Poolside Team

Laguna M.1/XS.2 Technical Report

We present Laguna M.1 and Laguna XS.2, two Mixture-of-Experts foundation models built for long-horizon, agentic coding: M.1 has 225.8B total parameters (23.4B activated per token) and XS.2 has 33.4B total (3B activated). Both models were trained from scratch end-to-end inside the same internal system that we refer to as our *Model Factory*: a tightly-integrated stack of versioned data, training, evaluation, and inference components that turn model development into an industrial process. We describe the principles and design choices of the Model Factory and also detail the end-to-end training process of our models, throughout pre-training data and architecture, post-training stages, evaluation, and quantization.

On agentic software engineering and terminal benchmarks (SWE-bench Verified, SWE-bench Multilingual, SWE-Bench Pro, and Terminal-Bench 2.0) M.1 and XS.2 are competitive with state-of-the-art open models in their respective weight classes. Laguna XS.2 weights are released under Apache 2.0 at https://huggingface.co/collections/poolside/laguna-xs2.

RESEARCH@POOLSIDE.AI

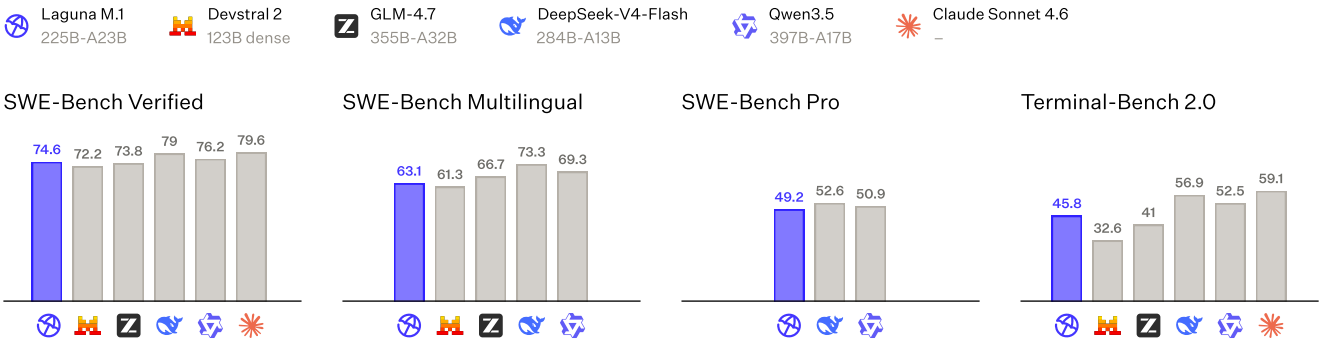
MAY 25, 2026

1 Introduction

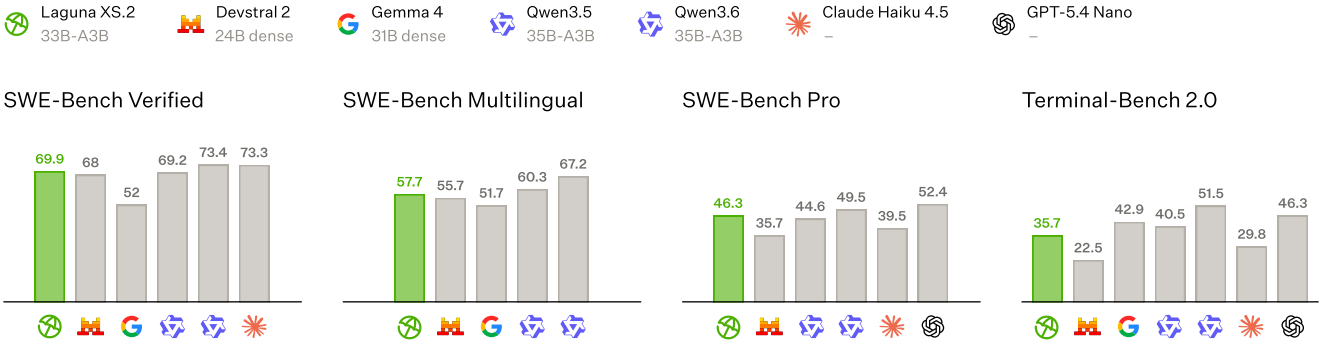
In this report, we present Laguna M.1 and Laguna XS.2, two foundation models built in quick succession, together with the engineering-first *Model Factory* methodology we used to build them. M.1 and XS.2 are capable agentic coding models built for long-horizon, multi-step work, competitive with state-of-the-art models in their weight class. We believe mastering agentic coding is the key to unlocking generalized autonomous problem solving for all knowledge tasks, since code is the most general medium for expressing and verifying solutions to structured problems.

M.1 and XS.2 are Mixture-of-Experts foundation models. M.1 has 225.8B total parameters with 23.4B activated per token; XS.2 has 33.4B total with 3B activated. Both models were trained from scratch, end-to-end, inside the same internal system, which we refer to as the *Model Factory*. One focus of this report is the construction process itself, illustrated by the lessons we learned building M.1, and implementing them in quick succession to build XS.2 afterwards. We started XS.2 just after pre-training of M.1 finished, and the end-to-end time from start of training to release of XS.2 spanned only five weeks. The weights of XS.2 are available to the public under the Apache 2.0 license, and can be accessed via https://huggingface.co/collections/poolside/laguna-xs2.

Results. Both models are competitive with state-of-the-art models of comparable size on agentic tasks. Figure 1 shows our evaluation results on agentic software engineering and terminal task benchmarks: SWE-bench Verified, SWE-bench Multilingual, SWE-Bench Pro, and Terminal-Bench 2.0. Details of our evaluation protocol can be found in Section 6.



(a) M.1 results.



(b) XS.2 results.

Figure 1: Laguna M.1 and XS.2 results on agentic benchmarks compared to Devstral 2 and Devstral Small 2 [57]; Qwen3.5 397B-A17B and 35B-A3B [68]; Qwen3.6 35B-A3B [69]; GLM-4.7 [28]; DeepSeek-V4 Flash [21]; Claude Sonnet 4.6 [7] and Haiku 4.5 [6]; Gemma 4 31B dense [29]; and GPT-5.4 Nano [59].

Model Factory Approach. In addition to sharing technical details of Laguna M.1 and XS.2, we want to draw attention to the underlying process that produced these models. Building XS.2 from inception to delivery in five weeks was only possible because we treat foundation model development as an industrial process. We have built a set of components and processes which we call our *Model Factory*, with the aim to accelerate the process of research and model building, so that our time can be focused on defining and investigating genuine research questions, while integration and plumbing are automated as much as possible. We describe the guiding principles of our Model Factory in Section 2, and show examples of how this works in practice throughout this report.

Technical Overview. We built M.1 and XS.2 from scratch, and share technical details on our Model Factory, architecture, pre-training data, post-training, quantization, and evaluation pipeline, in the hope of contributing to open research.

- **Model Factory.** We describe the design philosophy of our Model Factory, and illustrate the principles that we follow to make model building and research into a repeatable industrial process rather than artisanal projects in Section 2.
- **Pre-training.** We share details of our pre-training in Section 3. This includes architecture design choices for both models, findings on efficiency and stability when training large MoEs from scratch.
  In addition, we also go into our pre-training data strategy. Both Laguna models were trained across more than 30T tokens during pre-training, drawn from a mix of web, code, and synthetic sources. We describe our data curation and preparation pipeline, as well as our data mixture tuning.
- **Post-training.** We describe our post-training pipeline in Section 4, which is divided into imitation learning stages and a final reinforcement learning stage. We discuss how the recipe transferred between M.1 and XS.2, and the data and formatting choices that proved critical for stable training.
- **Quantization.** To make XS.2 deployable on low-VRAM devices, we quantize the model's MoE layers to `FP8`, `INT4`, and `NVFP4`, as well as the KV cache to `FP8`. We describe these quantization recipes and the mixed-precision strategy we adopted to preserve quality in Section 5.
- **Evaluation.** We describe our benchmark selection and evaluation harness, and present results on base models, as well as agentic software engineering and terminal task benchmarks in Section 6.

2 Model Factory Approach

We invest heavily into continuously improving the *process* of how we conduct research and model building, even moreso than into any individual model. This has resulted in a process that we internally call our *Model Factory*: a set of components and processes with the aim to maximize research iteration and integration speed, and to minimize the amount of attention researchers need to pay to bookkeeping and infrastructure. This process that allowed us to build Laguna XS.2 from scratch to delivery within five weeks, applying the lessons learned from training M.1.

2.1 Factory Principles

Running research and model building at scale comes with a few challenges: how do we coordinate across many independent research streams; how do we continue to improve research velocity; how do we quickly integrate research results into big production runs?

We share our principles to address these questions in the following sections. Throughout the report, we also show examples of these play out in practice in the respective technical sections.

2.1.1 Experiments as Code

All inputs and configurations to any run (data pipelines, ablation experiments, any training run) are expressed as code committed into a single repository. Each run is tracked with a unique ID, providing a stable handle for identifying, tracking, and reasoning about lineage across experiments and runs. Correspondingly, all experiments, their inputs, and artifacts are registered as persistently stored assets that track dependencies to each other. This gives us the following benefits.

Control Plane. We use Dagster [19] as the central control plane to navigate the directed acyclic graph (DAG) of assets, giving us answers to two questions: *what runs*, and *what does each run depend on*. It also provides a single entrypoint for researchers to manage their experiments: Any job is launched by registering a configuration asset,

and kicked off either via CLI or UI. Keeping track of runs allows us to maintain a full reproducible history of what experiments were run, with what configuration and inputs, and where to find results and outputs.

End-to-end Lineage. Maintaining a DAG of inputs, runs, and artifacts allows us to surface the complete lineage of any model or artifact in both directions: a token in a packed pre-training shard can be traced back through deduplication, filtering, and synthesis to its source document. Every checkpoint, evaluation result, and inference deployment can be traced back to the training run that produced it. This property is a force multiplier to our researchers, giving them full visibility into the combined research results of the whole team, enabling them to branch off any interesting idea they find without needing to manually piece together its lineage.

Agentic Workflows. Our control plane also allows AI agents a single entry point to participate in the research process. Through it, and our code base, they can access the full history of existing research, and corresponding assets and artifacts. Today we use agents to design and run ablation experiments, monitor runs and debug issues, and compile and analyze experiment results. Over time we expect them to take an even greater role in day-to-day research, autonomously pursuing and validating research directions.

2.1.2 Composable, Decoupled Components

We pursue the ideal that every part of the model pipeline should be built once, to be reused and composed many times in different places. This is not an obvious choice as often research and production live in different code bases, driven by different needs for flexibility.

We make the point that maintaining a single code base for research and production (including different research streams) gives us a crucial advantage in making any research progress immediately available to everyone in the team, and widely to users in production. This requires a continued discipline in maintaining a code base that can serve different purposes.

These components cover every step in our research and model pipeline, including data pipelines, training, reinforcement learning, inference, and evaluation. The composability allows us to iterate on components in isolation while making sure the end-to-end flow works, such that a successful innovation can be promoted into production by simply flipping a configuration flag. We highlight some examples of our components in the remaining sections, where they are used across pre-training, post-training, inference, and evaluations.

2.1.3 Reserve Human Attention for Novel Decisions

Another core principle of our Model Factory is to minimize the need for repetitive and mechanical manual work as much as possible, focusing researcher's attention on genuine research questions.

One of the key pieces that enables this is our custom scheduling system that allows researchers to deploy and run training and inference workloads with minimal need to care about the underlying orchestration and resource management. Great care is also spent on automated training failure recovery, and incident escalation all run without engineer involvement on the happy path. On-call is paged only when the automatic recovery system fails to make progress.

Custom Workload Scheduler. Jobs are scheduled by Kubernetes [85] across multiple node pool types. Topologies range from single-node debugging runs to full-cluster jobs on the order of 10^4 accelerators.

The cluster is managed by our custom cluster scheduler. An initial version was based on Volcano [88]. Over time two limitations became binding: First, Volcano makes eviction and reclaim decisions per node. When capacity needs to be freed for a higher-priority gang, entire nodes are drained, which displaces unrelated co-tenant pods and forces them to re-queue from scratch. Second, the scheduler relies on the Kubernetes API server, and therefore on etcd, to hold cluster topology. Under sustained churn at our scale, etcd latency dominated scheduling decisions and pushed end-to-end placement times into tens of minutes.

We replaced it with an in-house batch scheduler purpose-built around three observations:

1. **Per-job eviction and reclaim.** Capacity decisions are made at the granularity of jobs, not nodes. When a higher-priority job needs resources, the scheduler selects victim jobs whose preemption minimizes total displaced work, rather than draining whichever nodes happen to host the required slots. Co-tenant jobs whose pods are not directly required for reclaim continue running undisturbed.
2. **Topology in FoundationDB, populated by observers.** Cluster topology — nodes, pods, GPUs, NVLink and fabric groupings, taints, and capacity — is mirrored out of Kubernetes into FoundationDB [9] through

an observer-based controller that watches the API server and reconciles changes incrementally. The scheduler reads from FoundationDB, which sustains the read and transactional write rates we need without the latency tail we observed when scheduling decisions were issued directly against etcd.

3. **Sticky pod respawn.** When a pod within a running job dies — whether from a node-level fault, a transient NCCL failure, or a preemption that is later reversed — the scheduler prefers to respawn it on the same node it was previously bound to, provided the node is healthy. This dramatically reduces churn: caches stay warm, fabric topology assumptions remain valid, and the surrounding gang does not have to re-pack across the cluster.

The cumulative effect on tail latency was the change that mattered most operationally. With our custom scheduler, placement is consistently sub-minute. This made the rest of the factory's automation viable at our scale: hyperparameter sweeps, CI canaries, and preemption-driven backfill all rely on the scheduler being predictably fast.

3 Pre-training

In this section, we describe the details of our architecture and training setup, the construction of our pre-training data, and present evaluations of our base model on pre-training benchmarks. We outline the training instabilities we encountered with Laguna M.1, the lessons drawn from them, and how those lessons shaped Laguna XS.2. We report evaluation results in Section 6.

3.1 Architecture & Training

Laguna XS.2 and M.1 are both Mixture-of-Experts (MoE) [77] models using a pre-norm Transformer architecture [87, 96] with RMSNorm [103] as the normalization layer. Laguna XS.2 has a total parameter count of 33.4B with 3B parameters (including embeddings) activated per token, while Laguna M.1 has 225.8B total parameters out of which 23.4B (including embeddings) are activated per token. Laguna XS.2 uses token-choice routing [77] with 8 of 256 experts activated per token, plus a shared expert [20] that processes every token. The output of the routed experts is modulated by a coefficient of 2.5 before combining with the output of the shared expert. We use a linear layer followed by a sigmoid activation as the routing function, and apply normalization to the scores after top-k [22]. For load-balancing, we use an auxiliary loss from Qiu et al. [66] across the non-padding tokens of the global batch. For training stability, the first Transformer layer is dense. Both models share the same tokenizer with a vocabulary size of 100,352 tokens, trained with Byte-Pair Encoding [75] on our internal datasets.

Laguna XS.2 uses interleaved Sliding Window Attention (SWA) and Global Attention (GA) [11, 27] with a 3:1 ratio. In both types of layers, we use Grouped Query Attention (GQA) [3] with 8 KV heads, a head dimension of 128, and softplus-based per-head gating [67]. We employ Rotary Positional Encodings (RoPE) [81] to encode positional information. GA layers use 48 Q-heads, $\theta = 500{,}000$, and partial RoPE applied to the first 50% of the head dimension [89]. In layers using SWA, we use an attention window of 512 tokens, 64 Q-heads, and a smaller $\theta = 10{,}000$. The architecture choices above were selected via ablations on a smaller MoE proxy; ablation results are available in Appendix A.2.

From Laguna M.1 to Laguna XS.2. After training Laguna M.1, we revisited many of the architecture and data decisions, including some that had been made implicitly rather than deliberately. On the architecture and training side, Laguna XS.2 inherited many well-ablated choices from Laguna M.1 but introduced four notable changes. First, the more efficient attention mechanism described above replaces global attention on every layer in Laguna M.1. Second, we adopted a Warmup-Stable-Decay (WSD) learning rate schedule [34] instead of cosine. Third, we added routed expert modulation, similar to DeepSeek-V3 [22] and Nemotron 3 [58]. Fourth, we reduced the number of dense layers at the bottom of the model from 3 to 1. We paid particular attention to hyperparameters and numerical stability after the Laguna M.1 training instabilities described in Section 3.1.3, and established a scaling law for learning rate prediction (Section 3.1.1). On the data side, we added an AutoMixer (Section 3.2.3), substantially reduced repetitions across smaller high-quality datasets through synthetic rephrasing (Section 3.2.2), and increased diversity of our web data pipeline by reworking our filters.

Revisiting and adopting these decisions for XS.2 was cheap precisely because of our Model Factory approach (Section 2): the work on data pipelines, training stack, ablation proxy, and evaluation harness transfers seamlessly to any new training run, and the questions that remained were purely model-design questions. Each of the four deltas was selected by a series of targeted ablations on our smaller MoE proxy, and promoted to Laguna XS.2 as a configuration change against the Laguna M.1 baseline.

3.1.1 Training Recipe

The training recipe for Laguna XS.2 reflects the changes outlined above (Section 3.1): a WSD schedule rather than cosine, peak LR set by the scaling law in Section 3.1.1, and numerical conventions informed by the stability investigations in Section 3.1.3. Laguna M.1 and Laguna XS.2 were pre-trained on 6,144 and 2,048 NVIDIA H200 GPUs, respectively. We use the Muon optimizer [39], specifically the Moonlight variant of Liu et al. [49], across all training stages, including SFT and RL — Section 3.1.2 provides details on our distributed Muon implementation.

