Training Mixture-of-Experts Language Models

Hao Kang

CMU-CS-26-106

May 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Chenyan Xiong, Chair
Tianqi Chen

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science.

Copyright © 2026 **Hao Kang**

Keywords: Large Language Models, Mixture-of-Experts, Pre-training, Training Systems

To my dear friends and family,
whose support and belief carried me through.

Abstract

Mixture-of-Experts (MoE) has become the dominant architecture for frontier language models, offering a favorable trade-off between model capacity and per-token computation. This thesis studies the training of MoE language models from two perspectives: modeling and systems.

On the modeling side, we present FLAME-MoE, a transparent research platform providing a suite of MoE models across seven scales, with all code, data pipelines, intermediate checkpoints, and routing logs released publicly. We establish MoE scaling laws and show that the resulting models outperform dense baselines at matched compute. Using the full training trace, we conduct empirical analyses of expert behavior, finding that expert specialization emerges gradually, co-activation remains sparse but intensifies in deeper layers, and routing decisions converge early in training.

On the systems side, we present PithTrain, a Python-native MoE training framework that delivers production-grade throughput in roughly 10K lines of code. PithTrain supports 4D parallelism, a DualPipeV pipeline scheduler that overlaps computation with communication, FP8 training via DeepGEMM, and fused Triton kernels for expert dispatch. At this scale, the entire codebase fits within the context window of modern AI coding tools, making it end-to-end readable by both humans and agents. We also explore the implications of this minimal design for agentic development.

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Chenyan Xiong, for inviting me into language model research and for supporting both projects in this thesis throughout. His guidance shaped how I approach a research problem: what questions are worth asking, how to design experiments that answer them, and how to communicate the answers clearly. I am grateful for his patience, his detailed feedback on drafts, and for building a research environment in which thoughtful modeling work is possible at an academic scale.

I am equally grateful to Professor Tianqi Chen for serving on my committee and for first leading me into machine learning systems research. His course on Deep Learning Systems, taught from years of expertise and genuine passion for the material, is the most exciting course on campus; the hands-on experience of building a distributed extension to the *needle* framework over that semester was particularly helpful to the building of PithTrain.

FLAME-MoE grew out of a collaboration with Zichun Yu, from whom I learned a lot about modeling research; Zichun combines technical depth with an easygoing energy that lights up long debugging sessions. PithTrain was co-led with Ruihang Lai, from whom I learned how to do MLSys research; Ruihang cares deeply about mentoring and consistently pushes to dig below the surface: to read the profiles and extract insights that pure end-to-end measurements cannot give. Much of how I approach a performance problem today I learned from him. Working with Zichun and Ruihang has been among the most enjoyable parts of my time at CMU, and both happen to be excellent on the badminton court.

I thank the CMU Foundation and Language Model (FLAME) Center for providing the compute resources used to develop and evaluate both projects, including the GCP VMs on which the PithTrain throughput measurements were run. I am also grateful to Angy Malloy and Amanda Hornick for their help with program logistics; this thesis would not have come together as smoothly as it did without their support.

Looking back on five years at CMU, four as an undergrad and one as a Fifth-Year Master's student, I am grateful above all for the remarkable peers I was lucky enough to learn alongside. We didn't get through CMU alone; we got through by sharing strengths, lifting each other up, and learning to see the brilliance in perspectives not our own.

Finally, I thank my friends Yifan Su, Tevin Wang, Yutian Chen, Cindy Liu, Qinghan Chen, Xiaochuan Li, James Kim, Shanshan Zhong, Yiyan Zhai, Haozhan Tang, Helen Wang, Shyamal Vaderia, and Professor Eckhardt for making these years at CMU so much richer, and my family for their unwavering support through every stage of this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mixture-of-Experts (MoE) has become the architecture of choice for frontier language models. DeepSeek-V3.2 [1], Qwen3.5 [2], and gpt-oss [3] all use sparsely-activated expert layers to scale model capacity without proportionally increasing the computation required per token. The idea is simple: rather than passing every token through all parameters, a learned router selects a small subset of expert subnetworks for each token, so the model can grow to hundreds of billions of parameters while keeping the per-token cost fixed. This decoupling of capacity from compute is what makes MoE attractive: it allows models to store more knowledge and express more complex functions at a fraction of the training and inference cost of an equivalently large dense model.

Despite this prominence, much of what we know about training MoE language models comes from industrial labs that release models but not the full picture behind them. Training code, data pipelines, routing logs, and intermediate checkpoints (the artifacts that would allow the research community to reproduce results, study training dynamics, and build on existing work) are rarely made available. The few open MoE efforts that do exist [4, 5, 6] have made valuable contributions, but they typically focus on a single model scale and emphasize downstream performance over the kind of multi-scale, longitudinal analysis that would reveal how MoE models actually learn to route tokens and specialize their experts over the course of training.

On the systems side, training an MoE model efficiently is a much harder problem than training a dense model of comparable active size. Three challenges are specific to MoE. First, dynamic routing means tensor shapes vary from step to step, breaking static execution assumptions and requiring costly host-device synchronization. Second, tokens must be dispatched across GPUs to reach their assigned experts via all-to-all communication, and as expert parallelism scales, this traffic increasingly crosses slow inter-node links. Third, imbalanced workload across expert-parallel ranks leaves some GPUs idle while others are busy, reducing overall hardware utilization. The frameworks that handle these challenges today, such as Megatron-LM [7, 8] and DeepSpeed-MoE [9, 10], are production systems that have been developed over many years. They are highly capable, but their codebases often exceed a hundred thousand lines of code, with deep abstraction hierarchies and heavy C++ dependencies that make them difficult to understand, modify, or use as a starting point for modeling research. High-level trainers like Hugging Face Accelerate [11] are easy to set up, but oftentimes lack the parallelism, native FP8 support, and compute-communication overlap to reach competitive throughput. There is no middle ground: a minimal, readable codebase that still delivers the full MoE optimization stack.

This thesis approaches MoE training from two perspectives.

FLAME-MoE (Chapter 3) is a transparent, end-to-end research platform for MoE language models. It provides a suite of seven MoE models spanning 38M to 1.7B active parameters (100M to 10.3B total), trained on the DCLM corpus [12] with all code, data pipelines, intermediate checkpoints, and routing logs released publicly. We establish MoE scaling laws through IsoFLOP profiling [13] over 64 models and show that the resulting models outperform dense baselines by 0.7–3.2 accuracy points at matched compute budgets. Using the full training trace, we conduct three empirical analyses of expert behavior: we find that expert specialization emerges gradually and intensifies during training, that co-activation between experts remains sparse in early layers but grows in deeper layers, and that routing decisions converge early, with over 70% agreement by the midpoint of training. These characterizations offer a detailed picture of how MoE models develop their internal structure during pre-training, enabled by the kind of transparent, multi-scale platform that has not previously existed for MoE research.

PithTrain (Chapter 4) is an efficient, Python-native MoE training framework. It delivers production-grade throughput through 4D parallelism (pipeline $\times$ data $\times$ context $\times$ expert), a DualPipeV [14, 15] pipeline scheduler with five-stage compute-communication overlap, FP8 training via DeepGEMM [16], and fused Triton kernels for expert dispatch, all in roughly 10K lines of Python. PithTrain matches or exceeds the throughput of Megatron-LM [8] and TorchTitan [17] across a range of MoE models on Hopper and Blackwell GPUs. On a suite of New-Feature integration tasks performed by an autonomous coding agent, PithTrain reduces the agent's iteration cost compared to both production frameworks. The entire codebase, from the training loop to the pipeline scheduler to the GPU kernels, can be read end-to-end and modified without navigating layers of abstraction. As coding agents become an increasingly important part of the research workflow, a compact codebase that fits within an agent's context window matters as much as one a human can read; PithTrain is designed for both audiences from the ground up.

