PithTrain: A Compact and Agent-Native MoE Training System

Ruihang Lai[1*] **Hao Kang**[1*] **Haozhan Tang**[1] **Akaash R. Parthasarathy**[1]
Zichun Yu[1] **Junru Shao**[3] **Todd C. Mowry**[1] **Chenyan Xiong**[1,2†] **Tianqi Chen**[1,3†]
[1]Carnegie Mellon University [2]Xlue [3]NVIDIA

Abstract

Mixture-of-Experts (MoE) has become the dominant architecture for frontier language models. To meet this demand, production frameworks have built optimized MoE training stacks over years of engineering effort. Yet evolving these stacks for new architectures and system optimizations remains expensive. With the rise of AI coding agents, they could automate parts of training-framework development and accelerate this evolution. But applying them to these existing frameworks carries hidden costs, invisible to today's throughput-only evaluations. We name this missing dimension agent-task efficiency (ATE): the cost of using coding agents to understand, operate, and extend a framework. Grounded in four agent-native design principles, we build PithTrain, a compact, agent-native MoE training framework supporting pretraining, supervised fine-tuning, and reinforcement learning. We further introduce ATE-Bench, covering real-world training-framework tasks. Our evaluation shows PithTrain achieves throughput competitive with production frameworks and higher agent-task efficiency on ATE-Bench. On new-feature tasks, PithTrain uses up to 62% fewer Agent Turns than Megatron-LM, and up to 44% and 64% less Active GPU Time than Megatron-LM and TorchTitan, respectively.

GitHub repo: `https://github.com/mlc-ai/pith-train`

1 Introduction

Modern AI systems are increasingly powered by Mixture-of-Experts (MoE) language models such as DeepSeek-V3, Qwen3, Kimi-K2, and GPT-OSS [12, 47, 33, 44], whose training depends on systems that scale across distributed clusters. Production frameworks [43, 28, 40, 39] have built end-to-end MoE training stacks over years of engineering effort, pairing layered Python designs with extra compiled extensions to deliver broad model coverage, peak throughput, and multi-platform support needed for diverse training workloads. However, evolving these stacks for new model architectures and system optimizations demands in-depth expertise and substantial engineering effort.

AI coding agents [1, 34, 3, 13] have begun to automate parts of this work and could in principle accelerate training-system development. Most current practice applies agents to existing frameworks. But the design choices that helped human engineers cost agents differently. Plugin systems, registry-based indirection, and heavy compiled extensions raise the cost of locating relevant code, tracing what runs at a given call site, and verifying a change is complete. Today's throughput-only evaluations leave this cost unmeasured, and existing frameworks were not designed with it in mind.

This raises a design question: *can we redesign an MoE training framework that optimizes for agent-task efficiency?* We define *agent-task efficiency* (ATE) as the cost of using coding agents to understand, operate, and extend a framework, measured along dimensions such as session duration

*Equal contribution
†Equal advising

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

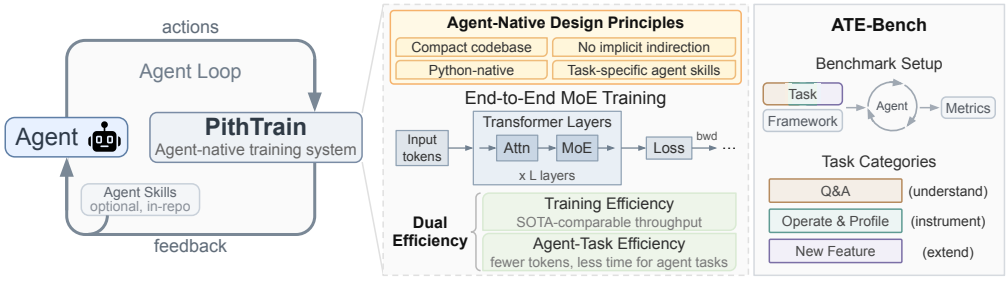


Figure 1: PithTrain overview. An agent issues actions and consumes feedback (left); built on four agent-native design principles, PithTrain delivers dual efficiency (middle); ATE-Bench evaluates agent-task efficiency across frameworks (right).

and output tokens. We answer this question with PithTrain, an end-to-end MoE training framework designed agent-native from the start. PithTrain is built on four design principles. First, we favor code compactness over coverage: PithTrain focuses on a compact MoE training stack rather than the broad model and feature coverage of production frameworks, while remaining straightforward for agents to extend with new features. Second, we use minimal, Python-native components covering the key layers of MoE training: operators, training engine, and applications. Third, we use direct calls and avoid implicit indirection in module composition, so that what runs at a given call site can be identified by static reading. Finally, we ship agent skills [2] for recurring training-framework tasks. PithTrain supports pretraining and supervised fine-tuning (SFT), and serves as a reinforcement learning (RL) backend in slime [56].

Beyond building the framework, we systematically evaluate how framework design affects agent-task efficiency on real training-system tasks. Existing AI-coding benchmarks [16, 8, 5] vary the agent on a fixed codebase to score agent capability, focusing on general software-engineering tasks such as issue resolution. We invert this with ATE-Bench, a comprehensive benchmark that varies the framework on real-world training-framework tasks, holding the agent fixed to compare framework-level development cost, including code structure, bundled guidance, and runtime behavior. Figure 1 summarizes PithTrain's overall design.

This paper makes three contributions:

- **PithTrain, a compact, Python-native, end-to-end MoE training framework.** Designed for coding agents, PithTrain achieves training throughput competitive with production frameworks.
- **Four design principles for agent-native ML training frameworks**, guiding PithTrain's design: code compactness, Python-native components, no implicit indirection, and agent skills.
- **Agent-task efficiency, ATE-Bench, and an empirical study.** Agent-task efficiency as a metric beyond training throughput, instantiated in ATE-Bench, with an empirical study showing PithTrain's higher agent-task efficiency over production frameworks, plus a skills ablation and case study.

PithTrain delivers *dual efficiency*: strong training throughput together with high agent-task efficiency. It achieves throughput competitive with production frameworks across a range of MoE models and training settings on NVIDIA H200 and B200 GPUs. On ATE-Bench's new-feature tasks, a coding agent uses up to 62% fewer Agent Turns on PithTrain than on Megatron-LM, and up to 44% and 64% less Active GPU Time than on Megatron-LM and TorchTitan, respectively. A qualitative case study illustrates how framework design shapes agent behavior.

2 Related Work

Mature production frameworks have driven large-scale MoE training. Megatron-LM [43, 28] established the pipeline-parallel Transformer recipe most subsequent frameworks build on, and DeepSpeed [40, 39] introduced ZeRO sharded-optimizer techniques. Both rely on layered designs with plugin systems, registry-based indirection, and compiled extension chains, delivering broad model coverage and peak throughput on production hardware. TorchTitan [21] more recently scaled PyTorch-native training to multi-thousand-GPU clusters, sharing PithTrain's Python-native goal but without making agent-native design a primary axis or matching production-framework throughput on our

Table 1: Adherence to agent-native design principles. ✓ satisfied; ◐ partial; ✗ not satisfied. Numbers are total framework LoC across Python, C++, and CUDA.[1]

Framework	Compact	Python-native	No implicit indirection	Agent skills
Megatron-LM	✗ 149K	✗	✗	◐
DeepSpeed	✗ 167K	✗	✗	✗
TorchTitan	◐ 38K	✓	◐	◐
PithTrain	✓ 11K	✓	✓	✓

configurations. PithTrain takes a different point on this tradeoff curve with agent-native design as a first-class goal, achieving higher agent-task efficiency along Agent Turns, Output Tokens, etc., while achieving training throughput competitive with production frameworks.

AI coding agents combine a tool-using language model with a small, stable vocabulary of primitives (read, edit, shell, search). Recent work designs better agent flows and harnesses for software engineering on top of these primitives [49, 48, 53]. PithTrain takes a complementary direction: rather than improving the agent flow, we design the software framework so that existing flows work better, lowering agent cost without changing the agent.

Existing agent benchmarks score capability on fixed codebases and tasks: SWE-bench [16] on GitHub-issue resolution, MLE-bench [7] on Kaggle-style ML engineering, and HumanEval [8] on function-level code generation. Closer to ML systems engineering, FlashInfer-Bench [46] and KernelBench [35] target inference operators and GPU kernels respectively. Other benchmarks also measure agent cost: GitTaskBench incorporates token cost into an economic-benefit metric [29], SWE-QA reports token usage for repository-level question answering [36], and public coding-agent leaderboards report token usage and cost alongside performance [4]. These evaluations vary the agent. ATE-Bench inverts this: holding the agent and task fixed, we vary the training framework to compare development cost on ML training-system tasks, including Active GPU Time consumed during testing and debugging. This axis is complementary to existing agent-facing benchmarks.

3 The PithTrain System

3.1 Agent-Native Design Principles

This subsection introduces the four agent-native design principles that guide PithTrain's system design, and contrasts how production training frameworks align with each principle today. Table 1 summarizes this comparison; production frameworks adopt different subsets of the four principles, reflecting different design priorities.

Compact codebase. Production frameworks such as Megatron-LM and DeepSpeed offer broad coverage of models, training features, and hardware platforms, accumulated over years of engineering effort, with core codebases exceeding 100K lines. A larger codebase inflates the cost of locating relevant code, tracking cross-file dependencies, and verifying a change is complete. A compact codebase reduces this cost; with frontier coding agents operating at context windows of 200K to 1M tokens, a sufficiently compact codebase can also fit in a single context pass. We treat compactness as a constraint on growth: PithTrain may grow over time, but we keep the code that defines models and coordinates training compact as kernel implementations grow.

Python-native codebase. Python is the dominant language in modern ML. A pure-Python framework lets an agent navigate the full framework in a single language, surfaces readable Python tracebacks instead of opaque native errors, and eliminates the compiled-extension rebuild cycle. These shorter feedback loops reduce the cost of iterative debugging. Megatron-LM composes its core transformer layers from out-of-tree TransformerEngine [30] modules, and DeepSpeed bundles extensive in-tree extensions. These deliver peak kernel performance and vendor-tuned numerics, but push an agent across language boundaries and force a rebuild on change.

No implicit indirection. Production frameworks compose many model variants from a shared layer skeleton via *implicit indirection* (a stored callable, plugin registry, or string-keyed resolution). This

[1] ATE-Bench evaluated PithTrain `23db182`, Megatron-LM `3bec9aa`, and TorchTitan `d84e83d`. DeepSpeed: `44c51e3`.

Model Construction via Implicit Indirection

```python
# In core/transformer/transformer_layer.py
class TransformerLayer(ModelModule):
  def __init__(self, config, submodules):
    self.input_layernorm = build_module(submodules.input_layernorm, ...)
    self.self_attention  = build_module(submodules.self_attention, ...)
    self.mlp             = build_module(submodules.mlp, ...)

# In core/models/gpt/gpt_layer_specs.py
def get_gpt_layer_with_spec(num_experts, ...):
  return ModuleSpec(
    module=TransformerLayer,
    submodules=TransformerLayerSubmodules(
      input_layernorm=Norm,
      self_attention=ModuleSpec(
        module=SelfAttention, submodules=SelfAttentionSubmodules(...)
      ),
      mlp=(ModuleSpec(module=MoELayer, ...) if num_experts > 0
           else ModuleSpec(module=MLP, ...)),
    ),
  )
```

"self.mlp" defined in another file

PithTrain

```python
# In models/qwen3_moe.py
class Qwen3DecoderLayer(nn.Module):
  def __init__(self, cfg, layer_idx):
    self.input_layernorm = RMSNorm(cfg.hidden_size)
    self.self_attn = Qwen3Attention(cfg)
    self.post_attention_layernorm = RMSNorm(cfg.hidden_size)
    use_moe = ...  # True for MoE layers, False for dense
    self.mlp = Qwen3MoE(cfg) if use_moe else Qwen3MLP(cfg)

class Qwen3MoE(nn.Module):  ...
class Qwen3MLP(nn.Module):  ...

class Qwen3Model(nn.Module):
  def __init__(self, cfg):
    self.embed_tokens = nn.Embedding(cfg.vocab_size, cfg.hidden_size)
    self.layers = nn.ModuleList([
      Qwen3DecoderLayer(cfg, i) for i in range(cfg.num_layers)
    ])
    self.norm = RMSNorm(cfg.hidden_size)
    self.lm_head = nn.Linear(cfg.hidden_size, cfg.vocab_size)
```

"self.mlp" is defined locally

Figure 2: Model construction patterns, illustrating the no-implicit-indirection principle. The implicit-indirection pattern resolves submodules through a runtime spec, supporting model variation from a shared layer skeleton; PithTrain instantiates layers directly, favoring local readability.