We use `BF16` mixed precision training throughout all stages with the master weights in `FP32` and most computations in `BF16`. Exceptions include RMSNorm layers and RoPE, which use selective `FP32` operations, and the LM head input-gradient all-reduce described in Section 3.1.3. The pre-training stage uses a context length of 4K tokens and a WSD schedule: linear warmup to a peak learning rate of 5×10^{-4}, a stable phase at peak, and a final decay phase covering the last 30% of training steps that follows a $1-\sqrt{\cdot}$ shape and ends at 5% of the peak (2.5×10^{-5}).

Peak Learning Rate from a WSD Scaling Law. WSD is attractive because a single stable-phase checkpoint can be reused across data-mix iterations via shorter cooldown runs. But, according to our ablations, final-loss calibration is less direct than for one-shot cosine, and naively tuned WSD runs sometimes underperform tuned cosine at matched compute. To pick the peak LR for Laguna XS.2 at production scale, we therefore fit a WSD-specific scaling law by sweeping four MoE sizes (2B–16B total / 0.3B–2.2B active) across six learning rates and five token budgets (30B–480B) at fixed batch $B_0 = 8$M tokens on our stage-1 mix. For each (N, D) we extracted the optimum $\mathrm{lr}^\star(N, D)$ from a parabola fit in $\log_{10}\mathrm{lr}$ space, then used ordinary least squares regression to fit a global power law (full procedure, per-cell plots, and robustness checks in Appendix A.1):

$$\mathrm{lr}^\star(N, D) \;=\; 10^{4.488} \cdot N^{-0.4639} \cdot D^{-0.2661}, \tag{1}$$

with N the number of active parameters and D the total token budget (cooldown included). For a different global batch B we scale the predicted LR by $\sqrt{B/B_0}$, following standard square-root batch–LR scaling for Adam-family optimizers [54]. Laguna XS.2 has $N = 3.0$B active parameters and trains at $B = 24$M tokens; the law predicts $\sim 5.5\times10^{-4}$, and we use 5×10^{-4} to leave a small safety margin.

Long-Context Training. We start the context extension from the end-of-decay checkpoint and split it into two equal-token sub-stages of 100B tokens each. The first sub-stage extends the context to 32K tokens; the second extends the context to 128K tokens. Both sub-stages apply YaRN [62] to global attention layers only, share a global batch size of 24M tokens, and use a cosine schedule that decays from 5% of the pre-training peak (2.5×10^{-5}) to 1% of the pre-training peak (5×10^{-6}). We do not re-warm up the learning rate and resume directly from the end-of-decay pre-training checkpoint; in our experiments this transferred better than reintroducing a warmup at the start of context extension. To reduce variation in the final base checkpoint, we apply an exponential moving average (EMA) over the 10 most recent checkpoints from the end of the 128K stage and use the EMA weights as the final base checkpoint. For the final checkpoint, we extend the context length further to 256K without any training by doubling the RoPE scale in the global attention layers.

3.1.2 Distributed Training

`MODEL FACTORY COMPONENT: TITAN`

Titan is a PyTorch-based [86] training library, originally seeded from TorchTitan [46] and extensively adapted (more than 2,200 changes) for our architecture, sharding strategies, checkpointing, and observability needs. It serves as the single training entry point for the factory, and handles pre-training and post-training needs, including reinforcement learning. It supports composing the full set of standard sharding paradigms — Distributed Data Parallel (DDP), Fully Sharded Data Parallel (FSDP) [104], Tensor Parallel (TP) [78], Expert Parallel (EP) [77], and Pipeline Parallel (PP) [35] — with careful overlap of communication and computation. The training loop, data loader, and metrics path are aggressively optimized for low CPU overhead so that the GPU runs consistently ahead. We have also paid close attention to numerical correctness throughout the codebase; some representative examples include resetting position indices at document boundaries to avoid RoPE-related rounding under document masking in attention, and making gradient-reduction dtypes configurable per collective with `FP32` as the default wherever a reduction is numerically sensitive (see Section 3.1.3).

Device Mesh. A full model/optimizer replica is distributed across FSDP, the TP/EP dimensions, and PP, with replication across DDP ranks. The mesh dimensions are (PP, DDP, FSDP, TP) for non-MoE layers, and (PP, DDP, FSDP, EGP, ETP) for MoE layers. Here, Expert Group Parallel (EGP) shards experts across ranks (analogous to standard EP) and Expert Tensor Parallel (ETP) shards the weights of a single expert along its tensor dimension. Together, EGP and ETP take the place of TP in MoE layers, with EGP $\times$ ETP $=$ TP. For Laguna XS.2 and Laguna M.1 we set ETP $= 1$, since the per-expert intermediate dimension is small enough that further tensor-parallel sharding does not pay off.

Tensor- and Sequence-parallel Non-MoE Components. Each component that lives on the TP dimension in a non-MoE layer (attention, normalizations, embeddings, LM head) can be configured individually as either tensor-parallel (TP) or sequence-parallel (SP). For most components SP shards activations along the sequence dimension in the usual sense [42]; for attention, however, we shard along the batch dimension in order to keep individual sequences intact, so that attention is computed without cross-rank communication. What we call sequence-parallel attention is sometimes referred to as data-parallel attention in the literature. When a component is switched from TP to SP, its weights are no longer sharded along the TP dimension and are instead replicated across TP ranks; each SP rank then produces only a partial weight gradient, computed on its local slice of the input stream, and these partial gradients must be summed across TP ranks before being passed to the FSDP/DDP gradient reductions. We perform these TP weight-gradient all-reduces asynchronously with respect to the compute stream, by carefully adapting PyTorch's FSDP post-backward logic so that the all-reduce is queued on a dedicated communication stream and overlapped with subsequent backward computation.

Distributed Muon. As noted above, we use the Muon optimizer throughout. Compared to many other optimizers, Muon incurs a significant computational overhead that we tackle through the distribution of the compute across ranks. At a high level, Muon needs to aggregate the gradients into a momentum buffer, apply Nesterov momentum, orthogonalize them via Newton–Schulz, and update the parameters. Naively, each rank would need to do this for every full parameter. Our implementation assigns each parameter and gradient to only one of the ranks sharding it, gathers the full gradient on that rank, performs Newton–Schulz, and redistributes the corresponding orthogonalized gradient shards back to all other ranks within the group, which then update their local parameter shards. That effectively removes the compute bottleneck of the Muon optimizer, at the cost of additional communication. Following Amsel et al. [5], we use a schedule of coefficients for the Newton–Schulz iterations rather than reusing the same coefficients across iterations.

Our implementation overlaps batched communication with the Newton–Schulz computations. We also support enabling CUDA graphs for the Newton–Schulz procedure to reduce the CPU overhead of launching many relatively small kernels; this is mainly beneficial for smaller models. Combined, these optimizations reduce the optimizer overhead to less than 1% of the training step time during Laguna M.1 pre-training.

Compute-communication Overlap for MoE. For MoE layers, we use Expert Parallelism, sharding expert weights across 8 H200 GPUs interconnected via NVLink. This requires two all-to-all collectives per layer: a *dispatch* to send tokens to the GPUs owning their expert weights, and a *combine* to retrieve the processed tokens back. Inspired by ParallelKittens [82], we fuse the *dispatch* and *combine* directly into the standard CUTLASS-based grouped GEMM kernels used by PyTorch, minimizing the idle time of Tensor Cores.

For *dispatch*, we dedicate 8 out of 132 Streaming Multiprocessors (SMs) on each H200 GPU to copying expert tokens from its peer GPUs to local High Bandwidth Memory (HBM). The tokens are copied in increasing expert order via vectorized 128-bit loads over NVLink, and a flag in the HBM is set once all tokens for a given expert have arrived. The remaining SMs run the original grouped GEMM algorithm, with the tile scheduler modified to wait until all expert flags for a tile are set, ensuring the tile has been fully copied before processing (see Figure 2).

For *combine*, we modify the grouped GEMM epilogue that normally stores an output tile from the SM registers to the HBM. The modified epilogue instead rearranges the results in a layout amenable to vectorized 128-bit stores through shared memory, then sends each processed token directly to its owner over NVLink. Since epilogues have no data dependencies on one another, all SMs can run in parallel.

This scheme overlaps grouped GEMM compute and NVLink communication at a fine granularity, and is straightforward to integrate into any grouped GEMM kernel that exposes a tile scheduler and an epilogue.

Additionally, we reserve 5 SMs on every GPU for NCCL kernels used in data-parallel communication. Specifically, 4 SMs are used for AllGather/ReduceScatter FSDP collectives, and 1 SM is used to aggregate auxiliary loss information.

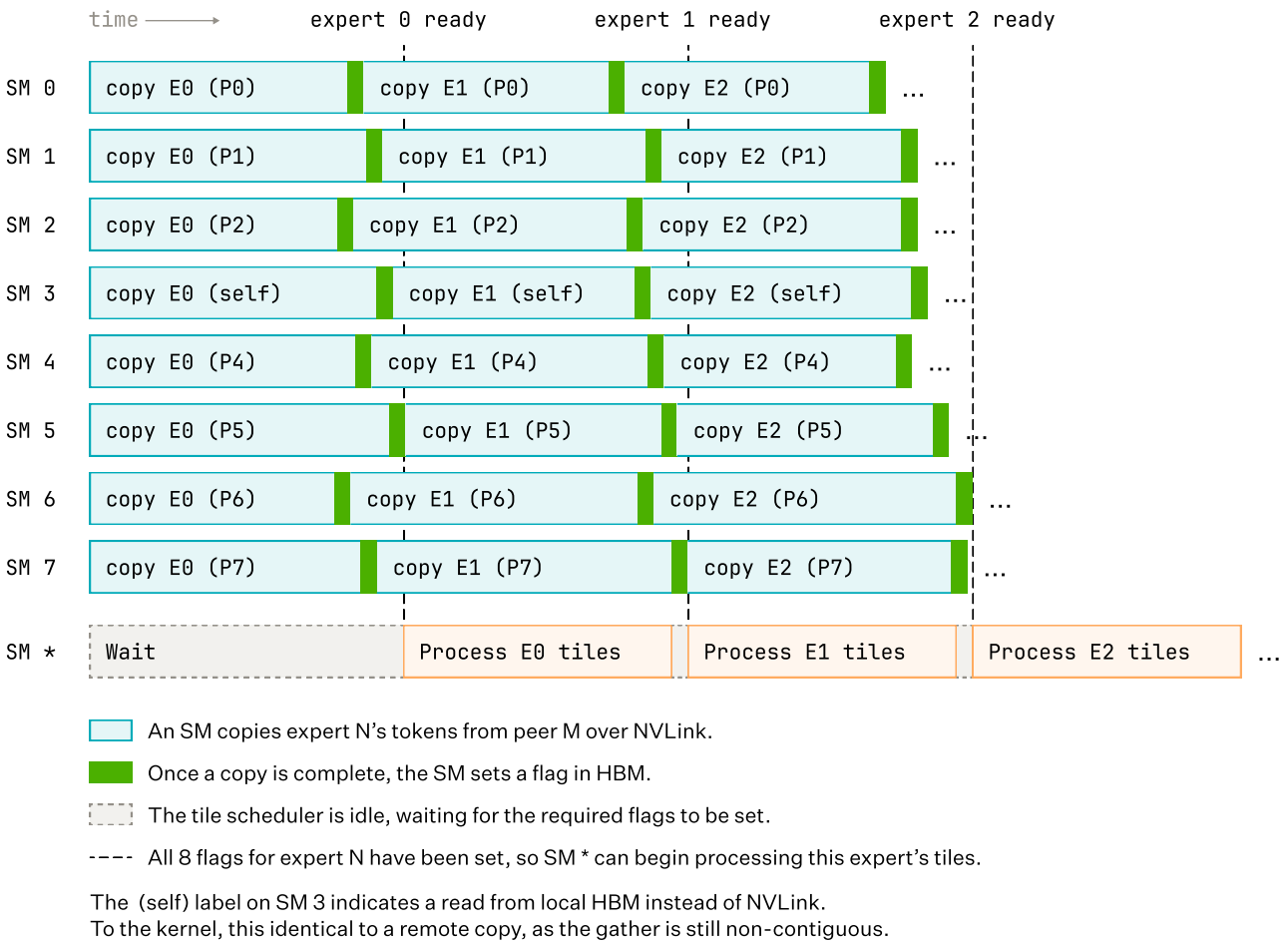


Figure 2: Our strategy for overlapping the *dispatch* with the grouped GEMM compute. Time flows from left to right. SMs numbered from 0 to 7 copy tokens. SM* illustrates all SMs running the grouped GEMM.

Dispatch/combine collectives utilize the scale-up network, while data-parallel communications use the scale-out network; hence, they do not contend for bandwidth.

MODEL FACTORY: RELIABILITY

Hardware faults at our cluster scale range from slowdowns to silent corruption of an entire training run. Pre-flight checks stress-test every node before admission; an in-flight recovery system detects hangs, NCCL failures, and node losses, replaces affected workers, and resumes from the latest checkpoint without human involvement. Critically, we found hash checks important to prevent silent and non-silent failures of large-scale training runs.

Cross-replica Hash Checks. Since updates and compute are replicated across model replicas (i.e., DDP ranks), we periodically hash model weights across replicas to assert that they remain bit-identical. These checks primarily catch silent data corruption (SDC) [24] from defective GPUs — specifically, errors originating in arithmetic logic and pipeline registers, which unlike DRAM and SRAM are not covered by ECC protection. E.g., during one run we identified a single machine producing silently corrupted arithmetic, surfacing as anomalous global gradient norms peaking at $\sim 10^6$. The same machine had participated in earlier runs but caused only subtle symptoms there, owing to lower activation magnitudes. After isolation, training continued

through a short correction period. Beyond their original purpose, the hash checks have also been valuable for surfacing subtle logical bugs in our distributed training implementation (data races, collective communication bugs, replica divergence) that would otherwise have manifested as slow, hard-to-debug drift. For instance, they caught several sources of non-determinism in Muon's Newton–Schulz iterations — whose extra norm reductions and matmuls offer more opportunities for non-determinism than AdamW — which we resolved by pinning the relevant kernels to deterministic algorithms. They also surfaced a subtle drift caused by our checkpointer silently inserting an extra cast in the optimizer state-dict path, leaving DDP rank 0 with `FP64` beta factors, while all other ranks used `FP32`.

Data Loading and Checkpointing. We stream training data rather than reading it from local disk. Streaming is handled by Blender, our internal data service, which mixes multiple sources according to specified mixture weights and curriculum, and exposes a gRPC API for fetching batches with consistent global composition and synchronization across workers. On the training side, a sidecar process prefetches batches from Blender and hands them to the main process through shared memory; the main process then copies them to the GPU via pinned memory. Checkpoints are persisted primarily to S3, with writes distributed across all nodes within a single model replica to make use of the available write bandwidth. Reliable S3 interaction at this scale required substantial engineering effort to handle request throttling and transient errors gracefully on both the read and write paths. On reads, to avoid throttling at restart, only a single replica fetches the checkpoint from S3 and broadcasts the weights to the remaining ranks via RDMA.

3.1.3 Training Stability

Pre-training Laguna M.1 surfaced several stability issues that required iteration on our recipe. The lessons from those investigations were carried into Laguna XS.2 from the start, and Laguna XS.2 pre-training proceeded without encountering further stability issues.

Expert Collapse. To align the effective weight decay between matrix parameters (updated by Muon) and non-matrix parameters (updated by AdamW [52]), we adopt Moonlight-style learning-rate scaling [49]. With this, Muon runs at AdamW-scale LRs, removing the order-of-magnitude mismatch in effective WD. In preliminary experiments, we found that Muon without this rescaling and with standard weight decay coefficients caused expert collapse beginning at around 450B tokens into training and propagating layer-by-layer through the first three MoE layers until the training diverged.

Precision of the LM Head Input-gradient All-reduce. A critical observation for the training stability of Laguna M.1 was the precision of the input-gradient all-reduce for the LM head. By default in our training codebase, the LM head runs under mixed precision [56], with `BF16` gradients on its inputs; under column-wise tensor parallelism those input gradients are also all-reduced across ranks in `BF16`, simply inheriting the gradient dtype. We have no optimization pressure against pre-softmax logit drift — no z-loss [16, 108], and RMSNorm does not subtract the mean — so the logits can freely grow during training. In Laguna M.1 this manifested as a strongly positive logit drift. With growing activations the input-gradient all-reduce becomes the dominant source of numerical error and propagates to the rest of the model, destabilizing training. We therefore enforce the LM head input-gradient all-reduce in `FP32` while keeping the layer tensor-parallel, resolving this numerical instability.

Padding in MoE Routing. A separate observation during Laguna M.1 training was that, despite sequence packing, ∼5% of training tokens were padding, and these were both routed and included in the load-balancing loss. Because we mask the language-modeling loss on padding tokens, the padding embedding is non-learnable; combined with the fact that padding tokens flow through attention without mixing with surrounding tokens, every padding token arrives at the router with the same representation. As a consequence, all padding tokens in a batch are routed to the same expert at the same time, potentially saturating that expert. For Laguna XS.2, we added an option to skip routing and load-balancing of padding tokens; further ablations confirmed this is preferable from a routing-stability perspective.

