# RL Systems Mind the Gap: Matching Trainer and Generator Throughput

# 《RL 系统填补鸿沟：对齐训练器与生成器的吞吐量》

RL Training Infrastructure, GRPO, PipelineRL, Async RL, Policy Staleness, RL Sandbox Infra, CPU Requirements, TCO Analysis, Thinking Machines Tinker

RL 训练基础设施、GRPO、PipelineRL、异步 RL、策略陈旧性、RL 沙盒基础设施、CPU 需求、TCO 分析、Thinking Machines Tinker

KIMBO CHEN, CHEANG KANG WEN, AND DYLAN PATEL

陈金波、陈江文、迪伦·帕特尔

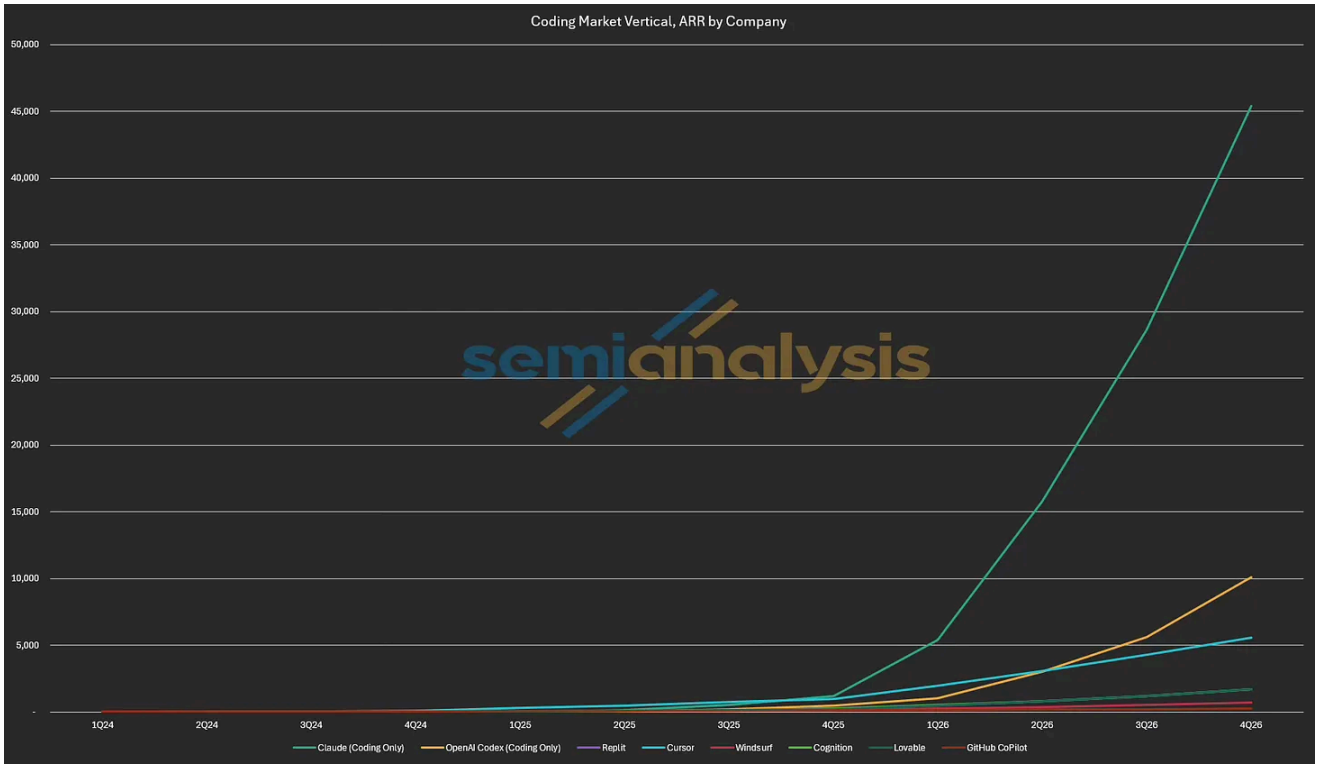
JUN 16, 2026 2026 年 6 月 16 日 · PAID · 已付费



# The Cost of Capability 能力的代价

Coding assistants are the greatest B2B SaaS application the world has ever seen: a $30B+ ARR market across the six largest players today, on track to clear $100B by year end, per our tokenomics model.

编程助手是当今全球最伟大的 B2B SaaS 应用：根据我们的代币经济学模型，六大主要玩家目前拥有超过 300 亿美元的年经常性收入（ARR），预计到年底将突破 1000 亿美元。




Source: SemiAnalysis Tokenomics Model Estimates

半分析代币经济模型估算

The agentic coding capabilities of those assistants don't come from pre-training alone. Post-training, and reinforcement learning (RL) in particular, is what elicits these capabilities from such pre-trained models. Concretely, Claude Opus 4.8 scores 69.2% on SWE-bench Pro and 74.6% on Terminal-Bench 2.1, and RL training is a major part of what drives the score.

这些编程助手的智能编码能力并非仅来自预训练。后训练阶段，特别是强化学习（RL），才是从预训练模型中激发这些能力的关键。具体而言，Claude Opus 4.8 在 SWE-bench Pro 上取得 69.2% 的得分，在 Terminal-Bench 2.1 上获得 74.6%成绩，而强化学习训练正是驱动这些高分的重要因素。

Dario Amodei, Anthropic's CEO, has described RL as showing the same kind of scaling pre-training once did, where performance climbs log-linearly with how long you train (link). However, that scaling is enormously expensive, which makes RL system efficiency critical: it sets how much RL you can afford, and with it how far

model capabilities can go.

Anthropic 首席执行官达里奥·阿莫迪将强化学习描述为展现出与预训练相似的扩展规律——模型性能会随训练时长呈对数线性增长（相关链接）。然而，这种扩展代价极其高昂，这使得强化学习系统效率至关重要：它决定了你能负担多少 RL 训练，进而决定了模型能力边界也就由此划定。

What governs the efficiency of an RL training system? In this article, we conducted RL training experiments on open models with open-source RL frameworks, and compared pricing to hosted RL training solutions such as Tinker. We show that system efficiency comes down to matching trainer and generator throughput.

什么因素决定了强化学习训练系统的效率？在本文中，我们使用开源 RL 框架对开源模型进行了训练实验，并将定价与 Tinker 等托管 RL 训练方案进行对比。研究表明，系统效率的核心在于实现训练器与生成器之间的吞吐量匹配。

# Acknowledgements 致谢

We'd like to thank the following for close collaboration:

我们要感谢以下各方提供的紧密合作：

- Prime Intellect: Matej Sirovatka, Ameen Patel, Sami Jaghouar, Johannes Hagemann. We thank their help with providing recipes, hardware resources, and article feedback

  Prime Intellect： Matej Sirovatka、 Ameen Patel、 Sami Jaghouar、 Johannes Hagemann。感谢他们提供方法指导、硬件资源以及文章反馈。

- Modal: Peyton Walters, Nan Jiang, Erik Dunteman. We also thank Modal's API credit sponsorship

  Modal： Peyton Walters、Nan Jiang、Erik Dunteman。同时感谢 Modal 提供的 API 额度赞助。

- vLLM / Inferact: Kaichao You, Ao Shen

  vLLM / Inferact： Kaichao You、Ao Shen。

- verl developers: Xibin Wu, Yuyang Ding, Yan Bai

  verl 开发者：吴西彬、丁宇阳、白岩

- slime developers slime 开发者
- Verda: Provided compute for experiments

  Verda：为实验提供了计算资源

We'd also like to thank the following for reviewing and offering feedback:

我们还要感谢以下人员参与审阅并提供反馈：

- Linden Li, Applied Compute: Gave great advice, and whose AIE talk inspired the article

  林登·李，应用计算：提供了极佳的建议，其 AIE 演讲启发了本文

- Periodic Labs: Dennis van der Staay, Byron Hsu

  Periodic Labs：丹尼斯·范德斯塔伊、拜伦·徐

- Randy Ardywibowo, Perplexity AI

  兰迪·阿迪维博沃，Perplexity AI

- λux, Non-Euclidean Pasture

  λux，非欧几里得牧场

- Simon Guo, Thinking Machines Lab

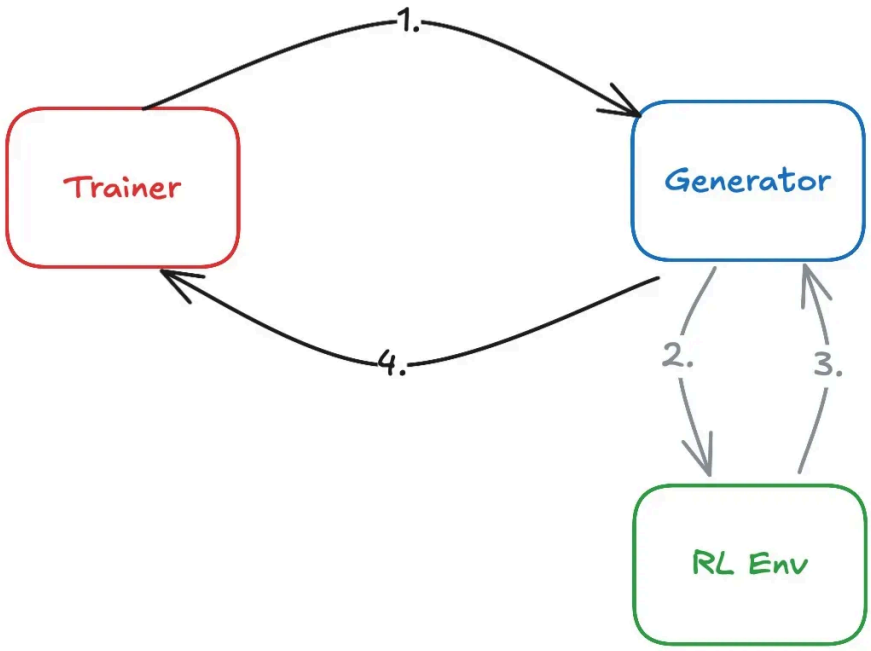
  Simon Guo，思维机器实验室

# The Three Actors 三个角色

An open-source RL training system has three actors: the **generator**, the **RL environment**, and the **trainer**. The **generator** performs inference on prompts from the dataset, generating a rollout: a prompt and a model's generated response. Unlike in pre-training, the dataset only provides prompts, not the full target. Instead, the system

generates the training signal. To generate rollouts, the generator interacts with the **RL environment**. The RL environment produces a reward based on the rollout. For example, a code environment executes the generated code in a sandbox and gives reward scores based on the test case pass rate. Then, the **trainer** ingests rollouts and rewards generated by the generator and trains the model, producing new model weights. Finally, the trainer pushes the new weights to the generator, closing the loop of an RL training step.

一个开源的强化学习训练系统包含三个角色：生成器、强化学习环境和训练器。生成器根据数据集中的提示进行推理，生成一个展开序列：即提示和模型生成的响应。与预训练不同，数据集仅提供提示，而非完整的目标。相反，系统会自行生成训练信号。为了生成展开序列，生成器与强化学习环境进行交互。强化学习环境根据展开序列产生奖励。例如，代码环境会在沙盒中执行生成的代码，并根据测试用例通过率给出奖励分数。随后，训练器接收生成器产生的展开序列和奖励，对模型进行训练，生成新的模型权重。最后，训练器将新权重推送给生成器，完成强化学习训练步骤的闭环。




Source: SemiAnalysis SemiAnalysis

# The RL Algorithm, Just Enough

# 强化学习算法，点到为止

Unlike pre-training, which maximizes the log-likelihood, RL training maximizes the expected reward. **Group-Relative Policy Optimization** (**GRPO**) is the open-source standard, and most open-weight models train with some variant of it.

与最大化对数似然的预训练不同，强化学习训练旨在最大化期望奖励。组相对策略优化（GRPO）是开源领域的标准方法，大多数开放权重模型都采用其变体进行训练。

GRPO samples multiple completions for each prompt, forming a group of **rollouts**. We assign each rollout a **reward**, a score of the rollout. We then compute each rollout's **advantage**: its reward relative to the group's average, capturing how much better or worse it did than a typical rollout for that prompt. Rollouts above the group average get reinforced, and those below get suppressed.

GRPO 为每个提示采样多个完整输出，形成一组展开轨迹。我们为每个展开轨迹分配一个奖励分数，然后计算每个轨迹的优势值：即其奖励相对于该组平均值的差值，用以衡量该轨迹相比该提示的典型表现是更好还是更差。高于组平均值的轨迹会得到强化，低于平均值的则会被抑制。

If every rollout in a group gets the same reward, each one equals the group average, so every advantage is zero and the group produces no training signal. This is governed by the group's **reward distribution**: a group only teaches the model something when its rollouts show different behaviors. The extreme case is a uniform distribution, which happens when a task is too easy (every rollout passes) or too hard (every rollout fails), i.e. when the solve rate is near 100% or near 0%.

如果组内所有展开轨迹获得相同奖励，则每个轨迹都等于组平均值，所有优势值均为零，该组便不会产生训练信号。这由组的奖励分布决定：只有当组内展开轨迹展现出不同行为时，该组才能让模型学到东西。极端情况是均匀分布，这发生在任务过于简单（所有轨迹都通过）或过于困难（所有轨迹都失败）时，即解决率接近 100%或接近 0%时。

# From Synchronous to Asynchronous RL

# 从同步到异步强化学习

Classic policy gradient algorithms (which GRPO is based upon) assume the rollouts in a group come from the same policy, i.e., the model with the same set of weights. In the context of the training system, this means the generator cannot update weights until it finishes the current batch of rollouts. As a result, if the trainer finishes a training step, it'll have to wait for the generator. This leads to synchronous execution of training and generation, greatly reducing system efficiency.

