MAI

RL infrastructure at Microsoft AI

Long, stable, and numerically accurate Reinforcement Learning (RL) is an essential piece of Microsoft AI's (MAI) path to Humanist Superintelligence. We aim to climb capabilities from the base model through RL itself, which means runs of many thousands of training steps spanning weeks to months on thousands of GPUs, rather than a short RL phase at the end of post-training. Over runs that long, small numerical mismatches and system inefficiencies compound, so we built Rocket, MAI's in-house RL framework, to enable us to post-models at increasing scale. We recently shared an update on our new three-trillion-parameter base model, MAI Base 2, in Frontier Pre-training at Microsoft AI. MAI Base 2 is now being post-trained with Rocket, climbing from the base model through RL without distillation from third-party models. This work is separate from the MAI Code 2 models.

Post-training MAI Base 2

The MAI Thinking 1 technical report introduced Rocket as the RL framework which post-trained our previous pre-trained model, MAI Base 1, into the reasoning model MAI Thinking 1. Since then, we have improved Rocket to operate at the larger scale demanded by MAI Base 2. RL infrastructure needs to address many challenges at increasing scale:

1. **Staleness.** Agentic rollouts can take hours while the learner keeps updating the policy. Because GRPO scores each rollout against the others for the same problem, a single slow or failed rollout holds back its entire group, so training drifts further off-policy.
2. **Numerical consistency.** Fitting a larger model and longer contexts into the sampler requires lower-precision storage, and differences in precision, kernels, and parallelism between the sampler and learner can slow learning and destabilize training.
3. **Fault tolerance and elasticity.** Runs that span thousands of GPUs for weeks or months will encounter hardware failures, while the best runtime configuration and data mixture change. Restarting the whole job for every failure or reconfiguration wastes GPU time, so the system must recover and reconfigure components while the run continues.

RL infrastructure built for MAI's training recipe

The right RL infrastructure depends on the training recipe it serves, and one factor that is easy to overlook is distillation. A model distilled from a stronger teacher begins RL close to its ceiling, so RL becomes a comparatively short refinement phase. In that regime, the infrastructure priorities differ: a failed run can simply be restarted, and small mismatches between sampler and learner have little time to compound.

MAI

Rocket is built for the more demanding case of climbing from base capabilities through RL itself, without a distilled head start. Being able to reach the frontier without depending on other labs' models is central to MAI's self-sufficiency. The infrastructure challenges above become the main design constraints rather than edge cases in this regime. Hardware failures are a certainty, small numerical mismatches accumulate over thousands of updates, and the best data mixture and GPU allocation shift as the policy improves.

Open-source frameworks such as verl and slime provide strong foundations for RL post-training, including distributed execution, customizable agent workflows, and asynchronous sampling. However, our long climbs from a base model push these capabilities beyond the regime they are typically exercised in. We built Rocket to tightly design our RL infrastructure around the unique needs of our RL workloads and MAI's models.

MAI develops and operates three tightly connected systems necessary for RL: the learner and training framework, Rocket, and an inference engine built on an MAI-maintained fork of SGLang. Owning all three allows us to co-design the interfaces between them rather than optimize each component in isolation.

This does not mean building every layer from scratch. Rocket leverages Kubernetes or Slurm to allocate nodes to a job, and Rocket's components run on Ray as actors. We adopted SGLang as our inference foundation and have substantially adapted it to MAI's models and long-running agentic RL workloads. Owning these integrations carries a maintenance cost, but lets us evolve the complete RL loop rather than optimize its components in isolation. A future blog post will detail the ways that we adapted SGLang to fit our workloads.

Rocket overview

Figure 1 shows Rocket's asynchronous training loop. A *problem* defines a task; a *rollout* is one attempt at solving it. Our long-running RL training is based on Group Relative Policy Optimization (GRPO), which computes advantages by comparing rewards across rollouts of the same problem. The controller (1) selects a problem from the curriculum and sends it to a problem worker (2), which dispatches a set of rollout requests to the rollout worker pool (3). Each problem and rollout worker handles many requests concurrently.