These two projects are independent efforts united by a common theme: training Mixture-of-Experts language models. FLAME-MoE is modeling research: it asks what happens when we train MoE models and makes the answer observable. PithTrain is systems research: it delivers production-grade MoE training through a lightweight, Python-native implementation. Together, they address MoE training from both the modeling and systems perspectives.

The remainder of this thesis is organized as follows. Chapter 2 provides background on the Mixture-of-Experts architecture. Chapter 3 presents FLAME-MoE, including the platform design, scaling law study, model training, and empirical analyses. Chapter 4 presents PithTrain, covering the system architecture, parallelism strategy, pipeline scheduler, FP8 integration, throughput evaluation, and an agent-task efficiency evaluation on New-Feature integration tasks. Chapter 5 concludes with reflections on lessons learned across the two projects.

Chapter 2

Background

2.1 Mixture-of-Experts Language Models

Modern language models are built on the transformer architecture [18], which processes sequences through alternating layers of self-attention and feedforward networks (FFNs). This thesis focuses on *decoder-only* transformers [19], trained autoregressively to predict the next token by minimizing the cross-entropy loss.

The Mixture-of-Experts (MoE) architecture [20] modifies the FFN sublayer by replacing it with a set of N_E expert networks and a routing mechanism, while leaving the attention mechanism unchanged. Rather than passing every token through a single FFN, a learned router selects a subset of experts for each token, so the model can increase its total parameter count while keeping the per-token computation fixed.

2.1.1 Sparse Routing

For an input token representation x, the router network r produces a score for each of the N_E experts. The top-k experts with the highest scores are selected, and the MoE layer output is a weighted sum of their outputs:

$$\text{MoE}(x) = \sum_{i \in \text{Top-}k(r(x))} \text{softmax}(r(x))_i \, E_i(x), \tag{2.1}$$

where E_i denotes the i-th expert network and $\text{softmax}(r(x))_i$ is the gating weight assigned to expert i. Some architectures also incorporate *shared experts* [21] that are activated for every token regardless of routing decisions, providing a stable baseline computation path alongside the dynamically routed experts.

2.1.2 Load Balancing

A known failure mode of MoE training is *routing collapse*, where the router learns to send most tokens to a small number of experts while the rest are underutilized [22]. The most common mitigation is an auxiliary load balancing loss [20] that encourages uniform token distribution. For

each expert i, let m_i denote the fraction of tokens it receives and P_i the average gating weight assigned to it by the router. The load balancing loss

$$\mathcal{L}_{\text{LB}} = N_E \cdot \sum_{i=1}^{N_E} m_i \cdot P_i \tag{2.2}$$

is minimized when both quantities are uniform across experts. The full training objective becomes $\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{CE}} + \gamma\, \mathcal{L}_{\text{LB}}$, where γ controls the strength of the balancing term.

A key design choice is the scope over which $\mathcal{L}_{\text{LB}}$ is computed. Switch Transformer [23] computes it at the micro-batch level, which can push the router to distribute tokens uniformly within each sequence. In contrast, Qiu et al. [24] show that computing the loss at the global-batch level, synchronizing expert selection frequencies across micro-batches, yields better pre-training performance by encouraging balance at the corpus level rather than within individual sequences.

Chapter 3

FLAME-MoE

FLAME-MoE is a transparent, end-to-end research platform for Mixture-of-Experts language models. It provides a suite of seven MoE models spanning 38M to 1.7B active parameters (100M to 10.3B total), trained on the DCLM corpus [12] with all code, data pipelines, intermediate checkpoints, and routing logs released publicly. The platform is built on Megatron-LM [7] and designed to support controlled experimentation across model scales, sparsity levels, and architectural configurations.

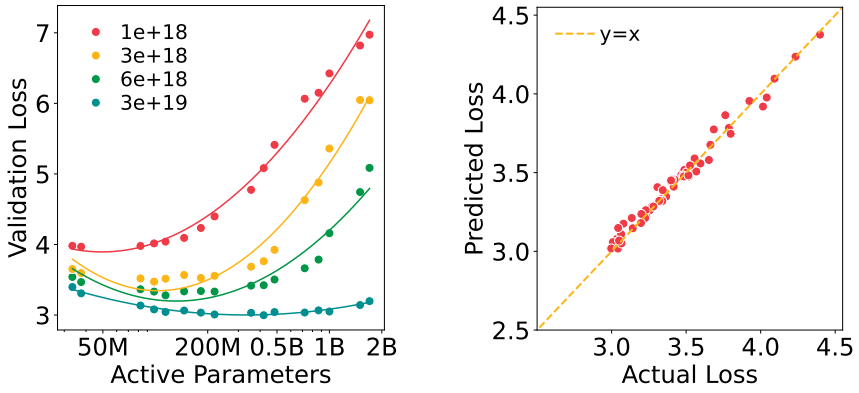
This chapter describes the model architecture (§3.1), the scaling law study used to determine compute-optimal model configurations (§3.2), the pre-training setup and downstream evaluation (§3.3), and three empirical analyses of expert behavior enabled by the platform's transparent training trace (§3.4).

3.1 Model Architecture

FLAME-MoE adopts a decoder-only transformer architecture with N_L layers, where all feedforward sublayers except the first are replaced by MoE layers. Each MoE layer contains $N_E = 64$ expert networks. For each token, $k = 8$ experts are activated: 2 shared experts that process every token, and 6 routed experts selected dynamically by the router based on the input. The shared experts follow the design of DeepSeek-V2 [21], providing a stable baseline computation path alongside the dynamically routed experts. The routing mechanism and gating weights follow the formulation described in Chapter 2 (Equation 2.1). The training objective combines the cross-entropy loss with an auxiliary load balancing loss [20] ($\gamma = 0.01$) and a router z-loss [25] ($\eta = 0.001$) that penalizes large router logits to improve numerical stability.

3.2 Scaling Law Study

Determining the optimal allocation of compute between model size and training tokens is important for training MoE models efficiently. We investigate this through two approaches from Chinchilla [13]: IsoFLOP profiling and parametric loss function fitting.



(a) IsoFLOP profiles (b) Parametric loss fit

Figure 3.1: Scaling law experiments. (a) IsoFLOP profiles for four compute budgets with parabolic fits. (b) Parametric loss function fit across all 64 models.

3.2.1 IsoFLOP Profiles

The first approach fixes a computational budget C and varies the number of active parameters N_{active}, training each (N_{active}, C) configuration for exactly $D = C/(6 \cdot N_{\text{active}})$ tokens, where the factor of 6 follows the standard FLOPs-per-token approximation [13]. Plotting the final validation loss against N_{active} for each budget produces an IsoFLOP curve whose minimum estimates the compute-optimal model size for that budget.

We use four computational budgets (1e18, 3e18, 6e18, and 3e19 FLOPs) and train 16 models per budget with N_{active} ranging from 33.4M to 1.7B, for a total of 64 models. Power laws are then fit to the optimal configurations:

$$N^*_{\text{active}}(C) \propto C^a, \quad D^*(C) \propto C^b. \tag{3.1}$$

3.2.2 Parametric Loss Fitting

The second approach fits a single parametric function to all 64 data points:

$$\mathcal{L}_{\text{val}}(N_{\text{active}}, D) = \frac{A}{N^{\alpha}_{\text{active}}} + \frac{B}{D^{\beta}} + L_0, \tag{3.2}$$

using Huber loss [26] ($\delta = 10^{-3}$) for robustness to outliers at small compute budgets. Once fitted, the optimal N^*_{active} and D^* for any budget C can be derived by minimizing $\mathcal{L}_{\text{val}}$ subject to $C = 6 \cdot N_{\text{active}} \cdot D$.

