per DP replica and a sorting window of eight global microbatches, our method achieves a $2.26\times$ throughput improvement (from 1,126 to 2,543 TGS). Smoothed training losses follow similar trajectories (Figure 23). Because this scheduling alters the global sample order, we evaluated final model quality separately, and benchmark differences with and without scheduling remained within run-to-run fluctuations.

7.2 Optimizer Allocation

We use Muon for expert and attention weight matrices, and AdamW for router, embedding, output, normalization, and bias parameters. To examine this allocation alongside the optimizer diagnosis in Table 3, we compare four

Table 5: Optimizer allocation in the second pre-training pilot. E denotes expert matrices and E-A denotes expert and attention matrices. LBL denotes the load-balancing scope. All Muon variants use RMS matching to AdamW (Liu et al., 2025), momentum 0.95, and five Newton–Schulz iterations; AdamW uses $(\beta_1, \beta_2) = (0.9, 0.95)$. Weight decay is 0.1. Router parameters use AdamW in every setting.

Setting	Experts	Attention	Router	Embedding/output/norm/bias	LBL
AdamW–Global	AdamW	AdamW	AdamW	AdamW	Global
Muon-E–Global	Muon	AdamW	AdamW	AdamW	Global
Muon-E-A–Global	Muon	Muon	AdamW	AdamW	Global
Muon-E-A–Local	Muon	Muon	AdamW	AdamW	Local

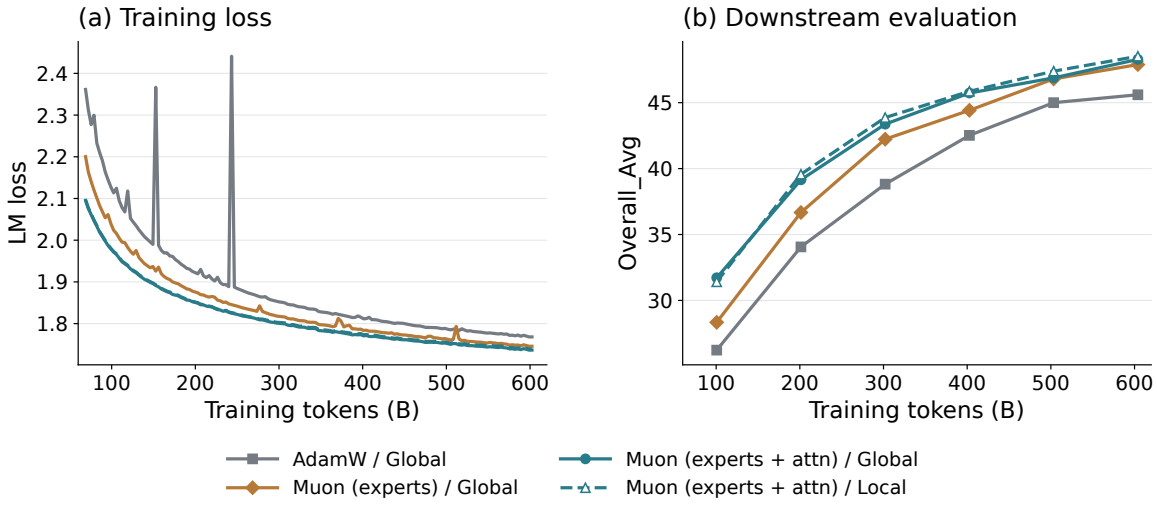


Figure 24: Optimizer allocation and load-balancing comparisons for the four settings in Table 5. *Left*: training LM loss, excluding auxiliary losses, averaged in consecutive 100-step bins after the 2,000-step warmup. *Right*: direct base-model Overall Avg scores at 100B-token intervals. The two expert-and-attention settings share a color, with a dashed line denoting local load balancing. Both global Muon allocations improve on all-AdamW in this interval, while the local/global score differences are small.

configurations from our second pre-training pilot (Table 5). Three use global load balancing and vary the parameters assigned to Muon; a fourth uses the same expert-and-attention allocation with local load balancing. All four use the same configured data mixture, a sequence length of 4,096, a global batch of 8,192 sequences (33.55M tokens), and a learning rate of 5×10^{-4} after 2,000 warmup steps. The load-balancing and router z-loss coefficients are 10^{-4} and 10^{-3}, respectively.

Figure 24 compares these runs over their evaluation interval, ending at step 18,000 (604B tokens). Under global load balancing, both hybrid configurations have higher downstream scores than all-AdamW at every evaluated checkpoint. At the endpoint, `Overall_Avg` is 45.60 for all-AdamW, 47.89 for Muon on experts, and 48.28 for Muon on experts and attention. Their mean LM losses over the last 100 steps are 1.768, 1.746, and 1.737, respectively. Extending Muon to attention provides an improvement in these runs, although the score gap narrows with training. The local and global expert-and-attention variants remain close: their endpoint scores are 48.52 and 48.28. These observations support the hybrid parameter allocation. Downstream evaluation, however, does not distinguish local from global load balancing over this interval. We nonetheless adopt global load balancing for main pre-training because its internal metrics are more stable. Each setting is represented by a single training run.

8 Evaluation

We evaluate 801-30B-A3B through complementary comparisons of base-model performance, general capabilities after supervised fine-tuning (SFT), and agentic capabilities against post-trained models. The base-model comparison assesses the pre-trained checkpoint directly, while the Base+SFT comparison examines general capabilities after adaptation. The agentic comparison further evaluates performance on SWE-bench Verified, SWE-bench Pro, CyberGym, and KernelBench. These settings are presented separately to distinguish direct base-model evaluation

from downstream performance after adaptation.

Given current resource and time constraints, we have focused our initial evaluation and optimization on the settings reported in this section, with particular emphasis on agentic tasks. We plan to broaden task coverage and provide more comprehensive evaluation support in future work.

8.1 Evaluation Settings

8.1.1 Base Evaluation Settings

We evaluate base checkpoints on 13 benchmarks across five categories: commonsense, with ARC-Easy and ARC-Challenge (Clark et al., 2018); reasoning, with MMLU-Pro (Wang et al., 2024b), MMLU-Redux (Gema et al., 2025), BBH (Suzgun et al., 2023), and AGIEval-EN (Zhong et al., 2024); code generation, with HumanEval (Chen et al., 2021), EvalPlus (Liu et al., 2023), and MultiPL-E (Cassano et al., 2023); mathematics, with GSM-8K (Cobbe et al., 2021) and MATH-500 (Lightman et al., 2024); and science, with GPQA-Main (Rein et al., 2024) and SuperGPQA (M-A-P Team et al., 2025). Table 6 summarizes the default prompting, scoring, and generation settings for each benchmark. The model-specific protocol-selection procedure is described at the end of this subsection.