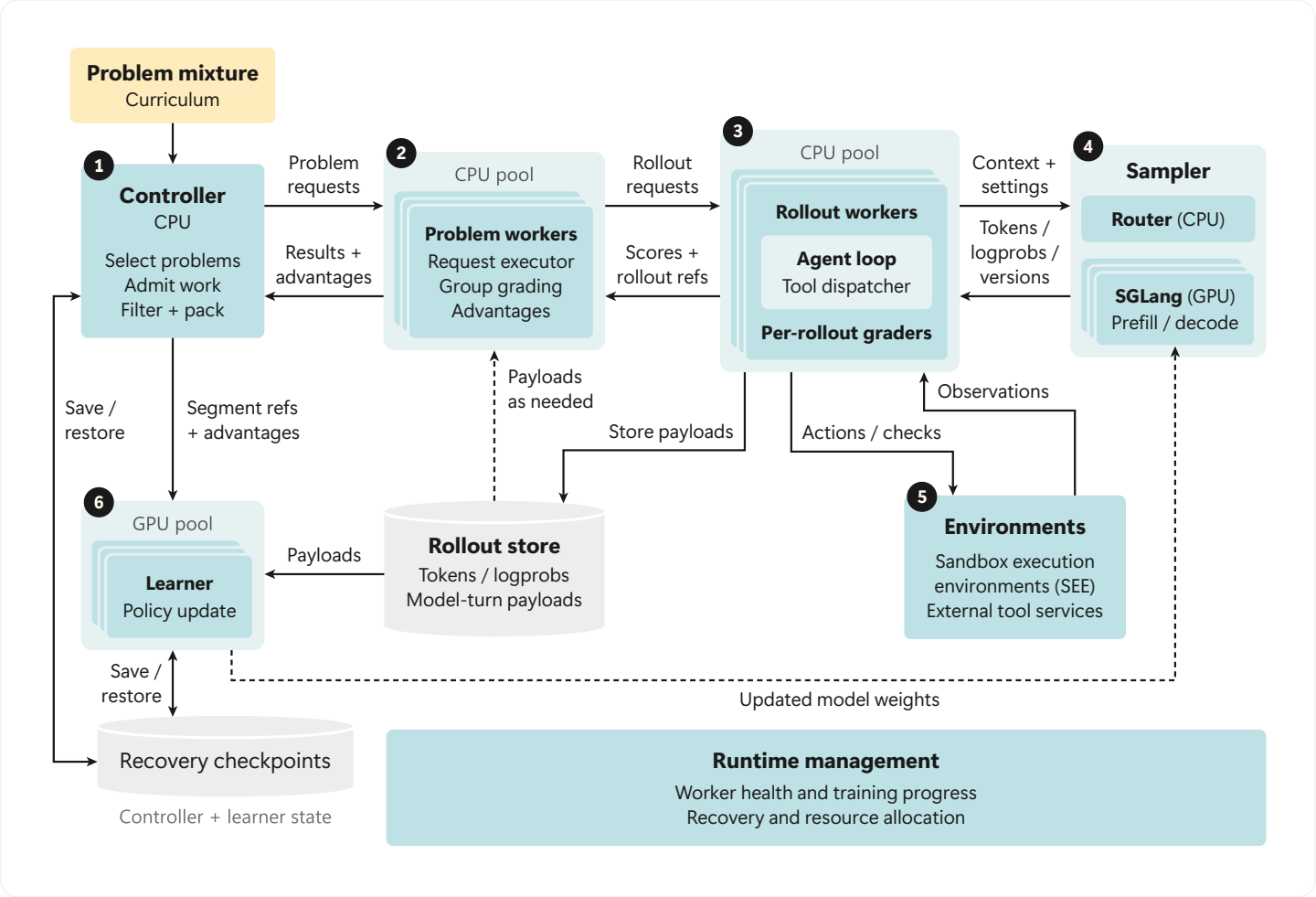


Figure 1: Overview of Rocket's asynchronous RL architecture.

Each rollout worker tracks the state of the ongoing rollouts it manages. For agentic tasks, an agent loop requests model responses from the sampler (4) and routes tool calls through the tool dispatcher to the task environment (5). The agent can execute actions in sandbox execution environments (SEE), query search services, or invoke tools exposed by MCP servers. Tool results become context for the next model call, and the loop continues until the agent finishes or reaches a configured limit.

Per-rollout graders then evaluate the trajectory and, where needed, the environment state. The rollout worker (3) stores the trajectory payload and returns scores and references to the stored rollouts to the problem worker (2). The problem worker collects a problem's rollouts, runs configured group graders on all the rollouts together, and then computes advantages for all rollouts in the GRPO group.

The problem worker returns these results to the controller (1), which selects eligible data and packs *training segments* into learner batches. Each segment is a training example containing one or more model turns. The batches carry segment references and advantages; the learner (6) retrieves the corresponding token IDs and sampling log probabilities from the rollout store.

The learner updates the policy, and the controller periodically initiates a weight transfer to refresh the sampler. Recovery checkpoints preserve both learner and controller state. Runtime management monitors worker health and coordinates recovery and resource allocation.

Controlling staleness

Our largest RL runs mix problems that finish in minutes, such as Instruction Following, with long horizon agentic tasks that last hours, such as software-engineering tasks (Figure 2). Rocket is an asynchronous RL framework by design, so the learner continues updating while rollouts run. We measure *policy staleness* as the gap between the policy version used to generate a token and the learner's version when it trains on that token.

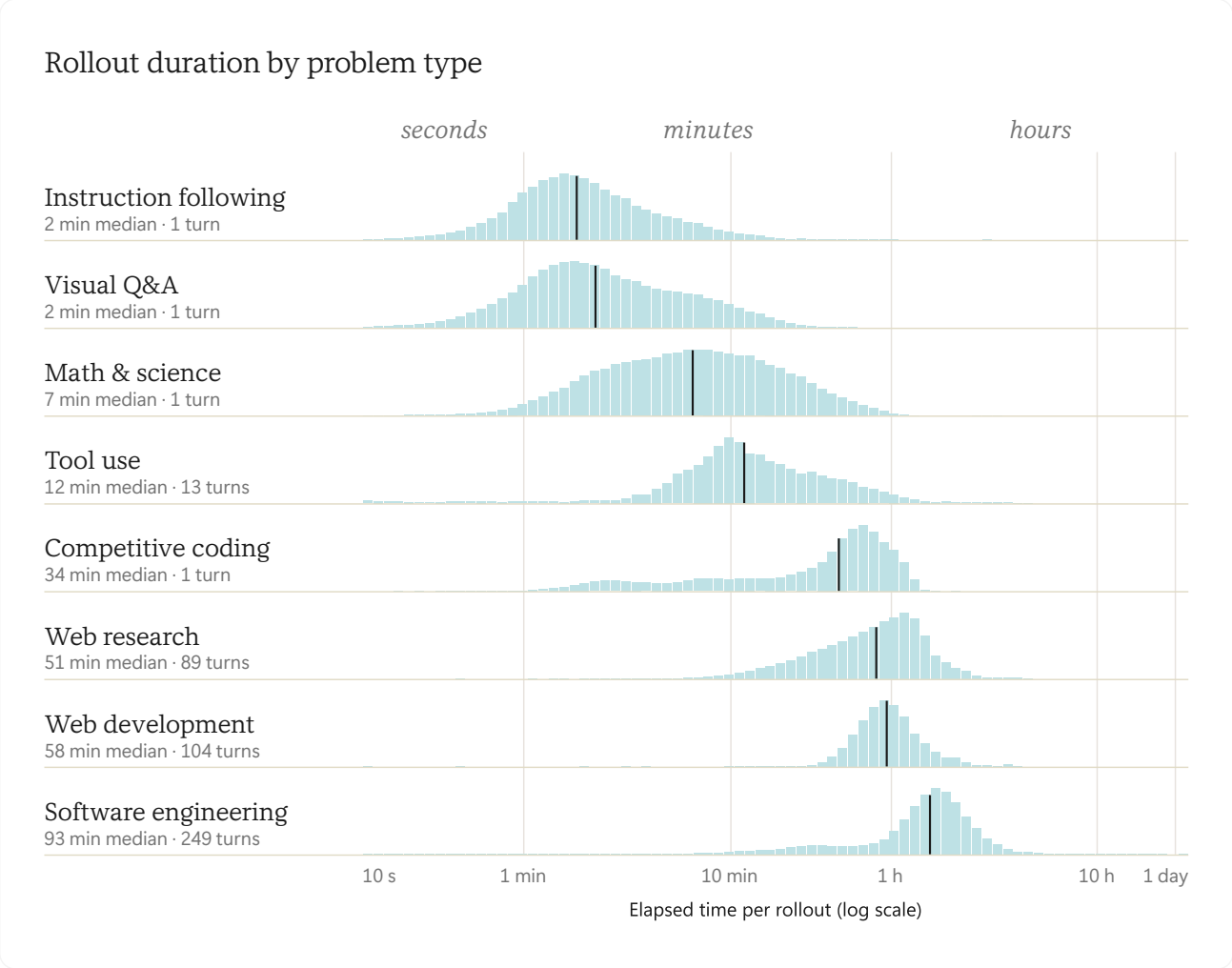


Figure 2: Rollout duration by problem type across 1.13M rollouts from one day of a production run. Elapsed time per rollout includes model inference, tool calls and waits; turns are the number of model turns per rollout.