The two approaches yield consistent results: the IsoFLOP-derived and parametrically-derived optimal configurations closely align. The exponents derived from the parametric fit are $a = 0.69$ and $b = 0.31$, indicating that as compute grows, a larger share should be allocated to increasing model size rather than training tokens. This is a stronger model-size bias than Chinchilla's dense scaling law, which finds roughly equal exponents ($a \approx 0.50$, $b \approx 0.50$). As a validation, the fitted scaling law achieves a Spearman correlation of 0.89 between predicted validation loss

Table 3.1: FLAME-MoE model configurations. All models use 64 experts with 8 active per token (2 shared + 6 routed).

Model	Layers	Hidden	FFN	MoE FFN	FLOPs	Tokens	H100 Hours
38M-100M	9	256	1368	176	1.0e18	4.4B	5.7
98M-349M	9	512	2736	352	3.0e18	5.0B	8.2
115M-459M	12	512	2736	352	6.0e18	8.7B	20.9
290M-1.3B	9	1024	5472	704	2.0e19	11.4B	63.4
419M-2.2B	15	1024	5472	704	3.0e19	11.9B	66.4
721M-3.8B	12	1536	8208	1056	8.0e19	18.4B	172.8
1.7B-10.3B	18	2048	10944	1408	2.4e20	23.1B	560.5

and downstream task performance on HellaSwag. Figure 3.1 shows the IsoFLOP profiles and parametric fit.

3.2.3 Model Family

We use the scaling law to construct the FLAME-MoE model family. Four models come directly from the IsoFLOP experiments at their compute-optimal configurations. Three additional models are predicted by the fitted parametric scaling law for larger budgets (2e19, 8e19, and 2.4e20 FLOPs) that correspond to compute scales available in the DCLM corpus.

Table 3.1 lists the seven models and their configurations. All models share the same architectural template: 64 experts per MoE layer, 8 active experts per token (2 shared, 6 routed), RMSNorm [27], rotary position embeddings [28], and SwiGLU activations [29].

3.3 Pre-training and Evaluation

3.3.1 Training Setup

All models are trained on the DCLM corpus using 32 NVIDIA H100 GPUs with expert parallelism across 8 GPUs per node (EP=8, PP=1). We use the Adam optimizer [30] with a maximum learning rate of 3×10^{-4}, minimum learning rate of 3×10^{-5}, and a warmup-stable-decay (WSD) schedule [31] with 1% warmup and 10% decay. The global batch size is 1024 with sequence length of 2048. Ten evenly-spaced checkpoints are saved during training for each model, along with full routing logs.

For each of the seven final models, we also train a dense baseline at the same FLOPs budget. The dense baselines use the same training recipe (optimizer, schedule, data) with a standard FFN in place of the MoE layer, allowing direct comparison of the MoE and dense architectures under matched compute. The training infrastructure is built on Megatron-LM; the complexity of adapting and operating this codebase at scale was one of the motivations for PithTrain (Chapter 4).

Table 3.2: Downstream evaluation results. Bold indicates the better result between MoE and dense at matched FLOPs.

FLOPs	Model	ARC-E	ARC-C	OBQA	HSwag	PIQA	WGrande	AVG
1.0e18	Dense-77M	0.327	0.219	0.246	0.259	0.558	0.495	0.351
	MoE-38M-100M	**0.361**	**0.224**	0.246	0.258	**0.562**	0.495	**0.358**
3.0e18	Dense-77M	0.375	0.207	0.244	0.263	0.583	0.504	0.363
	MoE-98M-349M	**0.422**	**0.215**	**0.266**	**0.279**	**0.614**	**0.524**	**0.387**
6.0e18	Dense-191M	0.432	0.210	**0.278**	0.277	0.602	0.492	0.382
	MoE-115M-459M	**0.465**	**0.241**	0.270	**0.310**	**0.624**	**0.520**	**0.405**
2.0e19	Dense-411M	0.480	0.242	0.294	0.324	0.640	**0.513**	0.415
	MoE-290M-1.3B	**0.520**	**0.247**	**0.302**	**0.357**	**0.668**	0.504	**0.433**
3.0e19	Dense-411M	0.505	0.257	0.294	0.335	0.639	**0.510**	0.423
	MoE-419M-2.2B	**0.548**	**0.284**	**0.310**	**0.394**	**0.681**	0.506	**0.454**
8.0e19	Dense-411M	0.548	0.257	0.294	0.378	0.665	**0.526**	0.445
	MoE-721M-3.8B	**0.596**	**0.288**	**0.320**	**0.454**	**0.699**	0.513	**0.478**
2.4e20	Dense-1.4B	0.604	0.302	0.336	0.465	0.684	0.518	0.485
	MoE-1.7B-10.3B	**0.647**	**0.317**	**0.350**	**0.530**	**0.724**	**0.536**	**0.517**

3.3.2 Downstream Evaluation

We evaluate all models on six downstream benchmarks: ARC-Easy and ARC-Challenge [32], OpenBookQA [33], HellaSwag [34], PIQA [35], and WinoGrande [36]. Following DCLM, ARC-Easy, ARC-Challenge, HellaSwag, and PIQA are evaluated with 10 shots; OpenBookQA and WinoGrande are evaluated zero-shot.

Table 3.2 reports the results. FLAME-MoE models outperform their dense baselines on nearly every task and at every scale. The average improvement ranges from 0.7 points at the smallest scale (1e18 FLOPs) to 3.2 points at the largest (2.4e20 FLOPs), with gains broadly increasing as the compute budget grows. At 2.4e20 FLOPs, FLAME-MoE-1.7B-10.3B achieves a 6.5-point advantage on HellaSwag over the dense baseline.

3.4 Empirical Analysis of Expert Behavior

The transparent training trace (routing logs and intermediate checkpoints across seven model scales) enables longitudinal analysis of how expert behavior develops during pre-training. We study three complementary aspects: how individual experts specialize on particular tokens, how experts are co-activated together, and how quickly routing decisions stabilize. Together, these characterize the emergence of MoE internal structure from three different angles. Unless otherwise noted, the analysis uses FLAME-MoE-1.7B-10.3B evaluated on the validation set.

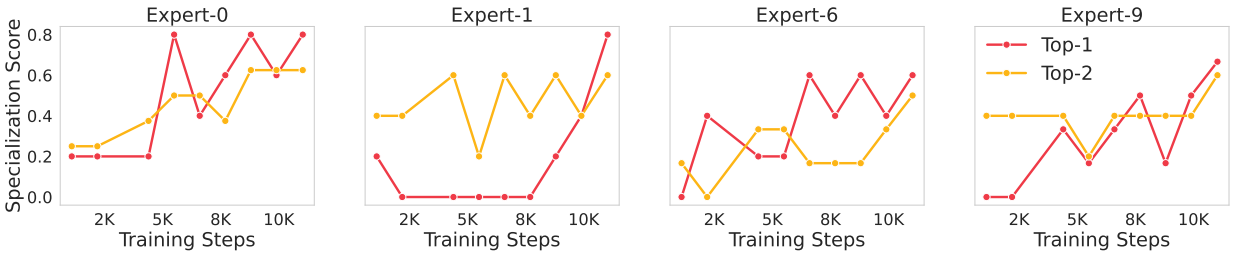


Figure 3.2: Evolution of specialization scores for the top-2 most specialized tokens across Experts 0, 1, 6, and 9 at the final layer in FLAME-MoE-1.7B-10.3B, evaluated on the validation set. Scores increase gradually throughout training.