经典的策略梯度算法（GRPO 即基于此）假设同一组中的轨迹来自同一策略，即具有相同权重集的模型。在训练系统的语境中，这意味着生成器在完成当前批次的轨迹生成之前无法更新权重。因此，如果训练器完成了一个训练步骤，它必须等待生成器。这导致训练与生成以同步方式执行，极大地降低了系统效率。

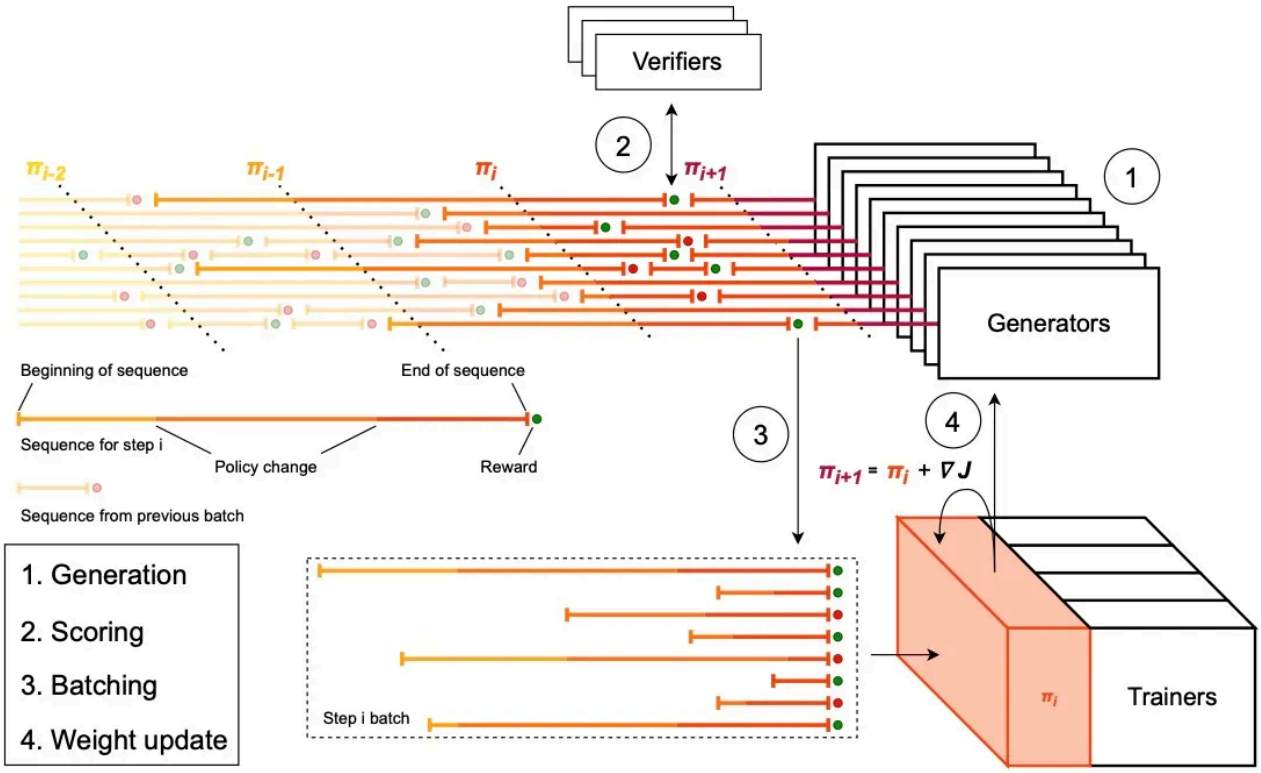
PipelineRL introduces asynchrony into the system by allowing the trainer to push new weights to the generator while rollouts are still in progress (in-flight weight updates). This way, the trainer can overlap its execution with the generator's rollout. However, this comes with a cost. A sample, the unit a trainer consumes, is defined as a rollout and the reward the RL environment assigned to it. With in-flight weight updates, each sample is generated by a mixture of old and new policies. We refer to the phenomenon as policy staleness.

PipelineRL 通过允许训练器在轨迹生成仍在进行时（即飞行中权重更新）将新权重推送给生成器，从而在系统中引入了异步机制。这样，训练器就能与生成器的轨迹生成过程重叠执行。然而，这需要付出代价。训练器所消耗的样本单元，定义为一条轨迹及其对应的强化学习环境赋予的奖励。在飞行中权重更新的情况下，每个样本都由新旧策略混合生成。我们将这种现象称为策略陈旧性。

PipelineRL shows that RL algorithms can tolerate policy staleness to an extent. Samples too stale degrade model learning. Synchronous execution wastes too much compute to be practical at scale, and async techniques are essential. In effect, PipelineRL is a throughput-matching scheme with bounded policy staleness. It allows the trainer and generator to run at different speeds, capped by how stale samples can get. This is why PipelineRL has become the de facto implementation in open-source

RL training.

PipelineRL 表明，RL 算法可以在一定程度上容忍策略陈旧性。过于陈旧的样本会损害模型学习效果。同步执行会浪费过多算力，难以在大规模场景中实用，因此异步技术至关重要。实际上，PipelineRL 是一种具有策略陈旧性上限的吞吐量匹配方案。它允许训练器和生成器以不同速度运行，仅受样本可达到的最长陈旧时间限制。这正是 PipelineRL 成为开源 RL 训练事实标准实现的原因。




Source: Magistral, Mistral AI

Magistral, Mistral AI

# The RL Environment and Sandbox

# 强化学习环境与沙箱

The RL environment provides feedback to the model for the model to learn how to do a task. Practically, the RL environment is implemented as a sandbox, a containerized runtime for code execution. Depending on the complexity of the RL environment, it could range from lightweight Firecracker micro-VMs (micro-Virtual Machines) to full-

blown QEMU VMs.

强化学习环境为模型提供反馈，使其学会如何执行任务。实际上，强化学习环境是通过沙箱实现的——这是一种用于代码执行的容器化运行时。根据强化学习环境的复杂程度，其实现方式可以从轻量级的 Firecracker 微虚拟机到完整的 QEMU 虚拟机不等。

Sandboxes for RL environments have unique system challenges. The interaction latency between the generator and the RL environment is critical to the end-to-end rollout latency, and sandbox startup latency is one of the major overheads. Sandbox service companies like Modal optimize the startup latency with techniques like content-addressed caching.

强化学习环境的沙箱面临独特的系统挑战。生成器与强化学习环境之间的交互延迟对端到端的部署延迟至关重要，而沙箱启动延迟是主要开销之一。像 Modal 这样的沙箱服务公司通过内容寻址缓存等技术来优化启动延迟。

Serving at scale is also challenging: sandboxes scale along with the number of concurrent rollouts, and the number of live sandboxes fluctuates greatly as rollouts are pruned or completed.

大规模服务同样具有挑战性：沙盒需随并发推演数量同步扩展，而随着推演被裁剪或完成，活跃沙盒的数量会剧烈波动。

Finally, sandboxes need to be robust against system failures. During learning, a model's unexpected behaviors may deplete a sandbox's designated resources. For instance, creating a million files can cause the sandbox to run out of memory. Sandbox orchestration needs to be able to detect and recover from those system failures.

最后，沙盒必须能够抵御系统故障。在训练过程中，模型意外行为可能耗尽沙盒的指定资源。例如，创建数百万个文件可能导致沙盒内存耗尽。沙盒编排系统需要能够检测并恢复此类系统故障。

# The Framework Throughput-Matching

# 吞吐量匹配框架

# Quantifying Trainer and Generator Throughput

## 量化训练器与生成器的吞吐量

We view the RL training system as a queue: the generator produces rollouts into a queue, and the trainer consumes from it. When the generator is slower than the trainer, the queue empties and the trainer starves, idling between steps. When the generator is faster, the queue grows and its samples age, causing policy staleness issues.

我们将强化学习训练系统视为一个队列：生成器将轨迹数据放入队列，训练器从中消费。当生成器速度慢于训练器时，队列会变空，训练器因等待而闲置，在训练步骤之间出现空闲。当生成器速度更快时，队列会增长，其中的样本逐渐过时，导致策略陈旧问题。

We model the system efficiency through the lens of matching the **producer** (**generator**) **and consumer** (**trainer**) **throughputs**. In an ideal RL training system, the trainer consumption rate should be roughly equal to the generator production rate. The trainer consumes samples, performs a training step, and then broadcasts weights to the generator. We measure the training step time, including the idle time waiting for new samples, the compute time, and the weight broadcast time. We further derive the sample throughput (samples / compute time) to represent the trainer consumption rate. We also calculate FLOP/s per GPU and model FLOPs utilization (MFU) to characterize hardware efficiency.

我们通过匹配生产者（生成器）和消费者（训练器）吞吐量的视角来建模系统效率。在理想的强化学习训练系统中，训练器的消费速率应大致等于生成器的生产速率。训练器消费样本、执行训练步骤，然后将权重广播给生成器。我们测量训练步骤的时间，包括等待新样本的空闲时间、计算时间和权重广播时间。进一步推导出样本吞吐量（样本数/计算时间）以表示训练器的消费速率。同时，我们计算每 GPU 的 FLOP/s 和模型 FLOPs 利用率（MFU）来表征硬件效率。

The generator produces samples through LLM inference and RL environment interactions. Since the **straggler**, the longest rollout in a group, sets the group's completion time, we measure end-to-end latency to estimate the average sample throughput. We can derive the sample throughput (samples / end-to-end latency) to represent the generator production rate.

生成器通过 LLM 推理和 RL 环境交互来生成样本。由于组内最慢的节点（即耗时最长的 rollout）决定了整个组的完成时间，我们通过端到端延迟来估算平均样本吞吐量。由此可推导出样本吞吐量（样本数/端到端延迟），用以表征生成器的产出速率。

We further break down the latency into LLM inference time and RL environment interaction time. RL environment interaction involves sandbox startup latency and sandbox execution time.

我们进一步将延迟拆解为 LLM 推理时间和 RL 环境交互时间。其中 RL 环境交互包含沙箱启动延迟和沙箱执行时间。

## Throughput Constraints 吞吐量约束

Multiple factors constrain the upper and lower bound of the throughput. The trainer consumption rate is **samples per step / training step time**.

吞吐量的上下限受多重因素制约。训练器的消耗速率为每步样本数/训练步耗时。

Factors that affect the number of samples include:

影响样本数量的因素包括：

- **Group size**: N samples per prompt, N is usually 8 or 16. Depending on the task difficulty, we increase N to strengthen the advantage signal, e.g., setting N=64 for GPU kernel writing tasks. Larger groups increase the samples per step

  组大小：每个提示生成 N 个样本，N 通常为 8 或 16。根据任务难度，我们会增加 N 以强化优势信号，例如在 GPU 内核编写任务中设置 N=64。更大的组会增加每步的样本量。

- **Reward distribution and advantage filtering**: Reward distribution of a rollout group affects the sample advantage, and it reduces the samples per step under

different advantage filtering strategies. For example, task difficulty: Tasks being too easy or too hard leads to uniform reward distribution (All zero or all full rewards), causing zero advantage. If we drop zero-advantage samples, it reduces the samples per step.

奖励分布与优势过滤：一组样本的奖励分布会影响样本优势，并在不同优势过滤策略下减少每步的样本量。例如，任务难度：任务过于简单或困难会导致奖励分布均匀（全零或全满分），从而产生零优势。如果丢弃零优势样本，则会减少每步的样本量。

- **Batch size**: There's a minimum effective batch size that enables stable learning. We typically deploy the trainer on sufficient GPU nodes to hold those batches

批次大小：存在一个最小有效批次大小，能够支持稳定的学习过程。我们通常会在足够的 GPU 节点上部署训练器，以容纳这些批次。

Factors that affect the training step time include:

影响训练步骤时间的因素包括：

- **Model**: Model architecture, including model size, activation size, model precision, dictates the memory usage, setting a minimum GPU count to perform forward and backward pass without running out of memory. It also determines what operations the GPU executes, affecting the training step time

模型：模型架构（包括模型规模、激活值大小、模型精度）决定了内存使用量，从而设定了执行前向和反向传播所需的最低 GPU 数量，以避免内存不足。同时，它也决定了 GPU 执行的具体运算，进而影响训练步骤的时间。

- **Parallelism and memory configuration**s: Tensor/Pipeline/Data/Expert Parallelism, Fully-Sharded Data Parallelism (FSDP), memory offloading, activation checkpointing. They are affected by the GPU count and determine the system efficiency and the training step time

并行化与内存配置：张量并行、流水线并行、数据并行、专家并行、全分片数据并行（FSDP）、内存卸载、激活检查点。这些配置受 GPU 数量影响，并决定了系统效率及训练步骤的时间。

On the generator side, we define the generator production rate as **the number of concurrent rollouts / end-to-end latency**.

在生成器端，我们将生成器产出率定义为并发运行次数/端到端延迟。

Factors that affect end-to-end latency include:

影响端到端延迟的因素包括：

- **Inference throughput**: Recipe tuning for the inference engine, including parallelism strategies, KV cache quantization, PD disaggregation, and speculative decoding

  推理吞吐量：针对推理引擎的配方调优，包括并行策略、KV 缓存量化、PD 分离以及推测解码

- **Sandbox latency**: Sandbox startup time and execution time

  沙盒延迟：沙盒启动时间和执行时间

- **Reward modeling type**: Math and coding tasks may have lightweight verifiers as reward models, while writing tasks may use LLM judges. Reward model types determine the rollout evaluation latency: lightweight verifiers are fast, and LLM judges are comparatively slow

  奖励模型类型：数学和编程任务可能使用轻量级验证器作为奖励模型，而写作任务可能采用 LLM 评判器。奖励模型类型决定了推理评估的延迟：轻量级验证器速度较快，而 LLM 评判器相对较慢。

- **Reward shape**: Process Reward Models (PRMs) may evaluate the rollout per turn, whereas some reward models evaluate only the full rollout. This determines the reward assignment latency

  奖励形状：过程奖励模型（PRM）可能按轮次评估生成结果，而某些奖励模型仅评估完整生成结果。这决定了奖励分配的延迟。

Concurrency, or the number of concurrent rollouts, is limited by KV cache memory space and average sequence length. The aggregate memory capacity of generation nodes - model weights and activations are the space for KV cache. KV cache dictates

the maximum number of tokens the generator can hold, and dividing it by the average sequence length gives us an estimate of max concurrency.

并发度（即同时进行的 rollout 数量）受限于 KV 缓存内存空间和平均序列长度。生成节点的总内存容量（模型权重与激活值）构成了 KV 缓存的存储空间。KV 缓存决定了生成器能容纳的最大 token 数量，将其除以平均序列长度即可估算出最大并发度。

Not all generated samples are consumed by the trainer due to training signal quality differences. We further define the effective generator production rate as acceptance rate × generator production rate. Factors that affect the acceptance rate include:

并非所有生成的样本都能被训练器采纳，这是由于训练信号质量的差异所致。我们进一步将有效生成器产出率定义为：接受率 × 生成器产出率。影响接受率的因素包括：

- Early pruning: We prune rollouts before completion based on heuristic length caps, value functions, or intermediate-result verifier checks. This dynamically reduces the number of concurrent rollouts

  早期剪枝：我们根据启发式的长度上限、价值函数或中间结果的验证器检查，在完成前对生成结果进行剪枝。这能动态减少并发生成数量

- **Adaptive sampling**: We reject rollouts based on criteria, such as online advantage filtering

  自适应采样：我们根据标准（例如在线优势过滤）来拒绝生成结果

# Policy Staleness 策略陈旧性

Staleness is the gap between the policy version that produced the sample and the one that the trainer uses when it applies the gradient. Staleness is a byproduct of async training: the trainer pushes weight updates in-flight while rollouts are being generated.

陈旧度指的是生成样本所用的策略版本与训练器在应用梯度时使用的策略版本之间的差距。这种陈旧度是异步训练的副产品：在生成器进行 rollout 的同时，训练器会实时推送权重更新。

Staleness happens at different granularities. At the trajectory level, staleness is the gap between the policy that the rollout starts generating with and the newer policy the trainer updates, e.g. the trainer is at version t+k while the trajectory started from version t. Token-level staleness occurs when in-flight weight updates happen mid-rollout, so different policy versions generate different parts of a rollout. Policy staleness also happens at an environment state-level, which we explain more in the Partial Rollout and Stateful Sandbox Design section.