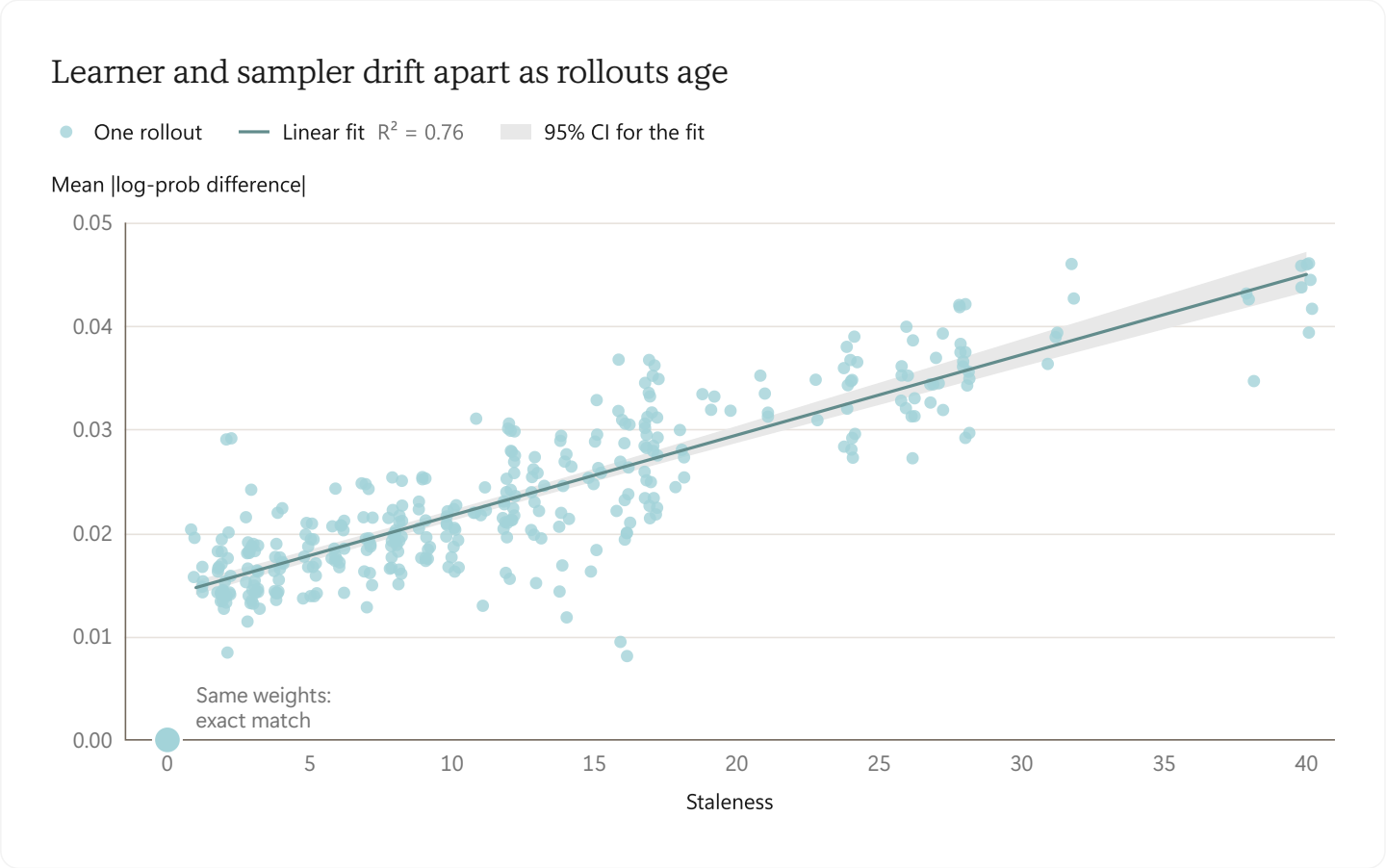


Figure 3: Higher staleness is correlated with larger differences between learner and sampler token probabilities.

When the sampler policy and learner policy differ because of staleness, training becomes off-policy, which increases the risk of instability due to token probability and activation differences between the two policies. Figure 3 shows that the mean absolute difference between the sampler's and learner's log-probabilities grows with staleness. When staleness is zero, there is no difference between token probabilities in the sampler and learner (more details in the Numerical parity section below).

