TRAINTAILOR: A CODING AGENT FOR SYNTHESIZING MODEL-SPECIFIC TRAINING ENGINES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are released at a rapid pace, requiring training frameworks to continually adapt to new architectures. However, adapting general-purpose training frameworks to these architectures requires substantial engineering effort, while their shared abstractions can introduce runtime overhead and constrain model-specific optimization. In this paper, we explore an alternative enabled by recent advances in coding agents: building a dedicated training engine from scratch for each model. We instantiate this idea as TrainTailor, an agent-driven system that synthesizes and optimizes a standalone training engine tailored to a given model, training task, and hardware environment. Specifically, TrainTailor first builds a functional training engine with an initial execution configuration for the target hardware and validates its training behavior against a trusted reference. It then iteratively refines the engine implementation and execution configuration together, guided by profiling and bottleneck analysis, to improve training efficiency. Freed from the need to support diverse architectures, the generated engine can eliminate unnecessary abstractions and enable model-specific optimizations, such as architecture-aware operator fusion. Experiments show that the engines generated by TrainTailor satisfy the prescribed numerical tolerances over 30 steps, while improving model FLOPs utilization (MFU) by approximately 30% on average.

1 INTRODUCTION

New models are being released at an increasing pace (Figure 1), requiring training frameworks to continually adapt to evolving architectures. Turning a new model architecture into an efficient training system requires integrating its computation with data processing, optimizer updates, and distributed execution. General-purpose frameworks reduce repeated engineering by providing reusable infrastructure across models and workloads (Narayanan et al., 2021; Zheng et al., 2024). However, supporting a new architecture can require coordinated changes across multiple layers of the framework, creating a recurring engineering burden as models evolve. Even after integration, obtaining an efficient implementation requires adapting execution to the model, training task, and available hardware.

Advances in coding agents (Yang et al., 2024; Wei et al., 2025) offer an opportunity to automate training-system development and make dedicated implementations more practical. ForgeTrain (He et al., 2026) demonstrates that agents can generate dedicated training engines from scratch using a trusted training implementation as a reference. An engine built for a specific model, task, and hardware environment need not accommodate unrelated architectures or workloads, allowing its computation, memory use, and communication to be organized around that target. We investigate *model-specific training engine synthesis*: constructing a standalone engine from a model definition and a training task, and jointly refining its implementation and execution configuration for the target hardware. The process does not require an existing optimized training implementation of the target model.

We present **TrainTailor**, a coding-agent system that synthesizes and optimizes model-specific training engines through two stages: *Build* and *Optimize*.[1] A model definition specifies the model's

[1]Code is available at `https://anonymous.4open.science/r/TrainTailor`.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

computation, but constructing a complete engine also requires integrating the surrounding workflow while preserving the training behavior required by the task. To guide this construction, Build analyzes the model source to understand its structure and computational behavior, organizing this knowledge into a structural blueprint and behavioral knowledge cards. These artifacts guide the agent in developing an initial implementation and execution configuration for the target hardware. Verification checks the engine against the task requirements and reference training behavior, with failures guiding repairs to the knowledge artifacts or implementation.

Optimizing this engine requires accounting for the interaction between its implementation and execution configuration. Choices such as parallel layout and microbatch size shape how the engine organizes training, while implementation changes can alter which configurations are feasible or efficient. Optimize therefore uses source analysis and profiling to identify bottlenecks, then jointly refines the implementation and configuration using execution feedback. As changes alter memory requirements and execution costs, the agent reconsiders configurations that were previously ruled out. With the model, training task, and hardware held fixed, TrainTailor applies the same verification criteria throughout optimization and retains the best verified engine found.

We evaluate TrainTailor on Qwen3.5-0.8B and a reduced Qwen3.8-Flash-Next model with approximately 15B parameters, covering architectures with and without native support in the evaluated training frameworks. The generated engines satisfy the prescribed numerical tolerances over 30 optimizer steps. On Qwen3.5-0.8B, the optimized engine achieves 26.94% model FLOPs utilization (MFU), improving MFU by 30.47% relative to LlamaFactory and 33.66% relative to Megatron-Bridge under the same training task and hardware. An optimization case study traces how implementation changes and configuration decisions contribute to the engine's performance.