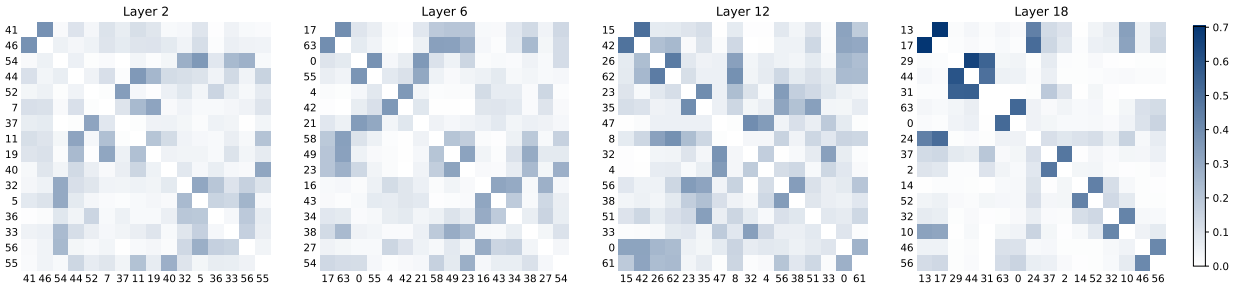


Figure 3.3: Expert co-activation in FLAME-MoE-1.7B-10.3B at convergence. Heatmaps show pairwise co-activation scores among the 16 most co-activated experts across layers 2, 6, 12, and 18. Co-activation is sparse in shallow layers and intensifies with depth.

3.4.1 Expert Specialization

We measure whether individual experts become responsible for particular tokens over the course of training. For a given token t and expert e, the specialization score is:

$$\text{Specialization}_e(t) = \frac{\text{number of times expert } e \text{ activates on } t}{\text{number of times } t \text{ appears in the evaluation set}}. \tag{3.3}$$

A high score means expert e is consistently selected when token t appears.

To track how specialization evolves, we identify the top-2 most specialized tokens for each expert at the final checkpoint and measure their specialization scores retrospectively at earlier checkpoints. We observe a consistent upward trend: specialization scores increase gradually throughout training, indicating that experts progressively focus on distinct token subsets rather than acquiring their roles abruptly. The trend is present across all experts analyzed, though the rate and magnitude of specialization vary between experts. The pattern is qualitatively consistent across model scales, from FLAME-MoE-38M-100M to FLAME-MoE-1.7B-10.3B. Figure 3.2 shows the evolution of specialization scores for selected experts in the largest model.

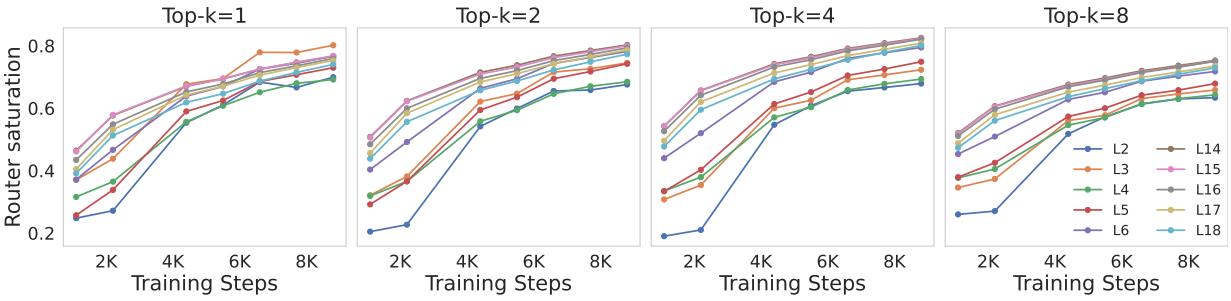


Figure 3.4: Router saturation across training for FLAME-MoE-1.7B-10.3B. Each subplot shows the average expert selection overlap with the final checkpoint using different top-k values (1, 2, 4, 8). Most layers exceed 70% saturation by the training midpoint.

3.4.2 Expert Co-activation

We next examine which experts tend to be selected together for the same tokens. The directional co-activation score from expert E_i to expert E_j is:

$$\text{CoAct}(E_i, E_j) = \frac{|\{x \in \mathcal{T} : E_i \in \text{Top-}k(x) \text{ and } E_j \in \text{Top-}k(x)\}|}{|\{x \in \mathcal{T} : E_i \in \text{Top-}k(x)\}|}, \tag{3.4}$$

where $\mathcal{T}$ is the evaluation token set. This metric is asymmetric: $\text{CoAct}(E_i, E_j) \neq \text{CoAct}(E_j, E_i)$ in general, since the two experts may have different activation frequencies.

Co-activation is generally sparse, with most expert pairs exhibiting low scores. However, the pattern varies with depth: shallow layers (e.g., layer 2) show maximum pairwise scores around 0.38, while deeper layers exhibit higher co-activation, reaching 0.70 at layer 18. Co-activation also intensifies during training: at layer 18, the peak score grows from 0.51 at 10% of training to 0.70 at convergence.

The sparsity of co-activation across most pairs suggests that experts learn diverse, complementary roles rather than forming redundant clusters. The increasing co-activation in deeper layers is consistent with the observation that later layers perform more specialized, compositional processing where certain expert combinations become reliably useful for particular input patterns. Across model scales, larger models tend to exhibit broader co-activation in shallow layers, while deep-layer co-activation patterns are similar across scales. Figure 3.3 shows the co-activation heatmaps across layers at convergence.

3.4.3 Router Saturation

Finally, we measure how quickly routing decisions stabilize during training. The saturation score at training step t for layer l is:

$$\text{Saturation}_l(t) = \frac{1}{|\mathcal{T}|} \sum_{x \in \mathcal{T}} \frac{|\text{Top-}k_l^{(t)}(x) \cap \text{Top-}k_l^{(T)}(x)|}{k}, \tag{3.5}$$

where $\text{Top-}k_l^{(t)}(x)$ is the set of experts selected for token x at step t, and T is the final training step. A saturation score of 1.0 means the router assigns exactly the same experts as at convergence.

Saturation increases steadily throughout training, with a sharp rise in the first few thousand steps. Most layers reach over 70% agreement with the final routing by the midpoint of training. Deeper layers saturate faster than shallower layers, and the pattern is consistent across all top-k settings ($k = 1, 2, 4, 8$).

This early convergence of routing decisions suggests that the router establishes its coarse assignment structure relatively quickly, with the remainder of training refining expert weights rather than substantially reorganizing which experts handle which tokens. Larger models exhibit somewhat slower saturation in early layers, suggesting greater routing plasticity early in training, but all scales converge to similar saturation levels by the end of training. Figure 3.4 shows saturation curves across layers and top-k settings.

Chapter 4

PithTrain

PithTrain is an open-source framework for end-to-end MoE language model training. It supports pipeline, data, context, and expert parallelism, FP8 training on Hopper and Blackwell GPUs, and a pipeline schedule that overlaps computation with communication, all in roughly 10K lines of Python. The framework builds on established components: DeepSeek's DualPipe [14] pipeline-orchestration scaffold, DeepGEMM [16] for FP8 GEMMs, FlashAttention [37] for attention, and PyTorch's FSDP2 [38] for sharding parameters, gradients, and optimizer states. PithTrain integrates these components into a training system whose entire code path, from the training loop to the GPU kernels, can be read end-to-end by both humans and AI coding agents.