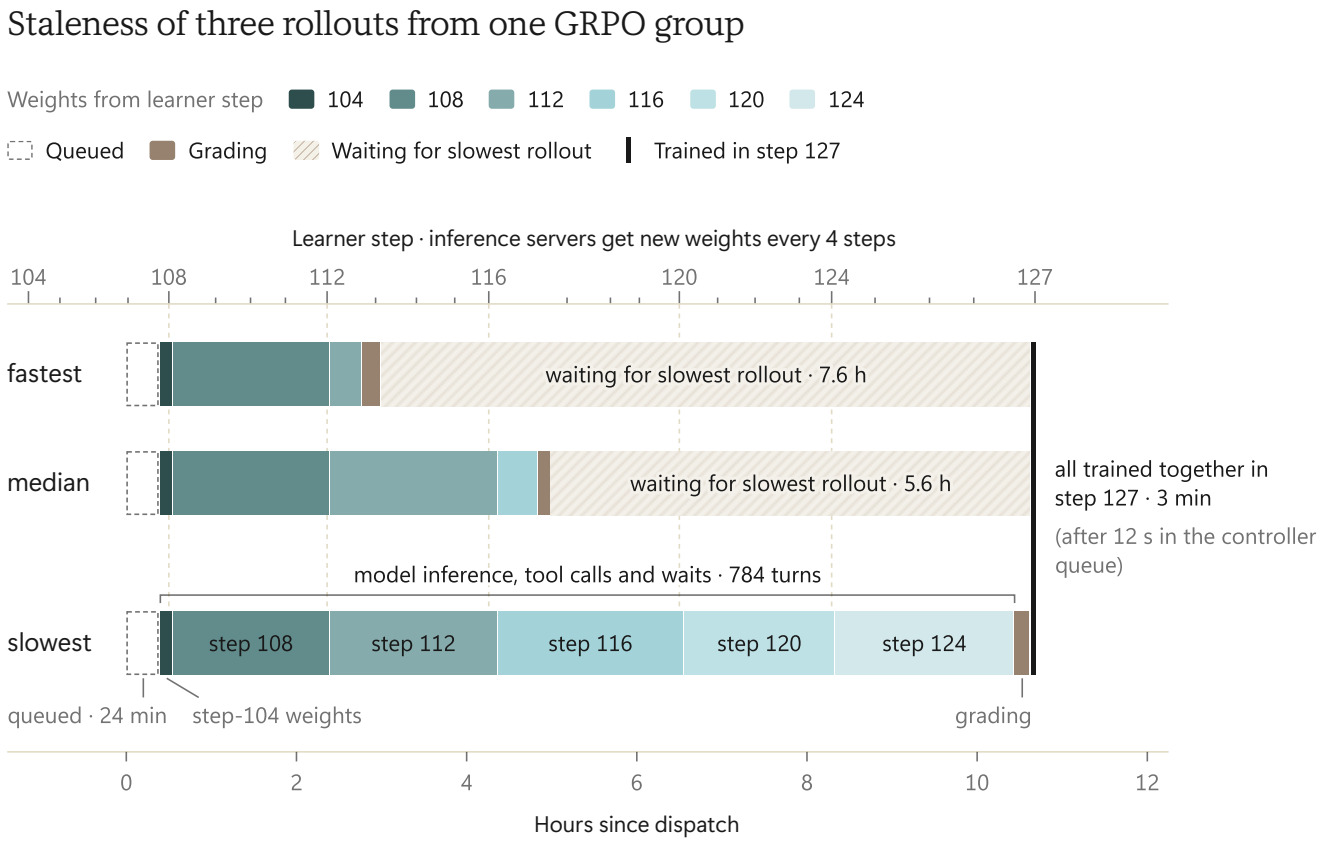


Figure 4: Three sample rollouts from a single GRPO group, colored by the version of the model weights that inference servers sampled from at each turn.

Figure 4 follows three rollouts from one GRPO group, from dispatch to the training step that consumes them. In this example, the sampler receives up-to-date learner weights every four steps. When the time between training steps is roughly constant, a token's staleness is approximately

$$\text{Sampler lag} + \frac{\text{Time when token is trained on} - \text{Time when token was generated}}{\text{Time between training steps}}$$

The first term, sampler lag, measures how far the sampler's policy version lags the learner's when a token is generated. The second captures staleness accumulated between generation and training: while the token's rollout finishes, while the completed rollout waits for the rest of its GRPO group, and while the group waits for the learner. The controls below target each of these in turn.

Keeping the sampler current. In Figure 4, the sampler receives up-to-date learner weights every 4 training steps, so sampler lag is 0–3 steps. We reduce this term through frequent sampler weight updates. The distributed learner must supply updated weights to every sampler replica, and sending weights separately to each one would make learner-side communication cost grow with the inference fleet size. Rocket therefore transfers weights to samplers in two stages: selected sampler engines first pull packed

parameter chunks from the learner, and then distribute them to corresponding ranks across sampler replicas. In our current configuration and training run, weight transfer takes 15 seconds, enabling frequent sampler weight updates to limit policy lag at generation.

Prioritizing existing rollouts over new ones. A long wait before a rollout's first token is generated does not contribute to staleness. Once a token is generated, however, it becomes increasingly stale as the rollout performs additional inference calls, tool calls, and requeues. The slowest rollout in Figure 4 samples from six successive weight versions before it completes. In early experiments, multi-turn rollouts sometimes paused for hours between inference requests while waiting for inference capacity. To mitigate this, we prioritize finishing existing rollouts over starting new ones. Each request inherits a priority based on its rollout's start time, so earlier rollouts are served first. We also automatically scale the number of in-flight rollouts up or down to match available inference capacity, keeping queues short under normal conditions. Even with this automatic scaling, new rollouts can wait for a long time before their first request. Each rollout in Figure 4 waits 24 minutes before its first inference request begins. Rollout-level timeouts must account for this wait, and rollouts may hold VMs and other resources before generation starts.

Dropping failed rollouts. Across a GRPO group, tokens from completed rollouts become stale while waiting for the rest of their group to finish so that training can occur. In Figure 4, the fastest rollout waits 7.6 hours for the slowest one to finish. Retrying failed rollouts can add a large gap between the retry finishing and the rollouts that succeeded on the first try. We therefore drop failed rollouts rather than hold back the group's training data. This reduces staleness, but dropping failed rollouts can add noise to or bias the baseline for GRPO's group advantage, since the failed rollouts may share features that never make their way to the learner.

Keeping jobs inference-bound. Completed rollouts that wait in a long internal queue before they're sent to the learner accumulate further staleness. We overallocate GPUs to our learner so that RL jobs are bottlenecked by inference capacity, not training capacity. In Figure 4, the completed group spends just 12 seconds in the controller queue before it is trained on. The trade-off is that the learner's GPUs sit partially idle.

The formula's denominator has a counterintuitive consequence: adding inference servers while leaving batch size unchanged can increase staleness. With more inference servers, it takes less time to collect enough rollouts for a training step. Since our jobs are typically inference-bound, this shortens the time between training steps. More training steps can be completed in the time it takes for a rollout to finish, even if the rollout's wall-clock time is unchanged.

Numerical precision

For MAI Thinking 1, we prioritized numerical fidelity by retaining BF16 weights and KV caches throughout the sampler. Since MAI Base 2 is a much larger model, we selectively use MXFP8 for MoE weights and KV caches to accelerate rollouts while maintaining training stability. Figure 5 summarizes the precision choices in the sampler and learner.