Our contributions are as follows:

- We introduce TrainTailor, which uses structured model knowledge to guide the construction and verification of a standalone training engine from a model definition and a training task.
- We develop a workflow that jointly refines the engine implementation and execution configuration, using performance feedback to guide optimization under shared verification criteria.
- We evaluate engine synthesis on two model architectures and demonstrate training-performance gains over established frameworks on Qwen3.5-0.8B, with a case study of the optimization process.

2 Background and Motivation

2.1 Supporting New Models in Training Frameworks

As shown in Figure 1, new models are released at a rapid pace (Luo et al., 2026), requiring general-purpose training frameworks such as Megatron-LM and LlamaFactory (Narayanan et al., 2021; Zheng et al., 2024) to keep pace with them. Supporting a new model often requires changes across components for parallel execution, memory optimization, and other training functionality. In these frameworks, the components are organized through multiple layers of abstraction and are highly interdependent, making the changes difficult to coordinate. For example, supporting Qwen3.5 (Qwen Team, 2026a) in Megatron-LM involved changes spanning model integration, positional encoding, and distributed training. The investigation of numerical issues in packed-sequence training also extended over several weeks.[2]

2.2 Coding Agents for Training Engine Development

Coding agents have demonstrated strong capabilities in software development (Yang et al., 2024). Recent work has also shown that agents can improve GPU kernel performance (Wei et al., 2025). These advances motivate extending agent-driven development from individual kernels to complete training engines. However, layered abstractions and implicit indirection in general-purpose training

[2]Megatron-LM PR #4751 summarizes the changes for Qwen3.5 support. PR #2644 documents the investigation of numerical issues in packed-sequence training and the associated workarounds.

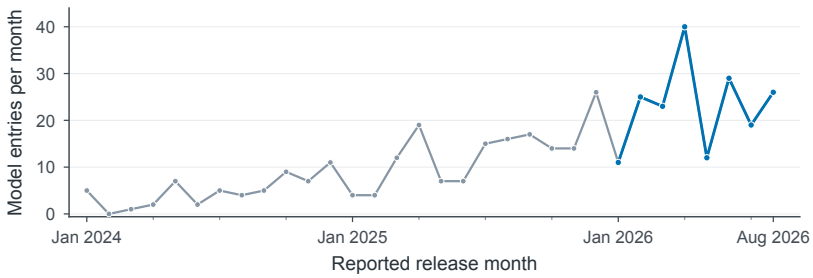


Figure 1: Monthly model entries by reported release date in Models.dev, January 2024–August 2026, with one count per model ID.

frameworks increase the effort agents need to understand and modify the code to support a target model (Lai et al., 2026). PithTrain addresses these difficulties through a compact training framework designed for coding agents, improving their efficiency on tasks such as integrating new model architectures.

ForgeTrain (He et al., 2026) explores an alternative approach by using coding agents to generate dedicated training engines from scratch. It relies on an existing training implementation of the target model in a general-purpose framework as a golden reference for both training behavior and performance.

Given the rapid pace of model releases and the challenges general-purpose training frameworks face in supporting new models, we take this direction a step further with *model-specific training engine synthesis*. An agent generates a standalone engine from a model definition and a training task for the target hardware, without relying on framework code already adapted to the model. We also treat execution configuration as part of the optimization space.

2.3 Challenges in Training Engine Synthesis

Synthesizing a correct and efficient training engine in this setting presents two technical challenges.

Generating a correct training engine. A model definition specifies the model's architecture and computation, but does not by itself provide a complete training engine. The agent must construct and integrate the surrounding training workflow while preserving the model's computation and the training behavior required by the task. Errors in the interactions between components can change training behavior without causing visible failures, so correctness must be established for the integrated engine as a whole.

Optimizing training performance. Achieving competitive training performance requires the agent to iteratively identify bottlenecks and optimize the generated engine. The agent must also explore execution configurations, including choices of parallel layout, state sharding, microbatch size, and recomputation policy. Configuration choices shape the generated code, while code optimizations can change which configurations are feasible or efficient (Korthikanti et al., 2023). This interdependence makes it difficult to settle configuration choices in advance, as configurations must be reconsidered as the implementation evolves. For example, reducing peak memory use can enable a larger microbatch and change which configuration performs best.