陈旧度存在于不同粒度层面。在轨迹层面，陈旧度体现为 rollout 开始生成时所用的策略与训练器更新的较新策略之间的差异，例如训练器已更新至 t+k 版本，而轨迹仍从 t 版本开始生成。当 rollout 过程中发生实时权重更新时，会出现词元级陈旧度，即同一 rollout 的不同部分由不同策略版本生成。此外，策略陈旧度还存在于环境状态层面，我们将在"部分 rollout 与带状态沙箱环境"章节中对此进行详细说明。

Policy staleness means the trainer trains the model on off-policy signals, which can destabilize training, so we typically set a policy staleness budget. The policy staleness budget bounds how stale a sample can be, i.e. how far the generator is allowed to run ahead of the trainer before its samples are rejected. This in turn caps the gap between the generator's production rate and the trainer's consumption rate, which is what lets the two run at different speeds.

策略陈旧性意味着训练器基于离策略信号训练模型，这可能导致训练不稳定，因此我们通常会设定一个策略陈旧性预算。该预算限制了样本可被允许的陈旧程度，即生成器在样本被拒绝前可领先训练器运行的距离。这反过来又约束了生成器产出速率与训练器消耗速率之间的差距，从而使两者能以不同速度运行。

The policy staleness budget limits the max difference between the trainer consumption rate and the effective generator production rate. Concretely, it limits how many steps the generator can be ahead of the trainer, so the RL algorithm can tolerate the max staleness in samples.

策略陈旧性预算限制了训练器消耗速率与有效生成器生产速率之间的最大差值。具体而言，它限制了生成器领先于训练器的步数上限，从而使强化学习算法能够容忍样本的最大陈旧度。

# The Moving Target: Model Capability and Behavior

## 移动目标：模型能力与行为

Model capability and behavior are first-class system variables in RL training. Model capability determines how well the model can solve a task, measured by **solve rate**: what percentage of the rollouts in a group passes the verification. When the solve rate is near 0 or near 100%, the reward distribution is uniform, leading to zero advantage, and collapses the training signal. This drives decisions including **group size** and **advantage filtering strategies**. It also shapes the **curriculum**: the order of tasks being presented to the model. The curriculum is chosen to keep the solve rate at a productive middle band, so the tasks are neither too easy nor too hard for the model's current capabilities.

在强化学习训练中，模型能力与行为是系统的一等变量。模型能力决定了模型解决任务的水平，通过解决率来衡量：即一组生成结果中通过验证的百分比。当解决率接近 0%或 100%时，奖励分布趋于均匀，导致优势值为零，从而削弱训练信号。这一特性影响着分组大小和优势过滤策略的决策，同时也塑造了课程设计——即模型接收任务的顺序。课程设计的目标是将解决率维持在富有成效的中间区间，确保任务对模型当前能力而言既不过于简单，也不过于困难。

A model's token usage behavior affects the output length, which indirectly determines the max concurrent rollouts in the generator. RL training typically elicits Chain-of-Thought reasoning, causing a model to generate long reasoning traces. This behavior drives the average output length, which increases KV cache usage. Under the same memory budget, this reduces the max concurrent rollouts and increases the sample generation end-to-end latency.

模型的令牌使用行为会影响输出长度，从而间接决定生成器中的最大并发推理数。强化学习训练通常会激发思维链推理，导致模型生成较长的推理轨迹。这种行为会推高平均输出长度，增加 KV 缓存的使用量。在相同内存预算下，这会减少最大并发推理数，并增加样本生成的端到端延迟。

A model's tool call behavior affects the sandbox time in the end-to-end latency. Concretely, more tool calls increase the sandbox time, and unexpected tool call operations strain the sandbox infra. We measure the tool call behavior with the

number of tool calls and the sandbox error rate.

模型的工具调用行为会影响端到端延迟中的沙箱时间。具体而言，工具调用次数越多，沙箱时间就越长；意外的工具调用操作会给沙箱基础设施带来压力。我们通过工具调用次数和沙箱错误率来衡量工具调用行为。

Both model capability and behavior shift during RL training. Model capability improves, and behavior drifts. The constraints move with them, and the training system has to adapt to ensure efficiency.

在强化学习训练过程中，模型的能力和行为都会发生变化。模型能力提升的同时，行为也会发生偏移。约束条件随之改变，训练系统必须适应这些变化以确保效率。

# Case Studies 案例分析

We ran experiments to ground our theory, and we shared our experiences with open-source RL frameworks.

我们通过实验来验证理论，并分享了我们在开源 RL 框架中的实践经验。

We select the following configurations:

我们选择了以下配置：

- Model: We chose Mixture-of-Experts (MoE) models since SoTA open-source models are mostly MoEs

  模型：我们选择混合专家（MoE）模型，因为当前最先进的开源模型大多采用 MoE 架构。

- RL environment and tasks: We chose agentic coding tasks for its popularity and its sandbox workload complexity, allowing us to see sandbox performance characteristics

  RL 环境与任务与 RL 环境：我们选择了智能体编程任务，因其广受欢迎且沙箱工作负载复杂，便于观察沙箱性能特征。

- Node count: We aim for experiments with at least 10 nodes to show the trainer-generator dynamics and fit training MoEs

  节点数量：我们旨在进行至少包含 10 个节点的实验，以展示训练器-生成动态效应并适配训练 MoE 模型

# Long Model Response and Early Exploration Training Phase

# 长模型响应与早期探索训练阶段

## Setup 设置

- Model: Qwen3-235B-A22B-Thinking-2507. We use BF16 precision for training and the FP8 variant for sample generation, a common setup in RL training

  模型：Qwen3-235B-A22B-Thinking-2507。训练采用 BF16 精度，样本生成使用 FP8 变体，这是强化学习训练中的常见配置。

- Trainer: 64 H200 GPUs, FSDP, EP8, CPU-offloaded optimizer states and activations

  训练器：64 块 H200 GPU，FSDP，EP8，CPU 卸载优化器状态和激活

- Generator: 192 GPUs, 24 inference instances: Each DP1, TP8, EP8

  生成器：192 块 GPU，24 个推理实例：每个实例 DP1、TP8、EP8

- Batch Size: 512 samples 批次大小：512 个样本
  - Group Size: 16 rollouts per problem

    组大小：每个问题 16 次生成

  - Problems per batch: 32 problems

    每批问题数：32 个

- Max sequence length: 32K 最大序列长度：32K

- Maximum policy staleness: 16 steps

  最大策略过时步数：16 步

- RL environment: Mini SWE Agent Plus, with Prime Intellect Sandboxes

  强化学习环境：Mini SWE Agent Plus，搭配 Prime Intellect 沙盒环境

- RL framework: Prime RL 强化学习框架：Prime RL
- Code: Recipe coming soon in repo's main branch. Original commit here

  代码：配方即将在仓库主分支中发布。原始提交在此

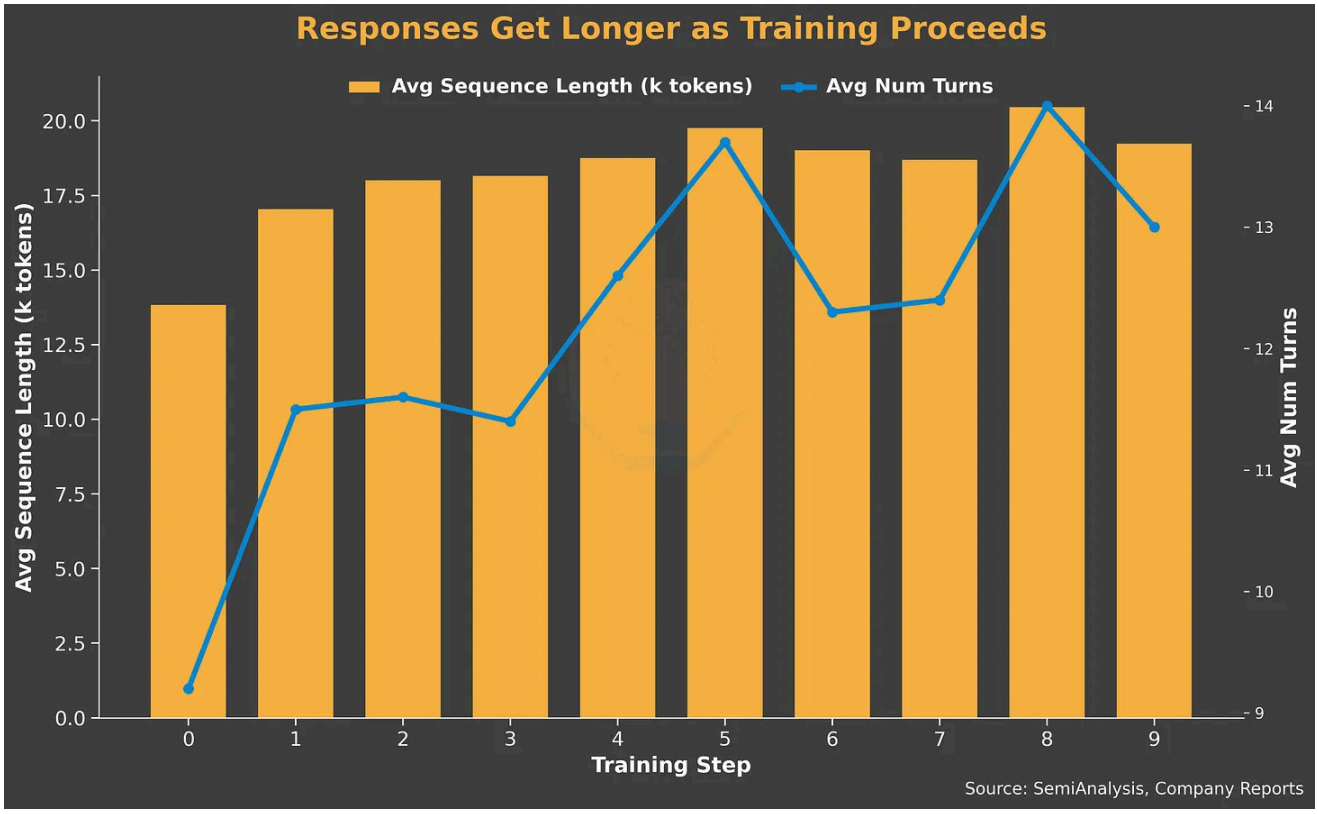
## Analysis 分析

In this case study, we show how two factors influence the system efficiency: how the model produces long responses with long thinking traces, and how well the model learns the task.

在本案例研究中，我们展示了两个因素如何影响系统效率：模型如何通过长思考轨迹生成长响应，以及模型学习任务的效果如何。

The model's long response behavior is the dominant driver of LLM inference time, which makes up the majority of sample generation time. Long-response variance also drives severe tail latency within each rollout group, which forces a system-side mitigation: oversampling. With oversampling the generator launches more concurrent rollouts than the trainer needs per batch and it discards unfinished or errored ones as soon as the target count is reached. In this run we discard a relatively high 60% of dispatched rollouts, reflecting how skewed the latency distribution becomes when the responses are long.

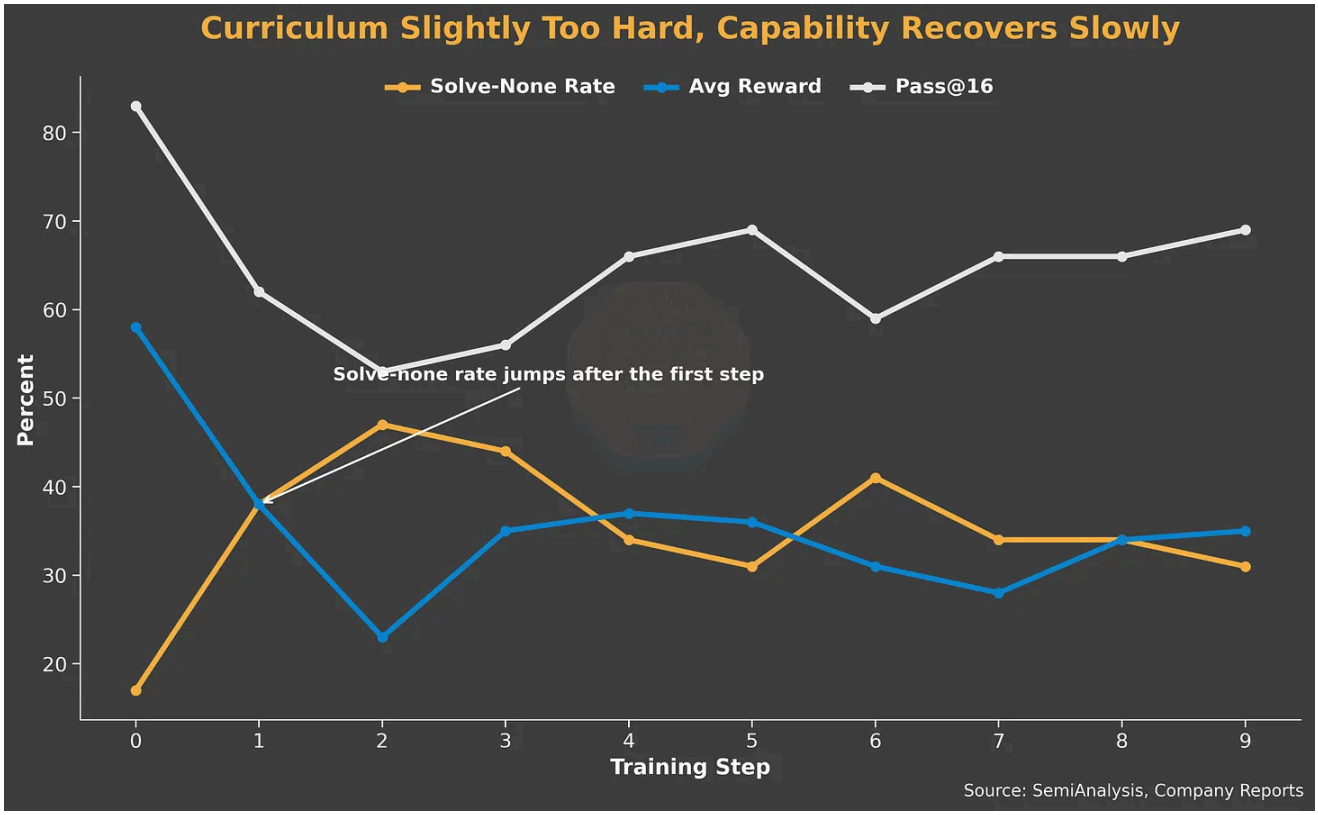
模型的长响应行为是 LLM 推理时间的主要驱动因素，而推理时间占据了样本生成时间的大部分。长响应方差还会导致每个展开组内出现严重的尾部延迟，这迫使系统采取一种缓解措施：过采样。通过过采样，生成器会启动比训练器每批次所需更多的并发展开，并在达到目标数量后立即丢弃未完成或出错的展开。在此次运行中，我们丢弃了相对较高的 60%已调度展开，这反映了当响应较长时延迟分布变得多么偏斜。




Source: SemiAnalysis SemiAnalysis

On the model capability side, the curriculum is slightly too hard for the base model. We observe the number of rollout groups that produce zero solutions jump sharply after the first training step, but average reward and pass@16 are slowly recovering. These are signs that the early RL exploration pushed the policy away from previously solvable problems before it could relearn them.