3.2 Pre-training Data

We trained both Laguna M.1 and Laguna XS.2 from scratch on a pre-training corpus sampled from a pool of ∼27T unique tokens. Our training data spans a broad collection of sources, including large-scale web corpora, code

repositories, curated educational datasets, academic text, and synthetically generated data. In total, both models were trained on more than 30T tokens.

Optimal dataset design for such long training regimes differs from those for shorter training horizons. Recent work on mixture pre-training under data constraints shows that the optimal mixture depends jointly on target data size, mixture ratio, model size, and compute budget [74]. Hence typical data strategies that optimize for high precision by aggressively removing noisy documents to improve average quality (e.g. in [61, 79]) are not optimal for larger token budgets.

We made this observation when training Laguna M.1, where we used a high-precision pipeline with a manually designed mixture, which exposed two bottlenecks: (1) excessive repetition in high-value subsets and (2) suboptimal allocation of data budget across different sources. We confirmed these observations also afterwards in targeted smaller scale experiments. The challenge shifted from maximizing precision under scarcity to controlling repetition and diversity under long-horizon training.

For Laguna XS.2 we addressed these challenges through three main efforts:

- **High-recall web data.** We shifted from a predominantly high-precision web pipeline toward a substantially higher-recall pipeline designed to preserve diversity while maintaining quality.
- **Synthetic data at scale.** We expanded the amount of usable training signal through targeted synthetic rephrasing at scale.
- **AutoMixer.** We replaced static manually designed mixtures with a custom automated data mixture process.

3.2.1 Web Data

Our web data pipeline for both Laguna models ingests large scale web data as raw HTML and we use a custom parser that maximizes data recall and optimizes boilerplate removal and the parsing of technical content. Further data cleaning is performed before language identification. This pipeline leverages GlotLID [40] with pycld2 as a fallback to filter English content. Snapshot-level fuzzy deduplication is preferred because it gives higher performance on knowledge benchmarks. Indeed, we observed that similar web pages across snapshots are statistically more likely to contain relevant facts.

Large-scale web data forms the broadest component of the pre-training corpus, providing coverage across topics, domains, languages, formats, and writing styles that would be infeasible to curate or enumerate manually. For large token budgets our objective is to maximize recall over useful documents, particularly within the high-quality segments, while intentionally retaining enough mid- and lower-quality material to preserve long-tail diversity and sustain scaling performance.

MODEL FACTORY COMPONENT: SPARK FOR DATA PIPELINES

We consolidate every data processing step to run on Spark, and the intermediate results of each data processing step are registered as a Dagster asset. At steady state the system processes on the order of 2×10^{13} tokens per day.

Data Labelling. One of the core challenges in large-scale web curation is defining a sufficiently stable and general criterion for judging document quality. We frame the core question as how to define *useful* versus *non-useful* content in a way that remains consistent across domains, formats, and scales of training.

To construct labels for filtering, we model document quality along two complementary axes: a *noise* axis (N) that captures whether a document is primarily noise or low-information content, while an information axis (I) captures whether the document contains educational, informational, or broader pre-training value. We use an integer range in $[0, 5]$ for both axes. This separation matters because many web documents are imperfect but still useful; a page may contain excessive boilerplate or formatting artifacts while still retaining substantial informational or educational value.

Each annotated document is additionally mapped onto the contribution scale defined in Table 1. Particular attention is given to the ambiguous low-quality region, especially documents in the $[0, 2]$ range of the contribution scale, where weak, noisy, and partially useful content overlap. We densely annotate this region within the $N \times I$ space to better separate truly unusable documents from imperfect but recoverable data. This calibration provides a sub-

stantially more stable target for downstream model-based quality scoring: aggressively filtering clear noise while preserving lower-confidence documents that still contribute useful long-tail training signal.

Table 1: Universal anchors used to build the web ground truth. The $N \times I$ grid is used to make the low-score boundary precise rather than treating all imperfect web pages as discardable.

Score	Anchor	Pre-training interpretation
0	Useless	Almost no training value; contributes no meaningful capability signal.
1	Clearly harmful	Overall content is harmful to training, e.g. severe logic errors, toxic patterns, or misinformation.
2	Borderline-negative	Uncertain but leaning negative; quality is too low or inconsistent to support effective pre-training.
3	Borderline-positive	Uncertain but leaning positive; core value is graspable and benefits outweigh weaknesses.
4	Confirmed beneficial	Clear positive impact despite some noise, such as useful logic, knowledge, or language patterns.
5	Flawless contribution	Minimal noise and exceptional expected value for improving model capabilities.

Ranking rather than Filtering. Rule-based filters are useful for removing obvious artifacts, but exposed a precision/coverage trade-off we observed in training Laguna M.1: broader rules removed more noise at the cost of discarding useful mid-quality documents, while highly precise rules covered only a small fraction of the noisy tail. For Laguna XS.2, the primary rejection path is therefore model-based and deliberately conservative. The pipeline removes documents only when we're confident they are pure noise. For the remaining documents, quality is treated as a ranking signal rather than fully filtered out. Documents are then sorted by a continuous contribution score, allowing us to fill data allocation quota by this ranking.

While ranking according to one general quality dimension would be ideal, we found that a single dimension for classification is too coarse and not accurately predictable enough. Hence we decompose document quality into a set of independently learnable properties, which we then recombine into a composite contribution score.

The ranking pipeline therefore leverages model-based annotations to remove high-confidence noise, assigns continuous quality signals, and samples from corresponding quality buckets. The document tags are produced with Propella [36], an open multi-property document annotation model that exposes separate dimensions rather than a single quality scalar. We keep Propella's native rating ranges, use a PCA-informed subset of its dimensions to de-correlate signals, and then form the composite score that is used both for filtering and sampling. Through empirical filtering and training experiments, we found that pre-training usefulness is best modeled through multiple complementary dimensions, including content quality, educational value, information density, content integrity, content ratio, and commercial bias.

Our composite score completely filters out 25.8% of web samples, while recovering roughly 34% of high-quality documents previously excluded by our previous static rules and lexical-based classifiers. Note that surviving the filtering does not mean the sample is necessarily taken into training. Instead they just become eligible to being sampled according to their contribution score.

Bucket Calibration. After filtering, retained documents are grouped by the composite score for sampling. For sampling, we define bucket boundaries by blind pairwise comparisons around candidate thresholds, so documents across a boundary are distinguishable by expected pre-training value. This produces variable-sized but more separable buckets. Finally, Figure 4 shows the final sampling distribution across quality buckets.

3.2.2 Synthetic Data

We use synthetic data to complement our organic data sources. Synthetic data builds on top of those sources rather than replacing them, regularizing presentation, filling under-represented formats, and exposing structure (plans, rationales, question-answer surfaces, etc.). In Laguna XS.2 it carries $\sim$13% of the mix across all stages, drawn from a pool of $\sim$ 4.4T generated tokens, spanning seed-heavy form rephrasing [53, 80] and expensive pipeline-heavy compositional distillation [1, 31]. We apply these pipelines across high-signal STEM and code domains and extend them into the broader data mix, so synthetic data enters early and runs consistently across all training stages.

We take a modular approach to synthesis: each pipeline composes from a small set of reusable components among six types:

1. A set of inputs $\mathscr{S}$.

Large-scale web data workflow

Each stage has a separate role: clean extraction, conservative rejection, ranked retention and quota-aware sampling.

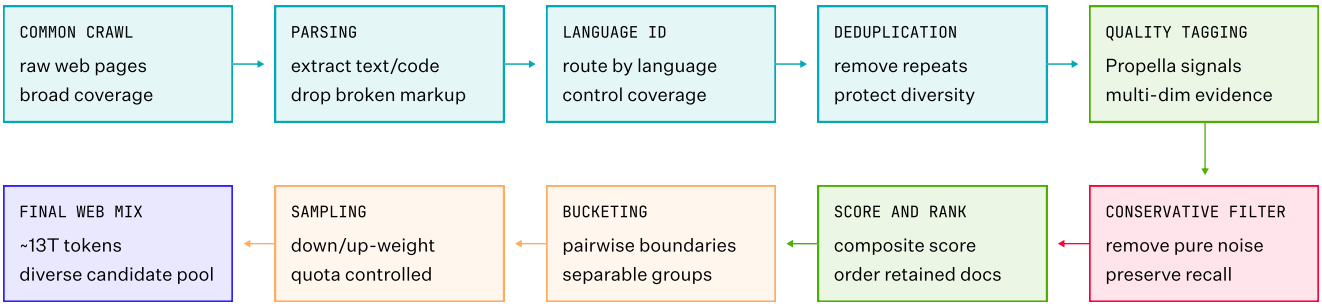


Figure 3: High-level workflow for large-scale web data. Early stages extract, route, and deduplicate Common Crawl documents. Quality tagging supplies multi-dimensional evidence, filtering conservatively removes pure noise, composite scoring ranks retained documents, and bucketing plus sampling turns the ranked pool into a quota-controlled web mixture.

Bucket distribution before and after final sampling

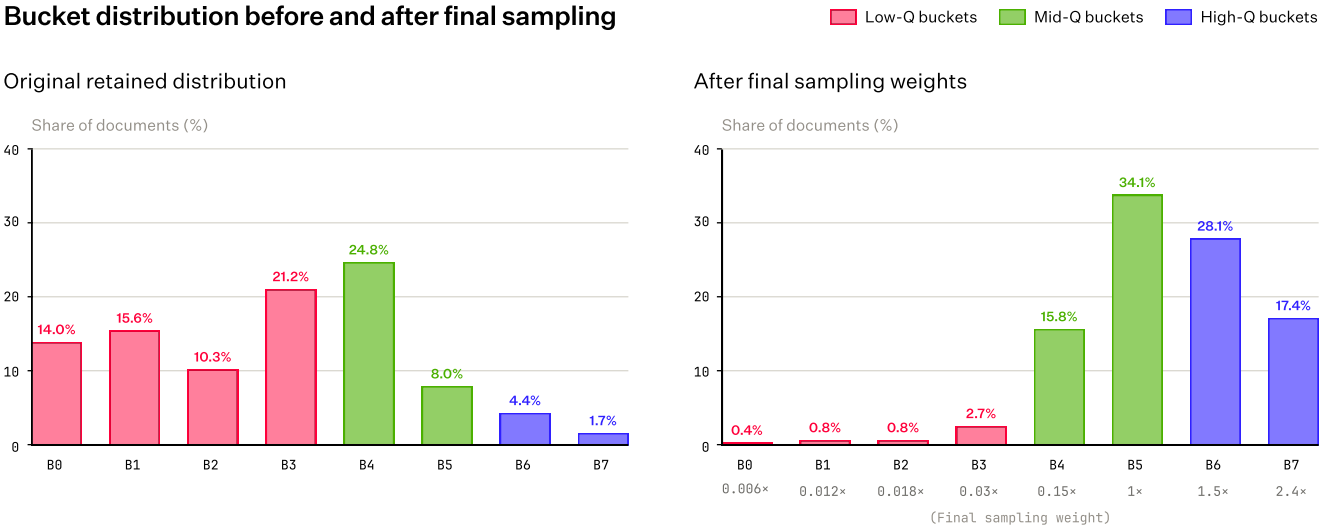


Figure 4: Document-level bucket distribution before and after applying the final sampling weights. Lower-quality buckets are progressively downsampled, while most sampling weight is allocated to higher-quality regions [B4-B7]. This preserves diversity while still prioritizing higher-value documents during training.

2. Metadata attached to each transformation in the pipeline $\mathscr{M}$.
3. A generator (LLM or a chain of LLMs) G we sample from to produce outputs $o \sim G(s, m)$ for some sample s and metadata m.
4. Filtering functions f.
5. Validation function V.
6. Pre- and post-processing functions pre and post.

This allows us to express each pipeline P as a composition of components, like below.

$$P = \text{post} \circ f_n \circ G_n \circ \cdots \circ f_1 \circ G_1 \circ \text{pre}.$$

In practice we draw a sample with metadata (s, m) from the input pool, draw a new sample conditioned on it $o \sim G(\cdot \mid s, m)$, and keep o if f accepts. The resulting synthetic corpus is the set of outputs generated by repeatedly sampling inputs from $\mathscr{S}$.

MODEL FACTORY COMPONENT: HIVE FOR SYNTHETIC DATA

To enable defining and running such pipelines easily, we built *Hive* as a component of the Model Factory (Section 2), a configurable framework that runs every pipeline as

$$P = \text{post}_H \circ H_T \circ \text{pre}_H,$$

where H_T is a dynamic agent-interaction loop over orchestrators, generators, judges [92, 106], and per-step early-exit gates given a common workflow. See Figure 5 for an illustration of the runtime. Iterating on a synthetic data generation pipeline is a configuration change rather than a re-implementation.

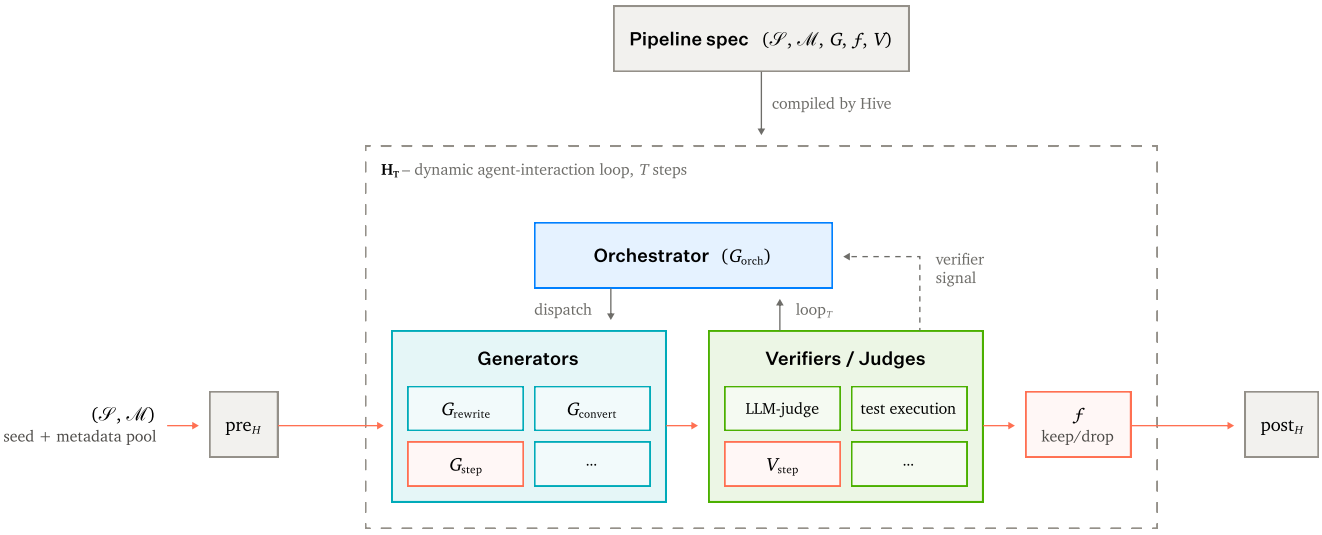


Figure 5: **Hive runtime.** A declarative pipeline spec $(\mathscr{S}, \mathscr{M}, G, f, V)$ is compiled into a single H_T loop. A seed and metadata pool $(\mathscr{S}, \mathscr{M})$ feeds pre_H; pre_H and post_H wrap the loop, while inside it an orchestrator dispatches across a library of generators and verifiers over T rounds and f gates keep or drop outputs.

Generating useful and diverse synthetic data is a challenging task. For our pipelines, we follow two broad principles:

- **Match pipeline complexity to teacher capability.** A task the teacher cannot solve in one shot becomes a source of biased generation, so we either decompose or strengthen the teacher.
- **Help LLMs through metadata.** We hand our generator chain G anything we already know about the output or how to achieve it through supplying metadata, reducing the challenge on the generator alone.

Following these principles, we define a series of synthetic data pipeline strategies that we used for Laguna XS.2. Each strategy balances between sometimes competing axes of *Grounding* (staying faithful to the information content in the seed/input data), and *Entropy* (diversity, injecting randomness and variation into the input data). Through composition of stages, filters, generators, and verifiers, we can define elaborate generation pipelines optimizing for both [44, 100]. We describe some of these strategies in Table 2.

3.2.3 AutoMixer

Pre-training data composition has a substantial impact on downstream model capabilities, with prior work showing that domain mixture proportions can significantly affect model quality [94]. Optimizing data mixtures manually becomes intractable as the number of datasets, capability trade-offs, and training constraints grows. Traditional approaches based on heuristics, qualitative strategies, or manual ablations are expensive, low-dimensional, and slow to iterate on, particularly at modern pre-training scales [99]. Recent work has therefore explored learned, proxy-based, or benchmark-driven approaches for pre-training data optimization and mixture design [10, 15, 45, 50].

Table 2: Pre-training synthetic data pipeline strategies. Token counts shown as order of magnitude of contribution to our pre-training corpus.