3 Method

As shown in Figure 2, TrainTailor takes a model, a fixed training task, and a target hardware environment as input and produces a standalone training engine through two stages: *Build* and *Optimize*. Below, we introduce our method in detail.

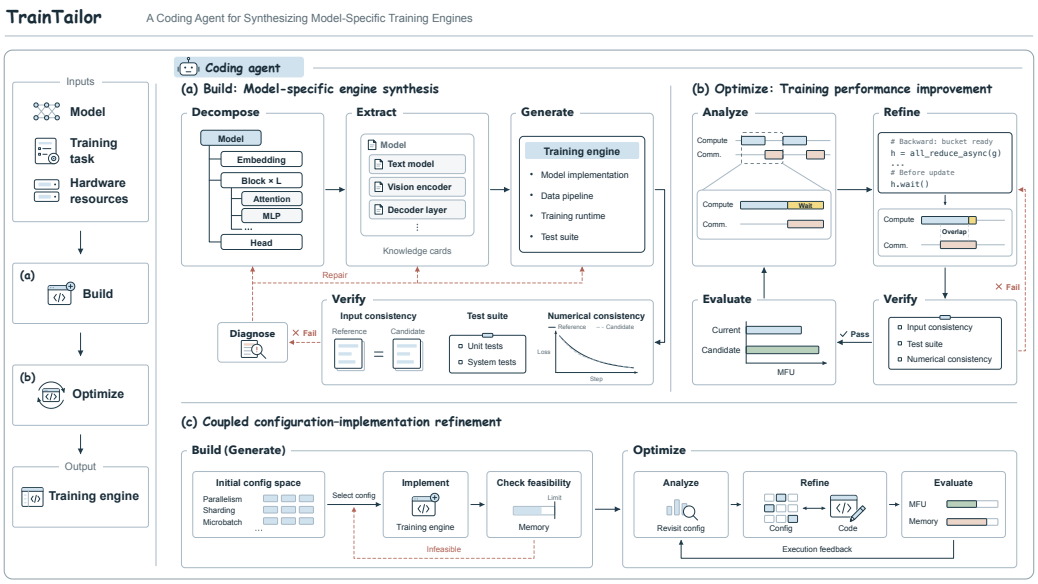


Figure 2: Overview of TrainTailor. (a) Build synthesizes a model-specific training engine. (b) Optimize improves its training performance through iterative refinement. (c) Build jointly constructs a feasible implementation and configuration, which Optimize refines together for performance.

3.1 Problem Formulation

We specify the target model M through a pinned Transformers implementation and its architecture configuration. We choose Transformers because its reference implementations expose model structure and computation directly, without requiring prior adaptation to a general-purpose training framework. The training task specifies the data, initial weights, and training hyperparameters. The hardware environment H specifies the available compute resources. We denote the training engine implementation by E and its execution configuration by C, which specifies parallelism, state placement, microbatching, and recomputation. Optimization jointly refines E and C while keeping M, H, and the training task fixed.

Let $\mathcal{C}(E; M, H)$ denote the execution configurations supported by E that satisfy the resource constraints of H when training M. Configuration choices impose requirements on how E organizes computation, storage, and scheduling, while changes to E can make new configurations feasible. These newly feasible configurations may in turn motivate further implementation changes to realize their performance benefits. For candidates that pass verification, the optimization objective is

$$\max_{E,\, C \in \mathcal{C}(E;M,H)} \quad \mathrm{MFU}(E, C; H), \tag{1}$$

where MFU measures model compute throughput relative to the hardware's peak throughput:

$$\mathrm{MFU}(E, C; H) = \frac{F_{\mathrm{step}}}{t_{\mathrm{step}}(E, C; H)\, n_{\mathrm{dev}}\, P_{\mathrm{peak}}}. \tag{2}$$

Here, F_{step} is the mean model FLOPs for forward and backward computation per global training step, and $t_{\mathrm{step}}(E, C; H)$ is the mean step time over the same measurement window. The hardware allocation consists of n_{dev} devices with peak compute throughput P_{peak} per device. The model FLOPs count is fixed across candidates, so higher MFU corresponds to shorter training time.