Prompting and generation. All benchmarks use zero-shot prompting except BBH, which uses three demonstrations, and AGIEval-EN, which uses five. For the non-code benchmarks, we use greedy decoding with temperature 0 and generate one candidate per problem. For HumanEval, EvalPlus, and MultiPL-E, we sample four candidates per problem with temperature 0.2 and top-p 0.95. Each code-generation candidate is limited to 2,048 generated tokens. The other generation limits range from 1,024 to 16,384 tokens, as listed in the table.

Table 6: Default base-evaluation settings (except Gemma-4-26B-A4B). Rows follow the five benchmark categories introduced in the text. T denotes temperature, candidates are counted per problem, and generation limits are in tokens. Greedy decoding uses $T = 0$. MultiPL-E scores are weighted by the number of problems in each language.

Category	Benchmark	Metric	Shots	Decoding	T	Top-p	Candidates	Max. tokens
Commonsense	ARC-Easy	Exact match	0	Greedy			1	16,384
	ARC-Challenge	Exact match	0	Greedy			1	16,384
Reasoning	MMLU-Pro	Exact match	0	Greedy			1	16,384
	MMLU-Redux	Exact match	0	Greedy			1	4,096
	BBH	Exact match	3	Greedy			1	1,024
	AGIEval-EN	Exact match	5	Greedy			1	2,048
Code	HumanEval	Avg@4	0	Sampling	0.2	0.95	4	2,048
	EvalPlus	Avg@4	0	Sampling	0.2	0.95	4	2,048
	MultiPL-E	Avg@4	0	Sampling	0.2	0.95	4	2,048
Math	GSM-8K	Exact match	0	Greedy			1	2,048
	MATH-500	Math Verify	0	Greedy			1	16,384
Science	GPQA-Main	Exact match	0	Greedy			1	16,384
	SuperGPQA	Exact match	0	Greedy			1	4,096

Scoring and aggregation. We use exact match for the non-code benchmarks except MATH-500, whose answers are scored with Math Verify. For HumanEval, EvalPlus, and MultiPL-E, we report Avg@4: test-passing outcomes are averaged over the four generated candidates for each problem and then across problems. Our MultiPL-E evaluation covers ten languages: Python, C++, C#, Go, Java, JavaScript, PHP, Rust, Bash, and TypeScript. We aggregate their scores with weights proportional to the number of evaluated problems in each language.

Code execution. For HumanEval, EvalPlus, and MultiPL-E, we set the compilation and execution timeouts to 15 seconds each. Runtime and compiler versions, along with preparation, compilation, and execution commands, are provided in Appendix D (Table 12).

Model-specific protocol selection for Gemma. For Gemma-4-26B-A4B, limited instruction-following ability makes the results particularly sensitive to prompting and evaluation conventions. We therefore evaluate multiple prompt and protocol configurations and report the maximum score obtained for each benchmark.

8.1.2 Post-training Evaluation

General Capabilities. We evaluate general capabilities on MMLU (Hendrycks et al., 2021), MMLU-Pro (Wang et al., 2024b), GPQA Diamond (Rein et al., 2024), AIME 2026 and HMMT February 2026 (Balunović et al., 2025), GSM8K (Cobbe et al., 2021), MATH-500 (Lightman et al., 2024), LiveCodeBench v5 and v6 (Jain et al., 2025), MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), IFBench (Pyatkin et al., 2025), IFEval (Zhou et al., 2023), AA-LCR v1.1 (Artificial Analysis Team, 2025), SimpleQA Verified (Haas et al., 2025), the public set of AA-Omniscience (Jackson et al., 2025), CL-bench (Dou et al., 2026b), and CL-bench Life (Dou et al., 2026a).

We use a sampling temperature of 0.6 throughout. The maximum output length is 262,144 tokens for AIME 2026, HMMT February 2026, LiveCodeBench v5 and v6, CL-bench, and CL-bench Life; 131,072 tokens for GPQA Diamond; 32,768 tokens for MMLU, MMLU-Pro, GSM8K, MATH-500, MBPP, HumanEval, IFBench, and IFEval; and 4,096 tokens for SimpleQA Verified and AA-Omniscience. For CL-bench and CL-bench Life, top-p is set to 0.95. We sample 32 responses per problem for AIME 2026 and HMMT February 2026; eight for LiveCodeBench v5 and v6, GPQA Diamond, MBPP, and HumanEval; four for IFBench, IFEval, AA-LCR v1.1, CL-bench, and CL-bench Life; and one for the remaining benchmarks.

We report accuracy for all benchmarks, averaged over the sampled responses. Mathematical answers are graded using Math-Verify, multiple-choice answers are compared against the reference choices, and generated code is evaluated against the benchmark test cases. AA-LCR v1.1 employs GPT-5.6 Luna (medium) as its judge, while SimpleQA Verified and AA-Omniscience rely on Qwen2.5-32B as the judge. For CL-bench and CL-bench Life, we use DeepSeek-V4.1 with thinking enabled and rubric-level majority voting: each response is judged independently three times, and each rubric's verdict is determined by a majority vote. Only final answer content is graded; responses containing only reasoning receive zero credit on every rubric.

Agentic Capabilities. We evaluate agentic capabilities on SWE-bench Verified (Chowdhury et al., 2024), SWE-bench Pro (Deng et al., 2026), CyberGym (Wang et al., 2026), and KernelBench (Ouyang et al., 2025).

For SWE-bench Verified and SWE-bench Pro, we use the SWE-agent harness (Yang et al., 2024), with a sampling temperature of 0.7, a repetition penalty of 1.1, and a maximum of 300 interaction steps. For SWE-bench Verified, we additionally apply anti-hacking safeguards against the shortcut behaviors summarized in Table 4.

For KernelBench, we use a sampling temperature of 0.6, top-p of 0.95, and a repetition penalty of 1.1. Each model turn allows up to 65,536 output tokens, with a maximum of 20 interaction rounds. Context compression is triggered when the context reaches 150K tokens.

For CyberGym, we use the Claude Code harness (Anthropic, 2025), with a sampling temperature of 1.0 and top-p of 0.95. The context window is 262,144 tokens, with automatic compression triggered at 90% utilization. Each task attempt has a two-hour time limit.

8.2 Base Model Evaluation

We first compare 801-30B-A3B with models for which base checkpoints are available: Qwen3.5-35B-A3B (Qwen Team, 2026a), Qwen3-30B-A3B (Yang et al., 2025a), Kimi-Linear-48B-A3B (Kimi Team et al., 2025), Nemotron-3-Nano-30B-A3B (NVIDIA et al., 2025), Nemotron-3.5-Lightning-30B-A3B (NVIDIA et al., 2026), and Gemma-4-26B-A4B (Gemma Team et al., 2026). Table 7 reports results on the 13 benchmarks spanning commonsense, reasoning, code generation, mathematics, and science, together with their average score.