Shape	Composition	What it does / examples	Tokens
Form-rewrite	$P = \text{post} \circ f_{\text{quality}} \circ G_{\text{rewrite}} \circ \text{pre}$	Single-call pass conditioned on $\mathscr{M}$ (mode, voice, structure); content inherited from $\mathscr{S}$. Used for: multi-mode rephrasing of web and STEM docs; code rephrased into code with natural-language.	$\sim 10^{12}$
Cross-domain transducer	$P = \text{post} \circ f_{\text{format}} \circ G_{\text{convert}} \circ \text{pre}_{\text{seed}}$	Casts a seed across modalities; pre enforces seed suitability. Used for: math $\leftrightarrow$ code, code-language porting.	$\sim 10^{10}$
Multi-stage cascade	$P = \text{post} \circ f_{\text{keep}} \circ G_n \circ \cdots \circ (f[V \geq \tau] \circ V \circ G_k) \circ \cdots \circ G_1 \circ \text{pre}$	Each G_i transforms the previous output or emits metadata for later stages; the parenthesized V-gate prunes intermediate outputs. Used for: academic and technical content adapted across multiple educational formats, textbook synthesis, diff-conditioned coding task generation, grounded QA generation.	$\sim 10^{11}$
Multi-turn rollout	$P = f_{\text{bounds}} \circ \text{loop}_T \left(\bigcirc_{i=1}^{k} (G_{\text{step}}^{(i)} \circ G_{\text{orch}}^{(i)}) \right) \circ G_{\text{init}} \circ f_{\text{relevance}}$	Closed-loop interaction; orchestrator decides flow and termination. Used for: stacktrace- and raw-code-grounded multi-turn chats, multi-turn math resolution, iterative eval-based doc evolution.	$\sim 10^{10}$

Our framework is inspired by these directions, but adapted to our setting. To solve the dataset mixing problem, we developed *AutoMixer*, a framework for automated data mixture optimization. AutoMixer operates by training a swarm of proxy models, each trained on a different data composition. For each data ablation run, we have trained $\sim$ 60 proxy models, each as a $\sim$0.5B parameter MoE model on $\sim$ 60B tokens sampled from different mixtures across our pre-training corpus.

The explored corpus spans more than 50 heterogeneous dataset groups, including general web corpora, curated educational text, academic papers, raw code, grounded code data, synthetic code data, mathematical web content, and conversational and knowledge-focused datasets. The overall framework is illustrated in Figure 6.

The core idea behind AutoMixer is to learn a surrogate mapping:

$$\mathscr{M} : x \rightarrow y$$

where $x \in \Delta^d$ denotes a data mixture over d dataset groups, and $y \in \mathbb{R}^k$ denotes downstream evaluation metrics across k capability groups. Through learning $\mathscr{M}$, AutoMixer learns how changes in mixture proportions affect downstream capabilities and we can then directly optimize over the learned surrogate.

We start the exploration of the data mixture search space by sampling over a manually designed prior mixture x_0. Candidate mixtures are sampled as:

$$x \sim \text{Dirichlet}(\alpha x_0)$$

subject to additional constraints:

$$\|x - x_0\|_1 < \epsilon$$

This allows efficient exploration of meaningful regions of the mixture space while avoiding unrealistic or degenerate configurations. For each capability group, we train a surrogate regressor:

$$f_j(x) \approx y_j$$

where x is the mixture vector and y_j is the downstream evaluation metric for capability group j.

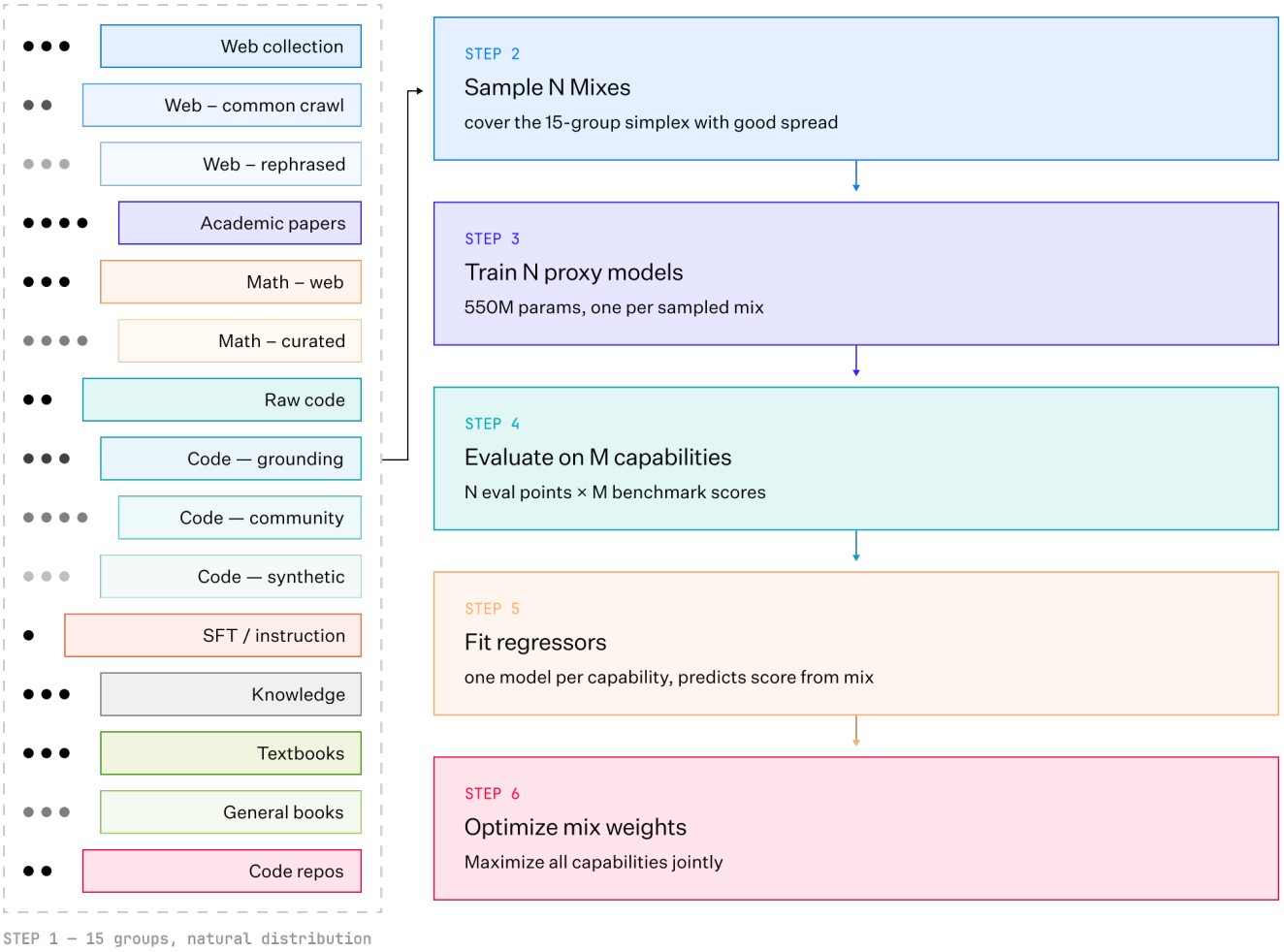


Figure 6: AutoMixer pipeline. A swarm of proxy models is trained under controlled mixture perturbations. Capability evaluations are used to fit surrogate regressors, which are subsequently optimized under regularization and practical mixture constraints.

Capabilities are grouped into coding, mathematical reasoning, STEM knowledge, commonsense reasoning, and general knowledge. We use a small set of pre-training benchmarks as a proxy for downstream capabilities.

The independent regressors allow the framework to recover capability-specific sensitivities and trade-offs. A simplified linearized form can be written as:

$$\hat{y}_j = \beta_j^\top x + b_j$$

although in practice the final implementation uses non-linear surrogate models.

The learned signals recover intuitive relationships between subsets and downstream capabilities. For instance, we observed synthetic and curated code data to strongly improve coding evaluations, while conversational and knowledge-centric corpora improve commonsense reasoning. Importantly, the framework recovers not only expected high-level relationships, but also substantially finer-grained interactions between subsets.

Once the surrogate regressors are trained, mixture optimization is formulated as:

$$\max_x \sum_{j=1}^{k} w_j f_j(x)$$

subject to:

$$\sum_i x_i = 1$$

Table 3: Performance impact of the optimized AutoMixer data mixture. Large gains are observed on optimization targets, particularly coding and mathematical reasoning benchmarks. Improvements also generalize to held-out evaluations not used during optimization, while modest regressions are observed on several commonsense-oriented tasks.

Category	Benchmark	Relative Change
Optimization Targets	HumanEval+	+43%
	MBPP+	+15%
	Crux-I	+54%
	Crux-O	+48%
	MultiPL-E	+27%
	GSM8K	+41%
	MMLU	+5%
Held-out Benchmarks	MATH	+25%
	LiveCodeBench	+39%
	APTBench-4k	+35%
	BigCodeBench	+16%
Commonsense Trade-offs	ARC-C	-6.8%
	WinoGrande	-1.4%
	PIQA	-0.9%
	HellaSwag	-1.3%

$$x_i \geq 0$$

$$\|x - x_0\|_1 < \epsilon$$

To avoid unrealistic shifts toward a small number of dominant sources, we regularize the optimization toward the baseline prior:

$$\mathcal{L}(x) = -\sum_j w_j f_j(x) + \lambda D_{KL}(x \| x_0)$$

where x_0 is the baseline mixture, D_{KL} is the KL-divergence regularization term, and λ controls deviation from the prior. In smaller scale ablations, we validated that the optimized mixture improves strongly over the manually designed prior mixture. Table 3 summarizes the downstream performance changes across both optimization benchmarks focused on coding, and held-out evaluations during a small-scale experiment on a 3B parameter model for 1.5T tokens.

We see that gains generalize to held-out benchmarks that were not directly optimized against, suggesting that the learned surrogate optimization generalizes meaningfully. We also observe modest regressions on several commonsense-oriented benchmarks, which is not surprising given the optimization objectives we chose.

Table 4: Aggregated pre-training data mix used for XS.2.

Data group	Mix weight
Raw code	30.6%
Web	25.2%
Synthetic/code-text	25.4%
Math	9.0%
Knowledge	6.6%
Instruction-like	1.4%
Academic papers	1.1%
Books	0.7%

AutoMixer plays a key part in our data strategy for pre-training. The final optimized mixture used for XS.2 is shown in Table 4. Relative to the data mixture used for Laguna M.1, the allocation defined by AutoMixer used by Laguna XS.2 shifts substantially toward broader web coverage, synthetic/code-text data, and mathematically oriented corpora, while still preserving a strong code-heavy foundation.

4 Post-training

Post-training for Laguna XS.2 and Laguna M.1 picks up after pre-training, including context-length extension (Section 3.1.1), and is divided into three stages:

- **Mid-training:** The largest stage by unique token count, containing roughly 60B tokens in a broad mix of instruction data spanning chat, reasoning, and agentic coding.
- **Supervised fine-tuning (SFT):** Three epochs of roughly 40B tokens each, focused primarily on agentic coding.
- **Reinforcement learning (RL):** Using only verifiable rewards and performed online with CISPO.

We iterate on data, hyperparameters and some architectural choices on top of Laguna M.1, which was available earlier. Due to the fast turnaround time in adapting pre-training changes from M.1 to XS.2, the smaller model was available shortly after, and we proceeded with parallel post-training of both models. Apart from hyperparameter changes and minor dataset fixes, the post-training recipe stayed the same between Laguna M.1 and XS.2 .

MODEL FACTORY: RESEARCH IS SELF-SERVICE

The initial imitation learning stages for Laguna XS.2 were run without any need for human synchronization and performed by a member outside of the core post-training team. This was made possible because the underlying tooling is shared across the whole company, and all training runs are tracked with their full lineage. Training runs can be easily replicated and adapted by any researcher in a self-service manner.

4.1 Special Tokens and Templates

4.1.1 Special Tokens

We introduce new XML-like special tokens to be used to format multi-turn chats in the mid-training stage: `<assistant>`, `</assistant>`, `<think>`, `</think>`, `<tool_call>`, `</tool_call>`. Their embeddings are randomly initialized before pre-training and remain unaltered until the mid-training stage. For the smaller Laguna XS.2 model, random initialization resulted in successful mid-training; however, in Laguna M.1 the mismatch between special and regular token embeddings caused gradient spikes, dead experts, and broader training instabilities.

We address this with initialization and a short warmup phase before mid-training. For initialization, we use subtoken averaging [25, 72] — each new token's embedding is initialized as the mean of its constituent subtokens' embeddings. For example, `<think>` is initialized as the mean of `<th`, `ink`, `>`. This gives the token a semantically grounded starting point consistent with the surrounding embedding space. We then run a 100-step warmup phase where the full network except input embeddings and LM head layers is frozen, allowing the new token representations to adapt before the main post-training phases begin. We found this strategy produces the most stable training dynamics and the lowest tool call formatting error rates from the start of training.

4.1.2 Formatting and Structured Outputs

We adopted chat and structured-output formats that align with open-source conventions: reasoning blocks use `<think>`/`</think>` tags, and tool calls follow an XML-like format compatible with the GLM [102] series.

To control reasoning, the chat template exposes an `enable_thinking` flag. When enabled, the chat template prefixes an opening `<think>` tag just before generation, allowing the model to either close the `<think>` block immediately or generate additional thinking tokens. During the SFT phase, however, we consistently omit empty reasoning blocks and the model learns to emit reasoning tokens whenever the flag is enabled. Conversely, when this flag is disabled, the chat template emits only a closing `</think>` tag. We train with *persistent thinking history*, where all prior reasoning blocks remain visible in the context during each generation step.

When integrating with open-source tool parsers, we discovered a number of failure cases due to the fact that these parsers are tailored to the set of features specific to their corresponding chat templates. Several edge cases, not present in other standard usage, made those failure cases often invisible or at least under-reported. For example, the parsers were sensitive to multi-token streaming deltas: when a single delta contained more than one token, the parser could emit invalid tool calls.

We upstreamed improved reasoning and tool parsers to vLLM,[1] derived from its `deepseek_v3` reasoning parser and `glm45` tool parser, to fix two issues:

- **Reasoning mode detection.** In streaming mode, the `deepseek_v3` reasoning parser decides whether a given turn's reasoning has ended by scanning the entire prompt for a closing `</think>`. Our parser instead scans only the current turn's content, so prior reasoning spans in the conversation history do not short-circuit the check.
- **Deltas spanning block boundaries.** The original parsers assumed each streamed delta contained exactly one token. This simplified logic because the parser did not need to handle transitions across block boundaries (e.g., reasoning → content → tool calls). In practice, deltas can contain multiple tokens — for example, under speculative decoding or when vLLM's single-slot output collector merges accumulated deltas because the producer outruns the consumer. Before our fix, any content trailing a boundary token was silently dropped, producing behavior identical to invalid special-token usage.

4.1.3 TITO and Chat Template Alignment

We use a token-in, token-out (TITO) API design for the actors used in our RL training. This ensures that token IDs are stable across multi-turn interactions, avoiding mismatches in the token ID representation of model outputs. While this approach resolves significant issues in multi-turn, agentic RL training, it introduces the possibility of misalignment between the multi-turn trajectories produced by such a system and the chat template used for production deployments. Subtle differences — such as a missing newline or a trailing space — can lead to significant degradation in model performance when the RL-trained model is deployed.

To prevent divergence while still allowing independent iteration of the RL renderer and the production chat template, we introduce a `render_assistant_messages_raw` flag. When enabled, the chat template inserts the raw rollout tokens verbatim into the assistant message blocks. After every generation step we render the conversation history with this flag enabled and assert that the resulting string exactly matches the decoded prefix stored during rollout. This invariant guarantees that the production template and its logic identically match the behavior seen during RL, eliminating the risk of deployment-time mismatch.

4.2 Mid-training

The mid-training stage uses an effective batch size of 128, a maximum sequence length of 131072 tokens, and a cosine learning-rate schedule with a 50-step warmup, a peak learning rate of 1×10^{-5}, and a final learning rate of 2×10^{-7}. We use Muon throughout. We pack sequences shorter than the maximum sequence length into a single microbatch. We run for a single epoch.

The data corpus used in this stage covers the following categories:

- **General chat**: open-domain question answering, writing, multi-turn dialogue, instruction-following, and long-context interactions;
- **Reasoning**: mathematical, scientific, and logical reasoning trajectories, with explicit chain-of-thought wrapped inside special tokens;
- **Coding and agent**: repository-level code, synthetic code, tool-calling, and long-horizon agentic trajectories targeting use cases for code repositories and terminal environments.

We find it critical to tune the following properties of our mid-training data mix:

- Presence of tool calls, count and variety of the tools provided in each trajectory, and average number of tool calls per trajectory.
- Presence of reasoning, and the ratio of samples with reasoning vs. samples without reasoning.

[1] https://github.com/vllm-project/vllm/pull/35208

- Length of reasoning and difficulty of question, measured by the difficulty labels of our input task sets.
- Total turn count per trajectory, and tokens per assistant and user turn.

In total, the mid-training stage consumes approximately 60B tokens. The mix is deliberately broad: general chat maintains conversational and instruction-following behavior, reasoning data teaches explicit thinking traces and difficult problem solving, and coding-and-agent data teaches repository-level tool use, terminal workflows, and long-horizon software engineering behavior.

In early iterations of our mid-training mix, we observed trajectory-length and tool-availability-dependent failure modes in tool calling and reasoning. For example, on some domain-specific evaluations, the model would either omit reasoning or generate degenerate reasoning spans when tools were not provided. Balancing these features across domains resolved the issue.

In terms of data mixture, during mid-training we spend approximately 40% of the training budget on logic and reasoning, 30% on coding-and-agent data, and 30% on general chat. In contrast, during SFT the mix is heavily weighted toward agentic coding: approximately 85% of tokens are agentic trajectories, with the remaining 15% shared across the other domains.

4.3 Supervised Fine-tuning

The SFT stage shares the same batch size, sequence length, packing, learning-rate schedule and optimizer as mid-training. We run for three epochs of 40B tokens each, with early stopping based on evaluation scores.