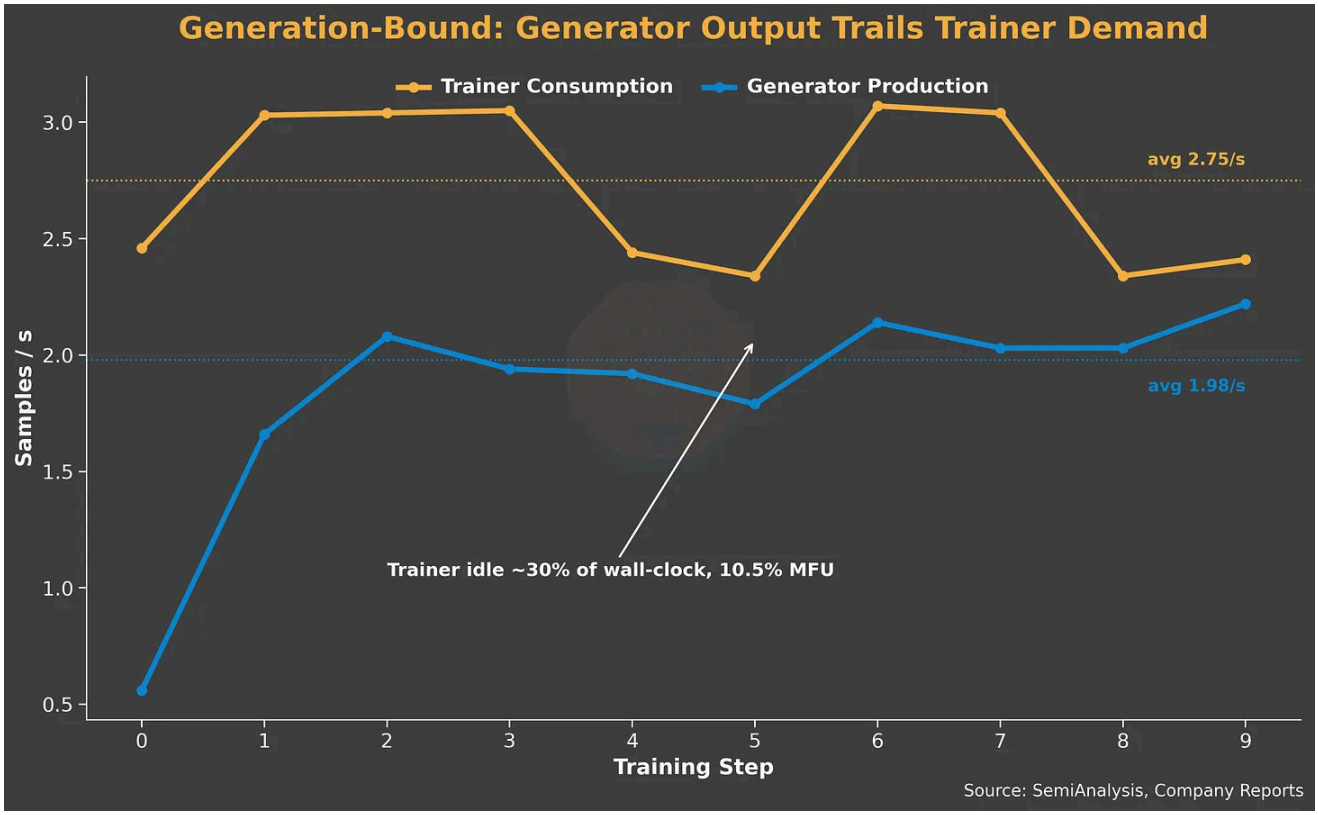
在模型能力方面，当前课程对基础模型而言难度略高。我们观察到，第一个训练步骤后，产生零解的回滚组数量急剧上升，但平均奖励和 pass@16 指标正在缓慢回升。这些迹象表明，在模型重新掌握解题能力之前，早期强化学习探索已使策略偏离了之前能解决的问题领域。



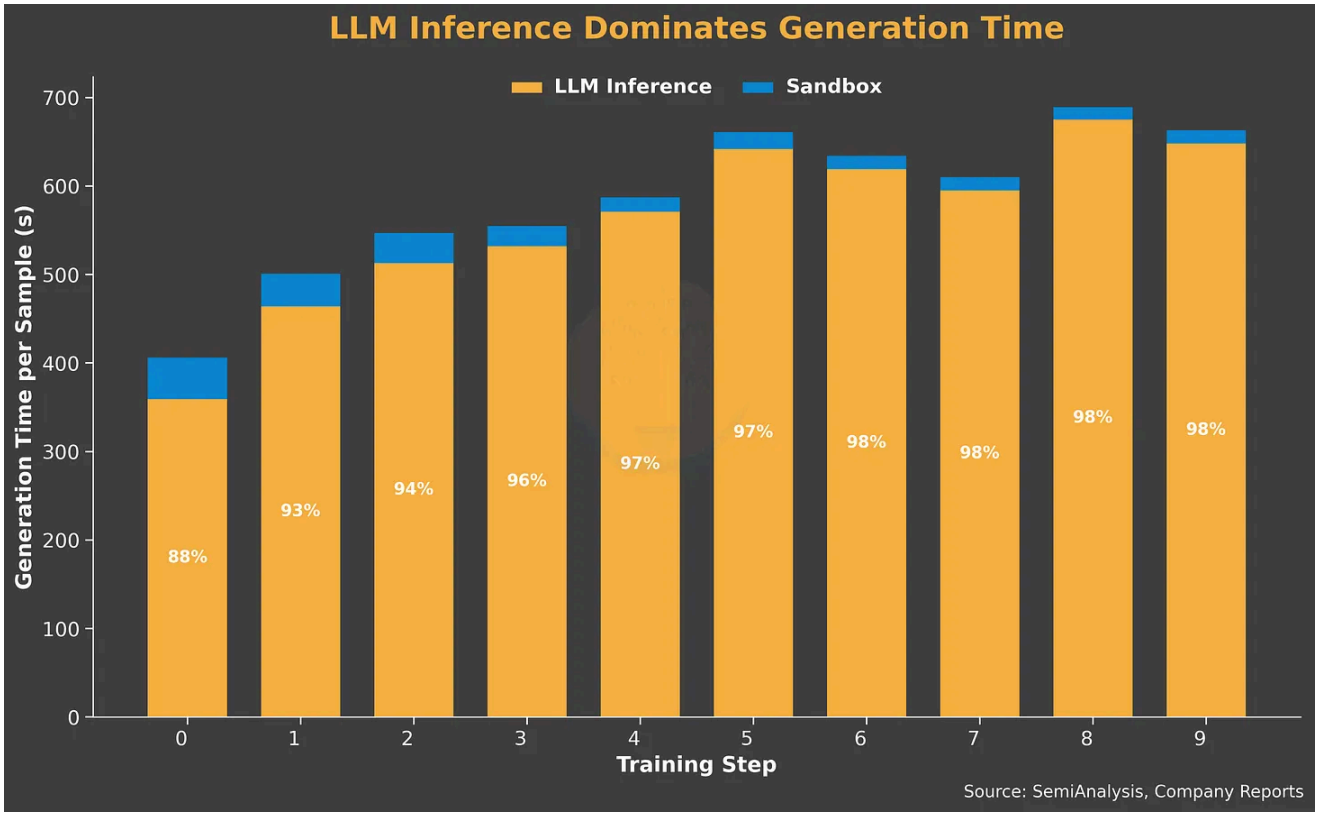

Source: SemiAnalysis SemiAnalysis

The combined effect is a **generation-bound system**: the queue runs dry and the trainer starves. To mitigate this issue, we trade off trainer efficiency for generator efficiency. The trainer consumes samples at 2.75 sample/s, waits 30% of the wall-clock time, and runs at 10.5% MFU. On the other hand, the generator delivers 1.95 sample/s and uses 3x the trainer's compute. This underscores how much inference efficiency matters during RL training.

最终导致系统被生成端所制约：队列耗尽，训练器陷入停滞。为缓解这一问题，我们牺牲训练器效率来换取生成器效率。训练器以 2.75 样本/秒的速度消耗样本，等待时间占实际运行时间的 30%，模型利用率（MFU）为 10.5%。而生成器则以 1.95 样本/秒的速度产出样本，消耗的计算资源是训练器的 3 倍于训练器的。这充分说明了推理效率在强化学习训练中的重要性。




Source: SemiAnalysis SemiAnalysis




Source: SemiAnalysis SemiAnalysis

# Frequent Tool Use and Easy Task

# 频繁使用工具与简单任务

## Setup 设置

- Model: GLM-5. We use BF16 precision for training and the FP8 variant for sample generation

  模型：采用 GLM-5 架构，训练使用 BF16 精度，样本生成采用 FP8 变体

- Trainer: 128 H200 GPUs, FSDP, CP2, EP8, CPU-offloaded optimizer states

  训练器：128 块 H200 GPU，FSDP 分布式策略，CP2 通信协议，EP8 模型并行，优化器状态 CPU 卸载

- Generator: 128 H200 GPUs, 2x 64-GPU inference replicas

  生成器：128 块 H200 GPU，配备两个 64GPU 推理副本

  - **PD Disaggregation:** 32 prefill GPUs, 32 decode GPUs, both DP32 and EP32

    PD 分离：32 块预填充 GPU、32 块解码 GPU，均采用 DP32 和 EP32 配置

  - 1TB KV cache offload per node

    每节点 1TB KV 缓存卸载

- Batch Size: 512 samples 批次大小：512 个样本
- Group Size: 16 rollouts per problem

  组大小：每个问题 16 次生成

- Problems per batch: 32 problems

  每批问题数：32 个

- Max sequence length: 64K 最大序列长度：64K
- Maximum policy staleness: 16 steps

  最大策略过时步数：16 步

- RL environment: Mini SWE Agent Plus, with Prime Intellect Sandboxes

  强化学习环境：Mini SWE Agent Plus，搭配 Prime Intellect 沙盒环境

- RL framework: Prime RL 强化学习框架：Prime RL
- Code: Recipe coming soon in repo's main branch. Original commit here

  代码：配方即将在仓库主分支中发布。原始提交在此

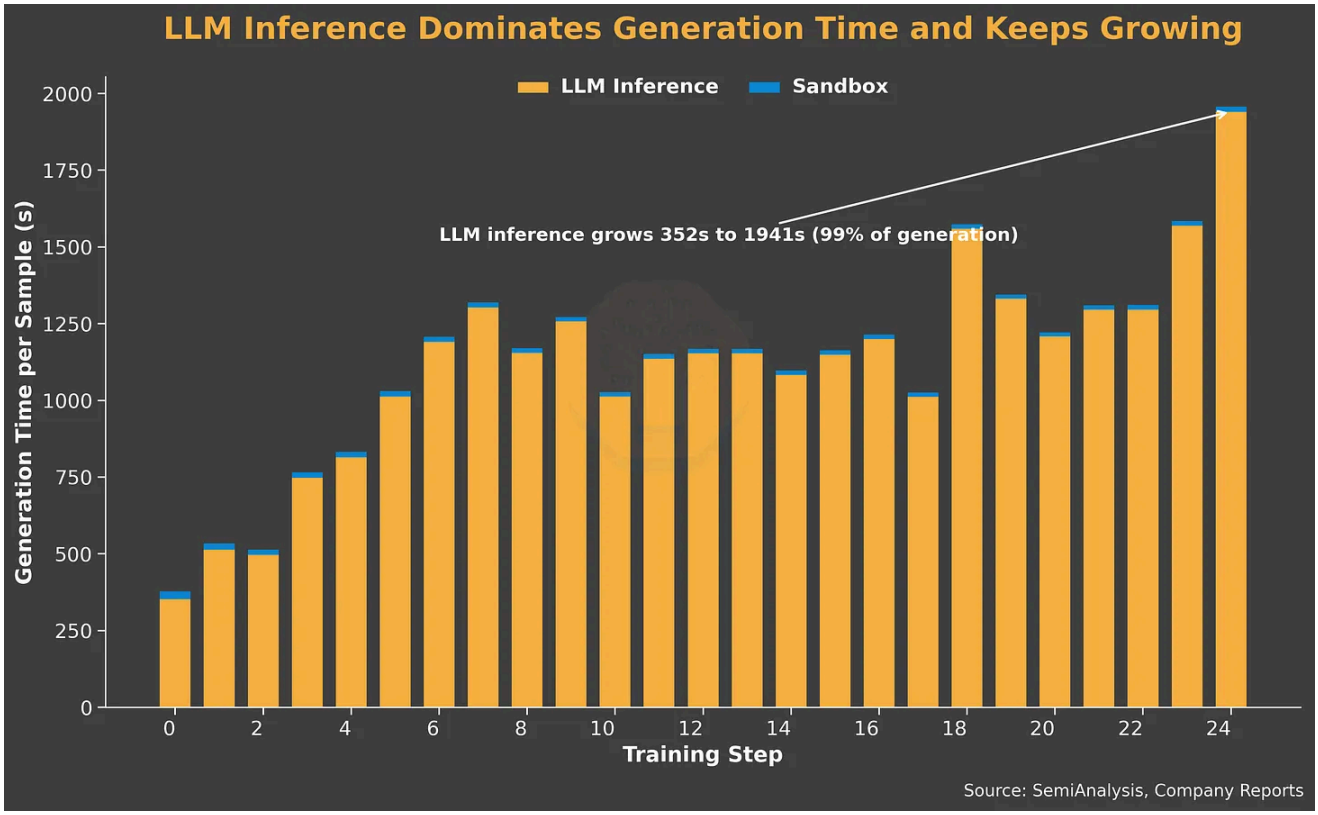
## Analysis 分析

In this run, we see two factors set the system efficiency: model behavior drift and the curriculum being too easy for the model.