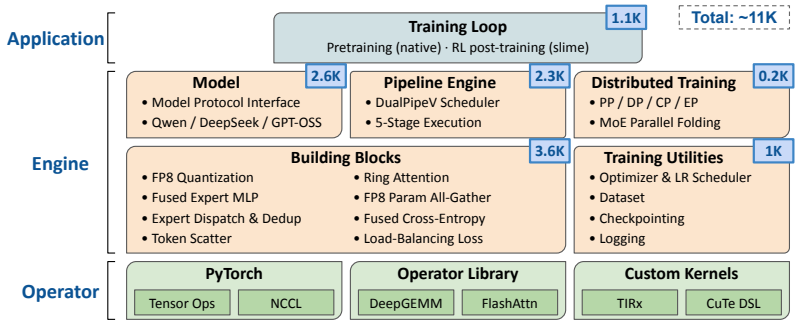


Figure 3: Architecture of PithTrain. Blue badges give per-component lines of code.[2]

pattern enables code reuse across models, while making what runs at a given call site harder to identify by local reading. Figure 2 shows an instance of model construction in a production framework: `TransformerLayer` resolves its submodules from a runtime spec in a separate file. An agent must resolve that spec before symbol search can reach the implementation. A flat code structure trades cross-model reuse for local readability, reducing the effort an agent spends building an end-to-end understanding. We balance this tradeoff through limited model-local duplication and shared functions with explicit call targets.

Task-specific agent skills. An *agent skill* [2] is a procedural playbook a coding agent loads on demand. Skills encode procedural knowledge an agent cannot recover from static reading alone, so it runs a verified procedure. Agent skills are a recent practice that existing training frameworks have not yet adopted at scale: Megatron-LM ships six skills for CI/PR hygiene; TorchTitan ships one developer-workflow skill plus four editor rules orienting the agent to the codebase; DeepSpeed ships two markdown files with generic rules. None of these target recurring training-framework tasks.

3.2 System Architecture and Optimizations

The codebase is organized in three layers (application, engine, operator), as shown in Figure 3. We identified the components necessary for distributed MoE training. Where production frameworks deliver broad out-of-the-box coverage of models, features, and hardware, PithTrain narrows scope to keep the codebase compact and reachable in a single context window. Table 2 maps each principle to where it is realized: we adopt a flat code structure with no plugin registries or runtime specs, with each MoE model living in a self-contained file under `models/` rather than being composed through a shared layer skeleton. This favors local readability over cross-model code reuse.

[2]Line counts are from `23db182`, before the adoption of TIRx and CuTe DSL.

Table 2: How PithTrain realizes each agent-native design principle.

Principle	Realization
Compact codebase	All boxes in Figure 3; line counts sum to $\sim$11K
Python-native codebase	Python throughout, including DSLs for kernels
No implicit indirection	Flat structure; self-contained model files
Task-specific agent skills	Specialized skills under the project root

PithTrain supports standard MoE training[3] with pipeline parallelism (PP), data parallelism (DP) via FSDP [54], context parallelism (CP) [23], expert parallelism (EP) [19], FP8 training [25], and DCP checkpointing [45], on NVIDIA Hopper and Blackwell GPUs. Beyond pretraining, PithTrain serves as a training backend for RL in slime [56], supporting asynchronous GRPO [42]. Despite its compact, Python-native codebase, PithTrain aims for training throughput competitive with production frameworks by adopting established MoE optimizations. We describe their integration for transparency and reproducibility; the optimizations themselves are not a separate contribution.

- **DualPipeV pipeline schedule.** PithTrain builds on DualPipe from DeepSeek-V3 [12], whose open-source version provides only the pipeline scaffold; PithTrain adds the compute–communication overlap. Each layer is split into five stages at EP communication boundaries, and the schedule overlaps the forward of one micro-batch with the backward of another to hide EP all-to-alls.
- **Agent-evolved GPU kernels.** We use agents to evolve fast kernels for Hopper and Blackwell GPUs with workflows such as TIRx Harness [27] and KDA [32]. Each evolved kernel is validated against a reference implementation before adoption, and kernels live in one file per architecture, so the code an agent edits stays small and self-contained.
- **Torch compile.** PithTrain applies `torch.compile(fullgraph=True)` to all transformer computation except the MoE forward and backward. This strict mode rejects graph breaks at compile time rather than silently degrading speedup. We exclude the MoE forward and backward because per-expert input shapes are data-dependent under expert parallelism.
- **Other optimizations.** PithTrain also implements MoE parallel folding [22] to decouple attention and expert parallelism; EP dispatch deduplication for lower all-to-all communication volume; FP8 parameter all-gather in FSDP to halve weight-gather traffic; and fused kernels written in TIRx [15] and CuTe DSL [31] for EP token scatter and FP8 quantization.

3.3 Agent Skills

A skill encodes the procedure for one recurring training-framework task. PithTrain ships a suite of skills covering several common ones (Table 3). Each skill is a self-contained folder with a top-level `SKILL.md` playbook, optionally additional markdown documents, and optionally helper scripts. Some skills are pure markdown, like `add-new-model`; others bundle scripts that offload deterministic work. Skills are composable: task-specific skills can invoke shared utility skills to reuse common procedures. These skills complement general improvements to agents and harnesses with framework-specific procedures maintained alongside the code.

Each skill in PithTrain is designed around three properties: *specific scope*, *explicit prerequisites*, and *verifiable success*. Figure 4 illustrates these on `validate-correctness`. The description and triggers together encode the skill's specific scope. The prerequisites section enumerates environment, data, and configuration assumptions, so missing state is caught before the skill begins to run. The procedure ends in a script call that returns a reproducible PASS/FAIL verdict rather than the agent's self-assessment. Skills designed around these properties are not technically hard to author, and we expect other training frameworks to ship comparable coverage as the practice matures.

4 ATE-Bench: A Benchmark for Agent-Task Efficiency

Evaluating the agent-task efficiency of a training framework requires varying the framework while holding the agent and task fixed. This is the inverse of standard benchmarks for AI coding agents [16, 8, 5], which hold the codebase and task fixed and vary the agent to score capability. We introduce

[3] PithTrain also supports dense (non-MoE) models, with MoE-specific machinery (e.g., expert parallelism) skipped.

Table 3: Representative skills in PithTrain.

Agent Skill	Example Usage
`add-new-model`	Port a new MoE architecture
`capture-nsys-profile`	Inspect a training step
`analyze-nsys-profile`	Find exposed communication
`validate-correctness`	Verify a refactor keeps loss
`validate-performance`	Benchmark an optimization
`launch-with-slurm`	Run multi-node training

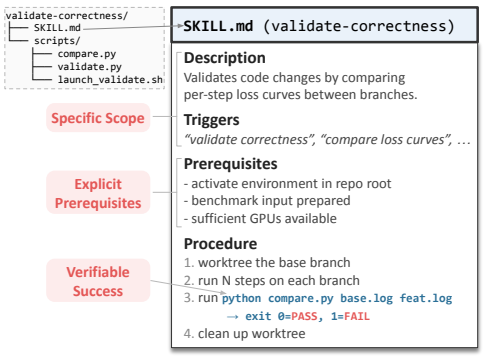


Figure 4: The `validate-correctness` skill in PithTrain. Pink labels mark properties.

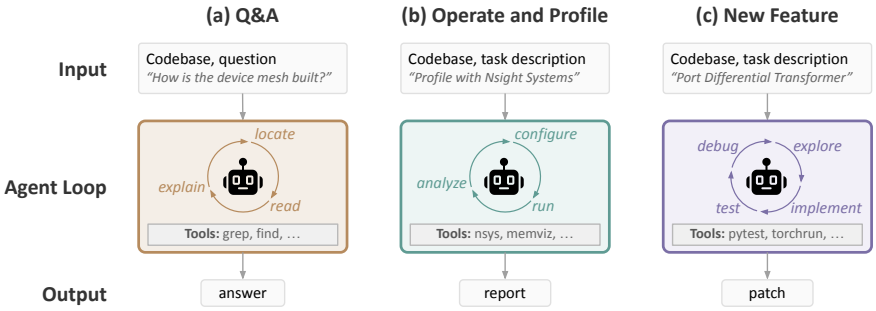


Figure 5: Agent loop for each ATE-Bench task category, from read-only Q&A to code-modifying New Feature tasks. Categories differ in loop steps, tools, and output.

ATE-Bench, a benchmark with a fixed agent and curated task suite run across frameworks to compare their agent-task efficiency. The suite spans the kinds of work researchers typically perform on training frameworks, organized around three recurring patterns: understanding the framework without modifying it, operating it as a tool for instrumentation and profiling, and extending it with new functionality. Tasks are distributed across three categories:

- Q&A (12 tasks): answer questions whose answer is a property of the code, not a runtime measurement (e.g., "how is the device mesh built?").
- Operate and Profile (4 tasks): run, instrument, and profile the framework as a tool (e.g., capture an Nsight Systems profile and identify the most expensive CUDA kernels).
- New Feature (4 tasks): port a new model architecture into the framework end-to-end against a published reference implementation (e.g., Mixture of Block Attention (MoBA) [24]).

Figure 5 illustrates the agent loop for each category, with agent involvement deepening from Q&A (read-only) to Operate and Profile (running, minor instrumentation) to New Feature (substantial modification, test-driven iteration). Full task descriptions and per-category correctness checks are in Appendix C. Using ATE-Bench, we evaluate PithTrain and production frameworks, reporting five effort metrics: session duration, active GPU time, agent turns, per-turn context, and output tokens. Without a single-scalar metric for agent-task efficiency, we report each dimension independently.

Q&A questions are chosen to be valid across all three frameworks, excluding framework-specific behaviors. New-feature tasks come from independent publications and reference implementations; none was integrated into any evaluated framework at study time. ATE-Bench does not cover tasks like cross-model propagation of a shared change, where production frameworks' implicit indirection may lower agent effort; we leave these as future work.

5 Evaluation

We evaluate PithTrain on both axes of dual efficiency: training efficiency and agent-task efficiency. We validate training correctness against Megatron-LM on both pretraining loss curves and down-

Table 4: Training throughput across frameworks. We report the aggregate tokens per second as the training throughput. Parentheses give BF16 MFU [9] against a 989.4 TFLOPS dense BF16 peak per H200; Appendix F gives the FLOP count.

Model	Hardware	Parallelism / SeqLen / Precision	Megatron-LM	TorchTitan	PithTrain
DeepSeek-V2-Lite	1×8-H200	PP1,DP8,CP1,EP8 / 4K / BF16	204.3K (42.3%)	171.0K (35.5%)	193.5K (40.1%)
Qwen3-30B-A3B	2×8-H200	PP2,DP8,CP1,EP8 / 4K / BF16	219.0K (31.9%)	170.2K (24.8%)	227.2K (33.1%)
Qwen3-30B-A3B	2×8-H200	PP2,DP2,CP4,EP8 / 32K / BF16	100.9K (36.3%)	89.6K (32.2%)	107.2K (38.5%)
GPT-OSS-120B	4×8-H200	PP4,DP8,CP1,EP8 / 4K / BF16	209.1K (21.6%)	238.2K (24.6%)	312.4K (32.3%)
Qwen3.5-35B-A3B	1×8-B200	PP1,DP1,CP8,EP8 / 64K / FP8	86.6K	47.3K	118.3K

stream accuracy in Appendix A. We also evaluate asynchronous GRPO on Qwen3-30B-A3B-Base, where PithTrain achieves training reward and AIME 2024 accuracy comparable to Megatron-LM (Appendix B). This section is organized to answer the following questions:

- Does PithTrain deliver competitive training throughput against production frameworks? (§5.1)
- Does PithTrain offer higher agent-task efficiency than production frameworks? (§5.2)
- How much do agent skills improve agent-task efficiency on PithTrain? (§5.3)
- Where do the per-framework differences in agent cost come from on a single concrete task? (§5.4)

5.1 Training Efficiency

We compare PithTrain (`cdf6f0e`), Megatron-LM [28] (`685fb8b`) and TorchTitan [21] (`45e33cd`) on four representative MoE models: DeepSeek-V2-Lite [11], Qwen3-30B-A3B [47], Qwen3.5-35B-A3B, and GPT-OSS-120B [33], under matched parallelism (PP, DP, CP, EP), sequence length, and precision. Configurations span single-node and multi-node deployments on NVIDIA H200 and B200 GPUs. For Megatron-LM, we follow NVIDIA's documented best practices[4]. DeepSpeed is excluded as it does not currently support PP combined with EP for MoE training[5].

To ensure the MoE router exhibits steady-state load-balanced routing across frameworks and thus comparable throughput, we load public model checkpoints rather than random initializations. Each experiment runs 25 steps, and we report the median step time over the last 10. We omit Model FLOPs Utilization (MFU) [9] for the FP8 configuration because Tensor Core peak FLOPS differs between BF16 and FP8, making the metric ambiguous for mixed-precision steps. Training hyperparameters follow Appendix A.

As Table 4 shows, PithTrain exceeds both baselines on four of five configurations and stays within 5.3% of Megatron-LM on DeepSeek-V2-Lite. This performance comes from optimizations such as DualPipeV's compute–communication overlap, agent-evolved fast GPU kernels, and `torch.compile(fullgraph=True)`. These results demonstrate that a compact, Python-native codebase can achieve competitive training throughput.