The dataset for this stage consists of four main parts:

1. **Agentic coding without reasoning.** Single-turn trajectories generated by an open-source teacher model. This component accounts for approximately 30% of the training mixture by tokens.
2. **Agentic coding with reasoning.** A corpus with the same number of samples and a similar source distribution as the previous component, but augmented with reasoning traces. Due to the reasoning, this component constitutes approximately 45% of the mixture by tokens.
3. **Math corpus.** A relatively small agentic mathematics corpus (∼3% of the mixture by tokens), containing tasks both with and without reasoning traces. The data was collected from several open-source datasets and filtered for verifiability, retaining only problems with numeric (rather than symbolic) answers. We additionally removed overly easy samples based on solve rates from open-weight LLMs.
4. **Non-agentic samples.** Approximately 22% of the mixture by tokens. This component is included to mitigate forgetting of non-agentic capabilities.

The agentic coding samples are collected from multiple sources, including open-source datasets and publicly available code repositories. A large portion of our SFT dataset is generated synthetically in-house, sharing the same pipeline that generated synthetic code environments.

In general, the tasks in the agentic coding corpus focus specifically on software engineering capabilities to operate with code repositories or directly on the text-based terminal.

4.3.1 Synthetic Code Environments

A large portion of our agentic software engineering training tasks is produced by an internal pipeline that turns real-world `git` commits from public repositories into verifiable training tasks. Each commit becomes a task whose problem statement, repository checkout, and hidden test patch are taken directly from the commit, with the commit diff kept as a gold solution. Tasks are filtered with a two-sided correctness check that requires the gold solution to pass the tests and an empty solution to fail them; this removes trivial tests and commits whose tests do not actually exercise the change. We additionally filter on repository popularity and code-quality percentiles, optionally keeping a single task per repository to enforce codebase diversity, retaining on the order of 30–60k tasks from a raw pool of ∼236k commits. The resulting tasks feed both the SFT mix — via teacher-generated trajectories, optionally augmented with multiple synthetic system messages for prompt diversity — and the RL task pool, where the per-repository test suite serves directly as the binary verifier.

4.3.2 Instruction Following

A key challenge in agentic SFT is *instruction following* (IF): ensuring the model respects behavioral constraints specified in the system prompt (e.g., tool restrictions, output format rules, persona requirements). Without explicit IF

supervision, RL fine-tuning tends to exhibit catastrophic forgetting of these behaviors, and over half of agentic tasks receive zero reward because the model violates system-message constraints before any coding work is evaluated.

Data Augmentation Pipeline. We augment agentic tasks with synthetically generated system messages before trajectory generation. For each task drawn from our agentic software engineering training set, we apply an EvolInstruct-style [97] generator to produce a set of behavioral requirements grounded in the task context. These requirements are appended to the original system message, so that the resulting trajectories demonstrate a model that simultaneously solves the coding task and satisfies explicit constraints. We generate multiple augmented system messages per task and use this approach to produce the final training dataset at scale.

We developed a dedicated IF judge that decomposes a system message into a list of independent behavioral requirements and scores each against the model's trajectory. Judge quality was validated against human-annotated examples and calibrated via prompt tuning; inter-model agreement was measured across several frontier models.

To track progress on ablation runs, we created an internal evaluation set obtained with the same pipeline, and applied to SWE-bench Verified. This allowed us to have complementary views of task completion (pass rate) and instruction adherence (follow rate).

Those ablations varied the training data along several axes: the number of requirements, system-message retention (we dropped the original one), the choice of teacher models, the necessity of generated rubrics (we used simple requirement templates in the end), and filtering strictness.

Those ablations allowed us not only to improve on the IF version of SWE-bench Verified, but also on the pass rate directly.

4.3.3 Multi-harness Training

To encourage generalization across diverse agent harnesses, our training data includes trajectories from external frameworks such as OpenHands [90], OpenCode[2], and Mini-SWE-Agent[3]. Specifically, we augment the supervised fine-tuning mix with 1.3B tokens of multi-harness agentic trajectories spanning a broad range of software engineering tasks. We intentionally preserve native harness behaviors during data collection — including custom subagent spawning, context compaction, planning scaffolds, and reminder systems — so the model sees a broad distribution of interaction patterns. These frameworks vary substantially in trajectory length, tool use, subagent orchestration, and harness-specific quirks, all of which improved generalization in internal evaluations.

4.4 Agentic RL

Following supervised post-training, we run a large-scale agentic reinforcement-learning phase in which the policy itself drives the deployed agent harness, producing multi-turn trajectories on real coding repositories, terminal sandboxes, and tool-augmented math problems.

The harness driving RL rollouts is the same one we ship in production: the policy interacts through the same tool API, chat template, and orchestration layer that customers exercise, so any change to the deployed harness is also a change the policy is trained against. Each rollout starts by spinning up a container drawn from the code-execution platform described in Section 2, which hosts both the agent's working environment for SWE, shell or tool-integrated reasoning tasks and the per-task verifier that produces the reward; the container is torn down once the verifier returns, with several thousand live across the platform at any moment.

MODEL FACTORY COMPONENT: CODE EXECUTION ENVIRONMENTS

We maintain a containerized code-execution platform as a primitive to be used in synthetic data generation, evaluations, and providing execution-based rewards in reinforcement learning.
Containers are produced from OCI builds against GitHub sources, with agent-driven generation increasingly replacing handwritten Dockerfiles for repositories that do not build out of the box. This platform spans roughly one million repositories at the moment.

[2] https://github.com/anomalyco/opencode
[3] https://github.com/SWE-agent/mini-swe-agent

4.4.1 Algorithm

We train with a token-level REINFORCE surrogate combining importance-sampling-ratio clipping (CISPO [14]) with a length-weighted leave-one-out [2] group-relative advantage estimator. Moonlight scaling [49] was deactivated during RL for Laguna M.1, but used for Laguna XS.2. We selected this recipe after ablating against other group-based policy-gradient algorithms (GRPO [76], GSPO [105]), which showed it offered the best combination of final evaluation quality and training stability.

For each prompt we sample a group of G trajectories $\{\tau_i\}_{i=1}^{G}$. Letting r_i denote the scalar terminal reward of τ_i and w_i its number of rewarded (assistant-generated) tokens, the length-weighted leave-one-out baseline and advantage are

$$b_i = \frac{\sum_{j \neq i} w_j r_j}{\sum_{j \neq i} w_j}, \qquad A_i = r_i - b_i. \tag{2}$$

With π_θ the current policy and $\pi_{\theta_{\text{old}}}$ the rollout policy at sampling time, the per-token surrogate loss applied uniformly across the assistant tokens of τ_i is

$$\mathcal{L}(\theta) = -\mathbb{E}\Big[\, \text{clip}\big(\rho_t,\, 1 - c_{\text{low}},\, 1 + c_{\text{high}}\big)\, A_i \, \log \pi_\theta(y_t \mid x, y_{<t}) \,\Big], \quad \rho_t = \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\theta_{\text{old}}}(y_t \mid x, y_{<t})}, \tag{3}$$

with asymmetric clipping bounds $(c_{\text{low}}, c_{\text{high}}) = (1, 4)$, yielding an effective importance-ratio clip of $[0, 5]$ that engages only on heavily off-policy tokens.

4.4.2 Reward Design

Rewards are produced by a deterministic chain of checkers applied to every terminated rollout in the following order, with the first failing check determining the reward.

Parsing error (-0.1): the rollout contains a malformed tool call or chat-template violation. Applying this penalty only to the last turn suppresses formatting drift without overwhelming the verifier signal.

Minimum-steps penalty (-0.1): the agent exits before issuing at least n_{min} tool calls. n_{min} was specifically tuned to suppress degenerate short-circuited trajectories that "give up" before verifying the solution.

Timeout, max steps reached (0.0): a wall-clock timeout or reaching the maximum number of steps zeros the reward without an additional penalty.

Binary task verifier (1.0 / 0.0): the task-specific verifier produces the only positive reward. For SWE-style tasks this is the repository's own unit-test suite run against the agent's patch; for terminal tasks it is the task's bash-side assertion harness; for tool-integrated math problems it is exact numeric-answer match against the gold solution.

Tool-error step penalty (-0.05, per-token): for every assistant step whose tool invocation led to the error in tool execution, we apply a small negative reward to exactly the tokens that constituted that step, rather than to the trajectory as a whole. This per-token shaping focuses credit assignment on the failing step and discourages long chains of malformed tool calls before falling back on the terminal verifier.

Shaping is therefore restricted to small negative penalties on parsing, degeneracy, and per-step tool errors; long-horizon credit assignment is carried entirely by the binary verifier at the end of the trajectory, propagated to every token of the trajectory through the advantage A_i.

4.4.3 Task Mix

The RL prompt distribution spans three families that share a single tool-use API.

Code repository tasks drawn from a large internal SWE collection as well as a mix of public datasets; rewards come from each task's own per-repository unit tests, executed in an ephemeral sandbox after applying the agent's patch.

Shell usage tasks executed inside ephemeral container sandboxes covering shell scripting, system administration, and long-horizon command-line workflows; rewards come from task-specific shell assertions evaluated against the final container state.

Tool-integrated math reasoning: math problems in which the agent must drive a code-execution tool to compute and verify its final answer, scored by exact numeric-answer match against the gold solution. These trajectories anchor the model's reasoning capability under the same tool-use API as the coding tasks, and prevent regression of the reasoning behavior established in mid-training.

We partition tasks into pass-rate buckets using prior attempts from the initial checkpoint. We drop tasks that were always solved or never solved, and sample the remaining tasks proportional to their $(1 - \text{pass\_rate})$. This allows us to get a distribution skewed to harder but still solvable tasks that preserved a training signal over the multiple epochs.

`MODEL FACTORY COMPONENT: ATLAS`

Atlas is our multi-accelerator inference library, built on top of vLLM [43] and consuming Titan's model definitions directly for bit-accurate reference with the trainer. Atlas serves every downstream consumer: automated evaluations, internal inference, customer-facing production traffic, but also crucially RL rollout generation. Any improvements in Atlas immediately benefit all downstream consumers. To manage traffic across Atlas deployments we run an Envoy-based [26] proxy paired with a custom orchestrator that can stand up new model deployments in seconds. The orchestrator owns session management and routing across the fleet, optimizing KV-cache utilization during agentic reinforcement learning.

4.4.4 Trainer-to-inference Weight Sync

The trainer and the inference fleet live on physically separate GPU pools but share a high-bandwidth interconnect. Updated weights are sent from trainer GPUs to every inference-replica GPU via our GPU↔GPU weight-transfer system [63]. The weight sync uses NCCL point-to-point primitives over GPUDirect RDMA with no intermediate offload to host memory or remote object storage. The system handles the asymmetric $n \to m$ fan-out between the sharded trainer and the inference fleet of full-weight replicas. In our experiments we kept m between $2n$ and $3n$ depending on model size, which allowed us to strike the right balance between speed and off-policy level. Transfers run asynchronously, so training continues while weights are in flight.

Broadcasts are issued every 2 optimizer steps. Two synchronization primitives keep all of the above safe under online RL, irrespective of which `FP8` mode the inference side is in. Weight broadcasts trigger an inference-side KV-cache reset, so the cache can never mix tokens encoded under different weight versions. Weight updates are configured to block any in-flight rollout step on the update, so no single rollout step straddles a refresh; from each rollout's perspective the policy is piecewise-constant in time, which is what the importance ratio ρ_t in the loss above implicitly assumes. We also cap the staleness of trajectories used for training at 10 optimizer steps, which we empirically found to be a good borderline between training speed and stability in our earlier experiments. However, with our release ratio between training and inference GPUs we never actually hit this limit, which effectively kept all trajectories non-wasteful.

4.4.5 FP8 Inference for RL Rollouts

The trainer keeps master weights and gradient computation in `BF16`; only the inference path uses `FP8`, and only on the KV cache for the release runs.

We store the attention KV cache in `FP8` across the agent's full 131,072-token context window. Agentic trajectories pack many short assistant turns interleaved with long tool observations; at this context length the KV cache dominates inference-replica memory, and storing it in `FP8` roughly doubles the number of concurrent trajectories a single replica can carry. The release-candidate runs for both Laguna XS.2 and Laguna M.1 used this `FP8` KV cache only, inference-side weights remained in `BF16` across the entire RL phase.

During pre-release ablations, we also quantized the inference-side weights to `FP8` with an in-flight block-wise scheme: weights were re-quantized on the inference replica during each weight sync, with per-block scales computed on the fly without any calibration data. We observed no measurable hit to stability as gradient norms and reward curves closely tracked the `BF16`-weight baseline, and the importance ratio ρ_t remained well within the clip bounds. The only observable effect was a larger training-inference mismatch measured by per-token KL divergence and absolute log-probability differences. However, as it was not obvious how this mismatch would play out over the many-day horizon of an RL run, we decided to keep `BF16` weights for the release runs.

5 Quantization

Since one of our goals in building LAGUNA XS.2 was to enable deployment on low-VRAM devices, the model's memory footprint was a key consideration. Quantization was therefore a crucial step that we needed to get right to broaden the model's compatibility with various hardware configurations.

LAGUNA XS.2 was pre-trained and fine-tuned using the `BF16` data type. We quantized the model's MoE layers to `FP8`, `INT4`, and `NVFP4`. We also quantized the KV cache to `FP8`. We primarily used the LLM Compressor framework [70] for quantization due to its tight integration with vLLM [43].

FP8 KV Cache. The KV cache was quantized to `FP8` using $a_{\max}$-based scaling with a calibration dataset of 128 long-context agentic trajectories. We use per-tensor scaling for KV cache to maximize compatibility with various `FP8` attention implementations.

FP8. We found that using SpinQuant [51] R1 rotation improved the quality of `FP8` quantization without introducing any runtime overhead, and therefore adopted it for our `FP8` quantization scheme.

For `FP8` quantization (`W8A8`), we used an $a_{\max}$-based quantization scheme with dynamic activation scaling that does not require calibration. Assigning a single scale factor to each 128×128 weight tile and 1×128 activation group resulted in no measurable quality degradation relative to the `BF16` baseline.

INT4. Similarly to our `FP8` quantization, we used SpinQuant R1 rotation as a pre-processing step before our `INT4` quantization.

For `INT4` weight quantization (`W4A16`), we applied AWQ [47] using a calibration set of 128 long-context agentic trajectories. This approach initially introduced a non-negligible quality drop. Analysis of the activation distributions in the residual stream of the model showed an accumulation of outlier activations beginning around layer 30 of the 40-layer network (see Figure 7).

To keep the quality degradation to a minimum, we adopted a mixed-precision quantization strategy: the first 30 layers were quantized to `INT4`, while the final 10 layers were quantized to `INT8` with 1×128 group-based weight scaling.

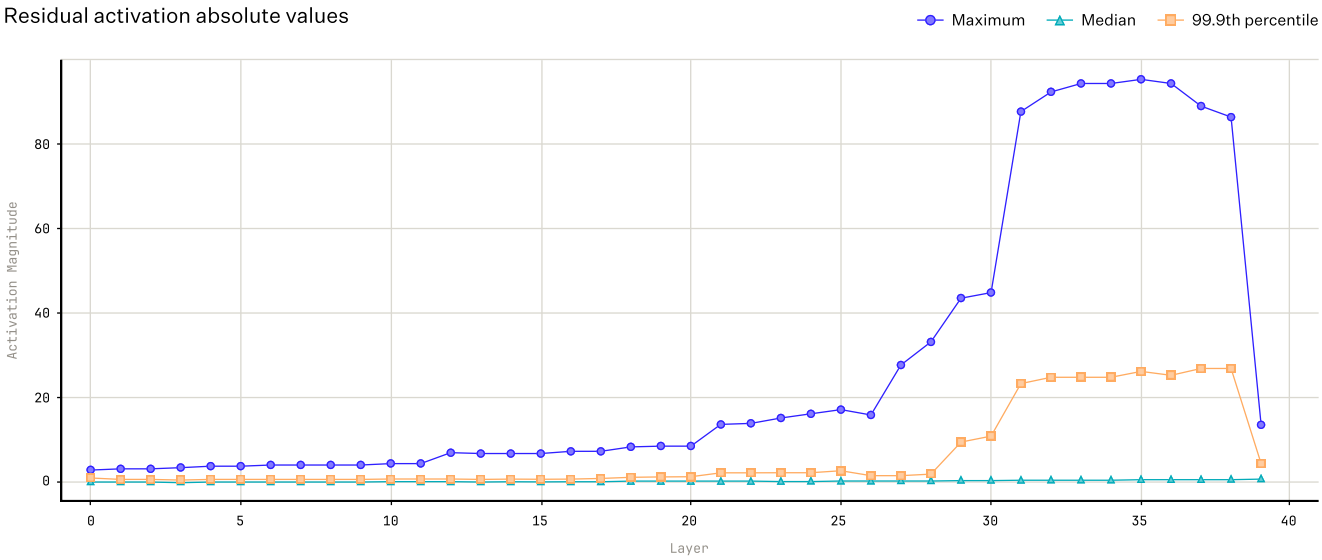


Figure 7: Accumulation of outlier activations across layers. The figure compares absolute maximum of the activations in the residual stream to the median of their absolute values.

NVFP4 Quantization. Direct post-training `NVFP4` quantization caused a non-negligible quality drop even in a mixed-precision `FP8/NVFP4` strategy mimicking the one we used for `INT4`. To mitigate this, we used quantization-aware distillation [95] (QAD) to recover quality. Unlike the `FP8` and `INT4` schemes above, we did not use SpinQuant

for NVFP4 as QAD was sufficient without additional pre-processing. We used the standard microscaling format for NVFP4 following Alvarez et al. [4]: 1×16 groups are used for local scaling factors in FP8 along with an FP32 global per-tensor scaling factor.