Competitive base-model performance. 801-30B-A3B achieves an average score of 73.13, the second-highest among the seven base models, behind Qwen3.5 at 76.67 and close to Nemotron-3.5-Lightning at 73.07. It exceeds Qwen3, Nemotron-3-Nano, and Kimi-Linear by 6.97, 4.64, and 8.76 percentage points, respectively. This result establishes a competitive starting point before the SFT evaluation considered below. 801-30B-A3B obtains the highest scores on GSM-8K and HumanEval, reaching 92.80% and 78.81%, respectively. On MMLU-Pro, it scores 68.38%, compared with 67.01% for Qwen3.5 and 69.13% for Nemotron-3.5-Lightning. The base-model results demonstrate a competitive overall profile and distinct task-level strengths of our model.

8.3 SFT Model Results

8.3.1 General Capabilities

We then examine whether the strength of the base model carries over after adaptation. Every model here is fine-tuned on the same SFT data, which is deliberately lightweight: 6.33 B tokens and no reinforcement learning (Section 4.1). Differences among the resulting models therefore mainly reflect what each base model brings to adaptation.

Table 7: Base-model evaluation. All entries are base checkpoints. Scores are percentages; AVG is the unweighted mean of the 13 benchmark scores. Bold indicates the highest score among the models shown. *Gemma-4-26B-A4B reports the maximum score across multiple prompts and evaluation protocols for each benchmark, as described in Section 8.1.1.

Category	Benchmark	801-30B-A3B Base	Qwen3.5 35B-A3B Base	Qwen3 30B-A3B Base	Kimi-Linear 48B-A3B Base	Nemotron 3-Nano 30B-A3B Base	Nemotron 3.5-Lightning 30B-A3B Base	Gemma-4 26B-A4B*
Commonsense	ARC-Easy	97.69	**98.27**	90.40	93.43	93.86	98.06	96.46
	ARC-Challenge	92.75	**97.01**	83.70	85.75	84.98	94.54	89.93
Reasoning	MMLU-Pro	68.38	67.01	45.15	49.56	59.26	**69.13**	43.55
	MMLU-Redux	83.14	**86.14**	62.26	73.44	76.95	83.74	73.30
	BBH	81.23	**86.35**	80.33	76.07	78.50	77.59	72.62
	AGIEval-EN	58.98	**66.56**	62.59	60.49	58.59	58.93	49.30
Code	HumanEval	**78.81**	71.80	71.95	68.14	64.79	69.97	63.11
	EvalPlus	75.46	**83.08**	79.42	75.15	73.02	80.79	53.20
	MultiPL-E	57.74	**66.48**	61.04	59.63	56.04	62.16	18.18
Math	GSM-8K	**92.80**	87.72	82.79	88.86	90.14	91.36	68.46
	MATH-500	73.20	**83.00**	76.60	62.40	76.00	74.00	36.20
Science	GPQA-Main	51.34	**55.58**	33.26	25.22	42.19	51.12	35.27
	SuperGPQA	39.16	**47.68**	30.60	18.63	36.06	38.57	24.90
AVG		73.13	76.67	66.16	64.37	68.49	73.07	55.73

Table 8 reports two groups of benchmarks. The lower block contains the *classic general benchmarks*, a suite of 13 widely used benchmarks covering the general capabilities commonly evaluated in models of this class, summarized by AVG13. The upper block contains five *specialized-capability evaluations*, covering three specific areas: long-context reasoning (AA-LCR v1.1), factual knowledge (AA-OmniScience on its public set and SimpleQA-verified), and learning from context (CL-bench and CL-bench Life). Their mean is reported as AVG5.

Performance on specialized-capability evaluations. 801-30B-A3B achieves an AVG5 of 28.70, leading Qwen3.5 (20.54) by 8.16 points and ranking first on four of the five evaluations. Its strongest results are on long-context reasoning and factual knowledge, particularly AA-LCR v1.1 (61.00%), AA-OmniScience (31.30%), and SimpleQA-verified (34.50%). The pre-training corpus was decontaminated against SimpleQA using 8-gram overlap matching, making test-set leakage an unlikely explanation for this result. The exception is CL-bench Life, where 801-30B-A3B trails Qwen3.5 and Nemotron-3-Nano.

Performance on classic general benchmarks. 801-30B-A3B achieves an AVG13 of 81.34, ranking second behind Qwen3.5-35B-A3B (83.76), consistent with its ranking in base-model evaluation (Table 7). It performs particularly well on mathematics and code, matching the highest Math500 score (98.40%) and reaching 95.75% on GSM8K and 95.12% on HumanEval. Its main weaknesses are in competitive programming and instruction following, as reflected by its results on LiveCodeBench v5 and IFEval Strict.

8.3.2 Agentic Capabilities

We further compare 801-30B-A3B, after the same lightweight SFT, with post-trained models released by their providers, which have generally gone through a full post-training process, on SWE-bench Verified, SWE-bench Pro, CyberGym, and KernelBench. In addition to general-purpose models of a similar size, namely Qwen3-30B-A3B-Thinking-2507 (Qwen Team, 2025), Qwen3.5-35B-A3B (Qwen Team, 2026a), Qwen3.6-35B-A3B (Qwen Team, 2026b), K2-Horizon-MoVA-36B-A4B (IFM Team, 2026), Gemma-4-26B-A4B-it (Gemma Team et al., 2026), and Nemotron-3.5-Lightning-30B-A3B (NVIDIA et al., 2026), the comparison includes Flint-40B-A2.5B (Recursive, 2026), a coding-specialized MoE model. Table 9 summarizes these results. Unlike the Base+SFT comparison above, this setting compares our model after 6.33B tokens of tiny SFT with released models that have gone through complete post-training pipelines, so we report it as a reference point for post-training headroom rather than as a like-for-like comparison.

801-30B-A3B achieves the highest score on KernelBench (77.60%) and the highest average across the four benchmarks (58.43). It also achieves competitive results on SWE-bench Pro and CyberGym. Notably, 801-30B-A3B

Table 8: General capabilities of Base+SFT models(%). All base models are fine-tuned on the same 6B-token SFT data without reinforcement learning. Scores are percentages; higher is better. AVG5 and AVG13 are the unweighted means of the upper and lower blocks, respectively. Bold indicates the highest score among the models shown, including ties.