5.2 Agent-Task Efficiency

In this section, we evaluate agent-task efficiency across frameworks. We run ATE-Bench (§4) on Megatron-LM, TorchTitan, and PithTrain with Claude Code (Opus 4.7 at `xhigh` effort[6]) as the fixed agent. Operate-and-profile and new-feature tasks run on a single node with eight NVIDIA H100 GPUs. Each task runs three times and we report medians; task descriptions, correctness criteria, and per-attempt values are described in Appendices C and D.1. Opus 4.7 completed every task across all attempts and frameworks. We repeat the suite with Codex and report its results and run-to-run variation in Appendix D.2 and D.3.

Q&A. Answering a question requires locating where a behavior lives in the codebase. We omit Session Duration and Active GPU Time (no training runs). All 12 questions are answered correctly across attempts and frameworks (grading details in Appendix C.1.2). Across the 12 questions, the agent uses up to 67% fewer Agent Turns to reach the final answer on PithTrain than on Megatron-LM. The lower Per-Turn Context is consistent with a smaller search space (Table 5).

[4] `https://docs.nvidia.com/nemo/megatron-bridge/nightly/training/moe-optimization.html`

[5] Verified at commit `44c51e3`: the Megatron-DeepSpeed integration script states "Currently we don't support PP for MoE."

[6] `https://platform.claude.com/docs/en/build-with-claude/effort`

Table 5: Per-question agent effort on the Q&A task suite (§4); Q1–Q12 are described in Appendix C.1. Each metric reports the median of three independent attempts. Lower is more efficient. Headers abbreviate Megatron-LM, TorchTitan, and PithTrain.

#	Agent Turns			Per-Turn Context			Output Tokens		
	Megatron	Titan	Pith	Megatron	Titan	Pith	Megatron	Titan	Pith
Q1	33	18	**15**	45.8K	44.3K	**33.4K**	9.0K	7.1K	**4.1K**
Q2	54	28	**18**	41.2K	43.5K	**35.9K**	10.7K	7.8K	**5.4K**
Q3	14	14	**13**	**31.3K**	36.1K	34.1K	**3.4K**	3.5K	3.4K
Q4	36	26	**16**	41.3K	39.7K	**31.1K**	10.0K	6.9K	**5.2K**
Q5	48	**19**	21	46.5K	41.7K	**33.8K**	10.7K	4.8K	**4.6K**
Q6	12	**4**	11	36.4K	29.9K	**27.9K**	4.5K	**2.3K**	3.3K
Q7	**9**	12	14	31.5K	**30.0K**	31.5K	**2.6K**	3.4K	4.2K
Q8	20	11	**9**	30.8K	32.7K	**28.7K**	4.7K	3.5K	**3.2K**
Q9	34	**16**	**16**	45.2K	38.6K	**37.0K**	7.6K	6.0K	**5.1K**
Q10	19	17	**11**	37.8K	38.8K	**29.4K**	5.9K	5.1K	**2.9K**
Q11	10	8	**6**	31.3K	30.1K	**25.6K**	3.1K	2.5K	**2.0K**
Q12	18	**5**	10	37.1K	33.9K	**31.1K**	5.1K	**2.2K**	2.6K

Table 6: Per-task agent effort across frameworks on operate-and-profile tasks (§4). Each metric reports the median of three independent attempts. Lower is more efficient. Session Duration and Active GPU Time are in minutes. Per-attempt metrics in Appendix D.1.

Task	Framework	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Getting Started	Megatron-LM	40.5	5.4	88	69.5K	26.9K
	TorchTitan	11.4	5.2	54	56.8K	15.8K
	PithTrain	**6.6**	**3.1**	**26**	**37.0K**	**5.8K**
Train and Evaluate	Megatron-LM	55.5	36.0	163	106.4K	52.9K
	TorchTitan	72.5	36.3	212	176.6K	97.8K
	PithTrain	**38.5**	**22.7**	**92**	**85.5K**	**34.2K**
Collect Routing Trace	Megatron-LM	33.3	5.5	112	144.3K	102.1K
	TorchTitan	32.8	10.4	103	166.0K	84.7K
	PithTrain	**16.3**	**2.8**	**58**	**118.7K**	**56.2K**
Report Heavy Kernels	Megatron-LM	22.1	12.1	60	52.6K	23.9K
	TorchTitan	15.0	6.7	**40**	66.1K	22.5K
	PithTrain	**11.8**	**3.6**	42	**49.2K**	**16.0K**

Operate and Profile. Across all tasks (Table 6), PithTrain's Agent Turns are up to 70% lower than Megatron-LM's and 57% lower than TorchTitan's, and its Output Tokens are up to 78% and 65% lower respectively. These comparisons evaluate each framework's codebase together with its in-repo documentation and skills. In addition, the agent invokes in-repo skills (§3.3) on its own when applicable; for example, the *Report Heavy Kernels* task triggers `capture-nsys-profile`.

New Feature. New-feature tasks exercise the test–debug cycle: edit, run training, read crash, edit again. Across new-feature tasks (Table 7), PithTrain's Active GPU Time is up to 44% lower than Megatron-LM's and 64% lower than TorchTitan's. Debugging TorchTitan's OOM failures on MoBA contributes to the latter gap. Excluding tasks where TorchTitan runs out of memory, the maximum reduction is 32% on Diff. Two patterns inflate Megatron-LM's reruns: a hidden argument registry causes the agent's manually-added CLI flags to collide with auto-derived ones (implicit indirection), and C++ paths like TransformerEngine's grouped-GEMM emit opaque segfaults that drive speculative configuration toggles (not Python-native). TorchTitan's reruns are dominated by memory-pressure debugging. On PithTrain, failures surface inside the file the agent just wrote with a readable Python traceback, and fixes stay in the same file. §5.4 provides a detailed case study on MoBA.

5.3 Ablation Study on Agent Skills

In this section, we isolate the effect of agent skills via ablation. They are a self-contained set of files shipped in the repository, so we can toggle them on and off against an otherwise fixed codebase.

Table 7: Per-task agent effort across frameworks on new-feature tasks (§4). Each metric reports the median of three independent attempts. Lower is more efficient. Session Duration and Active GPU Time are in minutes. Per-attempt metrics in Appendix D.1.

Task	Framework	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Differential Transformer (Diff) [50]	Megatron-LM	47.1	33.7	125	118.7K	57.1K
	TorchTitan	49.6	40.3	58	103.2K	36.0K
	PithTrain	**38.2**	**27.6**	**47**	**69.5K**	**25.4K**
Dynamic Mixture of Experts (DynMoE) [14]	Megatron-LM	83.8	49.1	199	208.0K	115.2K
	TorchTitan	140.6	94.4	197	228.8K	161.3K
	PithTrain	**60.4**	**41.9**	**76**	**146.0K**	**76.4K**
Mixture of Block Attention (MoBA) [24]	Megatron-LM	61.6	49.5	134	120.9K	53.8K
	TorchTitan	105.1	77.9	91	166.9K	111.8K
	PithTrain	**38.7**	**27.7**	**57**	**69.0K**	**32.4K**
MoE++ [17]	Megatron-LM	88.5	58.7	145	188.7K	117.0K
	TorchTitan	71.4	51.9	**87**	**164.6K**	**85.3K**
	PithTrain	**63.0**	**39.9**	90	176.6K	107.7K

Table 8: Agent effort on PithTrain with and without the corresponding skill. Each metric reports the median of three independent attempts. Session Duration and Active GPU Time are in minutes.

Task	Skills	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
`validate-correctness`	off	26.0	20.8	114	96.3K	30.2K
	on	22.9	22.5	34	43.5K	11.3K
`capture-nsys-profile`	off	9.4	5.6	75	62.1K	25.3K
	on	6.6	5.5	36	40.3K	14.5K

We pick two of PithTrain's skills, `validate-correctness` and `capture-nsys-profile`, which mirror the natural follow-up after a system optimization: validate that training correctness is preserved, then capture an Nsight Systems profile to examine whether the optimization is effective.

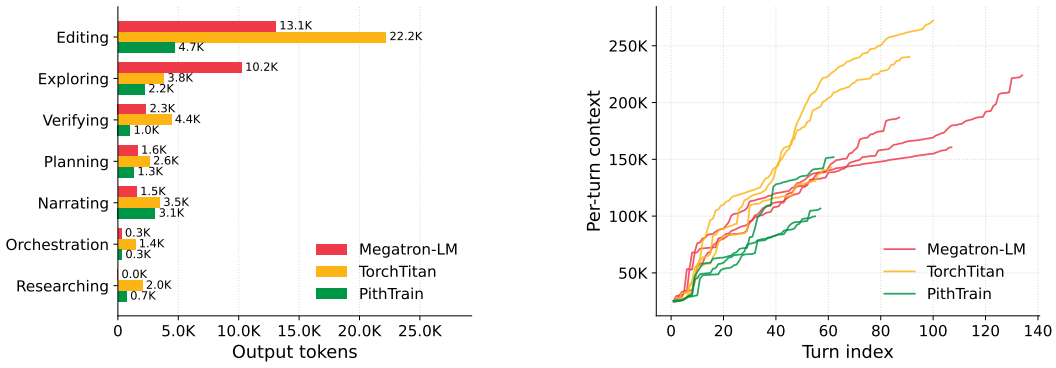
We run this ablation on the wgrad delay [37] commit in PithTrain, repeating each task three times with skills and three times without. The codebase, agent, and harness are otherwise identical, and we report the same five effort metrics as in §5.2. When skills are disabled, we strip them from both the working tree and the git history, so the agent cannot recover the procedure from either. All twelve runs completed successfully, reporting the correct verdict or generating a valid Nsight Systems profile.

Table 8 reports the results. Active GPU Time stays near parity across both tasks: each task runs a fixed set of training runs pinned by the workflow, so the GPU work is determined by the task rather than the agent. The four agent-side metrics, which capture the agent's reasoning overhead in setup, launch, and interpretation, all drop substantially with skills enabled. Agent turns drop the most (70% and 52% respectively), suggesting that with the procedure encoded in the skill, the agent acts on a fixed plan rather than iteratively deriving one through repeated tool calls. These results demonstrate that task-specific in-repo skills, comprising the markdown playbook and any helper scripts, substantially reduce agent effort on the recurring training-system tasks they target. We also evaluate profiling skills on Megatron-LM and TorchTitan in Appendix E.

5.4 Case Study: Integrating MoBA

To examine where per-framework differences in agent cost come from, we conduct a case study on integrating MoBA, decomposing the agent's output tokens by action category[7] (Figure 6a) and tracing its per-turn context window (Figure 6b). Editing dominates across all three frameworks (PithTrain 4.7K, Megatron-LM 13.1K, TorchTitan 22.2K). Megatron-LM also spends substantially more on Exploring (10.2K vs. 3.8K for TorchTitan and 2.2K for PithTrain), and its per-turn context sits well

[7]Thinking tokens are excluded as they are not surfaced by the API.



(a) Per-category breakdown of output tokens.

(b) Per-turn context window over the session.

Figure 6: Agent behavior on integrating MoBA across frameworks. (a) reports the median output tokens per action category of three independent attempts; (b) shows the per-turn input-side context for each of three independent attempts.

above PithTrain's. The agent reads the codebase to locate edits and interpret tracebacks, so a compact codebase with no implicit indirection lowers both Exploring and per-turn context. TorchTitan's elevated Editing and ~2× context spike in two of three runs have a different cause: the agent's initial implementation runs out of memory (OOM), forcing repeated debug-edit cycles. This points to runtime properties like memory headroom as a factor independent of codebase structure.

Beyond TorchTitan's memory failures, we examine the other failures the agent encountered. Across the three PithTrain runs, two complete without any failure; the third hits a tensor-stride mismatch in the agent's custom attention kernel, fixed in the same file as the traceback. Megatron-LM's three runs hit two distinct failures: two runs fail with a duplicate command-line flag registration that conflicts with one defined in framework code, and one run fails with a BF16 overflow in the agent's code. Each fix on Megatron-LM spans multiple files. This contrast reflects PithTrain's compactness and absence of implicit indirection, which keep each fix local to the agent's edit.

6 Discussion and Limitations

The four principles interact, and our comparisons evaluate their combined effect. ATE-Bench does not yet cover propagating shared changes across models, where implicit indirection may lower agent effort. Maintaining ATE as a framework grows is another open direction. For example, we use agent reviewers to flag compactness concerns and encourage agents to refine skills after use. Periodic ATE-Bench evaluation and focused model coverage could further help manage growth. We hope PithTrain and ATE-Bench help the community explore how to preserve agent-task efficiency as training frameworks evolve.

7 Conclusion

We presented PithTrain, a compact, agent-native MoE training framework built on four design principles. PithTrain offers competitive throughput across a range of models, and on ATE-Bench, a coding agent achieves higher agent-task efficiency on PithTrain than on production frameworks. We hope PithTrain serves as a starting point for future agent-native training frameworks.

Acknowledgments and Disclosure of Funding

This work is supported in part by the National Science Foundation under grant number IIS2543593, funding from Bosch, Amazon, and gifts from NVIDIA and Google. We thank the CMU Foundation and Language Model (FLAME) Center for providing compute resources to develop PithTrain, and NVIDIA for DGX B200 support. We also thank Zichu Wu, Gaokai Zhang, Rishith Prathi, and Yagneek Patlola for their contributions to the ongoing development of PithTrain.