Starting from the higher-precision checkpoint, we froze all other parameters and optimized the MLP BF16 weights by quantizing/dequantizing to NVFP4 in the forward pass. Following QAD-based accuracy recovery for quantized LLMs [95], the quantized student was trained to match the higher-precision teacher on a fixed distillation dataset. To make the full-vocabulary KL objective practical, we adopted the hidden-state caching strategy used by DeepSeek-V4 [21]: we cache final hidden states of the teacher and reconstruct teacher and student logits through the frozen output head.

Quality Evaluation. We observed only marginal quality degradation for our final quantization schemes. However, some of our intermediate quantization attempts resulted in a significant quality degradation, as described in the paragraphs above. Interestingly, the quality drop of the unsuccessful attempts was mostly visible on the agentic coding benchmarks, while more traditional single-turn benchmarks were affected to a smaller extent. This highlights the importance of using a diverse set of benchmarks to properly validate that the quantization process does not introduce any unintended side effects, especially for the tasks that require strict formatting such as agentic coding.

6 Evaluation

We present evaluation results for Laguna XS.2 and Laguna M.1 across a variety of benchmarks, including pre-training evaluations for Laguna XS.2 and agentic evaluations for both Laguna XS.2 and Laguna M.1.

MODEL FACTORY COMPONENT: EVALUATIONS

We schedule configurable evaluations automatically for every training experiment, run against in-flight checkpoints which execute on Atlas (Section 4.4.3).
A key challenge is *execution-grounded* evaluation: benchmarks where each task requires a real software environment that compiles, runs, and tests the model's output across heterogeneous environments. Here, we use the same code execution component (Section 4.4) as is used for agentic RL and synthetic data generation.

6.1 Base Model Evaluations

6.1.1 Evaluation Setup

Benchmarks. We evaluate Laguna XS.2 across general, mathematical, and coding capabilities. For general capabilities, we evaluate on BBH [83], MMLU-STEM [32], and MMLU-Pro [91]. For mathematical reasoning, we use GSM8K [18] and MATH [33]. For coding capabilities, we employ LiveCodeBench v6 [37] (questions from August 2024 to May 2025), MultiPL-E [13], BigCodeBench [107], and CRUXEval [30], reported separately with chain-of-thought on input prediction (CRUXEval-I (CoT)) and output prediction (CRUXEval-O (CoT)).

Baselines. We benchmark Laguna XS.2 against recent open-weight MoE base models of similar size: Qwen3.5-35B-A3B-Base [68], Gemma-4-26B-A4B [29], and Nemotron-3-Nano-30B-A3B-Base-BF16 [58]. We also include MiMo-V2-Flash-Base [93] as a reference point, despite its larger parameter footprint.

Evaluation Configurations. We use likelihood-based evaluation for MMLU-STEM and generation-based evaluation for BBH, MMLU-Pro, GSM8K, MATH, LiveCodeBench v6, MultiPL-E, BigCodeBench, CRUXEval-I (CoT), and CRUXEval-O (CoT). Per-benchmark few-shot counts and reported metrics are summarized in Table 5. Laguna XS.2 scores in Table 5 are produced by our internal evaluation framework. For the open-weight baselines, we report published scores when available, which we also reproduce in our internal framework for consistency; specifically, we draw published numbers from the MiMo [93] and Nemotron-3 Nano [58] technical reports. All remaining scores are computed with our internal evaluation framework under the configurations described above.

6.1.2 Evaluation Results

Table 5 presents a comparison of Laguna XS.2 against the four baselines introduced in Section 6.1.1.

Table 5: Comparison of Laguna XS.2 against open-weight base models. EM stands for Exact Match. **Bold** marks the best score among models of similar size (Laguna XS.2, Qwen3.5-35B-A3B, Nemotron-3-Nano-30B-A3B-BF16, Gemma-4-26B-A4B). †Score taken from the corresponding model's technical report (MiMo [93] or Nemotron 3 Nano [58]); all unmarked scores are computed with our internal evaluation framework. *For Gemma-4-26B-A4B on MultiPL-E, we use a slightly modified prompt format (a two-space indent appended as a trailing suffix) to work around a prompt-boundary artifact that otherwise significantly degrades performance on `cpp`, `sh`, and `ts`.

	Metric	Shots	Laguna XS.2	Qwen3.5	Nemotron-3 Nano	Gemma-4	MiMo-V2 Flash
# Total Params			33.4B	35B	31.6B	25.2B	309B
# Active Params			3B	2.6B	3.6B	3.8B	15.1B
General							
BBH	EM	3-shot	80.9	**86.3**	79.1	76.4	88.5†
MMLU-STEM	acc	5-shot	78.1	**80.2**	75.3	70.4	89.1
MMLU-Pro	EM	5-shot	53.0	62.5	**65.1**†	47.5	73.2†
Mathematics							
GSM8K	EM	8-shot	84.2	91.5	**92.3**†	75.4	92.3†
MATH	EM	4-shot	58.8	60.0	**82.9**†	38.9	71.0†
Coding							
LiveCodeBench v6	pass@1	1-shot	**29.3**	24.4	22.5	18.1	30.8†
MultiPL-E	pass@1	0-shot	**58.4**	57.9	56.1	45.1*	61.1
BigCodeBench	pass@1	0-shot	**53.8**	52.0	50.2	44.6	70.1†
CRUXEval-I (CoT)	pass@1	1-shot	61.9	**66.0**	63.2	54.7	67.5†
CRUXEval-O (CoT)	pass@1	1-shot	71.7	**71.9**	63.4	60.0	79.1†

Our base model results in Table 5 place Laguna XS.2 among comparable open-weight base models. It demonstrates strong performance on coding benchmarks, leading recent open-weight MoE base models of comparable active parameter count on the majority of coding tasks. Moreover, Laguna XS.2 approaches MiMo-V2-Flash-Base on LiveCodeBench v6 and MultiPL-E despite being a fraction of the size.

While pre-training benchmarks are the most informative signal available at this stage of training, they remain imperfect proxies for downstream task performance and real-world utility. We treat agentic evaluations of the final post-trained model as the primary signal we optimize against.

6.2 Agentic Evaluations

We report performance on the following four benchmarks to best represent the intended use cases of Laguna M.1 and Laguna XS.2 as agentic coding models: SWE-bench Verified [38, 60], SWE-bench Multilingual [98], SWE-Bench Pro [23], and Terminal-Bench 2.0 [55, 84].

6.2.1 Evaluation Setup

Baselines. Due to the complexity of agentic evaluations, performance can vary drastically depending on configuration, such as harness choice, sandbox resource allocations and sampling parameters. This makes publicly reported results for external models difficult to reproduce with certainty.

To avoid potential bias introduced by our evaluation setup, we report external model baseline scores only from official release blog posts by the authors or equivalent, with the exception of Gemma 4 31B dense, where the highest published scores were reported by the Qwen team [69], and Claude Haiku 4.5, where the highest published (verified) scores for SWE-Bench Pro and Terminal-Bench 2.0 are from their respective official leaderboards.

For Laguna XS.2, we report scores from Devstral Small 2 [57], Gemma 4 31B dense [29], Qwen3.5 [68], Qwen3.6 [69], Claude Haiku 4.5 [6], and GPT-5.4 Nano [59].

For Laguna M.1, we report scores from Devstral 2 [57], GLM-4.7 [28], DeepSeek-V4 Flash [21], Qwen3.5 [68], and Claude Sonnet 4.6 [7].

Settings. All agentic evaluations are run with *pool*,[4] our terminal coding agent harness, with a maximum of 500 steps per task. We sample Laguna XS.2 and Laguna M.1 with a temperature of 1.0 and top_k = 20, with thinking mode enabled and a context length of 256K tokens. All tasks are run in their own sandbox using our internal sandboxing service. Each sandbox is allocated 2 CPUs and 8 GB of RAM per task with the exception of Terminal-Bench 2.0, for which 32 CPUs and 48 GB of RAM are allocated per sandbox due to more resource-intensive tasks.

Agentic evaluations contain significant noise due to infrastructure limitations inherent to the benchmarks, which can result in large swings in single-trial scores [8]. To address this, we run each benchmark four times and report the mean pass@1 across the four runs.

Benchmark Improvements. Some base task images and verifiers are also patched to fix these infrastructure reliability issues inherent in task setup, such as rate limits on third-party dependencies in external registries used by the verifier. A detailed changelog is provided in Appendix A.3.

6.2.2 Benchmark Results

Table 6: Agentic coding results for Laguna XS.2 against open-weight baselines of comparable size (Devstral Small 2, Gemma 4, Qwen3.5-35B-A3B, Qwen3.6-35B-A3B), with Claude Haiku 4.5 and GPT-5.4 Nano shown as frontier proprietary references of comparable model size. "–" indicates a score not reported by the model provider.

	Laguna XS.2	Devstral Small 2	Gemma 4	Qwen3.5	Qwen3.6	Claude Haiku 4.5	GPT-5.4 Nano
# Total Params	33.4B	24B	31B	35B	35B	–	–
# Active Params	3B	24B	31B	3B	3B	–	–
SWE-bench Verified	69.9	68.0	52.0	69.2	**73.4**	73.3	–
SWE-bench Multilingual	57.7	55.7	51.7	60.3	**67.2**	–	–
SWE-Bench Pro	46.3	–	35.7	44.6	**49.5**	39.5	52.4
Terminal-Bench 2.0	35.7	22.5	42.9	40.5	**51.5**	29.8	46.3

Table 7: Agentic coding results for Laguna M.1 against open-weight MoE and dense baselines (Devstral 2, GLM-4.7, DeepSeek-V4-Flash, Qwen3.5-397B-A17B), with Claude Sonnet 4.6 shown as a frontier proprietary reference of comparable model size. "–" indicates a score not reported by the model provider.

	Laguna M.1	Devstral 2	GLM-4.7	DeepSeek-V4 Flash	Qwen3.5	Claude Sonnet 4.6
# Total Params	225B	123B	355B	284B	397B	–
# Active Params	23B	123B	32B	13B	17B	–
SWE-bench Verified	74.6	72.2	73.8	**79.0**	76.2	79.6
SWE-bench Multilingual	63.1	61.3	66.7	**73.3**	69.3	–
SWE-Bench Pro	49.2	–	–	**52.6**	50.9	–
Terminal-Bench 2.0	45.8	32.6	41.0	**56.9**	52.5	59.1

6.2.3 Reward Hacking

At the time of writing, we have discovered that the official versions for all four agentic benchmarks reported are vulnerable to benchmark hacking to some degree [64], either via leaked git history in the task images or via web search for reference solutions. Evidence of benchmark hacking has also been found on public leaderboard results[5].

[4] https://github.com/poolsideai/pool
[5] https://www.tbench.ai/news/leaderboard-integrity-update

To address this, we patched the base task images to remove the git history leak for all affected benchmarks, and opened issues and pull requests to contribute back these fixes to the official benchmark repositories. All scores in this report are run with the patched images.

Additionally, we have developed a reward hack judge to detect these specific cheating strategies, which we ran post-hoc on all Laguna M.1 and Laguna XS.2 evaluation runs for this report. After joint judge review and manual review, we did not find significant reward hacking in the runs.

7 Conclusion

We presented Laguna M.1 and Laguna XS.2, two Mixture-of-Experts foundation models for long-horizon, agentic coding. Both models are competitive with state-of-the-art open models in their respective weight classes on agentic software engineering and terminal benchmarks (Section 6).

We believe in the merits of open research and have shared deep technical details about the research that went into our models. In addition to this, we have also released the weights of Laguna XS.2 to the public under the Apache 2.0 license. We hope this contributes to the growth of a vibrant and diverse research community in AI.

Moreover, we described the principles of our *Model Factory* that accelerates our research and model building progress. While the architectural, data, and post-training choices that shaped Laguna M.1 and XS.2 are interesting in their own right, we also want to draw attention to the process behind them.

We believe this is one of the most consequential lever for frontier model development going forward. As models, data pipelines, and post-training recipes grow in complexity, the gap between teams that treat model building as a craft and those that treat it as an industrial process will continue to widen. Investing in the factory is how we expect to keep iteration speed ahead of model complexity.

Laguna M.1 and Laguna XS.2 are early outputs of this approach. We are continuing to build the factory, and look forward to sharing future models and our research.

8 Contribution

Authors are listed alphabetically by their last name. Names with * have since left Poolside.

Research & Engineering. Julien Abadji, Marah Abdin, Connor Adams, Eric Alcaide, Mustafa Altun, Michele Artoni, Junze Bao, Uday Barar, Vassilis Bekiaris, Arkadii Bessonov, Benjamin Bütikofer, Jonathan Chang, Yen-Chun Chen, Dmitry Chernenkov, Yang Chi, Filippos Christianos, Fenia Christopoulou, Razvan-Andrei Ciocoiu, Tzachi Cohen, Yohann Coppel, Dmitrii Emelianenko, Brandon Fergerson, Brian Fitzgerald, Matthias Gallé, Alex Golonzovskyi, George Grigorev, Yiyang Hao*, Christian Hensel, Jan Huenermann*, Ye Ji, Sarthak Joshi*, Eiso Kant, Kabir Khandpur, Seonghyeon Kim, Vladimir Kirichenko, Umut Kocasarac, Ilya Kochik, Ivan Komarov, Chaerin Kong, Anurag Koul*, François-Joseph Lacroix, Sergei Laktionov, Waren Long, Quentin Malartic*, Vadim Markovtsev, Afonso Marques, Robert McHardy, Carlos Mocholí, Dmitry Monakhov, Adam Morris, Martin Muller, Christian Mürtz, Robin Nabel, Thien Nguyen*, Rok Novosel, Szymon Ozog, Aalhad Patankar, Aleksei Petrov, Alexandre Piché*, Arthur Pignet, Teodor Poncu, Phil Potter, Alexander Rakowski, Pierre-Yves Ritschard, Jay Roberts, Joe Rowell, Piotr Sarna, Pierre-André Savalle*, Uladzislau Sazanovich, Nikita Shapovalov, Arsenii Shevchenko, Mikhail Shilkov, Andrei Sokol, Mohamed Soliman, Jack Stephenson, Victor Storchan, Dragos-Constantin Tantaru, Artem Tyurin, Adrian Wälchli, Pengming Wang, Jianxiao Yang, Renat Zayashnikov, Alexander Zelenka Martin, Nikolay Zinov.

Partnerships & Delivery. Caroline Bercier, José Caldeira, Margarida Garcia, Tom George, Kabeer Gharzai, Glenn Hitchcock, Carson Klingenberg, Ivo Pinto, Varun Randery, Noah Smith, Arina Sugako, Jason Warner.

References

[1] M. Abdin, J. Aneja, H. Behl, et al. *Phi-4 Technical Report*. 2024 (cit. on p. 11).

[2] A. Ahmadian, C. Cremer, M. Gallé, M. Fadaee, J. Kreutzer, O. Pietquin, A. Üstün, and S. Hooker. "Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024 (cit. on p. 21).

[3] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai. “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints”. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by H. Bouamor, J. Pino, and K. Bali. Singapore: Association for Computational Linguistics, Dec. 2023 (cit. on p. 5).

[4] E. Alvarez, O. Almog, E. Chung, S. Layton, D. Stosic, R. Krashinsky, and K. Aubrey. *Introducing NVFP4 for Efficient and Accurate Low-Precision Inference*. NVIDIA Developer Blog, June 2025 (cit. on p. 24).

[5] N. Amsel, D. Persson, C. Musco, and R. M. Gower. “The Polar Express: Optimal Matrix Sign Methods and their Application to the Muon Algorithm”. In: *The Fourteenth International Conference on Learning Representations*. 2026 (cit. on p. 7).

[6] Anthropic. *Introducing Claude Haiku 4.5*. Anthropic Blog. 2025 (cit. on pp. 2, 25).

[7] Anthropic. *Introducing Claude Sonnet 4.6*. Anthropic Blog. 2025 (cit. on pp. 2, 25).

[8] Anthropic. *Quantifying infrastructure noise in agentic coding evals*. Anthropic Engineering Blog. 2026 (cit. on p. 26).

[9] Apple Inc. *FoundationDB: A Distributed Database Designed for Key-Value Storage*. https://www.foundationdb.org. Accessed 2026-05-19. 2013 (cit. on p. 4).

[10] L. Belenki, A. Agarwal, T. Shi, and K. Toutanova. “Optimizing Pre-Training Data Mixtures with Mixtures of Data Expert Models”. In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar. Vienna, Austria: Association for Computational Linguistics, July 2025 (cit. on p. 13).

[11] I. Beltagy, M. E. Peters, and A. Cohan. *Longformer: The Long-Document Transformer*. 2020 (cit. on p. 5).

[12] Y. Bisk, R. Zellers, R. Le Bras, J. Gao, and Y. Choi. “PIQA: Reasoning about Physical Commonsense in Natural Language”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.05 (Apr. 2020) (cit. on p. 34).

[13] F. Cassano, J. Gouwar, D. Nguyen, et al. “MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation”. In: *IEEE Transactions on Software Engineering* (2023) (cit. on pp. 24, 34).

[14] A. Chen, A. Li, B. Gong, et al. *MiniMax-M1: Scaling Test-Time Compute Efficiently with Lightning Attention*. 2025 (cit. on p. 21).

[15] M. F. Chen, T. Murray, D. Heineman, M. Jordan, H. Hajishirzi, C. Ré, L. Soldaini, and K. Lo. *Olmix: A Framework for Data Mixing Throughout LM Development*. 2026 (cit. on p. 13).