Benchmark	801-30B-A3B Base+SFT	Qwen3.5 35B-A3B Base+SFT	Nemotron 3-Nano 30B-A3B Base+SFT	Qwen3 30B-A3B Base+SFT	Nemotron 3.5-Lightning 30B-A3B Base+SFT	Gemma-4 26B-A4B Base+SFT
Specialized-capability Evaluations						
AA-LCR v1.1	**61.00**	52.00	32.00	19.75	18.50	18.25
AA-OmniScience (public set)	**31.30**	21.70	16.70	18.80	17.80	19.00
SimpleQA-verified	**34.50**	12.00	7.00	9.10	9.90	10.20
CL-bench	**13.22**	12.80	7.74	6.95	5.00	4.42
CLbench-life	3.46	**4.20**	3.95	2.22	1.48	1.73
AVG5	🥇 **28.70**	🥈 20.54	🥉 13.48	11.36	10.54	10.72
Classic General Benchmarks						
AIME 2026	86.77	**87.60**	78.65	82.29	81.25	45.73
HMMT Feb 2026	68.37	**71.02**	60.04	65.62	64.02	31.06
LiveCodeBench v5	69.53	**77.37**	72.00	65.73	70.83	48.07
LiveCodeBench v6	66.41	**71.95**	68.19	62.02	64.22	41.41
GPQA Diamond	74.75	**75.76**	66.54	65.97	70.45	53.28
MMLU-Pro	80.00	**82.13**	76.58	77.40	79.22	72.62
IFBench	63.33	**64.92**	59.42	58.08	58.17	45.17
IFEval Strict	85.54	**89.33**	87.34	86.74	86.69	68.95
MMLU	84.62	**88.75**	83.34	84.01	86.60	82.87
GSM8K	95.75	**95.98**	94.62	95.60	**95.98**	93.78
Math500	**98.40**	**98.40**	96.80	96.40	97.20	84.80
MBPP	88.83	**90.30**	87.44	86.25	89.75	72.55
HumanEval	95.12	**95.35**	93.67	91.39	94.05	84.98
AVG13	🥈 81.34	🥇 83.76	78.82	78.27	🥉 79.88	63.48

Table 9: Agentic benchmark results (%). Bold indicates the highest score among the models shown; AVG is the unweighted mean of the four benchmark scores. All baselines are provider-released post-trained models; 801-30B-A3B is evaluated after the lightweight SFT stage only.

Model	SWE-bench Verified	SWE-bench Pro	CyberGym	KernelBench	AVG
Qwen3-30B-A3B-Thinking-2507	20.30	7.01	1.19	5.60	8.53
Qwen3.5-35B-A3B	69.10	39.23	13.47	17.20	34.75
Qwen3.6-35B-A3B	71.65	45.28	24.22	37.20	44.59
K2-Horizon-MoVA-36B-A4B	64.15	44.19	**39.35**	33.60	45.32
Gemma-4-26B-A4B-it	57.15	34.57	8.49	44.00	36.05
Nemotron-3.5-Lightning-30B-A3B	55.35	32.49	8.29	20.40	29.13
Flint-40B-A2.5B	**72.80**	**53.01**	33.91	41.20	50.23
801-30B-A3B Base+SFT	68.95	48.29	38.90	**77.60**	**58.43**

is evaluated after only a lightweight SFT stage using 6.33B tokens, without reinforcement learning, whereas the compared models have generally undergone much more extensive post-training. These results demonstrate the strong agentic capabilities that 801-30B-A3B already exhibits after lightweight post-training.

8.4 Learning aptitude

Lower start, faster learning on BFCL At the first checkpoint (step 55), 801-30B-A3B scores 31.32 on BFCL Full, below the 40.04 of Qwen3-30B-A3B. Its slope, however, is 8.62 points per tenfold increase in SFT steps against 3.33, about 2.6× steeper. From step 697 onward it leads at every shared checkpoint and ends at 48.65 against 46.26 at step

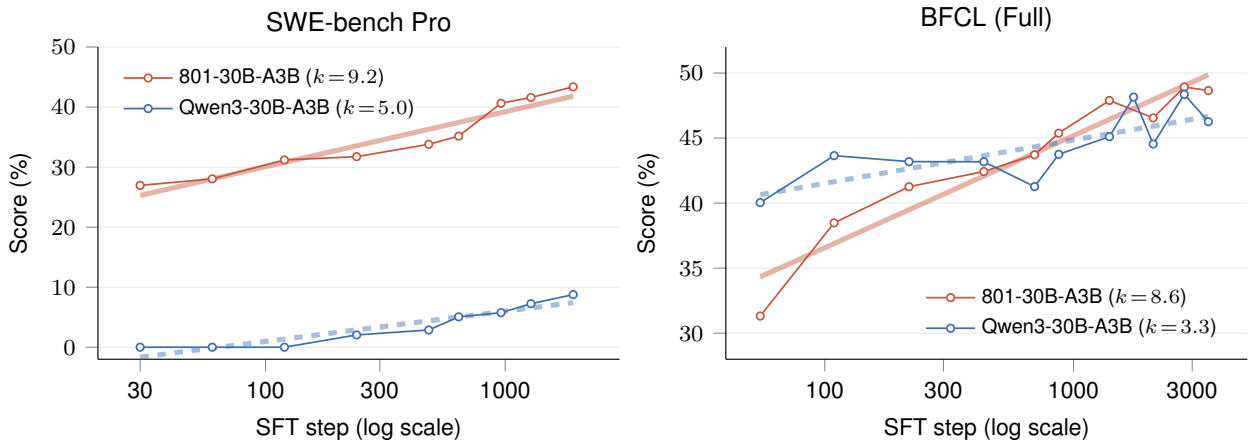


Figure 25: Learning aptitude of 801-30B-A3B and Qwen3-30B-A3B on SWE-bench Pro (left) and BFCL Full (right). Hollow markers are scores at successive SFT checkpoints; the pale line is the least-squares fit of Eq. (1), and k in the legend is its slope, the gain in points per tenfold increase in SFT steps.

3,485. This is the case that motivates learning aptitude: a checkpoint that starts lower but learns faster overtakes one that starts higher once fine-tuning runs long enough.

Higher start and faster learning on SWE-bench Pro On SWE-bench Pro, 801-30B-A3B leads from the first checkpoint, 26.95 against 0 at step 30, and learns faster, with a slope of 9.15 against 5.03. Qwen3-30B-A3B stays at zero for the first three checkpoints (up to step 120) and reaches 8.76 at step 1,922, where 801-30B-A3B reaches 43.37. Here the intercept and the slope point the same way, so the gap widens rather than closes as fine-tuning continues.

9 Conclusion

We attempted to answer a question that current RSI successes leave open: what can agents do for frontier pre-training, where the *propose* $\rightarrow$ *run* $\rightarrow$ *measure* $\rightarrow$ *update* loop cannot close because the run is not repeatable and the objective is not a score? Our answer, drawn from building 801-30B-A3B in three months, is to organize improvement around three types of loops. Debugging loops comprise the infra debugging loop and the data debugging loop, which connect unusual internal metrics and output patterns to their causes. Five cross-stack diagnoses, resolved in about one month, came out of these investigations. The headroom loop is instantiated as the recipe loop, using tiny-SFT, learning aptitude, and max-over-protocols evaluation to guide recipe revisions and monitor training progress. Experience-to-metric loops comprise the classifier development loop, the site discovery loop, and the crawl-coverage loop, turning experience-based judgment into feedback that agents can use to improve data workflows. All of them share a context infrastructure that integrates evidence across teams at scale and preserves negative results and their reasons.