References

[1] Anthropic. Claude Code. https://www.anthropic.com/claude-code, 2025.

[2] Anthropic. Equipping agents for the real world with Agent Skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, 2025.

[3] Anysphere. Cursor: The AI code editor. https://www.cursor.com, 2023.

[4] Artificial Analysis. Coding Agent Index. https://artificialanalysis.ai/agents/coding-agents/, 2026.

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

[6] Yonatan Bisk, Rowan Zellers, Ronan Le bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[7] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of machine learning research*, 24(240):1–113, 2023.

[10] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[11] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan

Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[12] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[13] GitHub. GitHub Copilot. `https://github.com/features/copilot`, 2021.

[14] Yongxin Guo, Zhenglin Cheng, Xiaoying Tang, Zhaopeng Tu, and Tao Lin. Dynamic mixture of experts: An auto-tuning approach for efficient transformer models. *arXiv preprint arXiv:2405.14297*, 2024.

[15] Bohan Hou, Hongyi Jin, Guanjie Wang, Shushi Hong, Jinqi Chen, Yaxing Cai, Lijie Yang, Zihao Ye, Yaoyao Ding, Ruihang Lai, Todd C. Mowry, and Tianqi Chen. TIRx: A unified tile primitive ml compiler across heterogeneous backends. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '27, page 955–969, New York, NY, USA, 2027. Association for Computing Machinery.

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

[17] Peng Jin, Bo Zhu, Li Yuan, and Shuicheng YAN. Moe++: Accelerating mixture-of-experts methods with zero-computation experts. In *The Thirteenth International Conference on Learning Representations*, 2025.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[19] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. {GS}hard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021.

[20] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, et al. DataComp-LM: In search of the next generation of training sets for language models. *arXiv preprint arXiv:2406.11794*, 2024.

[21] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. Torchtitan: One-stop pytorch native solution for production ready LLM pretraining. In *The Thirteenth International Conference on Learning Representations*, 2025.

[22] Dennis Liu, Zijie Yan, Xin Yao, Tong Liu, Vijay Korthikanti, Evan Wu, Shiqing Fan, Gao Deng, Hongxiao Bai, Jianbin Chang, Ashwath Aithal, Michael Andersch, Mohammad Shoeybi, Jiajie Yao, Chandler Zhou, David Wu, Xipeng Li, and June Yang. MoE parallel folding: Heterogeneous parallelism mappings for efficient large-scale MoE model training with megatron core, 2026.

[23] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*, 2024.

[24] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, et al. Moba: Mixture of block attention for long-context llms. *arXiv preprint arXiv:2502.13189*, 2025.

[25] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022.

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

[27] MLC Community. TIRx Harness: An open compiler harness for agentic GPU programming. `https://blog.mlc.ai/2026/09/29/tirx-harness-an-open-compiler-harness-for-agentic-gpu-programming`, September 2026.

[28] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, Sen Hu, Ronghao Chen, Bo Li, Xin Li, Chen Hu, Binxing Jiao, Daxin Jiang, and Pin Lyu. GitTaskBench: A benchmark for code agents solving real-world tasks through code repository leveraging, 2025.

[30] NVIDIA. TransformerEngine. `https://github.com/NVIDIA/TransformerEngine`, 2022.

[31] NVIDIA. CuTe DSL: DSL programming model. `https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html`, 2026.

[32] NVIDIA. Kernel design agents (KDA). `https://github.com/NVlabs/kda`, 2026.

[33] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b Model Card, 2025.

[34] OpenAI. Codex CLI. `https://github.com/openai/codex`, 2025.

[35] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025.

[36] Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions?, 2026.

[37] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024.

[38] Zihan Qiu, Zeyu Huang, Bo Zheng, Kaiyue Wen, Zekun Wang, Rui Men, Ivan Titov, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Demons in the detail: On implementing load balancing loss for training specialized mixture-of-expert models. *arXiv preprint arXiv:2501.11873*, 2025.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2020.

[42] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, 2024.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[44] Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026.

[45] The PyTorch Team. torch.distributed.checkpoint. `https://docs.pytorch.org/docs/stable/distributed.checkpoint.html`, 2023.

[46] Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems, 2026.

[47] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[48] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[49] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

[50] Tianzhu Ye, Li Dong, Yuqing Xia, Yutao Sun, Yi Zhu, Gao Huang, and Furu Wei. Differential transformer. *arXiv preprint arXiv:2410.05258*, 2024.

[51] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole

Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[52] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[53] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery.

[54] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.

[55] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[56] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. `https://github.com/THUDM/slime`, 2025. GitHub repository. Corresponding author: Xin Lv.

Hardware	4×8-H100
Parallelism	PP4, DP1, CP1, EP8
Dataset	DCLM [20]
Sequence	2048
Precision	BF16
Global Batch	1024
Training Steps	4096
Warmup Steps	128
Optimizer	Adam [18]
Max LR	2×10^{-4}
Min LR	1×10^{-5}
LR Schedule	Cosine Decay
Aux Loss	Global-Level [38]
Aux Coef	1×10^{-3}

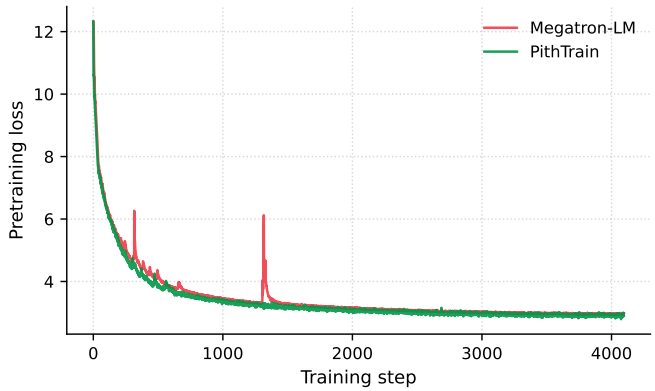


Figure 7: Training configuration (left) and pretraining loss curves (right) for Qwen3-30B-A3B trained with Megatron-LM and PithTrain.

A Pretraining Correctness

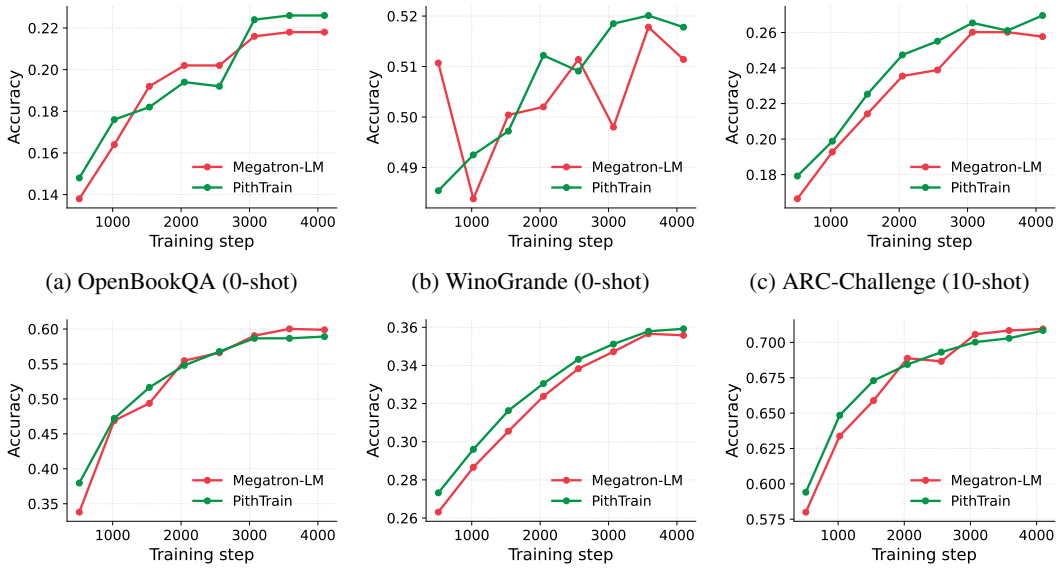
This appendix validates the pretraining correctness of PithTrain against Megatron-LM at matched configuration. We report two complementary measurements: pretraining loss trajectories (§A.1) and downstream task accuracy (§A.2).

A.1 Pretraining Loss

We pretrain Qwen3-30B-A3B with Megatron-LM and PithTrain under identical configuration. Figure 7 reports the full training configuration alongside the two loss curves; the trajectories align across the full run, with Megatron-LM showing two transient spikes that recover within a few steps.

A.2 Downstream Accuracy

We evaluate downstream task accuracy across six standard benchmarks: OpenBookQA [26] and WinoGrande [41] in the 0-shot setting, and ARC-Challenge, ARC-Easy [10], HellaSwag [52], and PIQA [6] in the 10-shot setting. Figure 8 plots accuracy for each task; Megatron-LM and PithTrain achieve comparable accuracy within statistical noise across all six benchmarks at every checkpoint.



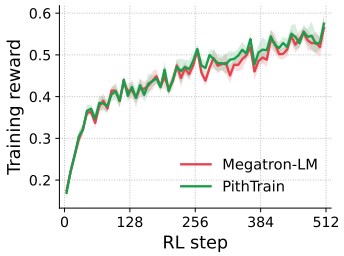
(a) OpenBookQA (0-shot) (b) WinoGrande (0-shot) (c) ARC-Challenge (10-shot)

(d) ARC-Easy (10-shot) (e) HellaSwag (10-shot) (f) PIQA (10-shot)

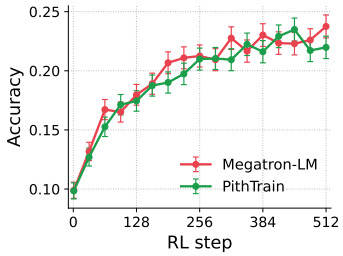
Figure 8: Downstream task accuracy for Qwen3-30B-A3B trained with Megatron-LM and PithTrain.

Hardware	2×8-H200
Parallelism	PP2, DP4, EP4
Dataset	DAPO-Math-17K [51]
Algorithm	Async GRPO
Batch	64 prompts × 8
Max Response	8192
Training Steps	512
Optimizer	AdamW
LR	1×10^{-6}
PPO Clip	0.2 / 0.28

(a) Configuration



(b) Training reward



(c) AIME 2024

Figure 9: Asynchronous GRPO on Qwen3-30B-A3B-Base with Megatron-LM and PithTrain. Training reward is an 8-step mean ± s.e.; AIME 2024 uses 30 problems × 64 samples.

B Reinforcement Learning

We integrate PithTrain as a training backend in slime [56], which pairs a trainer with SGLang [55] rollout engines, and run asynchronous GRPO [42] on Qwen3-30B-A3B-Base with Megatron-LM and PithTrain under identical configuration (Figure 9a). Rollouts are generated on a separate H200 node, and the trainer pushes updated weights to the engines after every step.

Identical data. We first record 32 asynchronous batches and replay them through both trainers, so that both optimize the same objective on the same data. Across the 32 steps, the per-step policy entropy, policy-gradient loss, gradient norm, clip fraction, and KL divergence show no growing discrepancy between backends. Policy entropy differs by about 0.05% between backends, compared with about 0.03% between two PithTrain launches.

End-to-end training. We then train each backend independently for 512 steps. Figure 9b–c show that training reward and AIME 2024 accuracy track each other throughout; the mean reward over the run is 0.454 for PithTrain and 0.447 for Megatron-LM. On a paired evaluation of the final checkpoints (30 AIME 2024 problems, 64 samples each), PithTrain and Megatron-LM reach 0.226 and 0.231 against 0.104 for the base model. The difference, -0.005 ± 0.007 (standard error; 95% CI $[-0.020, 0.008]$), is not significant, while both backends improve on the base model by about 12–13 percentage points.

C Task Suite

This appendix expands the per-category task descriptions and the correctness checks used to validate each attempt. All operate-and-profile and new-feature tasks share a fixed training configuration: the base model is DeepSeek-V2-Lite [11], the parallelism mesh is PP=4, EP=2, DP=1, sequence length 2048, global batch size 1024, precision BF16.

C.1 Q&A Tasks

Each Q&A task asks the agent to locate where a specific behavior lives in the framework codebase. The agent receives a single prompt consisting of the *universal instruction* below followed verbatim by the full query text for that task (Q1–Q12 in §C.1.1), and has read-only access to `Read`, `Grep`, `Glob`, and `Bash`; tools that modify the working tree (`Edit`, `Write`, `NotebookEdit`) are explicitly disabled. Correctness is validated by two independent human graders (§C.1.2).

Universal instruction (prepended to every query). Your task is to explore this training framework codebase and answer the question that follows. Every claim must be verified against the code on disk before you state it: use your CLI tools (Read, Grep, Glob, Bash) to locate the exact symbol, file, and line, then cite the `path/to/file.py:LINE` you actually read. Wrap your final response in `<final_answer>` tags, trace the execution path step by step, and give one citation per claim. If a feature is genuinely absent from this codebase, say so explicitly and cite negative evidence (the grep command and pattern that returned zero matches, or the file you inspected that does not contain it). A correct "absent, verified" answer is preferred over a fabricated one.