[16] A. Chowdhery, S. Narang, J. Devlin, et al. “PaLM: Scaling Language Modeling with Pathways”. In: *Journal of Machine Learning Research* 24.240 (2023) (cit. on p. 9).

[17] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. *Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge*. 2018 (cit. on p. 34).

[18] K. Cobbe, V. Kosaraju, M. Bavarian, et al. *Training Verifiers to Solve Math Word Problems*. 2021 (cit. on p. 24).

[19] Dagster Labs. *Dagster: The Data Orchestration Platform*. https://dagster.io. Accessed 2026-05-19. 2023 (cit. on p. 3).

[20] D. Dai, C. Deng, C. Zhao, et al. “DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024 (cit. on p. 5).

[21] DeepSeek-AI. *DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence*. Technical report. 2026 (cit. on pp. 2, 24, 25).

[22] DeepSeek-AI, A. Liu, B. Feng, et al. *DeepSeek-V3 Technical Report*. 2025 (cit. on p. 5).

[23] X. Deng, J. Da, E. Pan, et al. *SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?* 2025 (cit. on p. 25).

[24] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar. *Silent Data Corruptions at Scale*. 2021 (cit. on p. 8).

[25] K. Dobler, D. Elliott, and G. de Melo. “Token Distillation: Attention-Aware Input Embeddings for New Tokens”. In: *The Fourteenth International Conference on Learning Representations*. 2026 (cit. on p. 17).

[26] Envoy Project Authors. *Envoy: An Open Source Edge and Service Proxy*. https://www.envoyproxy.io. Accessed 2026-05-19. 2017 (cit. on p. 22).

[27] Gemma Team, M. Riviere, S. Pathak, et al. *Gemma 2: Improving Open Language Models at a Practical Size*. 2024 (cit. on p. 5).

[28] GLM Team. *GLM-4.7: Mid-Cycle Update to the GLM Coding Series*. Zhipu AI Technical Report. 2026 (cit. on pp. 2, 25).

[29] Google. *Gemma 4: Byte for byte, the most capable open models*. https://blog.google/innovation-and-ai/technology/developers-tools/gemma-4/. Google blog post. 2026 (cit. on pp. 2, 24, 25).

[30] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang. "CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution". In: *International Conference on Machine Learning*. 2024 (cit. on p. 24).

[31] S. Gunasekar, Y. Zhang, J. Aneja, et al. *Textbooks Are All You Need*. 2023 (cit. on p. 11).

[32] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. "Measuring Massive Multitask Language Understanding". In: *International Conference on Learning Representations*. 2021 (cit. on pp. 24, 34).

[33] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. "Measuring Mathematical Problem Solving With the MATH Dataset". In: *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*. 2021 (cit. on p. 24).

[34] S. Hu, Y. Tu, X. Han, et al. "MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies". In: *First Conference on Language Modeling*. 2024 (cit. on p. 5).

[35] Y. Huang, Y. Cheng, A. Bapna, et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019 (cit. on p. 6).

[36] M. Idahl, B. Droste, B. Plüster, and J. P. Harries. *propella-1: Multi-Property Document Annotation for LLM Data Curation at Scale*. 2026 (cit. on p. 11).

[37] N. Jain, K. Han, A. Gu, et al. "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code". In: *The Thirteenth International Conference on Learning Representations*. 2025 (cit. on p. 24).

[38] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024 (cit. on p. 25).

[39] K. Jordan, Y. Jin, V. Boza, J. You, F. Cesista, L. Newhouse, and J. Bernstein. *Muon: An optimizer for hidden layers in neural networks*. 2024 (cit. on p. 6).

[40] A. H. Kargaran, A. Imani, F. Yvon, and H. Schuetze. "GlotLID: Language Identification for Low-Resource Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by H. Bouamor, J. Pino, and K. Bali. Singapore: Association for Computational Linguistics, Dec. 2023 (cit. on p. 10).

[41] Kimi Team. *Kimi K2: Open Agentic Intelligence*. 2025 (cit. on p. 34).

[42] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro. "Reducing Activation Recomputation in Large Transformer Models". In: *Proceedings of Machine Learning and Systems*. Vol. 5. 2023 (cit. on p. 7).

[43] W. Kwon, Z. Li, S. Zhuang, et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023 (cit. on pp. 22, 23).

[44] A. Lee, B. Miranda, and S. Koyejo. *Beyond Scale: The Diversity Coefficient as a Data Quality Metric Demonstrates LLMs are Pre-Trained on Formally Diverse Data*. 2023 (cit. on p. 13).

[45] J. Li, A. Fang, G. Smyrnis, et al. "DataComp-LM: In search of the next generation of training sets for language models". In: *The Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2024 (cit. on p. 13).

[46] W. Liang, T. Liu, L. Wright, et al. "TorchTitan: One-stop PyTorch native solution for production ready LLM pretraining". In: *The Thirteenth International Conference on Learning Representations*. 2025 (cit. on p. 6).

[47] J. Lin, J. Tang, H. Tang, et al. "AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration". In: *Proceedings of Machine Learning and Systems*. Ed. by P. Gibbons, G. Pekhimenko, and C. D. Sa. Vol. 6. 2024 (cit. on p. 23).

[48] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. “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*. 2023 (cit. on p. 34).

[49] J. Liu, J. Su, X. Yao, et al. *Muon is Scalable for LLM Training*. 2025 (cit. on pp. 6, 9, 21).

[50] Q. Liu, X. Zheng, N. Muennighoff, G. Zeng, L. Dou, T. Pang, J. Jiang, and M. Lin. “RegMix: Data Mixture as Regression for Language Model Pre-training”. In: *The Thirteenth International Conference on Learning Representations*. 2025 (cit. on p. 13).

[51] Z. Liu, C. Zhao, I. Fedorov, et al. “SpinQuant: LLM Quantization with Learned Rotations”. In: *The Thirteenth International Conference on Learning Representations*. 2025 (cit. on p. 23).

[52] I. Loshchilov and F. Hutter. “Decoupled Weight Decay Regularization”. In: *International Conference on Learning Representations*. 2019 (cit. on p. 9).

[53] P. Maini, S. Seto, H. Bai, D. Grangier, Y. Zhang, and N. Jaitly. “Rephrasing the Web: A Recipe for Compute and Data-Efficient Language Modeling”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2024 (cit. on p. 11).

[54] S. Malladi, K. Lyu, A. Panigrahi, and S. Arora. “On the SDEs and Scaling Rules for Adaptive Gradient Algorithms”. In: *Advances in Neural Information Processing Systems*. 2022 (cit. on p. 6).

[55] M. A. Merrill, A. G. Shaw, N. Carlini, et al. *Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces*. 2026 (cit. on p. 25).

[56] P. Micikevicius, S. Narang, J. Alben, et al. “Mixed Precision Training”. In: *International Conference on Learning Representations*. 2018 (cit. on p. 9).

[57] Mistral AI. *Devstral 2: Mistral Vibe CLI*. Mistral AI Blog. 2025 (cit. on pp. 2, 25).

[58] NVIDIA. *Nemotron 3 Nano: Open, Efficient Mixture-of-Experts Hybrid Mamba-Transformer Model for Agentic Reasoning*. 2025 (cit. on pp. 5, 24, 25).

[59] OpenAI. *GPT-5.4 Nano System Card*. OpenAI System Card. 2026 (cit. on pp. 2, 25).

[60] OpenAI. *Introducing SWE-bench Verified*. OpenAI Blog. 2024 (cit. on p. 25).

[61] G. Penedo, H. Kydlíček, L. Ben allal, A. Lozhkov, M. Mitchell, C. Raffel, L. Von Werra, and T. Wolf. “The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale”. In: *The Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2024 (cit. on p. 10).

[62] B. Peng, J. Quesnelle, H. Fan, and E. Shippole. “YaRN: Efficient Context Window Extension of Large Language Models”. In: *The Twelfth International Conference on Learning Representations*. 2024 (cit. on p. 6).

[63] Poolside. *Post-Training in the Model Factory*. https://poolside.ai/blog/post-training-in-the-model-factory. Accessed 2026-05-14. 2025 (cit. on p. 22).

[64] Poolside. *Through the looking glass of benchmark hacking*. https://poolside.ai/blog/through-the-looking-glass. Accessed 2026-05-18. 2026 (cit. on p. 26).

[65] J. Qin, Y. Xi, J. Huang, et al. *APTBench: Benchmarking Agentic Potential of Base LLMs During Pre-Training*. 2025 (cit. on p. 34).

[66] Z. Qiu, Z. Huang, B. Zheng, et al. “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)*. Ed. by W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar. Vienna, Austria: Association for Computational Linguistics, July 2025 (cit. on p. 5).

[67] Z. Qiu, Z. Wang, B. Zheng, et al. “Gated Attention for Large Language Models: Non-linearity, Sparsity, and Attention-Sink-Free”. In: *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. 2026 (cit. on p. 5).

[68] Qwen Team. *Qwen3.5: Accelerating Productivity with Native Multimodal Agents*. Feb. 2026 (cit. on pp. 2, 24, 25).

[69] Qwen Team. *Qwen3.6-35B-A3B: Agentic Coding Power, Now Open to All*. Apr. 2026 (cit. on pp. 2, 25).

[70] Red Hat AI and vLLM Project. *LLM Compressor*. Aug. 2024 (cit. on p. 23).

[71] D. Rein, B. L. Hou, A. C. Stickland, J. Petty, R. Y. Pang, J. Dirani, J. Michael, and S. R. Bowman. “GPQA: A Graduate-Level Google-Proof Q&A Benchmark”. In: *First Conference on Language Modeling*. 2024 (cit. on p. 34).

[72] V. Sachidananda, J. Kessler, and Y.-A. Lai. “Efficient Domain Adaptation of Language Models via Adaptive Tokenization”. In: *Proceedings of the 2nd Workshop on Simple and Efficient Natural Language Processing (SustaiNLP), EMNLP*. 2021 (cit. on p. 17).

[73] K. Sakaguchi, R. Le Bras, C. Bhagavatula, and Y. Choi. *WinoGrande: An Adversarial Winograd Schema Challenge at Scale*. 2019 (cit. on p. 34).

[74] A. Sedova, S. Seto, N. Schluter, and P. Ablin. *Scaling Laws for Mixture Pretraining Under Data Constraints*. 2026 (cit. on p. 10).

[75] R. Sennrich, B. Haddow, and A. Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by K. Erk and N. A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016 (cit. on p. 5).

[76] Z. Shao, P. Wang, Q. Zhu, et al. *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. 2024 (cit. on p. 21).

[77] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”. In: *International Conference on Learning Representations*. 2017 (cit. on pp. 5, 6).

[78] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2019 (cit. on p. 6).

[79] L. Soldaini, R. Kinney, A. Bhagia, et al. “Dolma: an Open Corpus of Three Trillion Tokens for Language Model Pretraining Research”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024 (cit. on p. 10).

[80] D. Su, K. Kong, Y. Lin, et al. “Nemotron-CC: Transforming Common Crawl into a Refined Long-Horizon Pretraining Dataset”. In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2025 (cit. on p. 11).

[81] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. “RoFormer: Enhanced transformer with Rotary Position Embedding”. In: *Neurocomputing* 568 (2024) (cit. on p. 5).

[82] S. H. Sul, S. Arora, B. F. Spector, and C. Ré. *ParallelKittens: Systematic and Practical Simplification of Multi-GPU AI Kernels*. 2025 (cit. on p. 7).

[83] M. Suzgun, N. Scales, N. Schärli, et al. “Challenging BIG-Bench Tasks and Whether Chain-of-Thought Can Solve Them”. In: *Findings of the Association for Computational Linguistics: ACL 2023*. 2023 (cit. on p. 24).

[84] Tbench Team. *Terminal-Bench 2.0: A Benchmark for AI Agents in Terminal Environments*. https://www.tbench.ai/. 2025 (cit. on p. 25).

[85] The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io. Accessed 2026-05-19. 2014 (cit. on p. 4).

[86] The PyTorch Foundation. *PyTorch: An Open Source Machine Learning Library*. https://pytorch.org. Accessed 2026-05-19. 2016 (cit. on p. 6).

[87] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017 (cit. on p. 5).

[88] Volcano Community. *Volcano: A Cloud Native Batch System for High Performance Workloads*. https://volcano.sh. Accessed 2026-05-19. 2020 (cit. on p. 4).

[89] B. Wang and A. Komatsuzaki. *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model*. https://github.com/kingoflolz/mesh-transformer-jax. May 2021 (cit. on p. 5).

[90] X. Wang, B. Li, Y. Song, et al. *OpenHands: An Open Platform for AI Software Developers as Generalist Agents* (cit. on p. 20).

[91] Y. Wang, X. Ma, G. Zhang, et al. “MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark”. In: *The Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2024 (cit. on p. 24).

[92] Q. Wu, G. Bansal, J. Zhang, et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. 2023 (cit. on p. 13).

[93] Xiaomi LLM-Core Team. *MiMo: Unlocking the Reasoning Potential of Language Model – From Pretraining to Posttraining*. 2025 (cit. on pp. 24, 25).

[94] S. M. Xie, H. Pham, X. Dong, et al. "DoReMi: Optimizing Data Mixtures Speeds Up Language Model Pretraining". In: *Advances in Neural Information Processing Systems*. 2023 (cit. on p. 13).

[95] M. Xin, S. Priyadarshi, J. Xin, et al. *Quantization-Aware Distillation for NVFP4 Inference Accuracy Recovery*. 2026 (cit. on pp. 23, 24).

[96] R. Xiong, Y. Yang, D. He, et al. "On layer normalization in the transformer architecture". In: *Proceedings of the 37th International Conference on Machine Learning*. ICML'20. JMLR.org, 2020 (cit. on p. 5).

[97] C. Xu, Q. Sun, K. Zheng, et al. "WizardLM: Empowering Large Pre-Trained Language Models to Follow Complex Instructions". In: *The Twelfth International Conference on Learning Representations*. 2024 (cit. on p. 20).

[98] J. Yang, K. Lieret, C. E. Jimenez, et al. *SWE-smith: Scaling Data for Software Engineering Agents*. 2025 (cit. on p. 25).

[99] J. Ye, P. Liu, T. Sun, Y. Zhou, J. Zhan, and X. Qiu. "Data Mixing Laws: Optimizing Data Mixtures by Predicting Language Modeling Performance". In: *International Conference on Learning Representations*. 2025 (cit. on p. 13).

[100] Y. Yu, Y. Zhuang, J. Zhang, Y. Meng, A. J. Ratner, R. Krishna, J. Shen, and C. Zhang. "Large Language Model as Attributed Training Data Generator: A Tale of Diversity and Bias". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2023 (cit. on p. 13).

[101] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. "HellaSwag: Can a Machine Really Finish Your Sentence?" In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by A. Korhonen, D. Traum, and L. Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019 (cit. on p. 34).

[102] A. Zeng, X. Lv, Z. Hou, et al. *GLM-5: from Vibe Coding to Agentic Engineering*. 2026 (cit. on p. 17).

[103] B. Zhang and R. Sennrich. "Root Mean Square Layer Normalization". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019 (cit. on p. 5).

[104] Y. Zhao, A. Gu, R. Varma, et al. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel". In: *Proceedings of the VLDB Endowment* 16.12 (2023) (cit. on p. 6).

[105] C. Zheng, S. Liu, M. Li, et al. *Group Sequence Policy Optimization*. 2025 (cit. on p. 21).

[106] L. Zheng, W.-L. Chiang, Y. Sheng, et al. "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2023 (cit. on p. 13).

[107] T. Y. Zhuo, V. M. Chien, J. Chim, et al. "BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions". In: *The Thirteenth International Conference on Learning Representations*. 2025 (cit. on p. 24).

[108] B. Zoph, I. Bello, S. Kumar, N. Du, Y. Huang, J. Dean, N. Shazeer, and W. Fedus. *ST-MoE: Designing Stable and Transferable Sparse Expert Models*. 2022 (cit. on p. 9).

A Additional Details

A.1 WSD Optimal-LR Scaling Law

Supporting figures and tables for the WSD optimal-LR scaling law described in Section 3.1.1. All sweep runs use four model sizes (2B, 4B, 8B, 16B total parameters), six learning rates from 8×10^{-5} to 6×10^{-3}, fixed global batch $B_0 = 8.4$M tokens, and the stage-1 pre-training mix.

Per-cell fits. Per-(N, lr) power-law fits $L(D) = L_0 + A\,D^{-\gamma}$ across the five cooldown points are shown in Figure 8; Figure 9 zooms in on the 16B model.

For each $(N, D \in \{60, 120, 240, 480, 960\}\mathrm{B})$ we fit a parabola in $\log_{10}(\mathrm{lr})$ space and take the vertex as $\mathrm{lr}^\star(N, D)$ (Figure 10); rows with curvature $a < 0.065$ (flat parabolas, optimum poorly constrained) are excluded from the global fit.