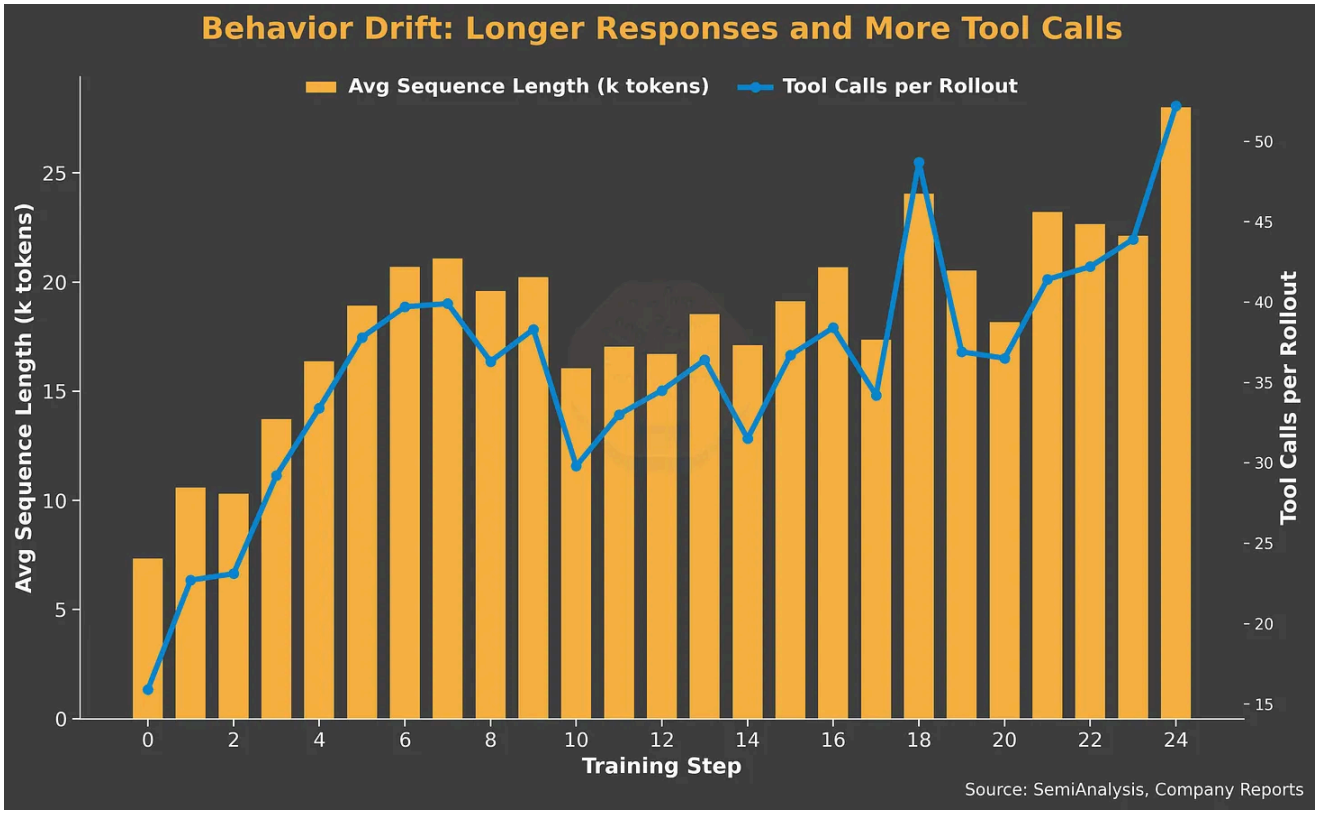
在这次运行中，我们看到两个因素决定了系统效率：模型行为漂移以及课程难度对模型而言过于简单。

We observe both the average response length per turn and the number of tool calls (20 to 51) triple over the run. Together they push sequence lengths up and shift the workload toward a prefill-heavy profile. This supports our choice of disaggregated serving, which improves the worst-case time-to-first-token (TTFT) for prefill requests.

我们观察到，每轮对话的平均回复长度和工具调用次数（从 20 次到 51 次）在运行过程中均增长了三倍。这两者共同推高了序列长度，并使工作负载转向以预填充为主的模式。这印证了我们采用分离式服务的决策，该方案有效缩短了预填充请求的最差情况首令牌生成时间（TTFT）。




Source: SemiAnalysis SemiAnalysis

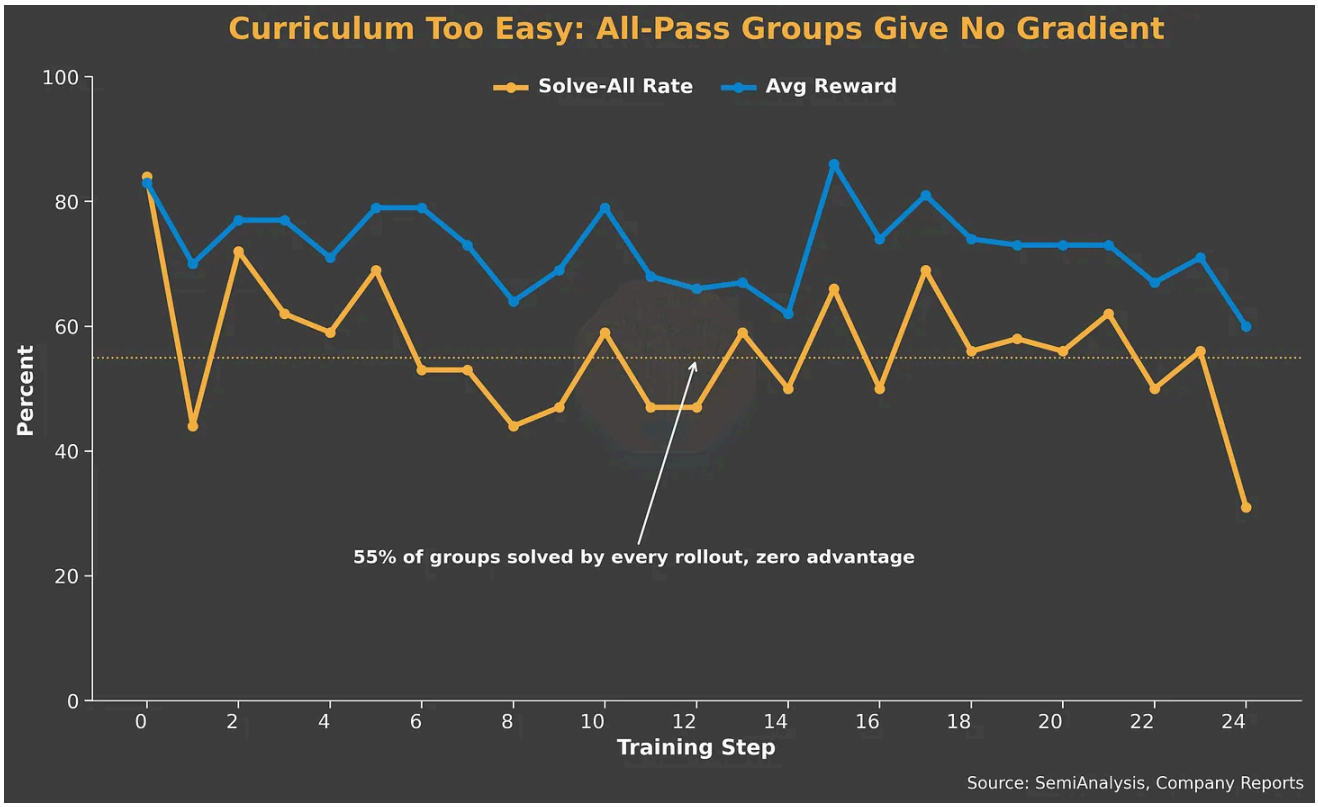



Source: SemiAnalysis SemiAnalysis

On the capability side, the curriculum is too easy for the model. On average, 55% of problems have a 100% solve rate, i.e., every rollout in the group passes. A rollout group where every rollout has the same reward produces zero advantage and contributes no

training signal. As a result, average reward stays flat. The curriculum mismatch reduces the effective training samples and subsequently the effective generator production rate.

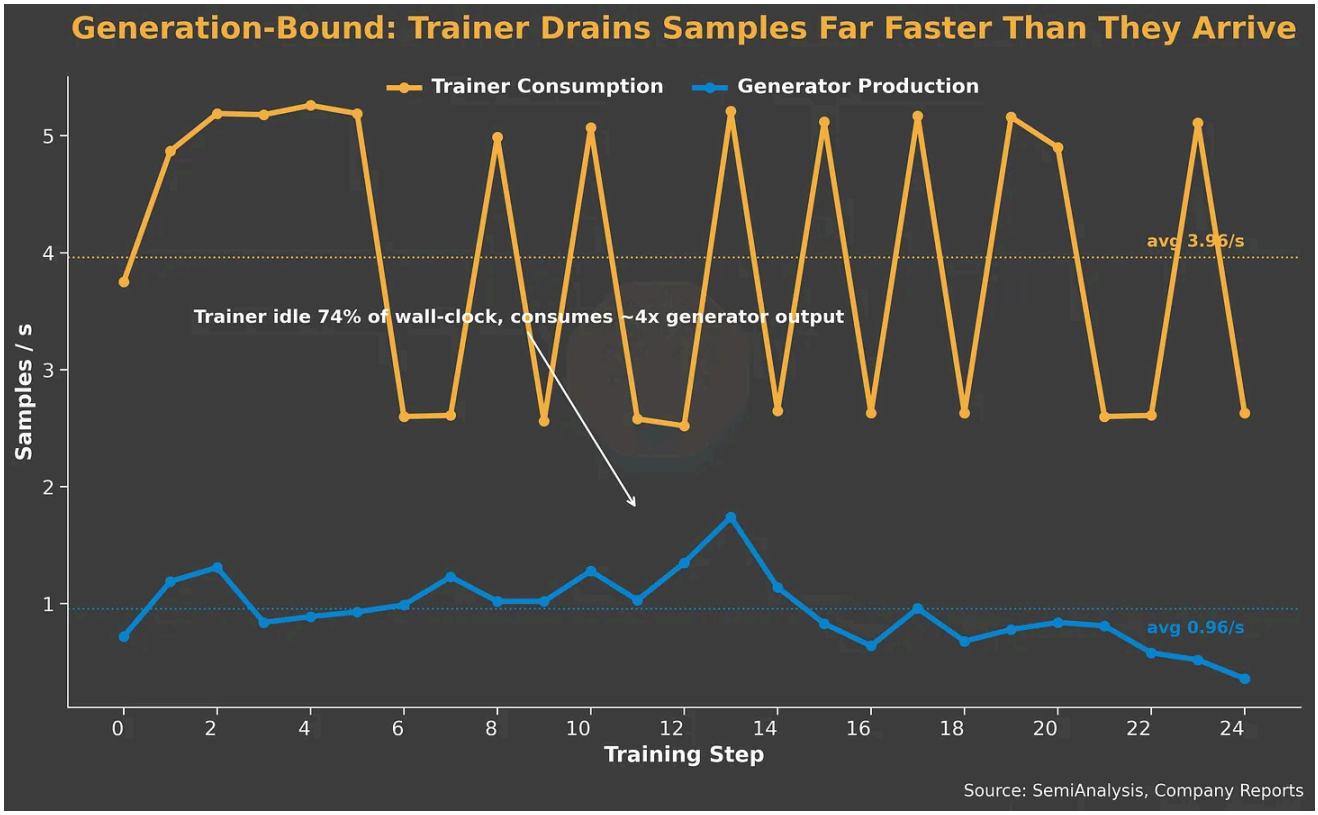
在能力方面，课程对模型而言过于简单。平均有 55%的问题达到 100%的解决率，即组内每次生成都通过。当生成组中所有生成结果获得相同奖励时，优势值为零，无法提供训练信号。这导致平均奖励保持平稳。课程不匹配降低了有效训练样本数量，进而降低了生成器的有效产出率。




Source: SemiAnalysis SemiAnalysis

All the effects result in a heavily generation-bound system. The queue fills up fast, but a large portion is filtered out, so the output is low. LLM inference dominates end-to-end generation time and keeps growing as training progresses. The trainer spends 74% of the wall-clock time waiting, and its consumption rate is 5× the generator's delivered production rate.

所有影响共同导致系统严重偏向生成环节。队列迅速积累，但大部分被过滤掉，因此输出量较低。LLM 推理占据了端到端生成时间的主导地位，并随着训练推进持续增长。训练器在 74%的实际时钟时间中处于等待状态，其消耗速率是生成器实际输出速率的 5 倍。




Source: SemiAnalysis SemiAnalysis

# Sandbox Scaling Challenge

# 沙箱扩展挑战

## Setup 设置

- Model: Qwen3 235B A22B Instruct 2507, both FP16

  模型：Qwen3 235B A22B Instruct 2507，均为 FP16

- Hardware: GB300 硬件：GB300
- RL framework: verl + uni-agent rollout

  强化学习框架：verl + uni-agent 推演

- RL environment: SWE-bench-style env, with Modal sandbox

  强化学习环境：SWE-bench 风格环境，搭配 Modal 沙盒

- Algorithm: GRPO 算法：GRPO

- Max sequence length: 128K

  最大序列长度：128K

## Training Stages 训练阶段

1st Stage: 第一阶段：

- Trainer: 32 GPUs, TP16 / EP32, CPU offload

  训练器：32 个 GPU，TP16/EP32，CPU 卸载

- Generator: 24 GPUs, 6 replicas, TP4

  生成器：24 个 GPU，6 个副本，TP4

- Group size: 8 组大小：8
- Number of concurrent rollouts: 96

  并发回滚数：96

2nd / 3rd Stage: 第二阶段/第三阶段：

- Trainer: 48 GPUs, TP4 / CP4 / PP3 / EP16, CPU offload

  训练器：48 块 GPU，TP4/CP4/PP3/EP16，CPU 卸载

- Generator: 24 GPUs, 6 replicas, TP4

  生成器：24 块 GPU，6 个副本，TP4

- Group size: 8 组大小：8
- Number of concurrent rollouts: 256 (2nd) / 360 (3rd)

  并发推演数量：256（第二轮）/ 360（第三轮）

## Analysis 分析

The training dynamics were roughly in line with our other Qwen3 235B training runs. The run was generation-bound: trainer idled 30% to 60% of the time. Generation length grew aggressively: 3 to 8% of the samples hit max sequence length 128K, and for

similar reasons, the tail latency blew up to 7500s for a noticeable straggler on step 9.

训练动态与我们其他的 Qwen3 235B 训练运行大致一致。本次训练受生成阶段约束：训练器有 30%至 60%的时间处于空闲状态。生成长度增长迅猛：3%到 8%的样本达到了最大序列长度 128K，类似原因导致尾延迟在第 9 步时激增至 7500 秒的显著掉队者。

The unique part of this case is that we pushed the concurrent rollouts and encountered sandbox scaling issues. Scaling concurrent rollouts is critical to sample generation throughput. It enables higher batch size, more reward signal variance, and subsequently provides a better training signal. However, each rollout corresponds to at least one sandbox, and scaling rollout concurrency challenges the sandbox infrastructure reliability. In this case, we experimented with concurrent rollouts ranging from 96 to 960. At 960, once we pushed past our account's configured limits, we encountered sandbox initialization dead errors and straggler 1-hour sandbox spin up latency, and to mitigate that, we scaled down to 96, but then observed low rollout efficiency. This experiment shows how critical sandbox scaling is to RL training efficiency.