3.2 BUILD: MODEL-SPECIFIC ENGINE SYNTHESIS

Build synthesizes a verified initial engine: an implementation E_0 and a configuration $C_0 \in \mathcal{C}(E_0; M, H)$ that together pass verification. Producing this engine addresses the first challenge in Section 2.3: constructing the complete training workflow while preserving the model's computation and the training behavior required by the task. As shown in Figure 2(a), Build divides this work into four phases: *Decompose* and *Extract* analyze the model implementation to capture the structure and computation that the engine must preserve, *Generate* constructs the engine and its initial configuration, and *Verify* checks the training behavior of the integrated engine.

Extracting model knowledge. *Decompose* resolves the architecture configuration against module definitions to produce a structural blueprint specifying component composition, repeated layer patterns, and parameter shapes. *Extract* traces computation across these components and produces linked knowledge cards describing input/output requirements, state updates, and required numerical behavior. For example, in the evaluated Qwen3.5 model, the blueprint specifies a repeated pattern of three Gated DeltaNet layers followed by one full-attention layer. The corresponding knowledge cards specify a padding mask without rotary embeddings for Gated DeltaNet and causal masking with rotary embeddings for full attention. Together, these artifacts guide *Generate* while leaving kernel choices and execution layout open.

Constructing the complete training engine. Building on this model knowledge, the agent must supply the surrounding training workflow and make it executable within the available hardware resources. Implementation skills provide a training-engine contract that specifies component responsibilities and minimum test requirements. Guided by this contract, *Generate* uses the blueprint and knowledge cards to build a standalone training engine from scratch. To establish feasibility on the target hardware, the agent develops the initial implementation and configuration jointly (Figure 2(c)). Model-state memory estimates guide the initial choices, while short training runs assess feasibility and guide revisions.

Verifying training correctness. Because training behavior depends on interactions across components, we validate both component behavior and the integrated training process. We obtain reference loss trajectories from an existing training framework when it supports the model; otherwise, we build a PyTorch training baseline around the original model implementation. *Verify* checks input consistency, executes the test suite, and evaluates numerical consistency:

1. *Input consistency.* We verify that the generated engine and reference use the same model inputs and supervision at each step, regardless of rank or microbatch layout.
2. *Test suite.* The agent derives unit and system tests from the training-engine contract and the knowledge cards. Unit tests check component behavior and local state transitions, while system tests check the actual training path, including gradient accumulation, distributed reduction, and parameter updates.
3. *Numerical consistency.* Following prior work on numerical validation of training implementations (Hsu et al., 2025; Lin et al., 2024), we compare loss trajectories under matched initialization and training settings while allowing bounded floating-point differences. Let $\delta_t = |\ell_t - \ell_t^{\mathrm{ref}}|$ be the absolute difference between candidate and reference losses at aligned step t. Over a K-step training window, the numerical gate requires

$$\begin{aligned} \delta_1 &\le \epsilon_{\mathrm{abs}}^{\mathrm{init}} + \epsilon_{\mathrm{rel}}^{\mathrm{init}} |\ell_1^{\mathrm{ref}}|, \\ \delta_t &\le \epsilon_{\mathrm{abs}} + \epsilon_{\mathrm{rel}} |\ell_t^{\mathrm{ref}}| \quad (1 \le t \le K). \end{aligned} \tag{3}$$

   We use fixed tolerances $(\epsilon_{\mathrm{abs}}^{\mathrm{init}}, \epsilon_{\mathrm{rel}}^{\mathrm{init}}) = (0.0005, 0.0025)$ and $(\epsilon_{\mathrm{abs}}, \epsilon_{\mathrm{rel}}) = (0.01, 0.05)$ across tasks. The initial-loss bound uses tighter tolerances to check forward agreement before the first parameter update. The per-step bound checks numerical agreement at every training step, with a relative term that scales with the reference loss and an absolute term that provides a tolerance floor for small losses. Appendix A provides empirical validation using established training frameworks.