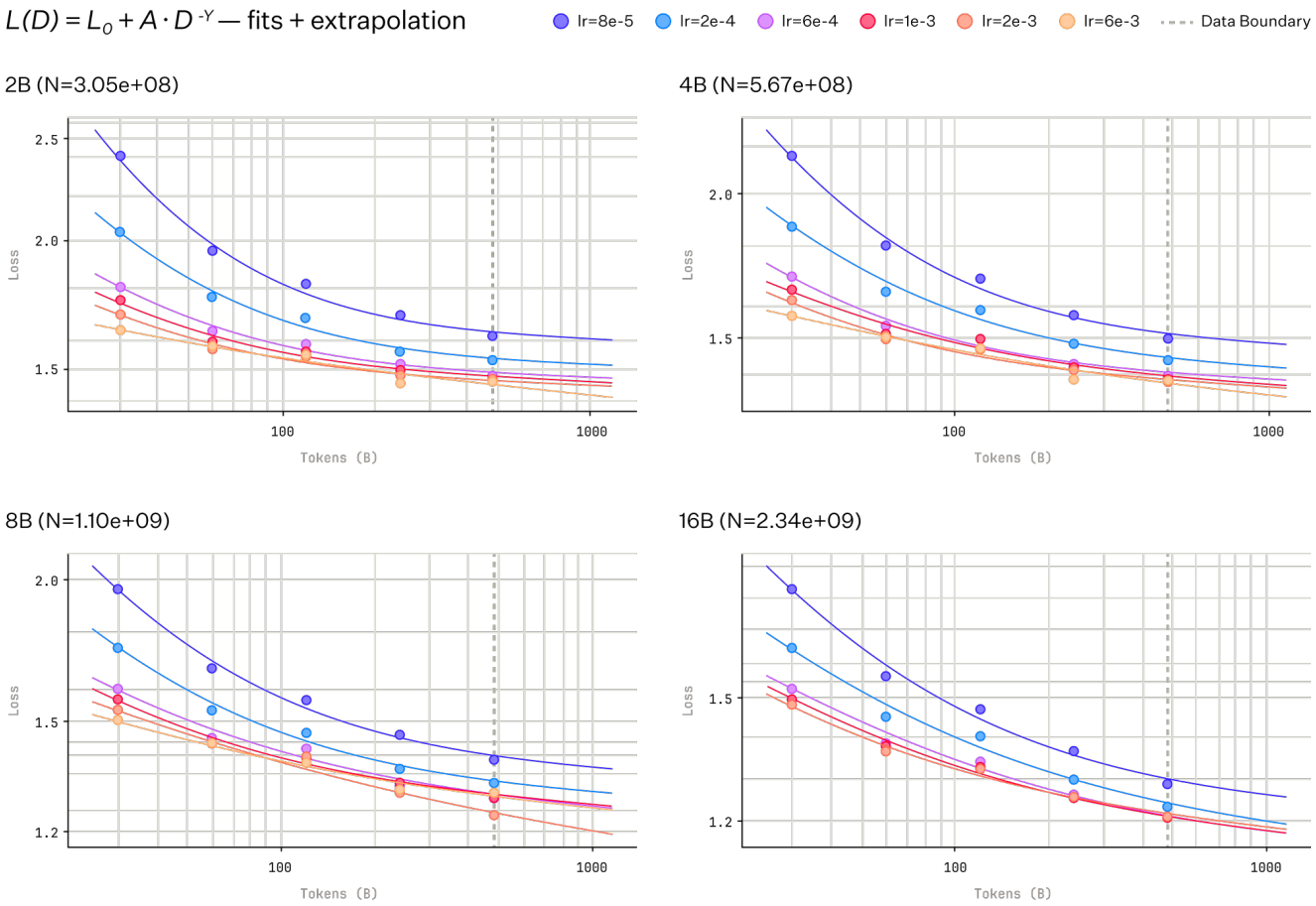


Figure 8: Loss-vs-tokens power-law fits $L(D) = L_0 + A\,D^{-\gamma}$ for each model size, one curve per learning rate. Dashed vertical lines mark the 480B-token data boundary; curves are extrapolated to ∼960B tokens.

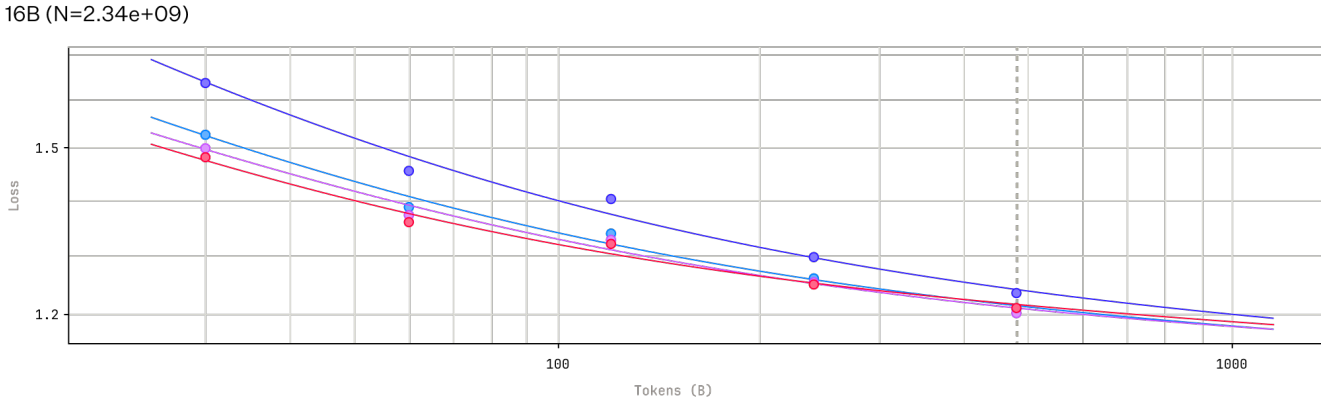


Figure 9: Same fit as Figure 8 for the 16B model with the lowest learning rate (8×10^{-5}) excluded.

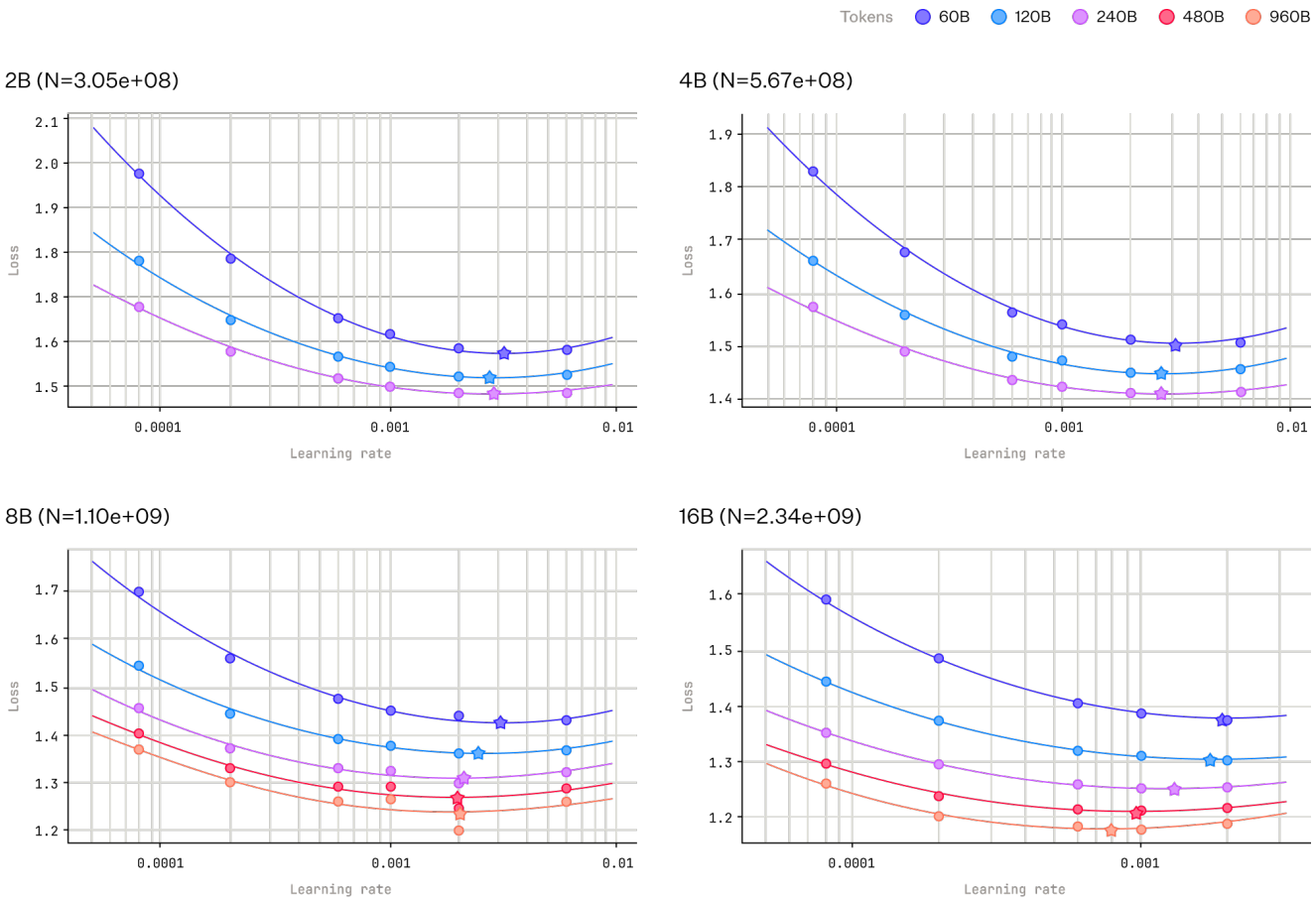


Figure 10: Per-(N, D) parabola fits with vertex lr* marked by stars.

Fit robustness. A companion fit using a stricter curvature threshold ($a \geq 0.065$, 16 retained points) yields $(\ell_0, \alpha, \beta) = (4.292, -0.4393, -0.2689)$ with $R^2 = 0.8865$ and per-size mean residuals (in $\log_{10}$ lr) of $+0.001/-0.041/+0.058/-0.022$ for the 2B / 4B / 8B / 16B groups (multiplicative LR errors $1.00\times/0.91\times/1.14\times/0.95\times$); the headline form in Equation (1) agrees within the LR-extrapolation uncertainty for Laguna XS.2.

External cross-check. Applied to Kimi K2 [41] ($N = 32.6$B active, $D = 15.5$T tokens, $B = 67$M), the law predicts $\sim 3.5\times10^{-4}$; Kimi K2 was trained at 2×10^{-4}. The two differ within an order of magnitude despite Kimi K2 using a 35% cooldown (vs. our 30%), a different MoE sparsity ratio, MuonClip, aux-loss-free load balancing, and different data, so this should be read as suggestive rather than a validation of the law outside our regime.

A.2 SWA Architecture Ablations

Supporting tables for the SWA architecture summary in Section 3.1. All ablation rows use a 16B-total / 2.3B-active MoE model; hyperparameters are included in Table 8. Only the attention design (window, gating, θ_{swa}, RoPE coverage, head allocation) varies between rows.

The 4K Avg is calculated over 10 base benchmarks (PiQA [12], WinoGrande [73], ARC-Easy & ARC-Challenge [17], HellaSwag [101], APTBench-4k [65], MMLU [32], GPQA-Diamond [71], MultiPL-E [13], EvalPlus [48]); the 32K and 128K Avgs are each calculated over 4 tasks at the corresponding window (APTBench, RULER QA, GSMInfinite, LongBenchV2-Academic).

Five reruns of the baseline give a standard deviation of $\sim$0.0026 on the 4K Avg, so we treat differences below $\sim$0.005 as within noise.

Table 8: Ablation model hyperparameters.

Hyperparameter	Value
Total / active parameters	16.6B / 2.3B
Layers	22
Hidden size d_{model}	2048
FFN intermediate (dense)	8192
FFN intermediate (per routed expert)	1024
Routed experts (top-k)	128 (top-8)
Shared experts	1
Optimizer	Muon
LR scheduler	Cosine
Pre-training	
Sequence length	4096
Global batch size	8.4M tokens
Total tokens / steps	335B / 40,000
Peak LR	2×10^{-3}
Warmup steps	1,000
32K / 128K continued pre-training (long-context)	
Sequence length	32,768 / 131,072
Global batch size	8.4M tokens
Total tokens / steps	50B / 6,000
Peak LR (32K / 128K)	7×10^{-5} / 7×10^{-6}
Warmup steps	200

Table 9: Architecture ablations across short and long contexts.

Architecture	4K Avg ↑	32K Avg ↑	128K Avg ↑
Dense GA, full RoPE, full gating	0.5389	**0.308**	0.290
+ SWA-1024 (interleaved 3:1)	0.5292	0.274	0.267
+ per-head gating, θ_{swa} = 10,000	0.5328	0.267	0.272
+ GA Partial RoPE (50%)	0.5425	0.266	0.284
+ SWA-512	0.5449	0.285	**0.304**
+ 48 GA / 64 SWA Q-heads, k_{dense}=1 (final ablation architecture)	**0.5455**	0.305	0.296

A.3 Benchmark Improvements

Many benchmarks contain components which encounter reliability issues when run at scale, such as rate limits on third-party dependencies, unpinned dependency versions, and behavior unique to our sandbox. To combat infrastructure noise outside of our control, we made numerous improvements to the agentic benchmarks reported on.

Terminal-Bench 2.0. Many of the tasks in Terminal-Bench 2.0 rely on third-party dependencies hosted on external registries, which have rate limits that can be easily hit when running the benchmark at scale. Just like many other benchmarks, external dependencies can also be pulled or become stale, so we also updated numerous tasks to rely on updated dependency versions.

- All tasks were run with a 3-hour timeout, 32 CPUs, 48 GB RAM, and 50 GB storage.

- We added in-task retries on components that rely on external services which are known to be flaky
- We vendored (stored copies in internal storage) all external dependencies for tasks `fix-ocaml-gc`, `reshard-c4-data`, `build-cython-ext`, `filter-js-from-html`, `sam-cell-seg`, `pytorch-model-cli`, `install-windows-3.11` and bundle the version of `uv`/`uvx` required by all images. This greatly reduced failures due to external rate limits.
- Tasks `qemu-alpine-ssh` and `qemu-startup` were updated to resolve a problem where open telnet sessions could not be opened in the verification stage, resulting in failures due to inability to grade results.
- Task `polyglot-c-py` replaced the strict assertion that `main.py.c` is the only file in the directory with an assertion that the file exists in the directory. Without this change, a valid solution (compiling in-place to produce a binary alongside `main.py.c`) could fail due to test-time requirements that contradict the instructions which allow this solution pattern.
- Fixed dependency drift on tasks: `build-pmars`, `crack-7z-hash`, `mteb-leaderboard`, `mteb-retrieve`, `rstan-to-pystan`, `dna-assembly`, `dna-insert`, `hf-model-inference`, and `kv-store-grpc`. Dependency drift occurs when a task's dependencies change from what the original build expected, resulting in broken builds or tests in the environment.

We are collaborating with the Harbor team to merge these changes upstream where possible.

SWE-bench Verified. The upstream SWE-bench Verified dataset is not frequently updated; running with the canonical dataset results in dependency and fixture drift. Our setup also faced rate limit failures from external providers, much like Terminal-Bench 2.0, due to the parallelism at which we run evaluation.

- The image for task `astropy-8872` is pinned to `setuptools==68.0.0` which raises a deprecated version warning and errantly fails the grader in Harbor. To fix this we pinned to `setuptools==58.0.0`.
- In Task `astropy-7606`, pytest emits `test_compose_roundtrip[unit0]` but the dataset expects `test_compose_roundtrip[]`. We fixed this mismatch in our setup.
- Task `sphinx-8475` has tests which rely on a `w3.org` URL which returns a 403. The task was fixed to allow for this instead of errantly failing.
- The pass-to-pass tests for Task `django-10097` fail with the golden patch due to Django/SQLite teardown issues unrelated to the task. If the known teardown flake occurs with the exact signature seen under the golden patch, and is the only failure, we ignore it.
- Eight of the `psf/requests` repo tasks call a `httpbin()` helper which defaults to a `httpbin.org` endpoint which under heavy load can return a transient 5xx error. We run a sidecar for HTTP and HTTPS requests to handle these requests instead with a fallback to the default `httpbin.org` endpoint.

SWE-bench Multilingual. Our SWE-bench Multilingual changes focused on improving the reliability of the dataset when run at scale with high parallelism.

- All images were updated to have future git tags removed to prevent possible reward hacking.
- Task `apache__lucene-12626` was updated to have additional network retries due to higher than normal network flakes.
- Task `tokio-rs__tokio-4384` fails due to dependency mismatches; we pin dependencies (`rand 0.8.5`, `rand_core 0.6.4`, and `getrandom 0.2.15`).
- Task `tokio-rs__tokio-6838` deterministically fails due to pass-to-pass test `uds_stream::epollhup` returning `Ok(_)` where the test expects `Err(ConnectionReset | ConnectionRefused)`. This issue was due to our sandbox kernel's epoll readiness not including the expected `EPOLLHUP` bit when a Unix-socket listener is dropped mid-connect. This resulted in an `Ok(_)` signal and subsequent unit test failures in the repo's default state. The task was updated to allow `Ok(_)` signals in the pass-to-pass check.
- Task `micropython__micropython-12158` deadlocks in our sandbox service due to thread starvation; we fixed this by updating the `tests/thread/thread_exc1.py` test to use `time.sleep(0)` instead of `pass`.

SWE-Bench Pro. For SWE-Bench Pro we made edits focused on fixing verifier selections and the previously mentioned reward hack vulnerabilities.

- All images were updated to have git history after the checked-out commit fully removed to prevent potential reward hacking. See https://github.com/harbor-framework/harbor/pull/1593
- Tasks `future-architect__vuls` and `ansible__ansible` had incorrect fail-to-pass or pass-to-pass names due to trailing punctuation.
- Task `tutao__tutanota` contained incorrect suffixes for its subset aggregations; this was fixed by dropping the `(\d+ assertions)` prior to comparison.