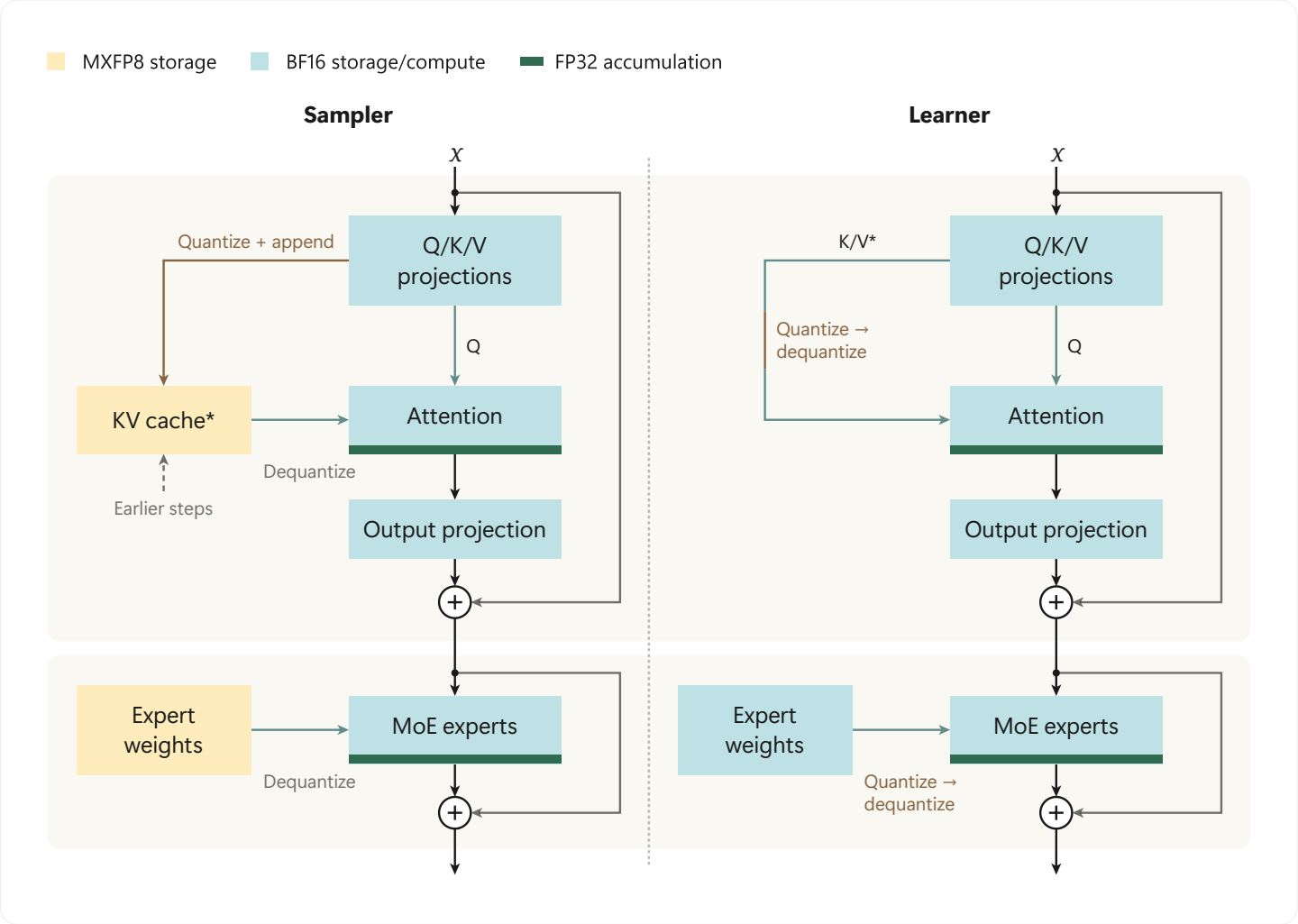


Figure 5: The storage precision choices in sampler and learner.
*Local-attention and the final three global-attention layers use BF16 with no conversion.

MoE expert weights account for over 95% of the model's parameters. Storing these weights in MXFP8 rather than BF16 nearly halves the sampler's weight memory footprint, enabling a smaller expert-parallel (EP) degree and fewer GPUs per sampler replica. In our experiments, performing the expert matrix multiplications in MXFP8 produced larger sampler–learner log-probability discrepancies than dequantizing the MXFP8 weights to BF16 and multiplying them with BF16 activations. We therefore store MoE weights in MXFP8 but dequantize them to BF16 for computation. This retains most of the speed benefits while keeping the sampler closer to the learner. To improve training stability, we reproduce the sampler's weight quantization in the learner by quantizing MoE weights to MXFP8 and immediately dequantizing them to BF16 for computation.

We apply the same separation of storage and compute precision to the sampler's global-attention KV cache. Within the attention kernel, quantized KV entries are dequantized to BF16 for computation. The learner applies corresponding KV fake quantization to reproduce the sampler's quantization effects. The

smaller KV cache footprint supports longer sequences and higher rollout concurrency, while reducing the need to offload KV entries or retract requests under memory pressure.

MAI Base 2 mixes local and global attention layers at a ratio of 5:1, and our ablations showed that quantizing both local- and global-attention KV caches introduced substantially larger sampler–learner log-probability discrepancies than quantizing global-attention KV caches alone. We therefore retain BF16 KV caches for local-attention, and, as a numerical precaution, for the final three global attention layers. All other global-attention layers use MXFP8 storage. The resulting KV cache configuration accommodates approximately 67% more global-context tokens than the BF16 KV cache configuration, while maintaining roughly the same local-attention cache capacity.

Numerical parity

Matching storage and compute precision between the sampler and learner is only one part of numerical consistency. Differences in kernels, parallelism, operation ordering, and intermediate rounding can yield different token probabilities even with identical checkpoints and inputs. These implementation-induced discrepancies can also evolve during training. In a controlled comparison with identical inputs, we observed a larger sampler–learner log-probability discrepancy at a later RL checkpoint than at initialization. This suggests that optimization may amplify or exploit numerical differences between the learner and sampler. Related work from Thinking Machines and DeepSeek has likewise emphasized consistency between training and inference implementations.