When verification fails, the agent diagnoses the failure and traces it to the earliest incorrect artifact: the blueprint, a knowledge card, or the engine implementation. The agent repairs that artifact,

reconciles downstream artifacts with the change, and verifies the revised engine again. The verified engine enters the Optimize stage, where subsequent candidates undergo the same correctness checks.

3.3 Optimize: Training Performance Improvement

Optimize refines the verified engine produced by Build to improve training performance, measured by MFU. The key difficulty is that configuration choices constrain how the engine is implemented, while implementation optimizations can change the feasible configuration space and which configurations perform best. TrainTailor jointly refines the implementation and configuration through an Analyze–Refine–Verify–Evaluate loop (Figure 2(b)).

Guiding performance optimization. Optimizing the generated engine requires the agent to locate performance bottlenecks and determine how to address them. During *Analyze*, the agent combines source analysis with training profiling to identify performance bottlenecks. TrainTailor provides optimization guidance on operator implementations, data layouts, memory use, and communication scheduling, which the agent uses to formulate a plan for the identified bottlenecks. During *Refine*, it implements and integrates the planned changes. The agent uses training measurements to assess their combined effect and guide subsequent optimization.

Joint configuration and implementation search. As shown in Figure 2(c), TrainTailor guides the agent to revisit configuration choices as implementation changes alter resource requirements or execution costs. The agent uses resource estimates and measured memory use to assess feasibility, and training measurements to compare performance. It retains the conditions behind earlier configuration exclusions and reconsiders affected alternatives when implementation changes alter those conditions. The agent explores promising configurations together with the implementation changes they require, using the resulting measurements to guide further refinement.

Throughout optimization, TrainTailor retains the best verified engine while allowing the agent to continue refining a separate development candidate without immediate performance gains. *Verify* applies the Build correctness checks, and *Evaluate* measures MFU; only a verified candidate with higher MFU replaces the best engine. TrainTailor returns the best verified engine found within the given optimization budget.

4 Experiments

We evaluate TrainTailor's ability to generate correct training engines and improve their training performance. The evaluation combines end-to-end results with an optimization case study on Qwen3.5 and component ablations.

4.1 Experimental Setup

Models. We select two model architectures with features beyond conventional Transformer blocks and different levels of training-framework support. **Qwen3.5** (Qwen Team, 2026a) interleaves Gated DeltaNet (GDN) with gated full attention. **Qwen3.8-Flash-Next** (Qwen Team, 2026b) combines GDN with Qwen Sparse Attention (QSA) and introduces gated residual streams and n-gram embeddings alongside routed mixture-of-experts layers.

We evaluate the released Qwen3.5-0.8B model and a reduced Qwen3.8-Flash-Next model with approximately 15B parameters on two and four NVIDIA RTX PRO 6000 Blackwell GPUs, respectively. Following prior work on reduced-scale evaluation (Zheng et al., 2022; Feng et al., 2025; Lourie et al., 2026; Hsu et al., 2025), we reduce Qwen3.8-Flash-Next's size while preserving its key architectural components to assess TrainTailor's applicability to this architecture. Appendix B.1 summarizes the model configurations.

Baselines. For Qwen3.5-0.8B, we compare against LlamaFactory and Megatron-Bridge, both of which support this model. We tune their execution configurations by searching microbatch sizes, parallel layouts, and supported kernel and optimizer options. All implementations use the same initial weights, logical training batches, input semantics, and training objective. Each framework re-

tains its native precision settings and is checked against the same numerical reference. For Qwen3.8-Flash-Next, the evaluated frameworks lack native training support, leaving no available performance baseline; we therefore directly evaluate the performance of the final engine produced by TrainTailor.

4.2 Results and Analysis

Numerical consistency. Using the prescribed tolerances, we evaluate numerical consistency over a fixed window of 30 training steps. All evaluated runs on Qwen3.5-0.8B and Qwen3.8-Flash-Next pass the numerical gate in Equation 3.

We use Qwen3.5-0.8B as a case study for the following analysis. The optimized engine's loss trajectory closely tracks both framework baselines throughout this window (Figure 3).