C.1.1 Task Descriptions

Q1: Process Groups / Device Mesh. Trace the sequence of function calls from the main training entry script down to the initialization of the PyTorch distributed process groups or DeviceMesh. Detail the exact file paths, function names, and line numbers where the world size and ranks for the parallel groups present in this codebase (any of DP, TP, PP, EP, CP that this repo actually supports) are assigned.

Q2: Configuration Propagation. Locate the exact file and line numbers where the user-provided configuration for `hidden_size` or `dim` is parsed. Trace how this specific variable propagates down to the instantiation of the first Transformer block.

Q3: Data Loading & Sharding. Trace the initialization of the dataset and dataloader. Identify the file, class, and line number where the global dataset is sliced or partitioned among the Data Parallel ranks to ensure each rank receives a unique, non-overlapping subset of data.

Q4: Distributed Seed Management. Find where the random seeds are set for data loading versus model initialization. Provide the exact file paths and line numbers, and trace how seeds are differentiated across different parallel groups (or, if they are deliberately shared, where and why).

Q5: Attention Kernel Dispatch. Locate the core attention module. Trace the logic that selects between an optimised kernel (FlashAttention, PyTorch SDPA flash backend, ring attention, etc.) and any fallback math implementation. Identify the exact file, class, line number, and configuration flag/condition controlling this dispatch. If the dispatch is delegated to the PyTorch `scaled_dot_product_attention`, say so and cite the call site.

Q6: RoPE Implementation. Find where the Rotary Positional Embedding (RoPE) is implemented. Locate the function that precomputes the frequency tensor (sine/cosine cache). Provide the file path, class/function name, and exact line numbers of the tensor operations where the rotary transform is applied to the Query and Key tensors.

Q7: SwiGLU / MLP Block. Locate the Feed-Forward Network (FFN) implementation. Identify the file and line number where the SwiGLU activation (or equivalent gated linear unit) is mathematically applied. Trace how the up-projection tensor is chunked or split before the activation function.

Q8: Normalization Placement. Find the implementation of the main Transformer layer. Provide the file and exact line numbers where the normalization (RMSNorm or LayerNorm) is applied before the attention module. Detail where the `epsilon` (eps) term is added for numerical stability.

Q9: Context / Sequence Parallelism. Find where sequence parallelism (or context parallelism) is handled. Identify the exact file and line numbers where the sequence dimension of the input tensor is sharded, gathered, or scattered across ranks during the forward and backward passes. If the codebase does not support CP/SP, say so and cite the negative evidence.

Q10: FSDP / DDP Wrapping. Locate the exact file and line numbers where the core model is wrapped with Fully Sharded Data Parallel (FSDP1 or FSDP2) or Distributed Data Parallel (DDP) wrappers, and identify the device-mesh axes used for sharding/replication.

Q11: Global Gradient Clipping. Trace the logic for global gradient norm clipping. Find the file, function, and exact line number where the global norm of the gradients is computed (including any reduction across distributed ranks) before the optimizer step.

Q12: Distributed Checkpoint Serialization. Locate the model checkpoint saving logic. Detail whether the system uses a rank-0 gather approach or distributed sharded saving (e.g., PyTorch DCP). Provide the exact file path and line numbers where the disk serialization occurs.

C.1.2 Correctness Checks

Each Q&A answer the agent returns cites the file paths, function names, and line numbers in the framework codebase that implement the behavior the question asks about. Two human graders independently trace these citations: they open each cited code location and verify that the code there does what the agent claims in the answer. An attempt is recorded as *satisfied* when both graders confirm every citation against the code on disk; disagreements are resolved by a third reader who looks only at the cited evidence. All 108 attempts (12 questions $\times$ 3 frameworks $\times$ 3 attempts) were judged satisfied by both graders.

C.2 Operate-and-profile Tasks

Each operate-and-profile task asks the agent to drive a real training-system workflow end-to-end, so success requires the agent to set up the environment, run the framework as intended, and produce the expected artifact. The four tasks span the workflows a researcher typically performs before any code change: getting the framework running, executing a full train-and-evaluate pipeline, instrumenting the model to capture behavior, and profiling the system to surface bottlenecks.

C.2.1 Task Descriptions

Getting Started. Set up the Python environment for the framework and run the provided 5-step smoke training script. Success means the script reaches step 5 with a finite loss. The agent must install all dependencies the script needs so that running `bash train.sh` as-is succeeds; `train.sh` itself documents best practices for training MoE models and is read-only. Pre-tokenized DCLM and the converted base-model checkpoint are pre-staged.

Train and Evaluate. Drive the full setup-train-export-evaluate pipeline for the base model. The agent trains from random initialization for 25 steps, exports the resulting checkpoint to HuggingFace format, and runs lm-evaluation-harness HellaSwag (zero-shot) on it via vLLM. The task tests pipeline correctness, not model quality: the HellaSwag score is expected to be near-random because 25 steps from random init produces a barely-trained model, and the agent must produce *some* score from a working pipeline. Initialization must be random; everything outside the fixed mesh and step count (LR, optimizer, scheduler, data preprocessing) is left to the agent.

Collect Routing Trace. Instrument training to dump the per-token MoE routing trace for the first 8M training tokens. The routing decision in each MoE layer, the top-k expert IDs and their gating weights, must be captured for every token in the global batch. Training resumes from the released HuggingFace checkpoint so the router is in its trained, load-balanced regime; routing decisions are model-intrinsic and valid from step 1, so no warmup is required. The output schema is one `step-<step_id:08d>.npz` file per step under `workspace/<framework>/routing-traces/`, each containing the expert-ID and gating-weight arrays.

Report Heavy Kernels. Profile a 7-step training run with Nsight Systems and identify the top 3 most expensive CUDA kernels by total GPU time, aggregated across all 8 ranks. Training resumes from the released HuggingFace checkpoint; the agent profiles only step 7 because steps 1–6 are warmup for cudagraph capture, NCCL handshake, and allocator priming. The output is a single CSV at `workspace/<framework>/heavy-kernels/top-kernels.csv` with header `kernel_name,total_time_ms,instances,mean_time_ms` and exactly three rows sorted by `total_time_ms` descending, plus the raw `profile.nsys-rep` so the result is reproducible.

C.2.2 Correctness Checks

Each operate-and-profile task verifies the artifact the agent produces, not the path the agent took to produce it. We use a mix of programmatic checks baked into the task harness and human inspection where the artifact is non-textual.

Getting Started. The task succeeds when running `train.sh` reaches step 5 with a finite loss. After the agent finishes installing dependencies, the harness re-runs the (read-only) training script and parses the resulting log; the artifact is accepted only if step 5 prints a finite loss value.

Train and Evaluate. The harness ships a read-only `evaluate.sh` script that consumes the pipeline output produced by the agent: the agent trains for 25 steps from random init, exports the resulting checkpoint to HuggingFace format, and `evaluate.sh` loads the export into vLLM and runs lm-evaluation-harness HellaSwag (zero-shot). An attempt is satisfied when `evaluate.sh` runs to completion and writes a finite HellaSwag score; the score is not required to clear any quality threshold, since 25 steps from random initialization is expected to yield near-random accuracy.

Collect Routing Trace. The harness verifies that four `step-<step_id:08d>.npz` files are present at `workspace/<framework>/routing-traces/` and that each carries expert-ID and gating-weight arrays of the correct shape for every MoE layer over every token in the global batch. A programmatic check additionally confirms that the expert-ID values are within the valid range `[0, num_experts)` for the MoE configuration of the model, and that the gating weights are non-negative and sum to 1 over the top-k selected experts for every token.

Report Heavy Kernels. The agent submits both `top-kernels.csv` and the raw `profile.nsys-rep`. The harness checks the CSV programmatically against the prescribed schema (header, exactly three rows sorted by total time descending). The kernel names themselves are validated by a human reader who opens `profile.nsys-rep` in the Nsight Systems profiler GUI, reads the CUDA GPU Kernel Summary in the Stats System view, and confirms that the top three kernel names in the CSV produced by the agent match the top three entries shown by the profiler.

All 36 attempts (4 tasks $\times$ 3 frameworks $\times$ 3 attempts) produced an artifact accepted by both the harness and the human reader.

C.3 New-feature Tasks

Each new-feature task mimics the workflow of a researcher or ML engineer integrating a recently published modeling architecture into a training framework. The agent is given the same materials such an engineer would normally consult: the arXiv paper and the reference implementation. From these inputs, the agent must produce a training script that integrates the new feature into the base model and runs on DCLM [20] under the fixed configuration above.

The four tasks are chosen for both coverage and a fair starting point. Two revise the attention mechanism and two require changes at the FFN (MoE) layer, so the suite spans the architectural subsystems a training framework must accommodate. None of the four architectures had been integrated into any of the three frameworks we compare at the time of this study, so all frameworks start the task from the same point and the comparison reflects only how the design of each framework affects integration effort.

Correctness is validated on two axes. First, the cross-entropy loss must decrease across the 64-step run and remain finite (no explosion, no NaN), evidence that the modified training pipeline produces a learnable model. Second, the changes made by the agent must satisfy three task-specific rules, each validating one required component of the new feature (§C.3.2); the rules target the architectural elements that distinguish each new feature from the baseline framework, ensuring the agent has implemented the intended mechanism rather than producing a passing-loss reading from an unchanged model. An attempt is recorded as *satisfied* when both axes hold.

C.3.1 Task Descriptions

Diff [50]. Differential attention replaces a single softmax-attention map with the difference of two parallel softmax maps, $\mathrm{Attn}(Q, K, V) = (\mathrm{softmax}(Q_1 K_1^\top) - \lambda\,\mathrm{softmax}(Q_2 K_2^\top))\,V$, where λ is a learned per-head scalar. The construction cancels common-mode attention noise, sharpening which tokens receive mass. Integrating it involves splitting the per-head Q/K projections in half and introducing λ as a trainable parameter with the published initialisation schedule.

DynMoE [14]. Standard MoE fixes the number of activated experts k per token. DynMoE replaces top-k selection with a per-expert sigmoid gate and a learned threshold, so the count of active experts varies per token. Integrating it involves replacing the top-k routing kernel of the framework and reformulating the load-balancing auxiliary loss for a variable-k regime.

MoBA [24]. MoBA partitions the key/value sequence into fixed-size blocks and routes each query to the top-k blocks selected by query–block affinity scores, yielding sub-quadratic attention cost in the sequence length. Integrating it involves inserting block-level routing between the Q/K projection and the attention computation, composing with the existing attention backend in the framework, and preserving causal masking inside each selected block.

MoE++ [17]. MoE++ augments a standard MoE expert pool with three zero-computation expert types (*zero*, *copy*, and *constant*), allowing easy tokens to be routed past the feed-forward layer entirely. Integrating it involves extending the expert pool definition with the zero-computation variants, widening the router output to cover them, and adjusting the load-balancing loss to prevent the zero experts from absorbing all easy tokens.

C.3.2 Rule-based Correctness Checks

Each new-feature task decomposes into three required components, the architectural elements that distinguish the new feature from the baseline framework. The rules below name each component and

describe how it should be implemented; they were fixed before inspecting any attempt. Each attempt is judged by an independent `claude-opus-4-7` session at `xhigh` effort, given the three rules and the git diff produced by the agent against the main branch of the framework; the judge returns PASS or FAIL per rule with a one-sentence justification quoting a specific code reference, and an attempt is recorded as *satisfied* when all three rules pass. We additionally inspect every verdict and its cited justification by hand: a human reader opens the diff at the cited file and confirms the line evidence for both PASS judgements and any FAIL or partial verdicts. Across all 36 attempts (4 tasks $\times$ 3 frameworks $\times$ 3 attempts), every attempt satisfies all three rules of its task.

Diff.

R1. The attention forward path produces two separate softmax outputs that are combined as $\mathrm{attn}_1 - \lambda \cdot \mathrm{attn}_2$ (literal subtraction with a learnable coefficient).

R2. A learnable parameter (λ, e.g. named `lambda` or `lambda_init`) is registered as an `nn.Parameter` with shape compatible with per-head broadcasting.

R3. The Q/K projections are split into two halves, either via an output dimension of $2 \cdot n_{\mathrm{heads}} \cdot d_{\mathrm{head}}$ followed by a split, or via two separate projection layers.

DynMoE.

R1. The router uses per-expert sigmoid gates (or sigmoid-gated activation), not pure softmax with top-k selection.

R2. A learnable threshold parameter for expert activation is registered as an `nn.Parameter` (often named `tau`, `threshold`, or `gate_threshold`).

R3. Active-expert selection is not a fixed `top_k = N`; the count depends on which experts pass the threshold (boolean mask or variable-length selection).

MoBA.

R1. The K/V sequence is partitioned into fixed-size blocks (a `reshape` or `view` into shape `[..., `n_{blocks}`, `B`, ...]` or equivalent).