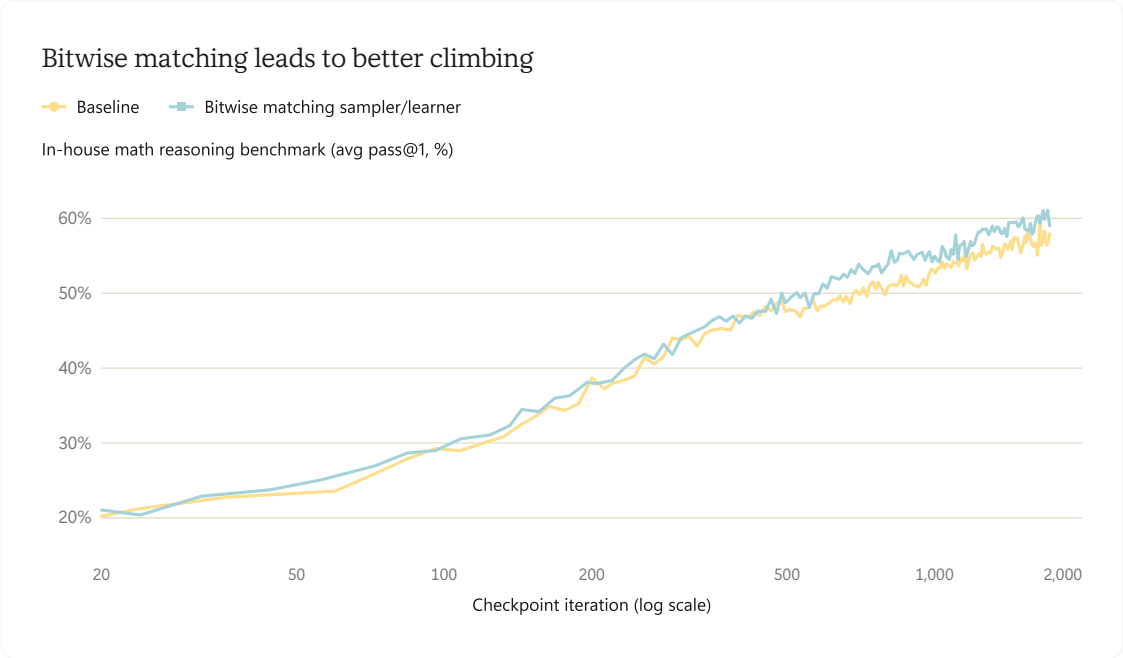


Figure 6: Two small-scale RL runs starting from the same checkpoint with (blue) and without (yellow) bitwise matching between the learner and sampler on L42, the 42-layer model from the MAI Base 2 scaling ladder. The y-axis is mean pass@1 on an in-house math reasoning benchmark.

We achieved bitwise parity between sampler and learner computations for identical checkpoints and inputs, with no noticeable performance degradation. Figure 6 compares two RL runs on a smaller model with 7B active and 180B total parameters (L42 on our scaling ladder), initialized from the same checkpoint and launched roughly one month apart. The baseline run (yellow) predates the numerical consistency improvements, while the bitwise-matched run (blue) incorporates them. The bitwise-matched run achieves higher evaluation scores at comparable training steps.

Achieving parity without sacrificing performance requires accommodating the distinct computational demands of sampler decode, sampler prefill, and learner forward passes. Each benefits from different parallelism strategies and kernel implementations, yet all must preserve the same numerical behavior. Below, we highlight several changes that made this possible.

- **Attention output projections.** Row-parallel projections under tensor parallelism (TP) require summing partial outputs across GPUs, a known source of numerical discrepancies (see Thinking Machines Lab and NeMo RL). We retain TP for attention in the sampler for performance, but replicate the attention output-projection weights to eliminate the reduction across GPUs after the projection. Before applying the projection, we all-gather the attention-head outputs so that each token's full projection can be computed on a single GPU. The learner operates without TP, allowing its summation order to match the sampler's.
- **MoE expert combination.** We distribute experts using expert parallelism (EP), without TP within individual experts. Combining the selected experts' outputs still requires a weighted sum with normalized routing weights. We use all-to-all communication to return individual expert outputs to each token's source device, then compute the weighted sum locally in FP32. Matching the accumulation precision and reduction order between the sampler and learner eliminates discrepancies introduced by expert combination.
- **Attention kernels.** FlashAttention's floating-point results depend on tiling, KV partitioning, and kernel arithmetic (see Is Flash Attention Stable?). We use a shared FA4 implementation and align its execution across learner forward, sampler prefill, and sampler decode. For sliding-window attention, we align the sampler's K-block boundaries with the learner's tiles and mask positions outside the valid window. This prevents the compact KV cache layout from changing token grouping in the online softmax and attention-value reductions. We jointly tune the number of KV tokens per split to maximize overall throughput in our training setup, then use this fixed value and matching split boundaries across sampler decode, sampler prefill, and learner forward passes to preserve a consistent reduction structure.

Long-running jobs, fault tolerance, and elasticity

The staleness and numerical controls above support stable learning. Sustaining that progress over weeks or months also requires recovering from failures and adaptation to changing resource needs. Rocket's job-management layer supervises the learner, controller, and inference workers, coordinating recovery, resource allocation, and runtime configuration. Separating supervision from the RL loop lets Rocket

recover or resize individual components without restarting the whole job. Table 1 estimates the training time and GPU-hours saved through targeted recovery and elastic runtime reconfiguration from a portion of RL training.

Event type	Estimated training time saved per event	Estimated time saved (h)	Percentage	Estimated GPU time saved (GPU-hours)
In-Job recovery (n = 21)	0.5 h	10.5	0.42%	63,000
Learner recovery (n = 88)	2 h	176	7.07%	1,056,000
Runtime elasticity (n = 123)	2 h	246	9.88%	1,476,000
Overall training time	—	2,490	100.00%	14,940,000
Total estimated savings	—	**432.5**	**17.37%**	**2,595,000**

Table 1: Estimated savings from targeted recovery and elastic runtime reconfiguration from a portion of RL training, assuming 6,000 GPUs. Per-event savings are estimates that vary with the workload mix and the point at which a failure or reconfiguration occurs.