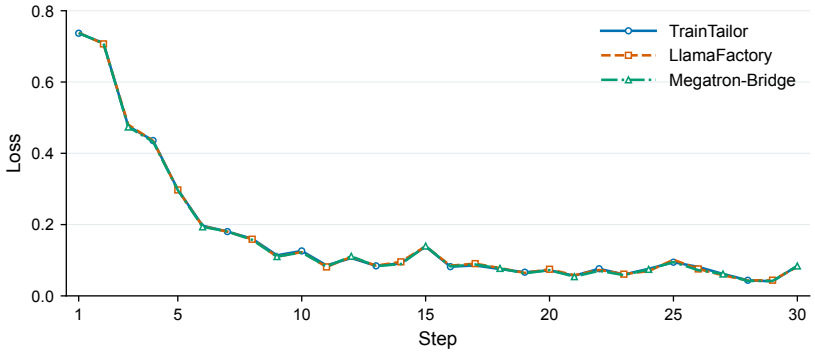


Figure 3: Training loss on Qwen3.5-0.8B.

Training performance. We compare the final engine produced by TrainTailor against the two framework baselines and the verified initial engine produced by Build. On Qwen3.5-0.8B, the optimized TrainTailor engine achieves 26.94% MFU, with relative improvements of 30.47% over LlamaFactory and 33.66% over Megatron-Bridge (Table 1). On Qwen3.8-Flash-Next, TrainTailor improves MFU from 0.52% after Build to 10.15% after Optimize.

Table 1: Training MFU (%).

Implementation	**Qwen3.5-0.8B**	**Qwen3.8-Flash-Next-15B**
LlamaFactory	20.65	–
Megatron-Bridge	20.16	–
TrainTailor (Build)	2.38	0.52
TrainTailor (Optimize)	**26.94**	**10.15**

Optimization case study. Starting from the verified engine produced by Build at 2.38% MFU, joint implementation and configuration optimization raises MFU to 26.94%, yielding 11.33× the initial value while preserving the training task. Figure 4 tracks the retained engine across 18 selected optimization rounds and highlights configuration decisions. The endpoints use the measurements in Table 1, while intermediate points use the corresponding round evaluations. Table 2 lists the adopted optimizations and the corresponding changes along the plotted MFU trajectory, expressed in percentage points.

During Build, TrainTailor selects two-way data parallelism with a microbatch size of 1. Memory estimates and a capacity check show sufficient headroom for this layout, so tensor and pipeline parallelism are not needed to fit the workload. Optimize retains this parallel layout and explores microbatch sizes as the implementation evolves.

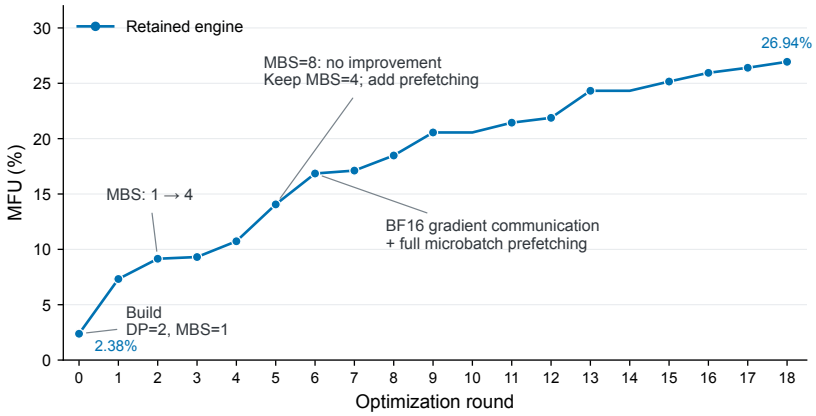


Figure 4: MFU evolution during Qwen3.5-0.8B optimization.

After the initial GDN kernel optimization, Optimize round 2 increases the microbatch size from 1 to 4, preserving a global batch size of 16. MFU rises from 7.32% to 9.15% along the optimization trajectory. In round 5, increasing the microbatch size further to 8 yields no improvement, as input preparation still blocks computation. TrainTailor therefore retains a microbatch size of 4 and introduces background example prefetching, bringing MFU from 10.73% to 14.06%.