Production MoE training frameworks such as Megatron-LM [7, 8] and DeepSpeed-MoE [9, 10] have built end-to-end MoE training stacks over years of engineering effort. They pair layered Python designs with C++/CUDA extensions to deliver broad model coverage, peak throughput, and the multi-platform support needed for diverse training workloads. These same patterns, however, raise the cost of working in the codebase: plugin systems, registry-based indirection, and heavy C++/CUDA extensions inflate the cost of locating relevant code, tracing what runs at a given call site, and verifying a change is complete. This cost is paid by both human engineers and, increasingly, by AI coding agents that automate parts of training-system development. PithTrain explores a different point on this design tradeoff: a compact, Python-native codebase that an agent or a researcher can read, trace, and operate efficiently, while still matching the throughput of production frameworks.

The system is organized around three design principles. *Compactness:* we narrow scope to a focused MoE training stack rather than the broad model and feature coverage of production frameworks, so the codebase fits in a single context window. *Python-native:* the entire framework is pure Python, with custom GPU kernels written in Triton; orthogonal optimized libraries (FlashAttention, DeepGEMM) provide vendor-tuned numerics without forcing the reader across language boundaries. *No implicit indirection:* module composition uses direct calls rather than runtime specs, plugin registries, or string-keyed resolution, so what runs at a given call site can be identified by static reading.

This chapter describes the system architecture (§4.1), the parallelism strategy (§4.2), the overlapped pipeline scheduler (§4.3), FP8 training (§4.4), the throughput evaluation (§4.5), and an evaluation of agent-task efficiency on a suite of New-Feature integration tasks (§4.6).

Table 4.1: PithTrain line count by component. Totals exclude third-party libraries.

Layer	Components	Lines
Upstream	Pre-training entry and training loop	∼0.5K
Core	Model definitions (Qwen3-30B-A3B, DeepSeek-V2-Lite)	∼1.8K
	Pipeline engine (DualPipeV scheduler, stage execution)	∼2.3K
	Training utilities (layers, optimizer, checkpointing)	∼1.7K
	Distributed setup (device mesh, parallelism groups)	∼0.2K
Operators	Triton & TileLang kernels	∼2.6K
	Total	∼9.1K

4.1 System Overview

PithTrain is organized in three layers. The *upstream* layer contains the pre-training entry point, which constructs the training configuration and dispatches to the core engine. The *core* layer implements model definitions, the pipeline engine, distributed setup, and training utilities such as the optimizer and checkpoint handling. The *operator* layer provides custom GPU kernels written in Triton and TileLang, JIT-compiled at runtime.

Table 4.1 summarizes the line counts by component. The framework adds no custom C++ build steps: orchestration, scheduling, and parallelism logic are pure Python; tensors and autograd are provided by PyTorch; attention and FP8 GEMMs are delegated to FlashAttention [37] and DeepGEMM [16]; and the remaining custom kernels are JIT-compiled at runtime from Triton and TileLang sources.

PithTrain exposes training through a Python configuration script rather than a YAML or JSON format: users set parallelism sizes, model paths, optimizer settings, and batch sizes as fields on a dataclass before calling `launch(cfg)`. Two model architectures are currently implemented: Qwen3-30B-A3B [39] and DeepSeek-V2-Lite [21]. Checkpoints are stored in PyTorch's distributed checkpoint (DCP) format during training, and a conversion utility exports them to Hugging Face format for use with downstream tools.

The relatively small codebase is not achieved by dropping features but by minimizing abstraction layers. PithTrain does not introduce a plugin system, a registry pattern, or config-driven dispatch; each component is implemented directly, and most behavior can be traced from the training loop to a GPU kernel through a small number of function calls.

4.2 Parallelism Strategy

Training MoE models at the scales we target requires composing multiple parallelism strategies. PithTrain constructs a four-dimensional device mesh with pipeline (PP) $\times$ data (DP) $\times$ context (CP) $\times$ expert (EP) parallelism, where data parallelism is implemented via FSDP2 [38]. The mesh is constructed once at training startup and exposed to the rest of the system through a flat distributed-context abstraction.

Table 4.2: Layer decomposition used by the PithTrain pipeline scheduler.

Stage	Name	Type	Operation
1	Attention	Compute	Pre-norm, self-attention, post-norm, router
2	Dispatch	Communication	All-to-all scatter of tokens to expert-parallel ranks
3	Expert	Compute	Grouped GEMM over routed experts
4	Combine	Communication	All-to-all gather of expert outputs
5	Aggregate	Compute	Weighted combination and residual connection

The mapping from mesh dimensions to physical hardware matters for performance because different parallelism strategies have different communication patterns. Expert parallelism is placed on the innermost dimension so that all-to-all traffic for token dispatch stays on high-bandwidth intra-node NVLink interconnects. Pipeline parallelism is placed on the outermost dimension, since point-to-point activation transfers between pipeline stages are infrequent relative to the compute they overlap and can tolerate lower inter-node bandwidth. Data and context parallelism occupy the intermediate dimensions.

4.3 Overlapped Pipeline Scheduling

The pipeline scheduler is the core mechanism through which PithTrain achieves competitive throughput. It builds on DeepSeek's DualPipe [14], which provides a reference implementation of bidirectional overlapped pipeline parallelism for simple MLPs. PithTrain adopts the DualPipeV variant [15], a V-shaped schedule introduced by Sea AI Lab that requires only PP/2 devices instead of PP, and extends it to full MoE transformer training with FSDP2 integration, optional FP8 support, and the ability to run complete Qwen3 and DeepSeek-V2 models.

4.3.1 Layer Decomposition

PithTrain decomposes each MoE transformer layer into five stages, shown in Table 4.2. Two are all-to-all communication (dispatch and combine); the remaining three are compute (attention, expert GEMMs, and final aggregation). The decomposition places the communication boundaries at the natural seams of the MoE layer, so that each stage can be scheduled independently by the pipeline engine.

Each stage is implemented as both a forward and backward function, together with merged variants for cross-layer fusion (for example, the aggregate of layer l and the attention of layer $l+1$ are fused when scheduled back-to-back). This produces a set of roughly a dozen stage functions that the scheduler composes into the full forward and backward pass.

This granularity is what enables compute-communication overlap. During the steady-state phase of the schedule, one micro-batch's dispatch or combine runs on a dedicated NCCL stream while a different micro-batch's compute stage runs on the default stream. Figure 4.1 shows this behavior in an Nsight Systems trace of a Qwen3-235B-A22B training run.

Figure 4.1: Nsight Systems trace of a Qwen3-235B-A22B training step. Compute kernels run on the default CUDA stream (top); dispatch and combine all-to-all kernels run on a separate NCCL stream (bottom).

4.3.2 Deferred Weight Gradients

A standard optimization in zero-bubble pipeline schedules is to decouple the computation of input gradients (which are on the pipeline's critical path) from weight gradients (which only need to be ready at the optimizer step). PithTrain applies this decoupling inside the expert stage: input gradients are computed as soon as the incoming error signal arrives, while weight gradient computation is deferred and scheduled into pipeline slots that would otherwise be idle.