Fault tolerance

Rocket recovers at the smallest scope needed to restore progress, preserving healthy components and valid in-flight work. Health checks track liveness and training progress, treating stalled processes as unhealthy.

Learner recovery. The controller, rollout workers, and inference fleet remain running when the learner fails (Figure 1). When the learner fails, Rocket pauses admission of new problems and terminates the learner ranks rather than waiting for each rank's NCCL timeout. It then rebuilds the learner, restores the latest checkpoint, and reconnects it to the existing controller. Curriculum state and valid pending work are retained, while queued and in-flight rollouts can continue on surviving services. In our runs, learner-only recovery takes roughly five minutes. We keep inference running during this interval, accepting the potential staleness cost rather than idling the fleet. Table 1 shows that learner recovery accounted for saving over 7% of the GPU hours in a recent run.

Inference recovery. A failed inference replica can be replaced without restarting healthy replicas or the learner. If repeated losses reduce the fraction of live inference workers below a configured threshold, Rocket restarts the job at the cluster-scheduler level to restore capacity.

Escalating recovery. When component-local repair is insufficient, Rocket can rebuild within the existing Ray job, denotated as "In-Job Recovery" in Table 1. If in-job recovery cannot restore progress or sufficient healthy capacity, Rocket can requeue the workload at the cluster level. In-job recovery avoids cluster scheduling and container-startup overhead and saves .42% of the training compute.

Control-plane isolation. Ray heads and Rocket's CPU actors run on CPU-only nodes, which have been more stable than GPU nodes in our deployments. This reduces the chance that a GPU-side failure takes down job supervision or other control services.

Elasticity

The same component boundaries that enable fault tolerance also let us adapt the job without a full relaunch. Changes in rollout length, model capability, and data mixture can shift the learner–inference balance during a run. Rocket supports *compute resource elasticity* to resize components and *runtime elasticity* to change training behavior.

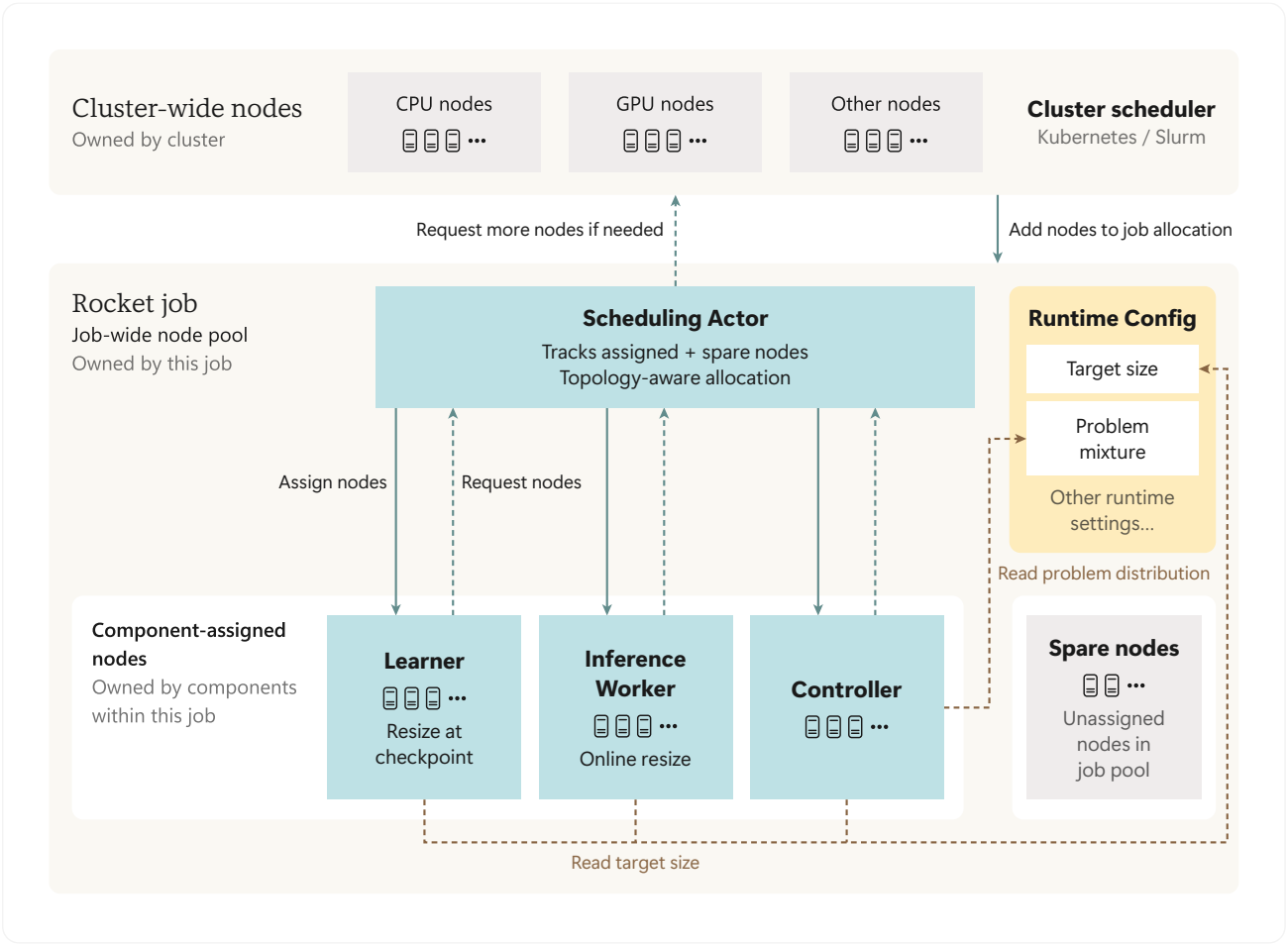


Figure 7: Two-layer scheduling separates cluster-level node allocation from Rocket's topology-aware component placement.

Compute resource elasticity

Rocket scales the learner, inference workers, and controller services independently, adding capacity at bottlenecks and releasing excess capacity elsewhere. Inference workers can be resized immediately when new configuration comes. For the learner, we wait and only resize it after the next checkpoint save in order to limit wasted compute.