R2. Each query computes a per-block score (typically $\mathrm{query} \cdot \mathrm{pooled_block_key}$) followed by top-$k$ block selection.

R3. Final attention is computed only over the selected blocks, with causal masking preserved at block boundaries (each query attends only to blocks at positions $\leq$ its own).

MoE++.

R1. The expert pool includes at least one zero-computation expert type (*zero*, *copy*, or *constant*), identifiable by class name, string literal, or a specialised expert factory.

R2. The router emits logits over a pool that contains both regular FFN experts and zero-computation experts, so the router learns to dispatch tokens to the no-op variants. The zero-computation experts may be added on top of the existing pool (router output dim grows to $n_{\mathrm{FFN}} + n_{\mathrm{zero}}$) or carved out of an unchanged total expert count.

R3. The load-balancing auxiliary loss handles zero experts explicitly: they are excluded from the balance term, weighted differently, or balanced under a separate penalty.

D Per-Task Results

D.1 Claude Code

This subsection reports per-attempt agent effort for Claude Code across all 20 tasks. Tables 9 and 10 cover the 12 Q&A tasks (split 6/6 across two pages), Table 11 covers the four operate-and-profile tasks, and Table 12 covers the four new-feature tasks. Each row is one independent attempt; per-task medians appear in the corresponding main-text tables in §5.2. Session Duration and Active GPU Time are reported in minutes. Lower is better on every metric.

Table 9: Per-attempt agent effort on the Q&A tasks Q1–Q6 (§5.2). Each row is one independent attempt. Session Duration is in minutes.

Task	Framework	Attempt	Session Duration	Agent Turns	Per-Turn Context	Output Tokens
Q1: Process Groups / Device Mesh	Megatron-LM	1st	2.0	33	47.8K	9.0K
		2nd	2.2	26	45.8K	9.7K
		3rd	2.1	34	44.5K	9.0K
	TorchTitan	1st	2.8	18	44.3K	7.1K
		2nd	3.1	19	45.2K	7.0K
		3rd	3.0	15	42.9K	7.3K
	PithTrain	1st	1.6	15	33.4K	4.1K
		2nd	1.4	13	33.1K	3.5K
		3rd	2.0	17	35.1K	5.3K
Q2: Configuration Propagation	Megatron-LM	1st	2.9	54	43.4K	10.7K
		2nd	1.9	37	37.6K	8.4K
		3rd	3.2	54	41.2K	12.0K
	TorchTitan	1st	2.8	25	42.6K	6.4K
		2nd	3.0	34	43.5K	7.8K
		3rd	3.5	28	45.0K	9.2K
	PithTrain	1st	2.0	18	35.7K	4.5K
		2nd	2.9	28	35.9K	7.6K
		3rd	2.2	18	36.8K	5.4K
Q3: Data Loading & Sharding	Megatron-LM	1st	1.0	10	32.1K	3.8K
		2nd	0.8	14	31.3K	3.0K
		3rd	0.9	14	30.8K	3.4K
	TorchTitan	1st	1.2	14	35.0K	3.2K
		2nd	1.8	21	38.2K	4.0K
		3rd	1.2	14	36.1K	3.5K
	PithTrain	1st	1.4	13	34.5K	3.5K
		2nd	1.3	15	29.1K	3.4K
		3rd	1.5	8	34.1K	2.9K
Q4: Distributed Seed Management	Megatron-LM	1st	2.2	36	37.5K	8.8K
		2nd	2.2	37	49.4K	10.0K
		3rd	2.5	34	41.3K	10.4K
	TorchTitan	1st	3.1	22	37.9K	6.5K
		2nd	3.3	26	39.7K	6.9K
		3rd	3.9	32	41.7K	7.9K
	PithTrain	1st	2.2	15	34.0K	5.0K
		2nd	2.3	19	31.1K	5.2K
		3rd	2.5	16	30.7K	5.6K
Q5: Attention Kernel Dispatch	Megatron-LM	1st	3.7	63	56.0K	14.0K
		2nd	2.8	48	46.5K	10.7K
		3rd	1.7	27	39.8K	6.6K
	TorchTitan	1st	2.1	20	41.7K	4.8K
		2nd	2.0	19	43.7K	5.1K
		3rd	2.0	15	41.2K	4.7K
	PithTrain	1st	1.9	20	34.2K	4.4K
		2nd	1.9	24	33.7K	4.6K
		3rd	2.0	21	33.8K	5.0K
Q6: RoPE Implementation	Megatron-LM	1st	1.1	12	36.4K	4.5K
		2nd	1.1	12	35.9K	4.5K
		3rd	0.9	12	36.4K	3.2K
	TorchTitan	1st	0.7	4	29.9K	1.9K
		2nd	1.0	4	29.9K	2.6K
		3rd	1.0	4	29.9K	2.3K
	PithTrain	1st	1.2	8	26.7K	2.6K
		2nd	1.3	11	27.9K	3.3K
		3rd	2.0	18	31.8K	5.6K

Table 10: Per-attempt agent effort on the Q&A tasks (continued, Q7–Q12).

Task	Framework	Attempt	Session Duration	Agent Turns	Per-Turn Context	Output Tokens
Q7: SwiGLU / MLP Block	Megatron-LM	1st	1.0	10	31.5K	3.2K
		2nd	0.7	8	31.4K	2.5K
		3rd	0.7	9	34.1K	2.6K
	TorchTitan	1st	1.6	11	30.0K	3.2K
		2nd	1.4	13	30.2K	3.8K
		3rd	0.9	12	30.0K	3.4K
	PithTrain	1st	2.1	15	31.5K	4.6K
		2nd	1.6	13	40.3K	4.2K
		3rd	1.5	14	30.1K	3.8K
Q8: Normalization Placement	Megatron-LM	1st	1.0	15	30.3K	4.1K
		2nd	2.6	20	30.8K	4.7K
		3rd	1.5	25	34.2K	5.5K
	TorchTitan	1st	1.0	11	32.7K	3.5K
		2nd	0.7	8	30.3K	2.5K
		3rd	1.1	14	33.7K	3.5K
	PithTrain	1st	1.1	9	27.7K	3.0K
		2nd	1.3	9	28.7K	3.2K
		3rd	1.4	11	29.9K	3.6K
Q9: Context / Sequence Parallelism	Megatron-LM	1st	4.7	60	45.6K	6.8K
		2nd	3.1	34	45.2K	7.6K
		3rd	3.3	29	44.6K	7.8K
	TorchTitan	1st	1.5	20	38.6K	5.8K
		2nd	2.0	16	38.6K	7.5K
		3rd	1.6	16	37.1K	6.0K
	PithTrain	1st	2.9	22	37.6K	7.1K
		2nd	2.2	16	37.0K	5.1K
		3rd	1.7	16	35.1K	4.1K
Q10: FSDP / DDP Wrapping	Megatron-LM	1st	2.8	19	37.8K	7.0K
		2nd	2.3	21	38.6K	5.6K
		3rd	2.7	16	35.5K	5.9K
	TorchTitan	1st	1.6	24	44.1K	6.0K
		2nd	1.3	17	36.7K	4.9K
		3rd	1.5	16	38.8K	5.1K
	PithTrain	1st	1.1	9	26.9K	2.6K
		2nd	1.2	12	30.2K	2.9K
		3rd	1.4	11	29.4K	3.0K
Q11: Global Gradient Clipping	Megatron-LM	1st	1.4	10	31.9K	3.2K
		2nd	1.2	12	31.3K	3.1K
		3rd	1.0	10	31.0K	2.9K
	TorchTitan	1st	0.7	9	30.0K	2.5K
		2nd	0.7	8	30.2K	2.7K
		3rd	0.7	7	30.1K	2.5K
	PithTrain	1st	0.8	6	25.6K	2.0K
		2nd	0.8	7	25.8K	1.9K
		3rd	0.8	6	25.5K	2.0K
Q12: Distributed Checkpoint Serialization	Megatron-LM	1st	2.0	19	37.2K	5.1K
		2nd	1.8	18	36.2K	4.5K
		3rd	2.8	17	37.1K	6.4K
	TorchTitan	1st	0.6	5	33.9K	2.2K
		2nd	0.8	5	33.9K	2.6K
		3rd	0.6	5	33.9K	2.2K
	PithTrain	1st	1.2	10	31.1K	2.5K
		2nd	1.1	9	30.0K	2.6K
		3rd	1.4	10	35.3K	2.8K

Table 11: Per-attempt agent effort on the operate-and-profile tasks (§5.2). Each row is one independent attempt. Session Duration and Active GPU Time are in minutes.

Task	Framework	Attempt	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Getting Started	Megatron-LM	1st	47.6	5.9	106	85.5K	31.9K
		2nd	40.5	5.4	88	69.5K	26.9K
		3rd	24.1	5.2	78	64.0K	25.8K
	TorchTitan	1st	10.8	5.0	54	62.0K	15.8K
		2nd	11.4	5.2	47	43.1K	15.4K
		3rd	13.5	5.6	68	56.8K	18.3K
	PithTrain	1st	7.6	3.9	20	33.8K	5.4K
		2nd	6.6	3.0	26	37.0K	5.8K
		3rd	5.8	3.1	28	37.9K	7.6K
Train and Evaluate	Megatron-LM	1st	52.3	29.3	163	109.2K	45.1K
		2nd	58.8	36.0	155	101.4K	53.2K
		3rd	55.5	38.1	170	106.4K	52.9K
	TorchTitan	1st	59.5	36.3	212	176.6K	75.8K
		2nd	101.6	87.5	268	149.8K	97.8K
		3rd	72.5	28.8	198	197.1K	109.3K
	PithTrain	1st	42.0	25.9	100	86.5K	36.9K
		2nd	37.1	22.7	87	83.1K	34.2K
		3rd	38.5	19.3	92	85.5K	32.3K
Collect Routing Trace	Megatron-LM	1st	33.3	8.3	112	144.3K	102.1K
		2nd	33.3	5.5	138	158.8K	102.5K
		3rd	25.1	5.3	87	106.4K	74.7K
	TorchTitan	1st	32.8	10.4	93	166.0K	84.7K
		2nd	41.4	11.9	152	181.7K	113.6K
		3rd	30.8	9.8	103	160.3K	74.8K
	PithTrain	1st	16.3	2.7	58	118.7K	56.2K
		2nd	14.3	2.8	52	107.8K	48.4K
		3rd	20.1	2.8	70	146.8K	71.9K
Report Heavy Kernels	Megatron-LM	1st	23.4	12.5	62	67.0K	31.0K
		2nd	22.1	12.1	53	48.1K	20.5K
		3rd	21.2	12.0	60	52.6K	23.9K
	TorchTitan	1st	14.8	8.1	44	65.1K	17.0K
		2nd	15.0	4.8	40	69.0K	22.5K
		3rd	15.2	6.7	39	66.1K	22.7K
	PithTrain	1st	11.8	3.7	42	50.2K	19.0K
		2nd	10.4	3.6	36	44.0K	11.6K
		3rd	12.5	3.5	43	49.2K	16.0K

Table 12: Per-attempt agent effort on the new-feature tasks (§5.2). Each row is one independent attempt. Session Duration and Active GPU Time are in minutes.

Task	Framework	Attempt	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Diff [50]	Megatron-LM	1st	47.1	33.7	62	118.7K	49.4K
		2nd	56.6	44.9	125	165.9K	58.3K
		3rd	46.3	33.6	141	103.6K	57.1K
	TorchTitan	1st	49.6	40.3	61	96.9K	35.4K
		2nd	57.1	49.2	53	105.9K	36.0K
		3rd	48.6	40.1	58	103.2K	38.7K
	PithTrain	1st	35.3	27.6	47	53.9K	29.4K
		2nd	38.2	33.2	35	69.5K	22.1K
		3rd	42.8	27.3	83	93.0K	25.4K
DynMoE [14]	Megatron-LM	1st	84.2	45.4	200	177.9K	125.3K
		2nd	76.7	49.1	199	238.1K	115.2K
		3rd	83.8	62.1	186	208.0K	94.5K
	TorchTitan	1st	143.6	97.8	253	306.1K	161.3K
		2nd	140.6	94.4	126	211.4K	178.2K
		3rd	115.5	90.3	197	228.8K	113.5K
	PithTrain	1st	57.1	41.9	76	146.0K	52.9K
		2nd	61.4	42.3	73	101.4K	76.4K
		3rd	60.4	41.3	118	224.9K	96.8K
MoBA [24]	Megatron-LM	1st	71.1	49.1	134	146.7K	53.8K
		2nd	61.6	49.5	135	108.1K	56.1K
		3rd	59.1	49.9	91	120.9K	45.7K
	TorchTitan	1st	53.7	39.4	61	110.2K	54.4K
		2nd	138.9	107.6	104	186.2K	111.8K
		3rd	105.1	77.9	91	166.9K	112.2K
	PithTrain	1st	38.0	27.7	57	66.4K	32.4K
		2nd	52.9	32.0	55	69.0K	22.8K
		3rd	38.7	26.9	66	81.5K	50.8K
MoE++ [17]	Megatron-LM	1st	144.6	78.0	829	516.5K	326.4K
		2nd	88.5	58.7	145	170.8K	117.0K
		3rd	75.5	53.6	140	188.7K	86.9K
	TorchTitan	1st	95.2	72.1	85	134.3K	66.0K
		2nd	71.4	51.9	87	164.6K	85.3K
		3rd	63.5	45.2	125	173.4K	90.4K
	PithTrain	1st	63.0	36.7	141	176.6K	107.7K
		2nd	63.0	39.9	85	201.0K	100.2K
		3rd	67.7	40.1	90	165.4K	119.1K

D.2 Codex

This subsection reports per-attempt agent effort and per-task medians for Codex across all 20 tasks. Tables 13–15 report the medians, and Tables 16–19 give all per-attempt values.