The resulting model is evidence that this regime works at scale. 801-30B-A3B is state-of-the-art among open base models at the 30B MoE scale in both direct base evaluation and matched-SFT comparison, and after a 6B-token SFT stage it approaches a fully post-trained frontier of its size class on agentic tasks. But we hold the model to be the byproduct and the account to be the contribution. Our experience suggests that RSI need not stop at frontier pre-training. It takes a different form: loops eliminate anomalies, headroom measurements guide training, and quantitative feedback improves data workflows. Shared context allows evidence and lessons to accumulate across teams and iterations. There may be no second run, but the one we have can still learn from its own progress.

Author Contributions

The authors are listed in alphabetical order by their last names.

Contributors: Xiaojie Cai, Jiayi Chen, Junwei Chen, Yu Chen, Sikai Cheng, Jinquan Dai, Siyuan Feng, Bin Fu, Dayuan Fu, Guosheng Gao, Qipeng Guo, Chenwei He, Zhulin Hu, Yaxing Huang, Shuomeng Jia, Mohan Jiang, Keyang Lei, Hongliang Li, Xuefeng Li, Zhaocheng Li, Jifan Lin, Pengfei Liu, Xinghan Liu, Yixiu Liu, Pengrui Lu, Siyuan Lu, Kai Lv, Tiantian Mi, Yang Nan, Yu Qiao, Yiwei Qin, Xipeng Qiu, Junhao Shi, Runzhou Shi, Jiadi Su, Bohao Tang, Hanning Wang, Keqi Wang, Pan Wang, Ran Wang, Shaojing Wang, Yishuo Wang, Zhixin Wang, Shenyu Wu, Muhang Xie, Tianze Xu, Hang Yan, Lvmanshan Ye, Hongche Yu, Junjie Yu, Lantian Zhang, Ruihang Zhang, Bin Zhao, Chenyang Zhou, Zijin Zhou, Wenhong Zhu, Zemu Zhu, Yumin Zhuang, Haoyang Zou

Affiliations: Shanghai Innovation Institute, Xinxu AI

References

[1] Wasi Uddin Ahmad, Somshubra Majumdar, et al. 2025. Opencodereasoning-ii: A simple test time scaling approach via self-critique.

[2] Anomaly. 2025. OpenCode: The open source AI coding agent. https://github.com/anomalyco/opencode. Accessed 2026-10-14.

[3] Anthropic. 2025. Claude Code. https://github.com/anthropics/claude-code. Accessed 2026-10-14.

[4] Anthropic. 2026. Messages API reference. https://docs.claude.com/en/api/messages. Accessed 2026-10-14.

[5] Artificial Analysis Team. 2025. Artificial analysis long context reasoning benchmark (AA-LCR). Hugging Face dataset https://huggingface.co/datasets/ArtificialAnalysis/AA-LCR. Version 1.1 released September 2026; methodology at https://artificialanalysis.ai/methodology/intelligence-benchmarking.

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models.

[7] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. ArXiv:2505.20411.

[8] Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. 2025. MathArena: Evaluating LLMs on uncontaminated math competitions. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.

[9] Tamay Besiroglu, Ege Erdil, Matthew Barnett, and Josh You. 2024. Chinchilla scaling: A replication attempt. *arXiv preprint arXiv:2404.10102*.

[10] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2019. PIQA: Reasoning about physical commonsense in natural language. *arXiv preprint arXiv:1911.11641*.

[11] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, et al. 2023. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

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

[13] Mingguang Chen, Licheng Wang, and Bo Qu. 2026. Recursive self-improvement in AI: From bounded self-refinement to autonomous research loops.

[14] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. 2024. Introducing SWE-bench Verified. OpenAI Blog. Updated February 24, 2025.

[15] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

[16] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

[17] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. 2026. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*. ArXiv:2509.16941.

[18] Shihan Dou, Yujiong Shen, Chenhao Huang, Junjie Ye, Jiayi Chen, Junzhe Wang, Qianyu He, Shichun Liu, Changze Lv, Jiahang Lin, et al. 2026a. CL-bench Life: Can language models learn from real-life context?

[19] Shihan Dou, Ming Zhang, Zhangyue Yin, Chenhao Huang, Yujiong Shen, Junzhe Wang, Jiayi Chen, Yuchen Ni, Junjie Ye, Cheng Zhang, et al. 2026b. CL-bench: A benchmark for context learning.

[20] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378.

[21] Earendil Works. 2025. Pi: A minimal, extensible agent harness. https://github.com/earendil-works/pi. Accessed 2026-10-14.

[22] Factory AI. 2025. Droid: Factory's agent-native development platform. https://github.com/factory-ai/factory. Accessed 2026-10-14.

[23] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39.

[24] Dayuan Fu, Mohan Jiang, Tongyu Wang, Dian Yang, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. 2026a. davinci-kernel: Co-evolving skill selection, summarization, and utilization via RL for GPU kernel optimization. *arXiv preprint arXiv:2606.16497*.

[25] Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. 2026b. daVinci-Env: Open SWE Environment Synthesis at Scale. *CoRR*, abs/2603.13023.

[26] Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, et al. 2025. Are we done with MMLU? In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 5069–5096. Association for Computational Linguistics.

[27] Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. 2026. Gemma 4 technical report.

[28] Taicheng Guo, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2026. Autollmresearch: Training research agents for automating LLM experiment configuration – learning from cheap, optimizing expensive. *arXiv preprint arXiv:2605.11518*.

[29] Lukas Haas, Gal Yona, Giovanni D'Antonio, Sasha Goldshtein, and Dipanjan Das. 2025. SimpleQA Verified: A reliable factuality benchmark to measure parametric knowledge.

[30] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *International Conference on Learning Representations (ICLR)*.

[31] Yue Huang, Yanbo Wang, Lichi Li, Yuanyuan Chen, Andreas Haupt, Xiaonan Luo, Zheyuan Liu, Jihye Choi, et al. 2026. RSI Arena: Towards recursive self-improvement with human feedback. *arXiv preprint arXiv:2610.33624*.

[32] IFM Team. 2026. Introducing K2 Horizon: Frontier performance, radically open. Official blog, https://ifm.ai/blog/k2/. Model card: https://huggingface.co/IFM/K2-Horizon-MoVA-36B-A4B.

[33] Declan Jackson, William Keating, George Cameron, and Micah Hill-Smith. 2025. AA-Omniscience: Evaluating cross-domain knowledge reliability in large language models. Public set: https://huggingface.co/datasets/ArtificialAnalysis/AA-Omniscience-Public.

[34] jacobkahn. 2025. Repo state loopholes during agentic evaluation. GitHub issue #465, SWE-bench/SWE-bench. Opened 2025-09-03 by GitHub user jacobkahn; marked fixed for SWE-bench Verified on 2026-02-03.