4.3.3 Token Dispatch

Token dispatch, the permutation of tokens by expert assignment that precedes the all-to-all communication, is implemented in a small set of fused Triton kernels. A naive implementation of this permutation involves a sequence of scatter, argsort, nonzero, and searchsorted operations that launch many small kernels per layer. PithTrain replaces this sequence with fused kernels that combine dedup bincount, prefix-sum metadata construction, and counting-sort scatter.

4.4 FP8 Training

PithTrain supports FP8 training through DeepGEMM [16], which provides block-scaled FP8 GEMM kernels for both Hopper (SM90) and Blackwell (SM100+) GPUs. The integration consists of an `FP8Linear` module that replaces `nn.Linear` in both the attention projections and the expert FFNs, and a grouped variant `FP8GroupLinear` used for grouped expert GEMMs. Quantization is performed at 128-element block granularity inside a small set of Triton kernels that fuse the row-wise quantization, transpose, and FP8 cast, with architecture-specific scale formats: FP32 power-of-2 scales on Hopper (SM90) and E8M0 scales on Blackwell (SM100+), the latter computed via the hardware PTX instruction `cvt.rp.satfinite.ue8m0x2.f32`. Weights quantized to FP8 are reused across the micro-batches within a single optimizer step to save redundant quantization work.

Table 4.3: End-to-end training throughput in tokens per second across frameworks. Parallelism is reported as PP×DP×CP×EP. "—" denotes not supported; "OOM" denotes out of memory.

Model	Hardware	Parallelism / SeqLen / Precision	Megatron-LM	TorchTitan	PithTrain
GPT-OSS-20B	1×8 B200	2-1-1-4 / 8192 / BF16	129.5K	—	140.9K
Qwen3-30B-A3B	1×8 B200	2-1-1-4 / 8192 / FP8	106.2K	OOM	134.5K
Qwen3-30B-A3B	2×8 H100	2-1-1-8 / 2048 / BF16	126.7K	90.5K	124.9K
Qwen3-30B-A3B	4×8 H100	4-1-1-8 / 4096 / BF16	264.1K	OOM	280.0K
DeepSeek-V2-Lite	1×8 H100	2-1-1-4 / 2048 / BF16	107.3K	74.1K	114.6K

4.5 Evaluation

We compare PithTrain, Megatron-LM [8], and TorchTitan [17] on three representative MoE models (GPT-OSS-20B [3], Qwen3-30B-A3B [39], and DeepSeek-V2-Lite [21]) under matched parallelism, sequence length, and precision. Configurations span single-node and multi-node deployments on NVIDIA H100 and B200 GPUs. For Megatron-LM, we follow NVIDIA's documented best practices for MoE optimization. DeepSpeed-MoE [10] is excluded because its current code path does not support pipeline parallelism combined with expert parallelism for MoE training, so it cannot run any of the configurations in our suite. To ensure the MoE router exhibits steady-state load-balanced routing across frameworks (and thus produces comparable throughput), we initialize each run from a public pre-trained checkpoint rather than from random initialization. Each measurement runs 25 steps and reports the median step time over the last 10.

We report aggregate tokens per second as the primary throughput metric. Model FLOPs Utilization is omitted: Tensor Core peak FLOPS differs between BF16 and FP8, which makes the metric ambiguous for mixed-precision steps and not directly comparable across configurations.

4.5.1 Training Throughput

Table 4.3 reports the results. PithTrain matches or exceeds Megatron-LM on four of the five configurations and stays within 1.4% on the fifth, while leading TorchTitan on every configuration where TorchTitan runs. This parity suggests that a compact, Python-native codebase can reach the throughput of mature production frameworks: the headline optimizations (DualPipeV's compute–communication overlap, `torch.compile` on attention and aggregation stages, FP8 GEMMs through DeepGEMM, fused token dispatch) do not need to be implemented inside a heavy C++/CUDA stack to deliver competitive performance.

4.5.2 Training Correctness

We pre-train Qwen3-30B-A3B from scratch on 8B tokens of the DCLM corpus with both PithTrain and Megatron-LM under the same configuration, and verify that the cross-entropy loss decreases smoothly and tracks between the two frameworks. The full set of hyperparameters used for the validation runs is listed in Table 4.4. Figure 4.2 shows the training loss and the accuracy on HellaSwag [34], a representative downstream benchmark; the two frameworks produce closely

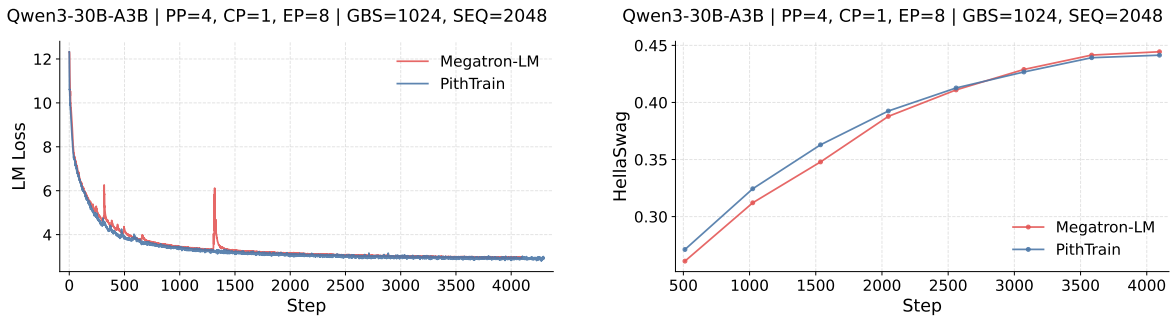
Table 4.4: Configuration used for the PithTrain and Megatron-LM training-correctness validation runs. Both frameworks run with the same settings.

Category	Setting	Value
Parallelism	Hardware	4×8 H100
	Pipeline parallelism	PP=4
	Expert parallelism	EP=8
	Context parallelism	CP=1
Schedule	Sequence length	2048
	Micro batch size	1
	Global batch size	1024
	Training steps	4,096
	Tokens processed	∼8.6B
Optimizer	Optimizer	Adam
	LR schedule	Cosine annealing
	Peak LR	2.0×10^{-4}
	Min LR	1.0×10^{-5}
	Warmup steps	128
	Weight decay	0
Precision	Compute dtype	BF16
	FP8	disabled
MoE	Experts / top-k	128 / 8
	Load-balance scope	Global batch [24]
	Load-balance coefficient	1.0×10^{-3}
Data	Corpus	DCLM-Baseline [12]
	Tokenizer	Qwen3-30B-A3B

aligned curves on both metrics. This is not a test of model quality at scale (8B tokens is well below the compute budget at which the 30B model would converge), but it confirms that the end-to-end training pipeline is functional. A converted checkpoint further loads into Hugging Face Transformers and serves through vLLM [40] and SGLang [41] without modification.

4.6 Agent-Task Efficiency on New-Feature Tasks

An argument for a lightweight, Python-native codebase is that it lowers the cost of extending the training system with new research ideas, particularly when the implementer is an AI coding agent. We evaluate this claim on a suite of *New-Feature* tasks, each of which asks an autonomous coding agent to port a published transformer-architecture variant end-to-end into a target framework. Holding the agent and task fixed and varying the framework isolates the framework's contribution to agent-task efficiency.



(a) Training loss

(b) HellaSwag accuracy

Figure 4.2: Training correctness validation. (a) Cross-entropy training loss over 4,096 steps of Qwen3-30B-A3B pre-training. (b) Accuracy on HellaSwag evaluated periodically during training. PithTrain and Megatron-LM produce closely aligned curves under the same configuration.