本案例的独特之处在于，我们推进了并发推演并遭遇了沙箱扩展问题。提升并发推演规模对样本生成吞吐量至关重要，它能支持更大的批处理规模、获得更多奖励信号方差，进而提供更优质的训练信号。然而，每次推演至少对应一个沙箱，扩展推演并发性会给沙箱基础设施的可靠性带来挑战。在此案例中，我们试验了 96 至 960 不等的并发推演规模。当达到 960 时，一旦超出账户配置限额，便出现沙箱初始化死错误以及长达 1 小时的沙箱启动延迟掉队现象。为缓解此问题，我们将规模降至 96，但发现推演效率低下。该实验充分说明了沙箱扩展能力对 RL 训练效率的关键作用。

We thank Ao Shen and Kaichao You from vLLM / Inferact for helping conduct this experiment. The detailed writeup of errors and attempts is here, the WandB logs of the final stage training are here, and the code repo is here (verl, uni-agent).

感谢 vLLM / Inferact 团队的沈奥和尤恺超协助开展此项实验。详细的错误记录与尝试过程请参阅此处，最终阶段训练的 WandB 日志见此处，代码仓库位于此（包含 verl 和 uni-agent）。

# Partial Rollout and Stateful Sandbox Design

# 部分展开与带状态的沙箱设计

## Setup 设置

- Model: Qwen3-235B-A22B-Thinking-2507-FP8 (FP8 generation, BF16 training)

  模型：Qwen3-235B-A22B-Thinking-2507-FP8（FP8 推理，BF16 训练）

- Trainer: 32x H200 GPUs, TP4 / CP4 / PP2 / EP8

  训练器：32 块 H200 GPU，TP4/CP4/PP2/EP8 配置

- Generator: 64x H200 GPUs, 8 replicas, TP8 / EP8

  生成器：64 块 H200 GPU，8 个副本，TP8/EP8 配置

- 32K max sequence length, global batch size 64

  最大序列长度 32K，全局批大小 64

- Algorithm: GSPO, group size 8

  算法：GSPO，组大小 8

- RL framework: slime 强化学习框架：slime
- Async setup: Fully asynchronous + partial rollout

  异步设置：完全异步 + 部分展开

- RL env: SWE-Bench Verified and Lite, with OpenHands harness and Modal sandbox

  RL 环境：SWE-Bench Verified 和 Lite，搭配 OpenHands 框架与 Modal 沙箱

- Code repo: slime SWE Bench example

  代码仓库：slime SWE Bench 示例

## Analysis 分析

Overall, this experiment has system characteristics similar to other Qwen3 235B runs: generation-bound with 60% trainer wait ratio, long 12K response length per turn, a moderate average 12 tool calls per sequence, and a relatively low resolve rate.

总体而言，本实验的系统特性与其他 Qwen3 235B 运行类似：受生成环节限制，训练器等待比例为 60%，每轮响应长度达 12K，每个序列平均工具调用次数为 12 次，且解决率相对较低。

We'd like to highlight slime's partial rollout feature and how the sandbox design has to adapt to support it. Partial rollout allows the straggler rollouts to be saved into a buffer and restarted in a later batch. This mitigates the long tail latency of straggler rollouts, which improves system efficiency.

我们想重点介绍 slime 的部分推出功能，以及沙箱设计如何适配以支持这一功能。部分推出允许将掉队的推出结果保存到缓冲区，并在后续批次中重新启动。这能缓解掉队推出带来的长尾延迟问题，从而提升系统效率。

In slime, partial rollout marks straggler rollouts as aborted, saves them to a replay buffer, and puts them into the queue in future batches. This creates challenges for the sandbox:

在 slime 中，部分回滚会将掉队回滚标记为已中止，将其保存到回放缓冲区，并在未来的批次中放入队列。这给沙盒带来了挑战：

- Sandbox needs to persist across rollout batches, since SWE-Bench problems can be stateful: a partial rollout may be in the middle of editing files. This also means we need additional logic to track the sandbox ID and sample mapping

  沙箱需要在多个批次间保持持久化，因为 SWE-Bench 问题可能具有状态性：部分批次可能正处于文件编辑过程中。这也意味着我们需要额外的逻辑来追踪沙箱 ID 与样本的映射关系

- Sandbox needs to be lazily created: sample processing logic is coupled with sandbox creation, but there may be cases where the sample fails to emit tool calls, and creating a sandbox in that case would be wasteful

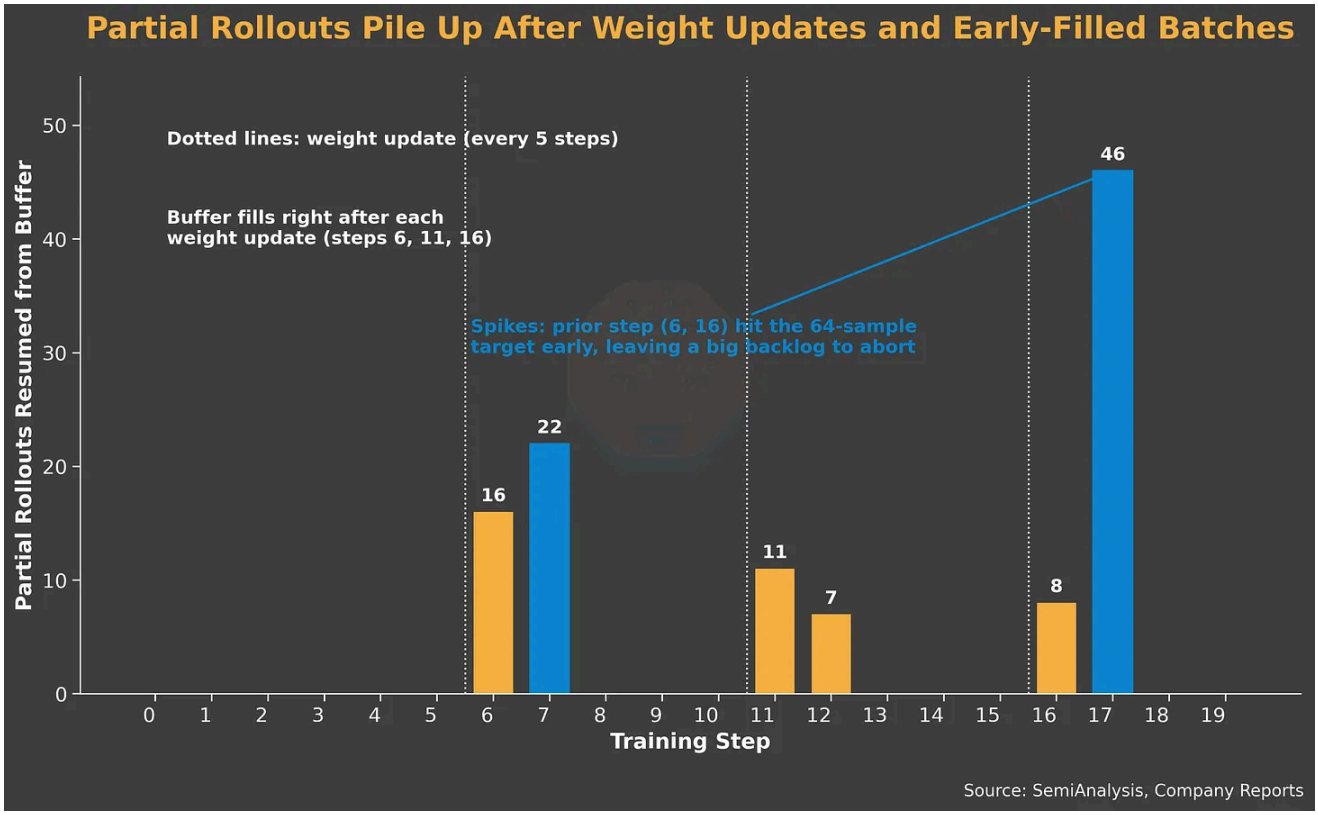
  沙箱需要采用延迟创建机制：样本处理逻辑与沙箱创建相耦合，但可能存在样本未能生成工具调用的情况，此时创建沙箱会造成资源浪费

- Sandbox robustness to failure: If sandbox fails during the time between a sample being aborted and being restarted, we need extra logic to ensure correctness. Since we didn't figure out how to do failure resumption on Modal, we opted to mark the sample as failed

  沙箱的故障鲁棒性：若沙箱在样本中止与重启之间发生故障，我们需要额外逻辑来确保正确性。由于我们尚未解决 Modal 平台上的故障恢复问题，因此选择将该样本标记为失败

Slime triggers partial rollout abortion in two situations: when a target rollout batch is filled (64 samples in our case), and when the trainer pushes a weight update. This leads to an interesting dynamic of partial rollout count in the replay buffer. We see the number increase after steps 5, 10, and 15, since we update weights every 5 intervals. We also see the number spikes at step 7 and 17, and we believe it's due to the previous step (step 6 and 16) completing as soon as it reaches the target rollout batch, leaving a big backlog.

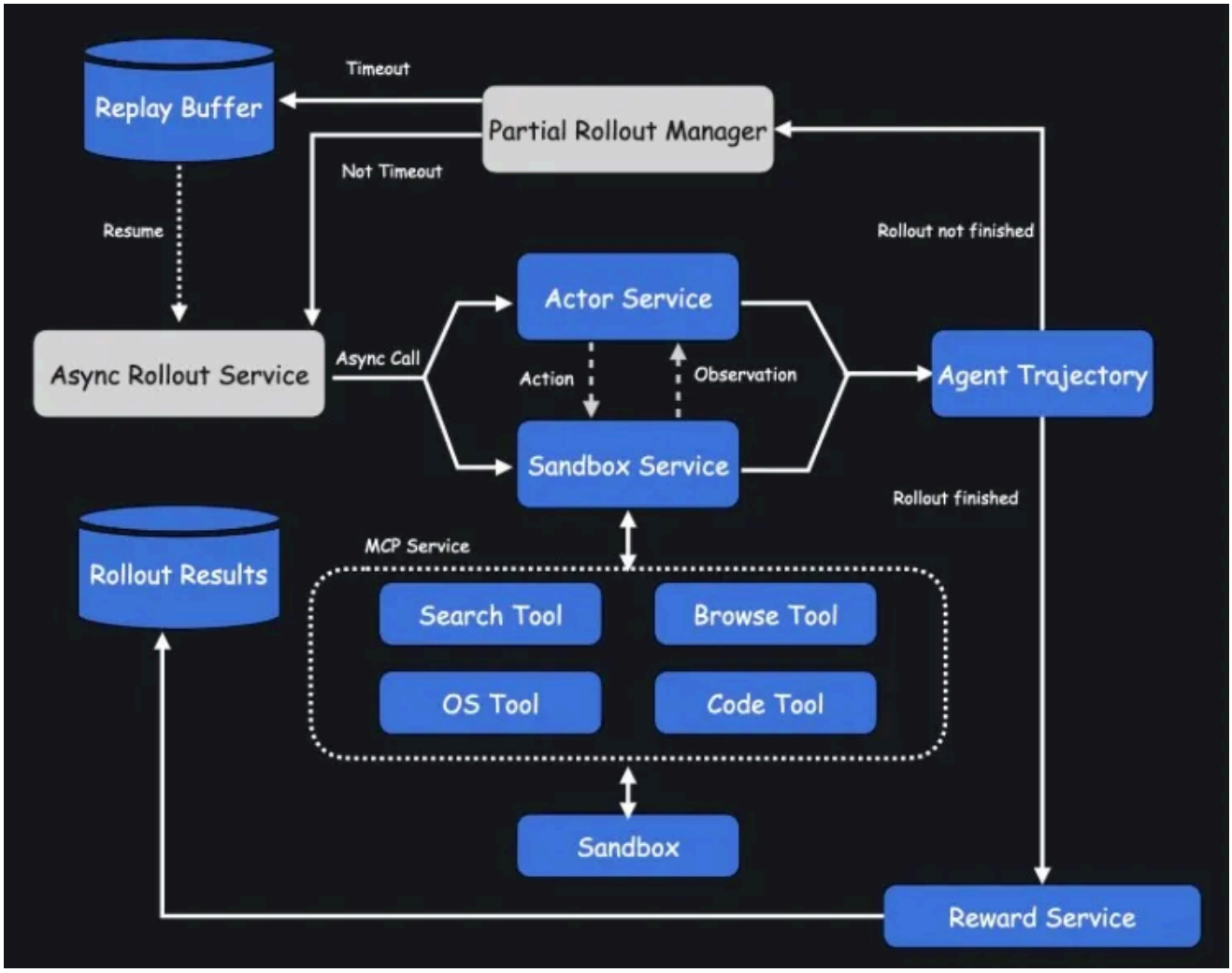
Slime 在两种情况下会触发部分生成中断：当目标生成批次已满（本系统中为 64 个样本），以及当训练器推送权重更新时。这导致回放缓冲区中出现有趣的部分生成动态变化。我们观察到在第 5、10、15 步之后数量增加，因为每 5 个间隔更新一次权重。同时在第 7 和第 17 步出现数量峰值，我们认为这是由于前一步骤（第 6 和第 16 步）在达到目标生成批次后立即完成所致。




Source: SemiAnalysis SemiAnalysis

Unlike PipelineRL, slime's SGLang configuration evicts the KV cache for aborted rollouts. This means resumed rollouts are essentially large prefill requests, which is exactly the workload type that benefits the most from PD disaggregation.

与 PipelineRL 不同，slime 的 SGLang 配置会清除已中止推演序列的 KV 缓存。这意味着恢复的推演本质上属于大型预填充请求，而这正是最受益于 PD 分离架构的工作负载类型。




Source: Kimi Researcher, Moonshot AI

Kimi 研究员，Moonshot AI

Partial rollout is where environment state-level staleness shows up. When an aborted rollout resumes in a later batch, the trainer has pushed new weights in between, so the resumed rollout faces not only the policy gap, but a state gap that's specific to stateful environments. In our SWE-Bench case, the sandbox it wakes up in is not a fresh repo. The sandbox holds the half-applied edits, created files, and working-directory state that the old policy produced over its earlier turns. The newer policy now must continue from a situation it didn't create and wouldn't necessarily have created itself. This can corrupt the training signal: during training, the advantage gets attributed to a trajectory the current policy only partially owns. Seeing the high resume rollout count after weight updates, we suspect environment state-level staleness contributes to the

run's low resolve rate.

部分回滚是环境状态级陈旧性显现的场景。当被中断的回滚在后续批次中恢复时，训练器在此期间已更新权重参数，因此恢复后的回滚不仅面临策略差距，还面临有状态环境特有的状态差距。以我们的 SWE-Bench 案例为例，沙箱唤醒时并非处于全新仓库状态。沙箱中保留着旧策略在先前轮次中产生的半完成编辑、已创建文件及工作目录状态。新策略现在必须从它未曾创建且未必会自行创建的情境中继续执行。这可能导致训练信号失真：训练过程中，优势值被归因于当前策略仅部分拥有的轨迹。观察到权重更新后回滚恢复次数居高不下，我们推测环境状态级陈旧性导致了本次运行的低解决率。

In summary, partial rollout rescues the stragglers to maintain the queue-filling throughput, but it comes at the cost of stateful-sandbox complexity and explicit environment state-level policy staleness.