[35] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

[36] Mohan Jiang, Dayuan Fu, Junhao Shi, Ji Zeng, Weiye Si, Keyu Li, Xuefeng Li, Yang Xiao, Wenjie Li, Dequan Wang, and Pengfei Liu. 2026. daVinci-Agency: Unlocking Long-Horizon Agency Data-Efficiently. *CoRR*, abs/2602.02619.

[37] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR)*. ArXiv:2310.06770.

[38] Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. 2024. Muon: An optimizer for hidden layers in neural networks. Blog post, https://kellerjordan.github.io/posts/muon/. December 8, 2024; reference implementation at https://github.com/KellerJordan/Muon.

[39] Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. 2017. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611.

[40] Andrej Karpathy. 2026. autoresearch: AI agents running research on single-GPU nanochat training automatically. https://github.com/karpathy/autoresearch.

[41] Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. 2025. Kimi Linear: An expressive, efficient attention architecture.

[42] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations (ICLR)*.

[43] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

[44] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. 2025. Muon is scalable for LLM training. *arXiv preprint arXiv:2502.16982*.

[45] Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*.

[46] M-A-P Team, Xinrun Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, King Zhu, Minghao Liu, Yiming Liang, Xiaolong Jin, et al. 2025. SuperGPQA: Scaling LLM evaluation across 285 graduate disciplines. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems (NeurIPS)*. ArXiv:2502.14739.

[47] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[48] Maxim AI. 2025. Bifrost: AI gateway. https://github.com/maximhq/bifrost. Accessed 2026-10-14.

[49] Yan Meng, Dhruv Srikanth, Bingchen Zhao, Zhengyao Jiang, and Yuxiang Wu. 2026. Autodata: Agentic search for pre-training data selection. *arXiv preprint arXiv:2609.19754*.

[50] Tiantian Mi, Dongming Shan, Zhen Huang, Yiwei Qin, Muhang Xie, Yuxuan Qiao, Yixiu Liu, Chenyang Zhou, and Pengfei Liu. 2026. Data Darwinism Part II: DataEvolve – AI can Autonomously Evolve Pretraining Data Curation. *CoRR*, abs/2603.14420.

[51] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2381–2391.

[52] NeoHorse Team, Guoliang Cao, Guohao Dai, Tianyu Guo, Kai Han, Hailin Hu, Zihan Jiang, Xiang Kuang, Boxun Li, Yulong Li, et al. 2026. NeoHorse-1: Towards recursive self-improvement via agentic post-training with routing harness.

[53] Nous Research. 2025. Hermes Agent: The self-improving AI agent. https://github.com/NousResearch/hermes-agent. Accessed 2026-10-14.

[54] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery.

[55] NVIDIA, Aaron Blakeman, Aaron Grattafiori, Aarti Basant, Abhibha Gupta, Abhinav Khattar, Adi Renduchintala, Aditya Vavre, Akanksha Shukla, Akhiad Bercovich, et al. 2025. Nemotron 3 Nano: Open, efficient mixture-of-experts hybrid Mamba-Transformer model for agentic reasoning.

[56] NVIDIA, Aastha Jhunjhunwala, Aayushi Ashokkumar Srivastava, Abhibha Gupta, Abhinav Garg, Abhinav Khattar, Achsah Wildish, Adam Rajfer, Adi Renduchintala, Adil Asif, Adriana Flores Miranda, et al. 2026. Nemotron 3.5 Super and Lightning: Open, efficient mixture-of-experts hybrid Mamba-Transformer models for agentic AI. Model card: https://huggingface.co/nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-Base-BF16.

[57] OpenAI. 2025. Codex CLI. https://github.com/openai/codex. Accessed 2026-10-14.

[58] OpenAI. 2026a. Chat completions API reference. https://platform.openai.com/docs/api-reference/chat. Accessed 2026-10-14.

[59] OpenAI. 2026b. Responses API reference. https://platform.openai.com/docs/api-reference/responses. Accessed 2026-10-14.

[60] OpenClaw contributors. 2026. OpenClaw: Your assistant, on your devices, in your chats. https://github.com/openclaw/openclaw. Accessed 2026-10-14.

[61] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pages 47356–47415. PMLR. ArXiv:2502.10517.

[62] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pages 47717–47737. PMLR. ArXiv:2412.21139.

[63] Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. 2025. Generalizing verifiable instruction following. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.

[64] Yiwei Qin, Zhen Huang, Tiantian Mi, Weiye Si, Chenyang Zhou, Qipeng Guo, Siyuan Feng, and Pengfei Liu. 2026a. Data Darwinism Part I: Unlocking the Value of Scientific Data for Pre-training. *CoRR*, abs/2602.07824.

[65] Yiwei Qin, Yixiu Liu, Tiantian Mi, Muhang Xie, Zhen Huang, Weiye Si, Pengrui Lu, Siyuan Feng, Xia Wu, Liming Liu, Ye Luo, Jinlong Hou, Qipeng Guo, Yu Qiao, and Pengfei Liu. 2026b. daVinci-LLM: Towards the Science of Pretraining. *CoRR*, abs/2603.27164.

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

[67] Qwen Team. 2025. Qwen3-30B-A3B-Thinking-2507. Hugging Face model card, https://huggingface.co/Qwen/Qwen3-30B-A3B-Thinking-2507.

[68] Qwen Team. 2026a. Qwen3.5: Towards native multimodal agents. Official blog, https://qwen.ai/blog?id=qwen3.5. Model cards: https://huggingface.co/Qwen/Qwen3.5-35B-A3B, https://huggingface.co/Qwen/Qwen3.5-35B-A3B-Base.

[69] Qwen Team. 2026b. Qwen3.6-35B-A3B: Agentic coding power, now open to all. Official blog, https://qwen.ai/blog?id=qwen3.6-35b-a3b. Model card: https://huggingface.co/Qwen/Qwen3.6-35B-A3B.

[70] Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. 2026. Posttrainbench: Can LLM agents automate LLM post-training? *arXiv preprint arXiv:2603.08640*.

[71] Recursive. 2026. Introducing Flint. Official blog, https://recursive.com/updates/introducing-recursive-flint. Model card: https://huggingface.co/RecursiveSI/flint.

[72] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2024. GPQA: A graduate-level Google-proof Q&A benchmark. In *First Conference on Language Modeling (COLM)*.

[73] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. WinoGrande: An adversarial Winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*.

[74] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, et al. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

[75] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*.

[76] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. 2024. AI models collapse when trained on recursively generated data. *Nature*, 631:755–759.

[77] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced Transformer with rotary position embedding. *Neurocomputing*, 568:127063.

[78] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2023. Challenging BIG-Bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051. Association for Computational Linguistics.

[79] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. 2024a. Auxiliary-loss-free load balancing strategy for mixture-of-experts. *arXiv preprint arXiv:2408.15664*.