Placement must respect interconnect topology. On rack-scale Blackwell clusters, enough free GPUs do not necessarily form a usable worker group under topology constraints. Figure 7 shows the two scheduling layers: the cluster scheduler allocates nodes to the job, while Rocket assigns topology-compatible groups to components and reuses released nodes within the job. Rocket tracks active allocations and spare capacity as component targets change; its scheduling actor requests additional resources when suitable capacity is insufficient.

We keep a spare group of nodes sized to replace one learner slice. If a slice fails, Rocket uses the spare group while autoscaling replenishes it. This trades reserved capacity for faster recovery and supports repeated node replacement while suitable cluster capacity remains available.

Runtime elasticity

Rocket exposes typed, validated runtime settings for inference capacity and concurrency, work admission, policy-staleness limits, batch packing, and problem-mixture weights. Figure 7 shows that the learner, inference worker, and controller read from the Rocket job's runtime config dynamically. Components apply updates at defined boundaries, such as after a training step or before a sampling operation. Runtime elasticity led to a 9.88% improvement in training compute as shown in Table 1.

Where we're headed

Rocket is not an abstract system, it is actively powering the model progress on our aggregate reasoning benchmark, a portion of which is shown in Figure 8. The figure shows the long log-linear climb that Rocket was built to sustain. The aggregate reasoning benchmark consolidates dozens of public and private evaluations into a single number. MAI's model recently passed Claude Opus 4.8 performance on this aggregate benchmark, and it is on track to overtake GPT-5.6 Sol in a few weeks.

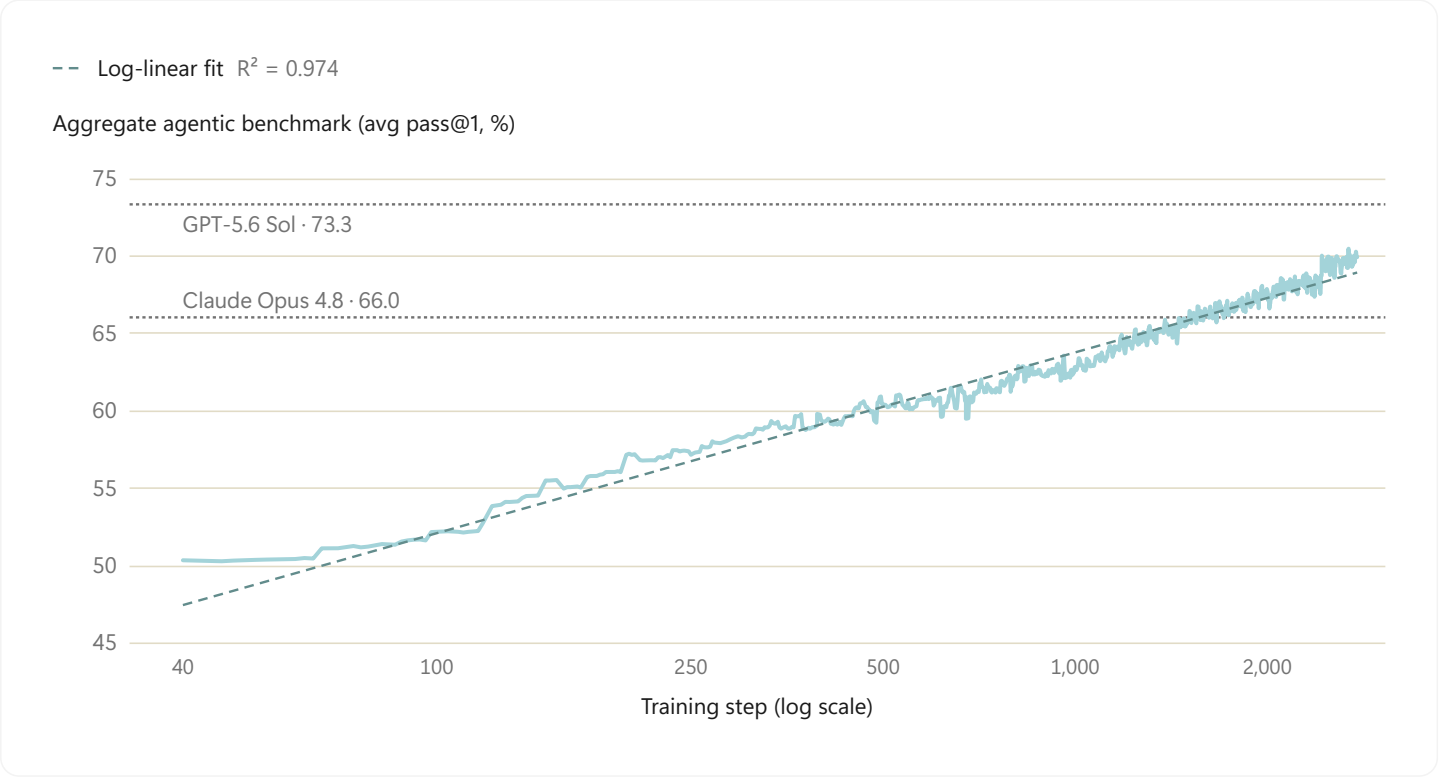


Figure 8: Rocket is one of the ingredients that enables hill-climbing model progress on our aggregate reasoning benchmark. MAI's next model recently passed Claude Opus 4.8 performance and is on track to overtake GPT-5.6 Sol soon.

The steadiness of our model progress matters as much as where it stands today. Across the run, the climb has continued without plateauing or destabilizing, which makes further progress a predictable function of RL data and compute. This gives us confidence that continuing to scale both will keep carrying our models toward the frontier. Infrastructure is one of the ingredients in this climb, and we look forward to sharing more details and model results soon.

We are continuously improving improving Rocket to support improved RL, longer rollouts, more complex agentic interactions, larger deployments, and other areas. We are particularly excited by the work to prepare Rocket for MAI's next generation of larger models as we scale towards Superintelligence. If these are problems you want to work on, we are hiring.

Citation

Please cite this work as:

> The Microsoft AI Team, "RL infrastructure at Microsoft AI," Microsoft AI, Nov 2026.

Or use the BibTeX citation:

MAI

```
@misc{mai_rl_infrastructure,
  title={RL infrastructure at Microsoft AI},
  author={{The Microsoft AI Team}},
  howpublished={Microsoft AI},
  year={2026},
  urldate={2026-11-XX},
  url={https://microsoft.ai/news/rl-infrastructure-at-microsoft-ai}
}
```