4.6.1 Setup

We compare PithTrain, Megatron-LM, and TorchTitan on four New-Feature tasks drawn from recent literature: Differential Transformer [42], Dynamic Mixture of Experts [43], Mixture of Block Attention [44], and MoE++ [45]. Each agent session is run with Claude Opus 4.7 [46] under Claude Code, with the 1M-token context window, given the task prompt, pointers to the published reference implementation, and read–write access to the target codebase. The agent is responsible for the full integration: design choices, code edits, debugging, and verification through a smoke test (`verify.sh`, 3 steps) followed by a longer training run (`train.sh`, 64 steps). Outcome is automated: the run is considered successful when the cross-entropy loss decreases at the rate the task specifies.

We report five effort metrics per (framework, task) trial. *Session duration* is wall-clock time from the agent's first action to its final report. *Active GPU time* sums GPU-minutes across the eight H100s used during `verify.sh` and `train.sh` executions, recovered from per-launch timestamps. *Agent turns* counts logical model turns. *Per-turn context* is the median input-token count fed to the model on each turn. *Output tokens* sums the model's generated tokens across the session. Each trial is repeated three times under independent random seeds; we report the median. Lower is more efficient on every metric.

4.6.2 Results

Table 4.5 reports the results. PithTrain leads on session duration and active GPU time across all four tasks, with median reductions of 1.4–1.7× over Megatron-LM and 1.3–1.7× over TorchTitan. On agent turns and output tokens, PithTrain leads on three of four tasks, with reductions of 1.6–2.6× over Megatron-LM. The exception is MoE++, where TorchTitan reaches a working implementation with fewer turns and tokens than PithTrain; this appears to reflect TorchTitan's modular MoE abstraction making zero/copy/constant expert types straightforward to add. The headline pattern is that the iteration cost (active GPU time) drops more than the writing cost

Table 4.5: Per-task agent effort across frameworks on four New-Feature tasks. Each cell is the median across three independent runs; **bold** marks the best framework on a metric for that task. Lower is more efficient.

Framework	Session Duration	Active GPU Time	Agent Turns	Per-Turn Context	Output Tokens
		Differential Transformer [42]			
Megatron-LM	46.3 min	57.4 min	125	118.7K	57.1K
TorchTitan	49.6 min	346.4 min	58	103.2K	36.0K
PithTrain	**38.2 min**	**25.9 min**	**47**	**69.5K**	**25.4K**
		Dynamic Mixture of Experts [43]			
Megatron-LM	83.8 min	374.0 min	199	208.0K	115.2K
TorchTitan	140.6 min	798.2 min	197	228.8K	161.3K
PithTrain	**60.4 min**	**335.3 min**	**76**	**146.0K**	**76.4K**
		Mixture of Block Attention [44]			
Megatron-LM	61.6 min	395.6 min	134	120.9K	53.8K
TorchTitan	105.1 min	769.8 min	91	166.9K	111.8K
PithTrain	**38.1 min**	**255.8 min**	**57**	**69.0K**	**32.4K**
		MoE++ [45]			
Megatron-LM	88.5 min	487.6 min	145	188.7K	117.0K
TorchTitan	71.4 min	422.5 min	**87**	**164.6K**	**85.3K**
PithTrain	**63.0 min**	**342.0 min**	90	176.6K	107.7K

(output tokens), suggesting that the bulk of the savings comes from converging on a working implementation in fewer trial runs, not from writing less code.

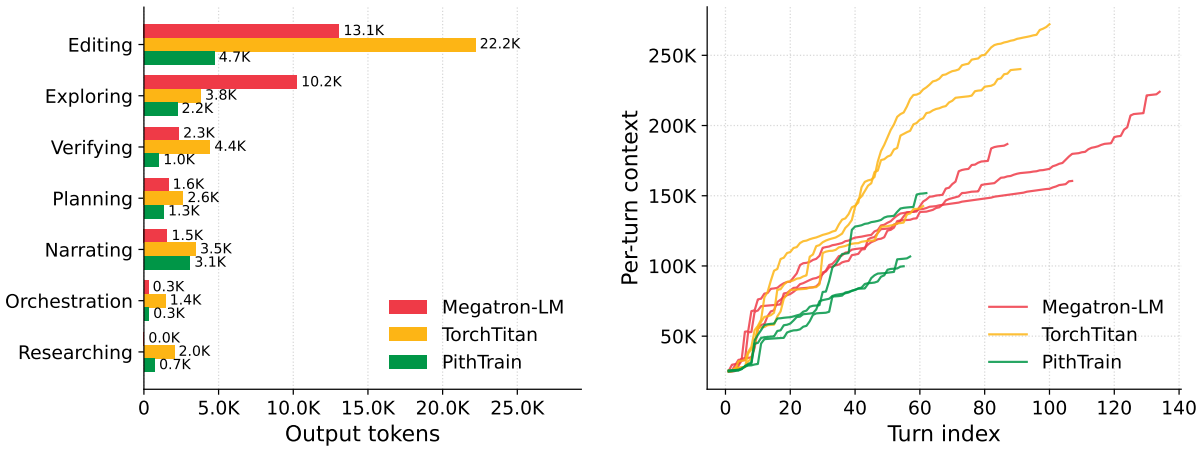
4.6.3 Case Study: Mixture of Block Attention

To understand where the cross-framework effort gap comes from on a single concrete task, we look at the integration of Mixture of Block Attention [44] in greater detail. PithTrain leads either baseline on every effort metric in Table 4.5: the iteration cost, measured in active GPU time, is 35% lower than Megatron-LM and 67% lower than TorchTitan.

Figure 4.3(a) decomposes where the extra output tokens go. Megatron-LM's two largest action-category bars are *editing* (13.1K) and *exploring* (10.2K), against PithTrain's 4.7K and 2.2K respectively. TorchTitan's action-category bars are close to PithTrain's, but Figure 4.3(b) shows its median per-turn context running over 2× PithTrain's, indicating that more files accumulate in the agent's working set even though the writing cost is similar.

To understand where the iteration cost gap comes from, we zoom in on the test–debug arc of the agent loop: after each crashing training run, the agent reads the traceback to decide what to fix before retrying. We classify each failed run by where its traceback terminates: *agent-code* when the terminal frame is in a file the agent edited and the fix site is the same file, and *framework-internal* when the terminal frame is in framework code and recovery requires reconstructing a contract buried inside the framework.

On PithTrain, the only failure observed is a tensor-stride mismatch from `torch.compile` on the agent's custom attention kernel; the traceback ends inside the file the agent just wrote, and



(a) Per-category breakdown of output tokens.

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

Figure 4.3: Agent behavior on integrating Mixture of Block Attention across frameworks. (a) shows how the agent's output tokens distribute across action categories; (b) shows how the per-turn input context grows over the session.

the fix is on the same line. On Megatron-LM, the two failures observed both surface through framework internals: a duplicate command-line flag registration that aborts the argument parser before it finishes building, and a BF16 overflow during a masked fill whose underlying dtype-promotion rule is hidden under a multi-rank distributed-launch wrapper before reaching the agent. On TorchTitan, the same stride mismatch encountered on PithTrain is followed by additional failures: the agent's reference attention path materializes a dense per-query mask, the same approach PithTrain and Megatron-LM adopt, but on TorchTitan its memory footprint exceeds available GPU memory, and subsequent trial runs fail in FSDP's BF16 gradient cast during backward. The agent's debugging effort therefore shifts from the attention kernel it just edited to the memory budget itself.