[80] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2025. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations (ICLR)*. ArXiv:2407.16741. Code: https://github.com/OpenHands/OpenHands.

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

[82] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. 2026. CyberGym: Evaluating AI agents' real-world cybersecurity capabilities at scale. In *The Fourteenth International Conference on Learning Representations (ICLR)*. ArXiv:2506.02548.

[83] Lilian Weng. 2026. Scaling laws, carefully. Lil'Log blog post, https://lilianweng.github.io/posts/2026-06-24-scaling-laws/. Accessed 2026-10-14.

[84] Peng Xia, Rujun Han, Zifeng Wang, Yanfei Chen, Yufan Zhuang, Yoonho Lee, Chengsong Huang, Han Yu, Zhongying CuiZhu, Yifei Ming, et al. 2026. RRSI: Regularized recursive self-improvement of agent harnesses.

[85] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. 2024. Effective long-context scaling of foundation models. 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 4643–4663, Mexico City, Mexico. Association for Computational Linguistics.

[86] Mingyu Xu, Cheng Fang, Keyue Jiang, Yuqian Zheng, Yanghua Xiao, Baojian Zhou, Qifang Zhao, Suhang Zheng, Xiuwen Zhu, Jiyang Tang, Yongchi Zhao, Yijia Luo, Zhiqi Bai, Yuchi Xu, Wenbo Su, Wei Wang, Bing Zhao, Lin Qu, and Xiaoxiao Xu. 2026a. Logics-stem: Empowering LLM reasoning via failure-driven post-training and document knowledge enhancement.

[87] Weixian Xu, Yixiu Liu, Yang Nan, Lyumanshan Ye, Xiangkun Hu, Zhen Qin, and Pengfei Liu. 2026b. Neural Architecture Discovery via Autonomous Evolution.

[88] Weixian Xu, Tiantian Mi, Yixiu Liu, Yang Nan, Zhimeng Zhou, Lyumanshan Ye, Lin Zhang, Yu Qiao, and Pengfei Liu. 2026c. ASI-Evolve: AI Accelerates AI. *CoRR*, abs/2603.29640.

[89] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

[90] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652. ArXiv:2405.15793. Code: https://github.com/SWE-agent/SWE-agent.

[91] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025b. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. ArXiv:2504.21798.

[92] Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, et al. 2026. Towards automated kernel generation in the era of LLMs.

[93] Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems (NeurIPS)*.

[94] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800.

[95] Ji Zeng, Dayuan Fu, Tiantian Mi, Yumin Zhuang, Yaxing Huang, Xuefeng Li, Lyumanshan Ye, Muhang Xie, Qishuo Hua, Zhen Huang, et al. 2026. davinci-dev: Agent-native mid-training for software engineering. *arXiv preprint arXiv:2601.18418*.

[96] Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. 2025. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[97] Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2024. AGIEval: A human-centric benchmark for evaluating foundation models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2299–2314. Association for Computational Linguistics.

[98] Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. 2025. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270*.

[99] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models.

[100] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. 2022. ST-MoE: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*.

A Layer-wise MaxVio Monitoring

Figure 26 reports the MaxVio trajectories for all 48 Mixture-of-Experts layers over training. Most layers remain within a relatively stable range, while a small number develop substantially larger violations; in particular, layer 15 exhibits the pronounced rise discussed in Figure 7. The vertical dashed lines indicate the stage boundaries, and the horizontal axis is training tokens in trillions.

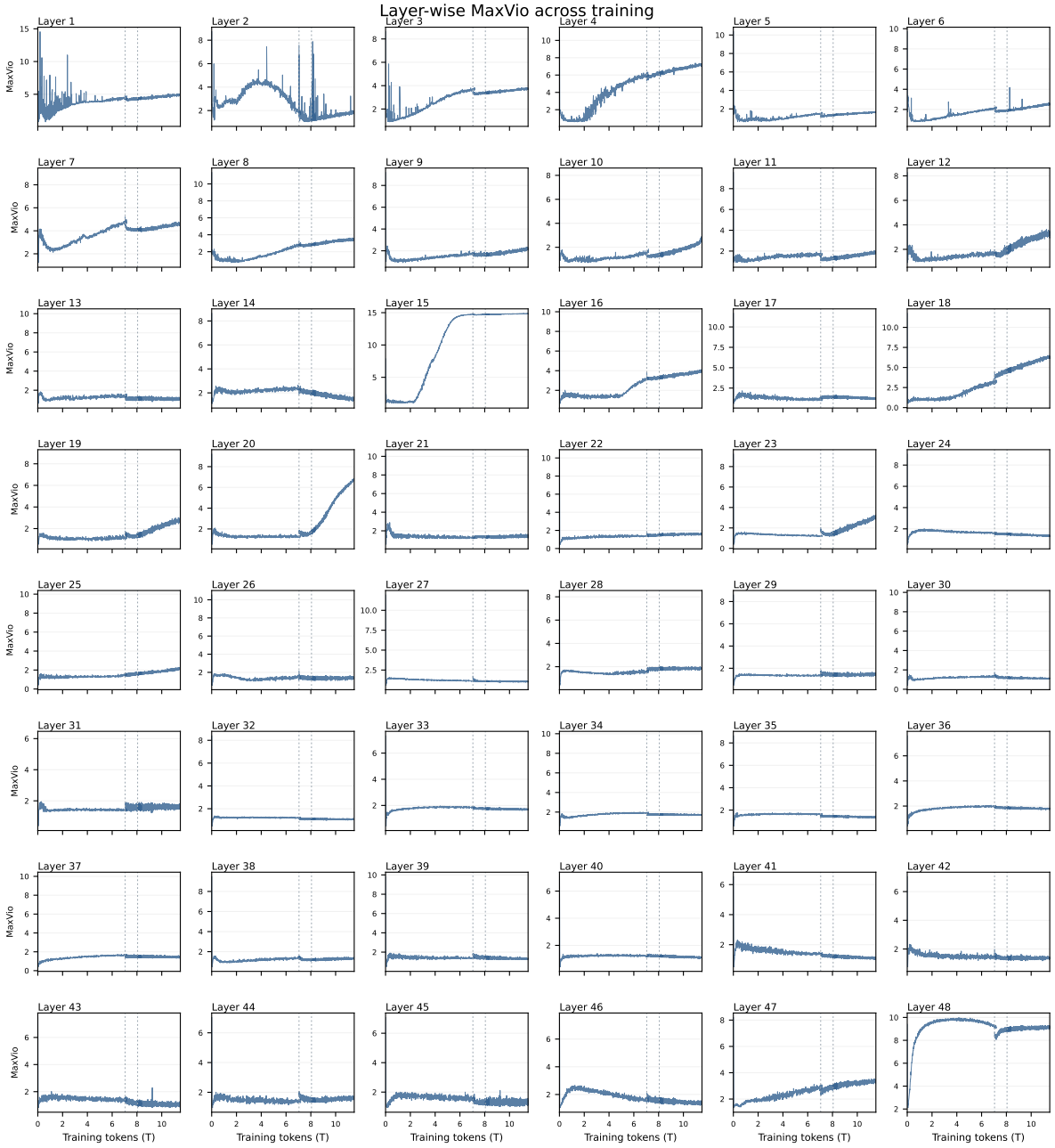


Figure 26: Layer-wise MaxVio trajectories for all 48 MoE layers.