To test whether the results depend on the agent, we repeat the suite with Codex (GPT-5.6 Sol at `high` reasoning effort). The framework commits, task instructions, hardware, and three-attempt protocol are unchanged. GPT-5.6 Sol judges new-feature correctness using the same rules as the Claude Code evaluation (§C.3.2).

The overall trend holds across agents. By median Agent Turns, PithTrain ranks first or ties for first on 18 of 20 Codex tasks, compared with 14 of 20 Claude Code tasks, although individual task orderings vary. Codex also reproduces TorchTitan's MoBA OOMs in two of three attempts; both recover and complete training.

Durations and Active GPU Time are in minutes; context and output are in tokens. Lower is better on every metric.

Table 13: Codex per-question agent effort on the Q&A task suite (§4); Q1–Q12 are described in Appendix C.1. Each metric reports the median of three independent attempts. Lower is more efficient. Headers abbreviate Megatron-LM, TorchTitan, and PithTrain.

	Agent Turns			Per-Turn Context			Output Tokens		
#	Megatron	Titan	Pith	Megatron	Titan	Pith	Megatron	Titan	Pith
Q1	20	15	**9**	63.7K	56.2K	**41.4K**	10.6K	9.6K	**6.8K**
Q2	21	13	**10**	68.1K	52.7K	**45.4K**	9.2K	6.7K	**4.9K**
Q3	**8**	11	**8**	41.1K	44.2K	**36.4K**	4.9K	4.0K	**3.1K**
Q4	21	**12**	**12**	56.2K	57.8K	**37.8K**	9.7K	8.5K	**7.2K**
Q5	26	9	**8**	74.6K	44.0K	**39.6K**	9.6K	5.3K	**4.9K**
Q6	7	**6**	**6**	44.3K	34.7K	**28.5K**	4.3K	**3.2K**	3.5K
Q7	8	8	**7**	41.6K	39.6K	**32.5K**	3.8K	3.5K	**3.2K**
Q8	15	**7**	10	48.8K	29.2K	**27.4K**	6.5K	**3.8K**	5.2K
Q9	12	14	**9**	52.0K	44.6K	**40.9K**	5.3K	7.3K	**5.2K**
Q10	17	13	**10**	60.7K	49.8K	**35.8K**	8.5K	7.6K	**4.4K**
Q11	10	**4**	5	33.8K	**20.5K**	22.8K	5.0K	2.8K	**2.3K**
Q12	14	8	**7**	53.3K	39.6K	**33.2K**	6.3K	4.3K	**3.7K**

Table 14: Codex per-task agent effort across frameworks on operate-and-profile tasks (§4). Each metric reports the median of three independent attempts. Lower is more efficient. Session Duration and Active GPU Time are in minutes. Per-attempt metrics in Table 18.

Task	Framework	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Getting Started	Megatron-LM	20.0	6.7	138	124.8K	18.2K
	TorchTitan	8.6	6.6	45	55.6K	6.1K
	PithTrain	**6.0**	**3.8**	**26**	**20.4K**	**4.0K**
Train and Evaluate	Megatron-LM	65.1	44.5	317	166.8K	50.9K
	TorchTitan	42.2	26.4	104	129.7K	27.3K
	PithTrain	**25.5**	**12.9**	**94**	**63.6K**	**20.6K**
Collect Routing Trace	Megatron-LM	10.1	6.0	39	**71.0K**	**17.4K**
	TorchTitan	13.9	7.5	72	103.6K	25.9K
	PithTrain	**8.9**	**3.9**	**32**	73.7K	18.7K
Report Heavy Kernels	Megatron-LM	21.3	17.9	89	43.9K	15.3K
	TorchTitan	24.3	17.3	99	67.6K	20.1K
	PithTrain	**11.1**	**6.8**	**42**	**43.0K**	**14.0K**

Table 15: Codex per-task agent effort across frameworks on new-feature tasks (§4). Each metric reports the median of three independent attempts. Lower is more efficient. Session Duration and Active GPU Time are in minutes. Per-attempt metrics in Table 19.

Task	Framework	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Differential Transformer (Diff) [50]	Megatron-LM	49.8	44.7	163	136.0K	27.0K
	TorchTitan	50.7	46.6	119	96.4K	17.4K
	PithTrain	**36.3**	**33.8**	**112**	**78.5K**	**14.6K**
Dynamic Mixture of Experts (DynMoE) [14]	Megatron-LM	66.9	56.6	203	152.5K	41.4K
	TorchTitan	105.4	93.8	257	206.3K	51.4K
	PithTrain	**46.9**	**39.5**	**148**	**141.8K**	**32.1K**
Mixture of Block Attention (MoBA) [24]	Megatron-LM	78.8	70.1	200	189.4K	51.3K
	TorchTitan	74.7	67.7	173	134.6K	38.1K
	PithTrain	**47.9**	**42.8**	**95**	**102.8K**	**26.9K**
MoE++ [17]	Megatron-LM	141.0	123.1	527	168.6K	74.6K
	TorchTitan	76.2	66.4	226	173.4K	44.8K
	PithTrain	**43.9**	**37.5**	**200**	**165.4K**	**41.8K**

Table 16: Codex per-attempt agent effort on the Q&A tasks Q1–Q6 (§D.2). Each row is one independent attempt. Session Duration is in minutes.

Task	Framework	Attempt	Session Duration	Agent Turns	Per-Turn Context	Output Tokens
Q1: Process Groups / Device Mesh	Megatron-LM	1st	4.8	20	68.9K	13.2K
		2nd	3.7	20	63.1K	10.6K
		3rd	3.4	18	63.7K	8.6K
	TorchTitan	1st	3.5	17	56.2K	9.6K
		2nd	2.4	13	54.5K	6.8K
		3rd	3.9	15	65.2K	10.0K
	PithTrain	1st	2.4	9	42.4K	6.8K
		2nd	2.0	7	33.0K	5.9K
		3rd	2.9	14	41.4K	7.9K
Q2: Configuration Propagation	Megatron-LM	1st	3.5	20	74.3K	9.2K
		2nd	4.7	27	68.1K	10.3K
		3rd	3.8	21	60.0K	8.3K
	TorchTitan	1st	2.6	13	49.3K	6.5K
		2nd	2.4	12	52.7K	6.7K
		3rd	2.5	14	55.3K	6.9K
	PithTrain	1st	1.5	9	42.6K	4.2K
		2nd	1.8	11	48.5K	5.0K
		3rd	1.9	10	45.4K	4.9K
Q3: Data Loading & Sharding	Megatron-LM	1st	1.0	7	41.1K	2.7K
		2nd	1.9	10	44.4K	4.9K
		3rd	1.8	8	37.9K	5.3K
	TorchTitan	1st	1.6	11	39.7K	4.0K
		2nd	2.3	18	49.6K	5.5K
		3rd	1.5	11	44.2K	3.9K
	PithTrain	1st	1.3	8	36.4K	3.1K
		2nd	1.7	10	40.1K	4.5K
		3rd	1.2	7	35.8K	3.1K
Q4: Distributed Seed Management	Megatron-LM	1st	3.7	21	47.6K	9.7K
		2nd	3.1	17	56.2K	7.9K
		3rd	5.1	24	71.2K	12.6K
	TorchTitan	1st	2.5	11	49.3K	6.5K
		2nd	3.4	15	57.8K	9.9K
		3rd	3.2	12	62.8K	8.5K
	PithTrain	1st	2.7	12	31.5K	7.2K
		2nd	2.6	12	50.7K	7.6K
		3rd	2.3	7	37.8K	7.1K
Q5: Attention Kernel Dispatch	Megatron-LM	1st	4.7	35	79.1K	11.2K
		2nd	2.8	15	61.1K	7.3K
		3rd	3.8	26	74.6K	9.6K
	TorchTitan	1st	1.8	9	40.4K	5.1K
		2nd	1.9	9	49.3K	5.3K
		3rd	1.9	9	44.0K	5.6K
	PithTrain	1st	1.6	8	39.0K	4.2K
		2nd	2.1	10	47.6K	5.5K
		3rd	1.7	8	39.6K	4.9K
Q6: RoPE Implementation	Megatron-LM	1st	1.6	9	45.2K	4.3K
		2nd	1.4	6	44.3K	4.0K
		3rd	1.6	7	35.6K	4.5K
	TorchTitan	1st	1.1	6	34.7K	3.0K
		2nd	1.2	7	35.9K	3.4K
		3rd	1.1	5	31.7K	3.2K
	PithTrain	1st	2.1	6	36.7K	6.3K
		2nd	1.4	7	28.5K	3.5K
		3rd	1.1	5	26.8K	3.3K

Table 17: Codex per-attempt agent effort on the Q&A tasks (continued, Q7–Q12).

Task	Framework	Attempt	Session Duration	Agent Turns	Per-Turn Context	Output Tokens
Q7: SwiGLU / MLP Block	Megatron-LM	1st	1.5	8	41.6K	4.0K
		2nd	1.4	8	39.6K	3.8K
		3rd	1.3	7	41.8K	3.4K
	TorchTitan	1st	1.3	8	40.3K	3.5K
		2nd	1.4	8	35.4K	3.9K
		3rd	1.4	6	39.6K	3.5K
	PithTrain	1st	1.6	9	38.7K	4.4K
		2nd	1.5	6	32.5K	3.2K
		3rd	1.3	7	29.7K	3.1K
Q8: Normalization Placement	Megatron-LM	1st	2.3	15	45.5K	6.3K
		2nd	2.5	15	48.8K	6.5K
		3rd	3.0	14	54.5K	7.8K
	TorchTitan	1st	2.4	13	35.7K	5.4K
		2nd	1.1	5	29.2K	3.1K
		3rd	1.5	7	28.6K	3.8K
	PithTrain	1st	3.6	14	31.2K	5.7K
		2nd	1.9	10	25.8K	5.2K
		3rd	1.5	10	27.4K	4.2K
Q9: Context / Sequence Parallelism	Megatron-LM	1st	1.8	12	52.0K	4.9K
		2nd	2.9	16	58.8K	8.1K
		3rd	1.9	11	44.8K	5.3K
	TorchTitan	1st	2.9	14	44.3K	7.4K
		2nd	3.1	14	46.4K	7.2K
		3rd	2.9	13	44.6K	7.3K
	PithTrain	1st	2.2	9	39.6K	5.2K
		2nd	2.9	13	44.1K	7.7K
		3rd	1.8	8	40.9K	5.0K
Q10: FSDP / DDP Wrapping	Megatron-LM	1st	3.2	14	60.7K	8.7K
		2nd	3.8	17	63.7K	8.5K
		3rd	2.9	18	57.6K	7.0K
	TorchTitan	1st	2.8	14	49.8K	7.8K
		2nd	2.3	12	58.0K	6.3K
		3rd	2.8	13	38.3K	7.6K
	PithTrain	1st	1.7	10	38.4K	4.4K
		2nd	1.4	6	28.2K	4.1K
		3rd	2.8	15	35.8K	7.9K
Q11: Global Gradient Clipping	Megatron-LM	1st	1.4	7	28.9K	3.8K
		2nd	1.8	10	33.8K	5.0K
		3rd	2.0	10	34.6K	5.6K
	TorchTitan	1st	0.8	4	20.5K	2.2K
		2nd	1.3	5	19.2K	3.7K
		3rd	1.1	4	23.6K	2.8K
	PithTrain	1st	0.9	5	23.0K	2.2K
		2nd	0.9	4	19.0K	2.3K
		3rd	1.2	6	22.8K	3.4K
Q12: Distributed Checkpoint Serialization	Megatron-LM	1st	2.6	18	53.3K	6.9K
		2nd	2.1	14	50.2K	5.4K
		3rd	2.5	14	60.8K	6.3K
	TorchTitan	1st	1.3	7	38.6K	3.6K
		2nd	1.6	8	39.6K	4.3K
		3rd	2.4	11	49.3K	6.7K
	PithTrain	1st	1.4	7	33.2K	3.7K
		2nd	1.3	6	35.4K	3.1K
		3rd	2.5	9	32.7K	5.7K

Table 18: Codex per-attempt agent effort on the operate-and-profile tasks (§D.2). Each row is one independent attempt. Session Duration and Active GPU Time are in minutes.