Taken together, the cross-framework effort gap reflects the friction the framework adds to the agent's debug loop, not the difficulty of the bugs themselves. The two baselines trade off differently across the two efficiency axes: Megatron-LM is training-efficient, but its hidden contracts inflate diagnostic loops at the agent-task layer; TorchTitan offers more accessible code paths but less memory headroom, requiring the agent to debug memory alongside the implementation itself. PithTrain is competitive on both axes: diagnostic loops stay local to the agent's edits, and training efficiency keeps the test–debug iteration short.

Chapter 5

Conclusion

This thesis presented two perspectives on training Mixture-of-Experts language models. FLAME-MoE is a transparent research platform that releases seven MoE models from 38M to 1.7B active parameters with full code, data pipelines, intermediate checkpoints, and routing logs, and uses the resulting trace to characterize how expert specialization, co-activation, and router saturation develop during pre-training. PithTrain is a Python-native MoE training framework, roughly 10K lines, that matches the throughput of mature production frameworks while remaining compact enough to be read end-to-end. The two projects are independent efforts that share a subject rather than a codebase; below we record two broad lessons that came out of working on both.

Model-infra co-design. Both parametric sparsity (MoE) and contextual sparsity push training systems beyond what a generic dense trainer handles. Dynamic shapes, all-to-all dispatch, and load imbalance are MoE-specific concerns that propagate through every layer of the stack: data loading, communication, kernel selection, memory accounting. Decisions made on the architecture side (number of routed experts, load-balancing scope, whether some experts perform zero computation) shift where the bottleneck lives in the systems below. The takeaway, for us, is the value of designing hardware-aligned architectures that expand the computational and representational efficiency of large-scale foundation models.

Agent-native development. Research codebases are increasingly read by AI coding agents in addition to human engineers, and the design choices that helped human engineers can cost agents differently. The agent-task evaluation in Chapter 4 compares Claude Opus 4.7 integrating four new architectural variants into PithTrain, Megatron-LM, and TorchTitan: the smaller codebase requires less iteration cost on every task, and a deeper case study on Mixture of Block Attention shows that on Megatron-LM the agent spends most of its debugging time recovering framework-internal contracts rather than fixing its own code. Patterns that are inexpensive for a human (a registry indexed by string keys, a configuration that resolves through three layers of indirection, an assertion that encodes an unwritten invariant) are expensive for an agent because they are visible nowhere except in the source itself. Agent-task efficiency is an important metric to measure alongside training throughput, and a property worth designing for rather than one that emerges incidentally.

Bibliography

[1] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025. 1

[2] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`. 1

[3] Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025. 1, 4.5

[4] Niklas Muennighoff, Luca Yang, Weijia Shi, Xiang Lisa Li, Jason Fries, Jiawei Shi, Kyle Lo, Luke Zettlemoyer, Hannaneh Hajishirzi, and Noah A. Smith. OLMoE: Open mixture-of-experts language models. *ArXiv preprint*, 2024. 1

[5] Fuzhao Xue, Zian Zheng, Yao Fu, Jinjie Ni, Zangwei Zheng, Wangchunshu Zhou, and Yang You. OpenMoE: An early effort on open mixture-of-experts language models. *ArXiv preprint*, 2024. 1

[6] Yikang Shen, Zhen Zhang, Tianyou Cao, Shawn Ott, Sashank Narang, and Jing Shang. JetMoE: Reaching llama2 performance with 0.1m dollars. *ArXiv preprint*, 2024. 1

[7] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. 1, 3, 4

[8] Zijie Yan, Hongxiao Bai, Xin Yao, Dennis Liu, Tong Liu, Hongbin Liu, Pingtian Li, Evan Wu, Shiqing Fan, Li Tao, et al. Scalable training of mixture-of-experts models with megatron core. *arXiv preprint arXiv:2603.07685*, 2026. 1, 4, 4.5

[9] 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*, pages 3505–3506, 2020. 1, 4

[10] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022. 1, 4, 4.5

[11] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab

Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022. 1

[12] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, George Arber, et al. DataComp-LM: In search of the next generation of training data for language models. *Proc. of NeurIPS*, 2024. 1, 3, 4.4

[13] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, DDL Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 10, 2022. 1, 3.2, 3.2.1

[14] DeepSeek-AI. DualPipe: Bidirectional pipeline parallelism for computation-communication overlap. https://github.com/deepseek-ai/DualPipe, 2025. 1, 4, 4.3

[15] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Dualpipe could be better without the dual. https://hackmd.io/@ufotalent/r1lVXsa9Jg, 2025. Blog. 1, 4.3

[16] DeepSeek-AI. DeepGEMM: Clean and efficient fp8 gemm kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025. 1, 4, 4.1, 4.4

[17] PyTorch Team. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. https://github.com/pytorch/torchtitan, 2024. 1, 4.5

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Proc. of NeurIPS*, 2017. 2.1

[19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019. 2.1

[20] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *ArXiv preprint*, 2017. 2.1, 2.1.2, 3.1

[21] DeepSeek-AI. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. *ArXiv preprint*, 2024. 2.1.1, 3.1, 4.1, 4.5

[22] Zewen Chi, Li Dong, Shaohan Huang, Damai Dai, Shuming Ma, Barun Patra, Saksham Singhal, Payal Bajaj, Xia Song, Xian-Ling Mao, et al. On the representation collapse of sparse mixture of experts. *Proc. of NeurIPS*, 2022. 2.1.2

[23] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *JMLR*, 2022. 2.1.2

[24] 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. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5005–5018, 2025. 2.1.2, 4.4

[25] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. ST-MoE: Designing stable and transferable sparse expert models. *ArXiv preprint*, 2022. 3.1

[26] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics: Methodology and distribution*, pages 492–518. Springer, 1992. 3.2.2

[27] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in neural information processing systems*, 32, 2019. 3.2.3

[28] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. 3.2.3

[29] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020. 3.2.3

[30] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 3.3.1

[31] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. MiniCPM: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024. 3.3.1

[32] 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*, 2018. 3.3.2

[33] 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 *Proc. of EMNLP*, 2018. 3.3.2

[34] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proc. of ACL*, 2019. 3.3.2, 4.5.2

[35] Yonatan Bisk, Rowan Zellers, Ronan LeBras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proc. of AAAI*, 2020. 3.3.2

[36] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. In *Proc. of AAAI*, 2020. 3.3.2

[37] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Proc. of NeurIPS*, 2022. 4, 4.1

[38] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proc. of VLDB*, 2023. 4, 4.2

[39] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. 4.1, 4.5

[40] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. 4.5.2

[41] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024. 4.5.2

[42] Tianzhu Ye, Li Dong, Yuqing Xia, Yutao Sun, Yi Zhu, Gao Huang, and Furu Wei. Differential transformer. *arXiv preprint arXiv:2410.05258*, 2024. 4.6.1, 4.5

[43] Yongxin Guo, Zhenglin Cheng, Xiaoying Tang, and Tao Lin. Dynamic mixture of experts: An auto-tuning approach for efficient transformer models. *arXiv preprint arXiv:2405.14297*, 2024. 4.6.1, 4.5

[44] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Zhu, 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. 4.6.1, 4.5, 4.6.3

[45] Peng Jin, Bo Zhu, Li Yuan, and Shuicheng Yan. MoE++: Accelerating mixture-of-experts methods with zero-computation experts. *arXiv preprint arXiv:2410.07348*, 2024. 4.6.1, 4.5

[46] Anthropic. Introducing claude opus 4.7. `https://www.anthropic.com/news/claude-opus-4-7`, 2026. 4.6.1