总之，部分回滚策略通过拯救掉队者来维持队列填充吞吐量，但代价是增加了有状态沙箱的复杂性，并引入了显式的环境状态级策略过时问题。

# Software User Experience 软件用户体验

## Prime RL

Prime RL has great user ergonomics. Users perform most commands with uv and configure setups in the .toml files. Prime RL also comes with agent skill files, allowing users to use AI agents more smoothly when working with the repo.

Prime RL 具备出色的用户交互体验。用户可通过 uv 执行大多数命令，并在 .toml 配置文件中完成设置。该框架还内置智能体技能文件，使用户在仓库协作中更流畅地运用 AI 智能体。

Regarding system design, we appreciate that Prime RL has an "orchestrator" actor to clarify who manages the trainer-generator dynamics. We love the hub of RL environments Environments Hub and look forward to it continuing to grow. Prime RL uses Torch Titan for training, vLLM for generation, and their in-house Prime Sandbox for hosting RL environments. Prime RL also supports PD disaggregation for the generator, which greatly boosts performance on agentic RL training. Overall, the

codebase is lean and has most state-of-the-art techniques.

在系统设计方面，我们欣赏 Prime RL 采用"编排器"角色来清晰管理训练器与生成器之间的动态关系。我们推崇其 RL 环境中心，并期待该平台持续发展。Prime RL 使用 Torch Titan 进行训练，vLLM 负责生成，并通过自主研发的 Prime Sandbox 托管 RL 环境。该框架还支持生成器的 PD 分离架构，这极大提升了智能体强化学习训练的性能。总体而言，其代码库简洁且集成了大多数前沿技术。

However, we also encountered some rough edges. Prime RL's heavy reliance on uv assumes users have great familiarity with uv. We spent a lot of time compiling and re-installing flash attention 3 because we couldn't figure out why uv uninstalled it. We encourage Prime Intellect to provide more resources to educate users on how uv works, in addition to the docs saying what uv commands not to execute.

然而，我们也遇到了一些不够完善的地方。Prime RL 对 uv 的高度依赖要求用户非常熟悉 uv 的使用。我们花了大量时间反复编译和重装 flash attention 3，因为始终搞不清 uv 卸载它的原因。我们建议 Prime Intellect 除了在文档中说明应避免执行哪些 uv 命令之外，还应当提供更多资源来帮助用户理解 uv 的工作机制。

Although Prime RL assumes it runs on a SLURM cluster, it assumes jobs run on bare metal instead of Pyxis. When we ran into CUDA library version mismatch for DeepGEMM libraries, running jobs in containers would solve the issue more easily.

尽管 Prime RL 假设其在 SLURM 集群上运行，但它假定作业在裸金属上执行而非通过 Pyxis。当我们遇到 DeepGEMM 库的 CUDA 库版本不匹配问题时，在容器中运行作业能更轻松地解决该问题。

We also had a lot of failed runs that turned out to be Prime Sandbox failures. We had a hard time parsing through Prime Sandbox's error messages, which usually happens late into the run.

我们也有许多失败的运行最终被证实是 Prime Sandbox 的故障。解析 Prime Sandbox 的错误信息对我们来说非常困难，这些问题通常会在运行后期才暴露出来。

Errors include dangling sandboxes using up sandbox quota, out-of-resource errors, out-of-credit issues, many of which can be detected before launching a run. Prime

Sandbox is still in beta, so we expect it to improve in the future.

错误包括悬空沙盒占用沙盒配额、资源不足错误、信用额度不足等问题，其中许多问题可以在启动运行前检测到。Prime Sandbox 仍处于测试阶段，因此我们预计其未来会有所改进。

We thank Prime Intellect for supporting the effort, including software and hardware resources. Prime Intellect provided training recipes based on this PR (Commit hash here).

我们感谢 Prime Intellect 提供的支持，包括软件和硬件资源。Prime Intellect 基于此 PR（提交哈希值见此处）提供了训练方案。

## slime 粘液怪

slime is known for its clean and minimal abstractions, and we believe it lives up to the hype. We particularly like slime's hook abstractions. They allow us to write customized functions for rollout processing, reward functions, and metric logging utilities. The function signature and contracts are clear, and it's a breeze to integrate and add new features during development.

slime 以简洁干净的抽象层而闻名，我们认为它名副其实。我们尤其欣赏 slime 的钩子抽象机制，这使得我们能够编写定制化的回滚处理函数、奖励函数以及指标记录工具。函数签名和契约非常清晰，在开发过程中集成和添加新功能也变得轻而易举。

We also want to highlight how friendly and helpful slime's developer team is. Zilin, slime's core maintainer, has offered guidance on configuration tuning and answered slime design questions.

我们还想强调 slime 开发团队的友好与乐于助人。slime 的核心维护者 Zilin 在配置调优方面提供了指导，并解答了关于 slime 设计的问题。

Our main gripe is slime's focus on co-located mode, and as a result sparse documentation of asynchronous modes. We figured the difference between async and fully async and the mechanisms of partial rollout mostly through trial and error. We

hope slime can provide better documentation on those topics.

我们主要的不满在于 slime 对同地模式的侧重，导致异步模式的文档资料稀缺。我们主要通过反复试验才弄清了异步与完全异步的区别以及部分推演机制的运作方式。希望 slime 能就此提供更完善的文档说明。

## Modal

Modal has great API documentation: We referenced Modal's Sandbox API extensively when implementing the sandbox integration in slime, and the documentation is clear and helpful. Modal services have also been relatively robust throughout our experiments. Under small scale, Modal has been reliable, and creating / terminating sandboxes is easy and responsive. The Modal team has also been very friendly and helpful. Modal offered support early on in our effort with API credits, and Modal's slime integration team Peyton and Nan offered great advice and feedback on our experiment results.

Modal 拥有出色的 API 文档：我们在 slime 中实现沙盒集成时大量参考了 Modal 的 Sandbox API，其文档清晰且实用。在实验过程中，Modal 的服务也始终保持相对稳定。小规模场景下，Modal 表现可靠，创建/终止沙盒的操作便捷且响应迅速。Modal 团队同样非常友好且乐于助人。他们早期便通过 API 信用额度为我们提供支持，而提供支持，其 slime 集成团队成员 Peyton 和 Nan 就我们的实验结果给出了宝贵的建议与反馈。

The main challenges for Modal arose when we ran many concurrent sandboxes. As we reported above, we saw dead initialization errors and long-tail sandbox start-up latencies. We suspect this was our own misuse of the API rather than a hard limit in Modal, but we couldn't pinpoint the root cause on our end during training. We later reached out to Modal and collaborated on identifying the issue. It turned out to be the resource limit on our account, and we were able to verify the stability of high sandbox concurrency after Modal raised our limits (Reproduction code here). We appreciate the responsiveness of Modal's developer team, and we hope Modal can offer more sandbox

observability tools and more documentation on how to scale sandboxes in the future.

Modal 面临的主要挑战在于我们同时运行大量沙盒时出现。如前所述，我们遇到了初始化错误和沙盒启动延迟的长尾问题。我们推测这可能是 API 使用不当所致，而非 Modal 本身的硬性限制，但训练期间我们始终无法定位根本原因。后续我们联系了 Modal 团队并共同排查问题，最终发现是账户的资源限制所致。在 Modal 提升限制后，我们验证了高并发沙盒环境的稳定性（复现代码见此处）。我们感谢 Modal 开发团队的快速响应，同时期待 Modal 未来能提供更完善的沙盒可观测性工具，以及更多关于沙盒扩展的文档说明。

# History of Open-Source RL Frameworks

# 开源 RL 框架的历史

The release of DeepSeek R1 sparked the open-source community effort to reproduce the algorithm and infrastructure. One of the early efforts is OpenRLHF, a large-scale RL training framework that implemented PPO, REINFORCE++, and then GRPO. OpenRLHF profoundly influenced the subsequent RL framework developments. Numerous OpenRLHF maintainers later developed popular RL frameworks, including slime and verl. These frameworks created vibrant Chinese communities of RL training, which we believe positively contributed to recent advances of Chinese models. These open-source frameworks also enabled academic researchers to develop new algorithms and techniques, bringing RL research within reach of academia.

DeepSeek R1 的发布激发了开源社区在算法与基础设施复现方面的努力。其中一项早期成果是 OpenRLHF——一个实现 PPO、REINFORCE++及 GRPO 的大规模强化学习训练框架。该框架深刻影响了后续 RL 框架的发展，多位 OpenRLHF 维护者后来开发了 slime、verl 等热门 RL 框架。这些框架构建了活跃的中文社区，我们认为这为中国模型的近突破做出了积极贡献。这些开源框架也使学术研究者得以开发新算法与技术，让 RL 研究触手可及。

We thank Kaichao, who led vLLM's early-stage RL integration in OpenRLHF, verl, etc., for sharing this history. We hope this shows that sincere collaboration and sharing, rather than hostile competitions and information hiding, propels science

forward.

感谢 Kaichao 分享了 vLLM 在 OpenRLHF、verl 等项目早期阶段集成强化学习的历史。我们希望这能表明，真诚的合作与分享，而非敌对的竞争与信息隐藏，才能推动科学进步。

# Conclusion 结论

RL training system efficiency is a matter of queue health. Oversampling, early pruning, and partial rollout controls what enters the queue, adaptive sampling and policy staleness controls what leaves the queue, and in-flight weight updates allow trainer and generator to operate at different rates.

RL 训练系统的效率本质上是队列的健康状态问题。过采样、早期剪枝和部分执行控制着进入队列的内容，自适应采样和策略陈旧度管理控制着离开队列的内容，而飞行中权重更新使得训练器和生成器能够以不同速率运行。

RL training is as much of an infrastructure problem as an algorithm one. In this post, we focused on systems, explaining how system and algorithm designs are intertwined, and grounding them with real-world experiments. We hope this ignites interest in RL infrastructure and bridges RL algorithm researchers with system engineers.

强化学习既是一个算法问题，也是一个基础设施问题。在这篇文章中，我们聚焦于系统层面，解释系统设计与算法设计如何相互交织，并通过实际实验加以验证。我们希望这能激发对强化学习基础设施的兴趣，并架起强化学习算法研究者与系统工程师之间的桥梁。

We plan to continue exploring the systems and infrastructure of RL training in our future articles. We will explore in depth techniques such as PD disaggregation and speculative decoding, and system challenges of large-scale RL training such as training Nemotron 3 Ultra. We are also looking into benchmarking sandbox

infrastructure and analyzing model rollout traces.

我们计划在后续文章中继续深入探讨强化学习训练的系统与基础设施。我们将详细研究 PD 分离、推测解码等技术，以及大规模 RL 训练（如训练 Nemotron 3 Ultra）面临的系统挑战。同时，我们也在着手对沙箱基础设施进行基准测试，并分析模型推理轨迹。

If you're interested in working on these topics, we're hiring! Please contact us at letsgo@semianalysis.com . Attach your resume and ask us questions about this article.

如果你对这些话题感兴趣，我们正在招聘！请通过 letsgo@semianalysis.com 联系我们。附上你的简历，并就本文内容向我们提问。

For our last section, we present a TCO analysis of RL training, and we compare it to RL training platforms Tinker from Thinking Machines Lab. We quantify how much more cost efficient Tinker is, and we hypothesize the reasons behind it.

在最后一节中，我们对强化学习训练进行了总拥有成本（TCO）分析，并将其与 Thinking Machines Lab 的 RL 训练平台 Tinker 进行了对比。我们量化了 Tinker 在成本效率上的优势，并推测了其背后的原因。

# RL Training TCO Analysis RL 训练总拥有成本分析

## Server Cost 服务器成本

H200 SXM InfiniBand has a total upfront cluster capex of $361k per server, or $45.2k per logical GPU, for a Neocloud Giant customer profile. Most of the capital cost is still the server itself at $258k per server, or roughly 71% of total upfront capex. On a per-GPU basis, this translates to $45.2k of upfront capital cost per logical GPU. Using a 10.25% WACC and 6-year useful life, we get a capital cost of $1.15 per hour per GPU.

对于 Neocloud Giant 客户群体，H200 SXM InfiniBand 集群的服务器前期资本支出为每台 36.1 万美元，折合每逻辑 GPU 为 4.52 万美元。大部分资本成本仍来自服务器本身，每台 25.8 万美元，约占前期总支出的 71%。按每 GPU 计算，这意味着每逻辑 GPU 的前期资本成本为 4.52 万美元。采用 10.25%的加权平均资本成本率和 6 年使用年限，可得出每 GPU 每小时 1.15 美元的资本成本。

**AI Cloud Capital Cost of Ownership Summary**

| Chip<br>Customer Profile | Unit | H200 SXM (InfiniBand)<br>Neocloud Giant |
|---|---|---|
| Cluster Capital Costs | | |
| **Server Cost** | **USD** | **$258,001** |
| Server Service | USD | $8,000 |
| **Networking Cost** | **USD** | **$76,155** |
| Storage Cost | USD | $11,732 |
| Software Licenses and Other Costs | USD | $2,867 |
| Other Installation | USD | $4,500 |
| **Service, Networking, Storage, Software, Others** | **USD** | **$103,253** |
| **Total Upfront Cluster Capex, per Server** | **USD** | **$361,255** |
| **Total Upfront Cluster Capex, per Logical GPU** | **USD** | **$45,157** |
| Weighted Average Cost of Capital | % | 10.25% |
| Useful Life in Years | Years | 6 |
| **Total Cluster Capital Costs per Month per Server** | **USD/mth** | **$6,738.18** |
| **Capital Cost per Unit, per Hour** | **USD/hr/GPU** | **$1.15** |

Source: SemiAnalysis AI TCO Model

SemiAnalysis 人工智能总拥有成本模型

Operating cost is primarily a function of server power draw and the facility cost applied to that power. The model assumes 10.93kW of all-in power consumption per 8-GPU server, or roughly 1.37kW per GPU. At $0.087/kWh, 80% utilization, and 1.35 PUE, electricity equates to $68.59 per kW of critical IT power per month. Colocation for a Neocloud typically comes out to about $150/kW/month in the US, Applying that to the 10.93kW server power footprint results in $2,388/month of power and colocation cost per server, or roughly $299/month per GPU. On an hourly basis, this translates to $0.44/hr/GPU of operating cost.