B Benchmarks for Pre-training Monitoring

Table 10 summarizes the 22 evaluation items used to monitor pre-training, grouped as in the training-progress plots. Overall_Avg is the unweighted mean of their 22 reported percentage scores.

Table 10: The 22-benchmark pre-training monitoring suite, with brief introductions to the evaluated tasks.

Benchmark	Description
General commonsense (7)	
ARC-Easy (Clark et al., 2018)	School-level science questions, easy split.
ARC-Challenge (Clark et al., 2018)	More challenging science questions requiring reasoning.
HellaSwag (Zellers et al., 2019)	Selecting plausible continuations of everyday events.
OpenBookQA (Mihaylov et al., 2018)	Combining basic science facts with commonsense.
PIQA (Bisk et al., 2019)	Physical commonsense and practical problem solving.
WinoGrande (Sakaguchi et al., 2019)	Resolving ambiguous references using commonsense.
TriviaQA (Joshi et al., 2017)	Factual short-answer questions without evidence context.
General reasoning (6)	
MMLU (Hendrycks et al., 2021)	Multiple-choice knowledge questions across 57 subjects.
MMLU-Pro (Wang et al., 2024b)	Harder, reasoning-oriented questions across 14 subjects.
MMLU-Redux (Gema et al., 2025)	Human-reviewed MMLU questions.
BBH (Suzgun et al., 2023)	Diverse challenging reasoning tasks; 27 configured subtasks.
AGIEval-EN (Zhong et al., 2024)	Ten English exam subsets covering logic, reading, and math.
DROP (Dua et al., 2019)	Reading comprehension with discrete reasoning.
Code (3)	
HumanEval (Chen et al., 2021)	Implementing Python functions from signatures and natural-language specifications to assess programming and problem-solving ability.
EvalPlus (Liu et al., 2023)	Code-generation evaluation with augmented tests, emphasizing functional correctness and edge-case handling.
MBPP (Austin et al., 2021)	Writing short Python programs from natural-language requirements.
Math (2)	
GSM-8K (Cobbe et al., 2021)	Grade-school word problems requiring multi-step arithmetic.
MATH-500 (Lightman et al., 2024)	Competition-style mathematics.
Science (4)	
GPQA-Main (Rein et al., 2024)	Graduate-level biology, physics, and chemistry questions.
SuperGPQA (M-A-P Team et al., 2025)	Graduate-level knowledge and reasoning across diverse disciplines, including science, engineering, medicine, law, economics, and the humanities.
MMLU-STEM	The 19 MMLU subjects: abstract algebra, anatomy, astronomy, college biology, college chemistry, college computer science, college mathematics, college physics, computer security, conceptual physics, electrical engineering, elementary mathematics, high school biology, high school chemistry, high school computer science, high school mathematics, high school physics, high school statistics, and machine learning.
MMLU-Pro-STEM	Six MMLU-Pro domains: biology, chemistry, computer science, engineering, mathematics, and physics.

C Evaluation Protocols for Base Models

Figure 13 evaluates the same checkpoints under several protocols. A protocol is determined by four choices: perplexity-based (PPL) or generation-based scoring, the number of in-context examples (k-shot), the required final-answer format, and the answer-matching method. Table 11 defines each protocol label used in the figure.

Table 11: Evaluation protocols in Figure 13.

Protocol	In-context examples	Final-answer format	Scoring
Non-code tasks			
5-shot	5 solved examples: final answers only for ARC, GPQA-Main, MMLU-Redux, and AGIEval-EN; worked solutions for MMLU-Pro and SuperGPQA	"The answer is (X).", shown by the examples	Extracted letter, exact match
4-shot Minerva	4 worked solutions in Minerva format (MATH-500)	Follows the examples	Math equivalence
8-shot CoT	8 worked solutions (GSM-8K)	"The answer is N."	Strict match, exact match
3-shot CoT	3 official worked solutions per subtask (BBH)	"the answer is . . . "	Extracted answer, exact match
0-shot answer-is	None	Instructed to end with "the answer is X"	Extracted answer, exact match[†]
0-shot boxed	None	Instructed to put the answer in `\boxed{}`	Extracted answer, exact match[†]
PPL acc_norm	None	No generation	Highest length-normalized option log-likelihood
Code tasks			
3-shot completion	3	Continue the bare function signature and docstring	Unit tests, pass@1
k-shot instruction + signature	3 or 0	Instruction prompt; signature pre-filled in the answer	Unit tests, pass@1
k-shot instruction, no signature	3 or 0	Instruction prompt; the model writes the full function	Unit tests, pass@1

[†]Math equivalence for MATH-500.

D Code Evaluation Environments

Table 12 provides the code execution environments for the base-model evaluations in Section 8.1.1. HumanEval and EvalPlus use the Python environment, while MultiPL-E uses all ten language environments.

Table 12: Code execution environments. Preparation or compilation commands precede execution. Java uses JVM build `25.0.1-internal-adhoc.conda.src`. `JAVATUPLES_JAR` and `TSC_JS` identify the JavaTuples dependency and the TypeScript compiler entry point, respectively.

Language / runtime	Preparation and execution commands
Python CPython 3.10.19	`exec(check_program, exec_globals)`
C++ GCC 13.3.0 libstdc++ 15.2.0	`g++ -std=c++17 program.cpp -o program` `./program`

C#
Mono JIT / mcs 6.12.0.199
	`mcs -define:DEBUG -r:System.Numerics.dll \`
	`  -out:Program.exe Program.cs`
	`MONO_TRACE_LISTENER=Console.Error mono Program.exe`

Go Go 1.23.4 (linux/amd64)	`go mod init solution` `go test -p=8`
Java OpenJDK / javac 25.0.1-internal	`javac -cp ".:$JAVATUPLES_JAR" Problem.java` `java -ea -cp ".:$JAVATUPLES_JAR" Problem`
JavaScript Node.js 20.12.2	`node program.js`
PHP PHP CLI 8.5.4 Zend Engine 4.5.4	`php program.php`
Rust rustc 1.96.0; LLVM 22.1.2	`rustc --edition 2021 program.rs -o program` `./program`
Bash GNU Bash 5.1.16	`bash program.sh`

TypeScript
TypeScript 5.4.5
Node.js 20.12.2
	`node "$TSC_JS" program.ts --target es2020 \`
	`  --module commonjs --lib es2020,dom --skipLibCheck`
	`node program.js`