Task	Framework	Attempt	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Getting Started	Megatron-LM	1st	15.2	6.7	69	79.8K	10.8K
		2nd	33.2	8.0	284	127.6K	27.6K
		3rd	20.0	6.7	138	124.8K	18.2K
	TorchTitan	1st	8.6	6.6	36	49.5K	6.1K
		2nd	8.7	6.5	45	61.7K	8.7K
		3rd	8.2	6.6	45	55.6K	5.9K
	PithTrain	1st	6.0	3.8	28	20.4K	4.0K
		2nd	6.0	3.8	26	35.2K	4.7K
		3rd	4.2	3.3	21	18.7K	3.6K
Train and Evaluate	Megatron-LM	1st	69.6	44.5	317	190.9K	50.9K
		2nd	65.1	49.4	221	113.7K	34.3K
		3rd	57.1	35.5	321	166.8K	53.6K
	TorchTitan	1st	43.8	31.2	104	142.0K	27.3K
		2nd	42.2	26.4	146	129.7K	36.4K
		3rd	27.2	15.8	103	121.3K	26.0K
	PithTrain	1st	26.7	12.7	108	61.4K	22.7K
		2nd	25.5	12.9	94	63.6K	19.6K
		3rd	25.0	13.1	72	71.0K	20.6K
Collect Routing Trace	Megatron-LM	1st	10.1	6.0	47	79.9K	17.4K
		2nd	10.2	6.0	39	71.0K	18.5K
		3rd	9.5	5.2	33	69.3K	16.9K
	TorchTitan	1st	12.7	7.5	62	106.4K	20.6K
		2nd	26.1	20.5	113	98.9K	25.9K
		3rd	13.9	7.4	72	103.6K	26.6K
	PithTrain	1st	7.8	3.3	30	75.5K	18.7K
		2nd	12.3	3.9	32	71.2K	21.4K
		3rd	8.9	3.9	35	73.7K	17.0K
Report Heavy Kernels	Megatron-LM	1st	21.3	18.8	50	43.9K	13.9K
		2nd	20.9	17.9	93	44.9K	15.3K
		3rd	22.1	14.4	89	42.6K	17.6K
	TorchTitan	1st	24.3	17.3	99	68.8K	20.1K
		2nd	21.1	15.1	62	67.6K	21.1K
		3rd	24.5	18.6	105	52.3K	18.3K
	PithTrain	1st	11.9	6.6	41	43.0K	13.3K
		2nd	11.1	6.8	51	43.7K	16.2K
		3rd	9.4	6.8	42	37.1K	14.0K

Table 19: Codex per-attempt agent effort on the new-feature tasks (§D.2). Each row is one independent attempt. Session Duration and Active GPU Time are in minutes.

Task	Framework	Attempt	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
Diff [50]	Megatron-LM	1st	54.8	48.9	163	128.4K	27.0K
		2nd	49.2	44.4	117	137.0K	22.3K
		3rd	49.8	44.7	164	136.0K	27.8K
	TorchTitan	1st	49.9	46.9	119	86.5K	15.3K
		2nd	50.7	46.6	119	97.3K	17.4K
		3rd	51.0	46.6	113	96.4K	19.4K
	PithTrain	1st	35.2	33.1	143	68.0K	14.6K
		2nd	36.3	33.8	91	86.7K	12.8K
		3rd	36.4	33.8	112	78.5K	14.9K
DynMoE [14]	Megatron-LM	1st	59.1	50.3	170	152.5K	40.3K
		2nd	85.6	76.9	215	171.7K	41.4K
		3rd	66.9	56.6	203	60.8K	43.8K
	TorchTitan	1st	78.5	67.2	257	206.3K	47.3K
		2nd	109.6	98.1	208	202.6K	51.4K
		3rd	105.4	93.8	310	210.0K	53.0K
	PithTrain	1st	46.9	39.7	148	162.5K	32.1K
		2nd	46.9	39.5	98	141.8K	31.1K
		3rd	47.0	38.2	186	116.6K	37.6K
MoBA [24]	Megatron-LM	1st	78.8	70.1	163	165.6K	51.3K
		2nd	62.7	53.6	200	191.4K	37.9K
		3rd	103.7	90.8	223	189.4K	58.5K
	TorchTitan	1st	56.2	47.4	152	134.6K	38.1K
		2nd	74.7	67.7	195	139.8K	31.8K
		3rd	87.8	78.4	173	125.2K	46.0K
	PithTrain	1st	47.4	42.8	82	86.4K	24.7K
		2nd	47.9	42.7	119	109.3K	29.0K
		3rd	48.3	43.3	95	102.8K	26.9K
MoE++ [17]	Megatron-LM	1st	141.0	123.1	527	168.6K	74.6K
		2nd	71.2	62.5	203	182.4K	38.3K
		3rd	161.7	133.6	775	146.4K	128.7K
	TorchTitan	1st	68.6	61.2	281	168.3K	42.8K
		2nd	76.2	66.4	226	173.4K	45.4K
		3rd	82.4	72.7	225	197.3K	44.8K
	PithTrain	1st	43.0	37.5	200	165.4K	41.8K
		2nd	43.9	35.8	125	151.2K	30.8K
		3rd	64.5	48.0	233	214.6K	50.0K

D.3 Run-to-Run Variation

We report per-task medians to reduce sensitivity to extreme attempts. For example, Megatron-LM's three MoE++ attempts use 140, 145, and 829 Agent Turns, compared with 85, 90, and 141 on PithTrain. Tables 20 and 21 report Agent Turns as median [minimum–maximum] over the three independent attempts. Tables 22 and 23 give the same summary for new-feature Active GPU Time. On all four new-feature tasks, every PithTrain attempt uses less Active GPU Time than every attempt on either baseline, under both agents. Agent-Turn ranges overlap on some tasks, illustrating the variation behind the median comparisons.

Table 20: Claude Agent Turns: median [minimum–maximum] across three attempts.

Task	Megatron-LM	TorchTitan	PithTrain
Q1	33 [26–34]	18 [15–19]	15 [13–17]
Q2	54 [37–54]	28 [25–34]	18 [18–28]
Q3	14 [10–14]	14 [14–21]	13 [8–15]
Q4	36 [34–37]	26 [22–32]	16 [15–19]
Q5	48 [27–63]	19 [15–20]	21 [20–24]
Q6	12 [12–12]	4 [4–4]	11 [8–18]
Q7	9 [8–10]	12 [11–13]	14 [13–15]
Q8	20 [15–25]	11 [8–14]	9 [9–11]
Q9	34 [29–60]	16 [16–20]	16 [16–22]
Q10	19 [16–21]	17 [16–24]	11 [9–12]
Q11	10 [10–12]	8 [7–9]	6 [6–7]
Q12	18 [17–19]	5 [5–5]	10 [9–10]
Start	88 [78–106]	54 [47–68]	26 [20–28]
Train/eval	163 [155–170]	212 [198–268]	92 [87–100]
Routing	112 [87–138]	103 [93–152]	58 [52–70]
Kernels	60 [53–62]	40 [39–44]	42 [36–43]
Diff	125 [62–141]	58 [53–61]	47 [35–83]
DynMoE	199 [186–200]	197 [126–253]	76 [73–118]
MoBA	134 [91–135]	91 [61–104]	57 [55–66]
MoE++	145 [140–829]	87 [85–125]	90 [85–141]

Table 21: Codex Agent Turns: median [minimum–maximum] across three attempts.

Task	Megatron-LM	TorchTitan	PithTrain
Q1	20 [18–20]	15 [13–17]	9 [7–14]
Q2	21 [20–27]	13 [12–14]	10 [9–11]
Q3	8 [7–10]	11 [11–18]	8 [7–10]
Q4	21 [17–24]	12 [11–15]	12 [7–12]
Q5	26 [15–35]	9 [9–9]	8 [8–10]
Q6	7 [6–9]	6 [5–7]	6 [5–7]
Q7	8 [7–8]	8 [6–8]	7 [6–9]
Q8	15 [14–15]	7 [5–13]	10 [10–14]
Q9	12 [11–16]	14 [13–14]	9 [8–13]
Q10	17 [14–18]	13 [12–14]	10 [6–15]
Q11	10 [7–10]	4 [4–5]	5 [4–6]
Q12	14 [14–18]	8 [7–11]	7 [6–9]
Start	138 [69–284]	45 [36–45]	26 [21–28]
Train/eval	317 [221–321]	104 [103–146]	94 [72–108]
Routing	39 [33–47]	72 [62–113]	32 [30–35]
Kernels	89 [50–93]	99 [62–105]	42 [41–51]
Diff	163 [117–164]	119 [113–119]	112 [91–143]
DynMoE	203 [170–215]	257 [208–310]	148 [98–186]
MoBA	200 [163–223]	173 [152–195]	95 [82–119]
MoE++	527 [203–775]	226 [225–281]	200 [125–233]

Table 22: Claude new-feature Active GPU Time: median [minimum–maximum] in minutes.

Task	Megatron-LM	TorchTitan	PithTrain
Diff	33.7 [33.6–44.9]	40.3 [40.1–49.2]	27.6 [27.3–33.2]
DynMoE	49.1 [45.4–62.1]	94.4 [90.3–97.8]	41.9 [41.3–42.3]
MoBA	49.5 [49.1–49.9]	77.9 [39.4–107.6]	27.7 [26.9–32.0]
MoE++	58.7 [53.6–78.0]	51.9 [45.2–72.1]	39.9 [36.7–40.1]

Table 23: Codex new-feature Active GPU Time: median [minimum–maximum] in minutes.

Task	Megatron-LM	TorchTitan	PithTrain
Diff	44.7 [44.4–48.9]	46.6 [46.6–46.9]	33.8 [33.1–33.8]
DynMoE	56.6 [50.3–76.9]	93.8 [67.2–98.1]	39.5 [38.2–39.7]
MoBA	70.1 [53.6–90.8]	67.7 [47.4–78.4]	42.8 [42.7–43.3]
MoE++	123.1 [62.5–133.6]	66.4 [61.2–72.7]	37.5 [35.8–48.0]

E Agent Skills on Production Frameworks

We tailor `capture-nsys-profile` to Megatron-LM and TorchTitan, then repeat Report Heavy Kernels with and without the skill. Each condition uses Codex for three attempts with a fixed harness, following the skill-removal protocol in §5.3. Table 24 reports the medians.

On TorchTitan, the skill reduces Agent Turns from 99 to 53 and Active GPU Time from 17.3 to 9.1 minutes. Without the skill, the agent builds step-scoped profiling hooks in all three attempts, each with a different design, and runs training repeatedly to get them right. The skill provides this procedure directly. On Megatron-LM, Agent Turns increase from 89 to 98 while GPU time stays near parity (17.9 versus 17.6 minutes). Its existing profiling flags already expose the procedure, so reading the skill adds overhead. Skills help when procedural knowledge is missing; their effect depends on what the framework already exposes.

Table 24: Baseline profiling-skills ablation: medians of three Codex attempts. Durations are in minutes, and context and output are in tokens.

Framework	Skill	Duration	GPU time	Turns	Context	Output
Megatron-LM	off	21.3	17.9	89	43.9K	15.3K
	on	23.5	17.6	98	37.9K	16.1K
TorchTitan	off	24.3	17.3	99	67.6K	20.1K
	on	12.7	9.1	53	39.0K	14.0K

F Model FLOPs Utilization

We compute MFU [9] as $T \cdot F_{\text{token}}/(N_{\text{GPU}} \cdot P_{\text{peak}})$, where T is the aggregate throughput in tokens per second and $P_{\text{peak}} = 989.4$ TFLOPS is the dense BF16 peak of an H200. F_{token} follows the `num_floating_point_operations` formula of Megatron-LM `core_v0.20.0` with three corrections: gated experts count three weight matrices (gate, up, and down), the router GEMM is included, and sliding-window layers attend over their window rather than the full causal context[8]. This gives 23.08 and 56.91 GFLOP per token for Qwen3-30B-A3B at 4K and 32K, 32.71 GFLOP per token for GPT-OSS-120B at 4K, and 16.41 GFLOP per token for DeepSeek-V2-Lite at 4K. Without the corrections, the formula undercounts GPT-OSS-120B by about 17%; for Qwen3-30B-A3B, only the router term changes.

[8] `https://github.com/NVIDIA/Megatron-LM/issues/1725`

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims are backed up by the experiments in §5.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in §6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Evaluation protocols, configurations, and task criteria are described in §5 and Appendices A–D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Code and usage instructions are available at `https://github.com/mlc-ai/pith-train` and `https://github.com/mlc-ai/ate-bench`.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Experimental settings are described in §5 and Appendices A–C.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Appendix D reports per-attempt results, medians, and minimum–maximum ranges across three attempts for both agents.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: GPU configurations are given in §5.1 and §5.2; per-attempt execution times are in Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in the paper conforms, in every respect, with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This is foundational systems research on an agent-native MoE training framework, with no direct path to specific societal applications beyond those of LLM training in general.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We release a training framework and a benchmark; neither is a high-risk asset.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All existing assets used in the paper are cited.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: PithTrain and ATE-Bench are publicly available with installation and usage instructions.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.

- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Agent evaluation, correctness judging, and kernel development are described in §3, §5.2, and Appendices C and D.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.