运营成本主要取决于服务器功耗及该功耗对应的设施成本。该模型假设每台 8-GPU 服务器的总功耗为 10.93kW，即每块 GPU 约 1.37kW。按 0.087 美元/千瓦时电价、80%利用率及 1.35 的 PUE 值计算，每千瓦关键 IT 电力月均电费为 68.59 美元。在美国，Neocloud 的托管费用通常约为 150 美元/千瓦/月。将此标准应用于 10.93kW 的服务器功耗，每台服务器每月电费与托管费合计为 2,388 美元，即每块 GPU 约 299 美元/月。按小时折算，每块 GPU 的运营成本为 0.44 美元/小时。

| AI Cloud Operating Cost of Ownership Summary | | |
|---|---|---|
| Chip | Unit | H200 SXM (InfiniBand) |
| Customer Profile | | Neocloud Giant |
| Cluster Operating Costs | | |
| Electricity Cost | USD/kWh | $0.087 |
| Utilization Rate | % | 80.0% |
| Power Usage Effectiveness (PUE) | Ratio | 1.35 |
| Electricity Cost per kW of Critical IT per mth | USD/kW/mth | $68.59 |
| Colocation Cost | USD/kW/mth | $150.00 |
| **Total Cost per kW Critical IT Power per Month** | **USD/kW/mth** | **$218.59** |
| All-in Power Consumption | kW | 10.93 kW |
| **Total Server Costs per Month** | **USD/mth** | **$2,388.27** |
| Remote Hands + Support Engineer | USD/mth | $131.00 |
| Internet Connection | USD/mth | $39.00 |
| **Total Cluster Operating Cost per Month, per Serve** | **USD/mth** | **$2,558.27** |
| **Total Cluster Operating Cost per Month, per GPU** | **USD/mth** | **$319.78** |
| **Operating Cost per Unit, per Hour** | **USD/hr/GPU** | **$0.44** |

Source: SemiAnalysis AI TCO Model

SemiAnalysis 人工智能总拥有成本模型

Total ownership cost comes out to **$1.59/hr/GPU, with Capital Cost contributing to 72.5% of per GPU per hour cost.**

总拥有成本为每 GPU 每小时 1.59 美元，其中资本成本占每 GPU 每小时成本的 72.5%。

| AI Cloud Operating Cost of Ownership Summary | | |
|---|---|---|
| Chip | Unit | H200 SXM (InfiniBand) |
| Customer Profile | | Neocloud Giant |
| Capital Cost per Unit per Hour | USD/hr/GPU | $1.15 |
| Operating Cost per Unit per Hour | USD/hr/GPU | $0.44 |
| **Total Cost per Unit per Hour** | **USD/hr/GPU** | **$1.59** |
| Capital Cost as % of Total Ownership Cost | % | 72.5% |

Source: SemiAnalysis AI TCO Model

SemiAnalysis 人工智能总拥有成本模型

# RL Training Run TCO RL 训练运行总体拥有成本

For our experiments, we did a reinforcement learning run on Qwen3-235B-Thinking-2507 and applied our H200 TCO against the RL duration and Number of H200s required - then compared our own RL TCO estimate to Tinker published pricing (as of June 11th, 2026). We note that our TCO captures one PrimeRL and slime configuration. We expect these numbers to drop accordingly, as better recipes are coming from the open source community and vendors like NVIDIA.

在我们的实验中，我们对 Qwen3-235B-Thinking-2507 模型进行了强化学习训练，并基于训练持续时间和所需 H200 数量，应用了 H200 的总拥有成本（TCO）进行估算。随后，我们将自行估算的 RL TCO 与 Tinker 公布的定价（截至 2026 年 6 月 11 日）进行了对比。需要说明的是，我们的 TCO 仅涵盖一套 PrimeRL 与 slime 配置方案。我们预期随着开源社区及 NVIDIA 等供应商推出更优的配方，这些数字将相应下降。

| Assumptions (Qwen 3 slime) | |
|---|---|
| H200 Total cost per GPU hour | $1.59 |
| Rollout batch size | 64 |
| Generator number of GPUs | 64 |
| Trainer number of GPUs | 32 |
| RL Process Total cost per GPU hour | $152.64 |

| Tinker Qwen3 235B A22B Costs[1] | |
|---|---|
| Context Length | 32,000 |
| Prefill ($/Mtok) | $0.68 |
| Sample ($/Mtok) | $1.70 |
| Train ($/Mtok) | $2.04 |

*1. https://tinker-docs.thinkingmachines.ai/tinker/models/*

Source: SemiAnalysis SemiAnalysis

| Assumptions (Qwen 3 Prime RL) | |
|---|---|
| H200 Total cost per GPU hour | 1.59 |
| Rollout batch size | 512 |
| Generator number of GPUs | 192 |
| Trainer number of GPUs | 64 |
| RL Process Total cost per GPU hour | $407.04 |

| Tinker Qwen3 235B A22B Costs[1] | |
|---|---|
| Context Length | 32,000 |
| Prefill ($/Mtok) | $0.68 |
| Sample ($/Mtok) | $1.70 |
| Train ($/Mtok) | $2.04 |

*1. https://tinker-docs.thinkingmachines.ai/tinker/models/*

Source: SemiAnalysis SemiAnalysis

The system runs trainer and generator GPUs concurrently. Within every step, while the generator is producing rollouts for step N, the trainer is doing fwd/bwd on the rollouts from step N−1. The two pools overlap almost entirely in wall-clock; they share the same step boundary.

系统同时运行训练器和生成器 GPU。在每个 step 中，当生成器正在为第 N 个 step 生成 rollouts 时，训练器正在对第 N-1 个 step 的 rollouts 进行前向/反向传播。这两个计算池在墙上时钟上几乎完全重叠，共享相同的 step 边界。

Hence, we look at the max time taken on each step by either the training or generator GPUs and apply the time required across the TCO across both cards as even though one might be waiting on the other, you are effectively still paying the TCO per hour to have it on hand. Even if the training GPU might not run for the same duration as the generator GPUs, the trainer pool is reserved for the entire run; it is on the meter whether it is computing or waiting. For example, if the generator runs for 600s and the trainer runs for 270s during the same 600s, the run consumes 600s of wall-clock, not 870s.

因此，我们关注每个 step 中训练器或生成器 GPU 各自消耗的最大时间，并将该时间应用于两张 GPU 的全周期成本（TCO）。因为即使其中一方可能在等待另一方，实际上你仍在按小时为这些硬件支付 TCO。即便训练器 GPU 的运行时长可能短于生成器 GPU，训练器池在整个运行周期内都是被预留的，无论它是在计算还是等待，都在持续计费。例如，如果生成器在同一 600 秒内运行了 600 秒而训练器只运行了 270 秒，实际消耗的是 600 秒的墙上时钟时间，而非 870 秒。

Looking at our experiment on Qwen3 235B A22B slime, we land ~4.86x higher than Tinker's published $4.86/Mtok target at $16.23/Mtok.

观察我们在 Qwen3 235B A22B slime 上的实验，最终成本为每百万 token 16.23 美元，比 Tinker 公布的 4.86 美元/百万 token 目标高出 4.86 倍。

## RL Process (slime)

| Step | Avg Response Length | Avg Sequence Length | Rollout Time (s) |
|---|---|---|---|
| 0 | 11,128 | 19,702 | 593.4 |
| 1 | 13,227 | 21,473 | 479.2 |
| 2 | 10,946 | 19,777 | 518.9 |
| 3 | 12,691 | 20,357 | 479.2 |
| 4 | 13,644 | 20,628 | 426.4 |
| 5 | 12,185 | 21,004 | 522.1 |
| 6 | 12,766 | 20,257 | 555.3 |
| 7 | 12,718 | 21,383 | 523.7 |
| 8 | 11,203 | 19,127 | 374.4 |
| 9 | 10,451 | 17,228 | 527.1 |
| 10 | 12,896 | 19,333 | 455.4 |
| 11 | 11,696 | 19,603 | 457.8 |
| 12 | 10,273 | 19,183 | 433.1 |
| 13 | 13,168 | 20,730 | 513.4 |
| 14 | 12,864 | 21,939 | 481.5 |
| 15 | 12,852 | 20,193 | 423.5 |
| 16 | 11,590 | 18,860 | 620.7 |
| 17 | 13,847 | 21,686 | 508.9 |
| 18 | 11,630 | 19,900 | 496.1 |
| 19 | 12,856 | 20,922 | 454.4 |
| **Steady Average** | **12,232** | **20,164** | **492.2** |

Source: SemiAnalysis SemiAnalysis

## Tinker TCO

| Step | Prefill cost | Decode cost | Train cost | Total cost | Total cost per Mtok |
|---|---|---|---|---|---|
| 0 | $0.37 | $1.21 | $2.57 | $4.16 | $3.30 |
| 1 | $0.36 | $1.44 | $2.80 | $4.60 | $3.35 |
| 2 | $0.38 | $1.19 | $2.58 | $4.16 | $3.28 |
| 3 | $0.33 | $1.38 | $2.66 | $4.37 | $3.36 |
| 4 | $0.30 | $1.48 | $2.69 | $4.48 | $3.39 |
| 5 | $0.38 | $1.33 | $2.74 | $4.45 | $3.31 |
| 6 | $0.33 | $1.39 | $2.64 | $4.36 | $3.36 |
| 7 | $0.38 | $1.38 | $2.79 | $4.55 | $3.33 |
| 8 | $0.34 | $1.22 | $2.50 | $4.06 | $3.32 |
| 9 | $0.29 | $1.14 | $2.25 | $3.68 | $3.34 |
| 10 | $0.28 | $1.40 | $2.52 | $4.21 | $3.40 |
| 11 | $0.34 | $1.27 | $2.56 | $4.18 | $3.33 |
| 12 | $0.39 | $1.12 | $2.50 | $4.01 | $3.27 |
| 13 | $0.33 | $1.43 | $2.71 | $4.47 | $3.37 |
| 14 | $0.39 | $1.40 | $2.86 | $4.66 | $3.32 |
| 15 | $0.32 | $1.40 | $2.64 | $4.35 | $3.37 |
| 16 | $0.32 | $1.26 | $2.46 | $4.04 | $3.35 |
| 17 | $0.34 | $1.51 | $2.83 | $4.68 | $3.37 |
| 18 | $0.36 | $1.27 | $2.60 | $4.22 | $3.32 |
| 19 | $0.35 | $1.40 | $2.73 | $4.48 | $3.35 |
| **Steady Average** | **$0.35** | **$1.33** | **$2.63** | **$4.31** | **$3.34** |

Source: SemiAnalysis SemiAnalysis

For our experiment on Qwen3 235B A22B Prime RL, we landed at ~2.01x higher than Tinker's published $3.43/Mtok target at $6.90/Mtok.

在我们对 Qwen3 235B A22B Prime RL 的实验中，最终成本为每百万 token 6.90 美元，比 Tinker 公布的每百万 token 3.43 美元目标高出约 2.01 倍（即~2.01x）。

**RL Process (Prime RL)**

| Step | Avg Prompt Length | Avg Response Length | Avg Sequence Length | Rollout Time (s) |
|---|---|---|---|---|
| 0 | 3,629 | 10,203 | 13,833 | 371 |
| 1 | 5,141 | 11,898 | 17,039 | 479 |
| 2 | 5,689 | 12,317 | 18,006 | 529 |
| 3 | 5,118 | 13,042 | 18,160 | 544 |
| 4 | 5,394 | 13,360 | 18,755 | 587 |
| 5 | 6,904 | 12,851 | 19,755 | 656 |
| 6 | 5,897 | 13,116 | 19,013 | 632 |
| 7 | 5,272 | 13,418 | 18,691 | 608 |
| 8 | 6,892 | 13,569 | 20,461 | 688 |
| 9 | 5,971 | 13,262 | 19,233 | 663 |
| **Steady Average** | **5,591** | **12,704** | **18,294** | **576** |

Source: SemiAnalysis SemiAnalysis

**Our TCO Estimate (Prime RL)**

| Step | Cost per Step | Tokens per Step | Cost per Mtok |
|---|---|---|---|
| 0 | $41.93 | 7,082,240 | $5.92 |
| 1 | $54.16 | 8,723,866 | $6.21 |
| 2 | $59.77 | 9,218,970 | $6.48 |
| 3 | $61.46 | 9,297,818 | $6.61 |
| 4 | $66.34 | 9,602,406 | $6.91 |
| 5 | $74.22 | 10,114,406 | $7.34 |
| 6 | $71.45 | 9,734,502 | $7.34 |
| 7 | $68.74 | 9,569,536 | $7.18 |
| 8 | $77.76 | 10,476,237 | $7.42 |
| 9 | $74.92 | 9,847,450 | $7.61 |
| **Steady Average** | **$65.07** | **9,366,743** | **$6.90** |

Source: SemiAnalysis SemiAnalysis

**Tinker TCO**

| Step | Prefill cost | Decode cost | Train cost | Total cost | Total cost per Mtok |
|---|---|---|---|---|---|
| 0 | $1.26 | $8.88 | $14.45 | $24.59 | $3.47 |
| 1 | $1.79 | $10.36 | $17.80 | $29.94 | $3.43 |
| 2 | $1.98 | $10.72 | $18.81 | $31.51 | $3.42 |
| 3 | $1.78 | $11.35 | $18.97 | $32.10 | $3.45 |
| 4 | $1.88 | $11.63 | $19.59 | $33.10 | $3.45 |
| 5 | $2.40 | $11.19 | $20.63 | $34.22 | $3.38 |
| 6 | $2.05 | $11.42 | $19.86 | $33.33 | $3.42 |
| 7 | $1.84 | $11.68 | $19.52 | $33.04 | $3.45 |
| 8 | $2.40 | $11.81 | $21.37 | $35.58 | $3.40 |
| 9 | $2.08 | $11.54 | $20.09 | $33.71 | $3.42 |
| **Steady Average** | **$1.95** | **$11.06** | **$19.11** | **$32.11** | **$3.43** |

Source: SemiAnalysis SemiAnalysis

# Tinker's Cost Edge Tinker 的成本优势

We hypothesize that Tinker's cost efficiency comes from multi-tenancy. Tinker provides a Low-Rank Adaptation (LoRA) training API, and LoRA allows multiple users to train models that share most of their weights. This means on the trainer side, Tinker can greatly increase efficiency by batching training requests across users, with some LoRA-specific modifications. On the generation side, multi-tenancy mitigates the straggler effect by backfilling idle slots with other tenants' rollouts when a run stalls on a straggler.

我们推测，Tinker 的成本效率源于其多租户架构。Tinker 提供低秩适配（LoRA）训练接口，而 LoRA 允许不同用户共享大部分权重进行模型训练。这意味着在训练端，Tinker 可通过批量处理跨用户的训练请求（配合 LoRA 特定调整）大幅提升效率。在生成端，多租户机制通过当某个任务因掉队者而停滞时，用其他租户的推理结果填充空闲计算槽位，有效缓解了掉队者效应。

Multi-tenancy is likely the primary driver, with inference-stack and hardware advantages compounding it. The strikingly different TCO in our slime and Prime RL experiments show how much inference efficiency determines total cost. We expect Thinking Machines Lab to apply inference optimizations such as PD disaggregation. They may also be running Blackwell GPUs, which offer significant inference gains

over Hopper, according to our InferenceX analysis.

多租户架构很可能是主要驱动因素，推理栈和硬件优势则进一步放大了这一效应。我们在 slime 和 Prime RL 实验中显著不同的总拥有成本（TCO）表明，推理效率对总成本的决定性作用。预计 Thinking Machines Lab 将采用 PD 分离等推理优化技术。根据我们的 InferenceX 分析，他们可能还部署了 Blackwell GPU——相较于 Hopper 架构，该芯片在推理性能上具有显著优势。

Previous 上一篇

## Discussion about this post

## 关于此帖的讨论

Comments 评论 Restacks

Write a comment...