Round 6 further couples input-pipeline changes with communication precision. TrainTailor extends prefetching to complete microbatches and moves index preparation off the training thread, while switching gradient communication to BF16. Across rounds 5 and 6, the retained engine progresses from 10.73% to 16.85% MFU with the microbatch size fixed at 4, while satisfying the numerical gate. These rounds demonstrate how TrainTailor uses configuration feedback to revise the implementation and coordinates both kinds of changes to improve training performance.

Table 2: Optimization progress on Qwen3.5-0.8B.

Round	**Optimization**	**MFU (%)**	**Change (pp)**
1	GDN kernels; batched gradient reduction	$2.38 \rightarrow 7.32$	$+4.94$
2	Microbatch size $1 \rightarrow 4$	$7.32 \rightarrow 9.15$	$+1.83$
3	Host synchronization removal	$9.15 \rightarrow 9.31$	$+0.16$
4	Selective projection; fused operators	$9.31 \rightarrow 10.73$	$+1.42$
5	Background example prefetching	$10.73 \rightarrow 14.06$	$+3.33$
6	Microbatch prefetch; BF16 communication	$14.06 \rightarrow 16.85$	$+2.79$
7	Variable-length vision attention	$16.85 \rightarrow 17.11$	$+0.26$
8	Fused AdamW; batched state copies	$17.11 \rightarrow 18.48$	$+1.37$
9	Text packing	$18.48 \rightarrow 20.56$	$+2.08$
11	Overlap of AllReduce and backpropagation	$20.56 \rightarrow 21.44$	$+0.88$
12	Earlier embedding-gradient readiness	$21.44 \rightarrow 21.88$	$+0.44$
13	Overlap of input copies and computation	$21.88 \rightarrow 24.32$	$+2.44$
15	Padding-free GDN convolution	$24.32 \rightarrow 25.16$	$+0.84$
16	Host-side packing and vision planning	$25.16 \rightarrow 25.94$	$+0.78$
17	Direct master-parameter conversion	$25.94 \rightarrow 26.40$	$+0.46$
18	Gradient staging and fused averaging	$26.40 \rightarrow 26.94$	$+0.54$

4.3 Ablation Study

We conduct ablation experiments using the Qwen3.5-0.8B experimental setup.

Decompose and Extract. We compare the full Build pipeline with *direct generation*, which bypasses Decompose and Extract and supplies the original model source directly to Generate without a structural blueprint or knowledge cards. Both retain Verify and diagnostic repair. Direct generation requires four rounds of diagnosis and revision before passing Verify, whereas the full pipeline requires one. The full pipeline also checks GDN's chunked computation against its recurrent formulation, a component test absent from the directly generated engine. These results suggest that resolving model structure and computational requirements through Decompose and Extract helps Generate construct the engine and targeted component tests with fewer subsequent repairs.

Configuration–implementation coupling. We compare the full pipeline with an ablated pipeline that removes the joint configuration and implementation optimization component. In the ablated variant, Build selects the initial configuration based on memory availability, and Optimize keeps the microbatch size fixed at 2 while modifying only the implementation. The ablated pipeline achieves 13.21% MFU, compared with 26.94% for the full pipeline.

5 Discussion

Supporting newly designed architectures. TrainTailor currently uses Transformers definitions of released models. Decompose and Extract capture their structure and computational behavior, providing a basis for extending the workflow to researcher-defined architectures. Supporting custom model definitions and prototype implementations would allow researchers to generate dedicated training engines for new designs, facilitating training and evaluation during architecture exploration.

Toward a self-evolving system. Inspired by recent work on agent self-evolution (Lee et al., 2026; Zhang et al., 2026b;c;a; Liu et al., 2026), TrainTailor could improve continuously through accumulated development experience. Build could maintain a reusable structural knowledge base of recurring modules and architectural patterns, refining the associated knowledge cards through verification feedback to support subsequent engine synthesis. Guided by performance optimization skills, Optimize could also retain validated implementation and configuration strategies together with their applicable conditions. Accumulating this experience from profiling and evaluation across tasks could help the agent improve the training performance of subsequent engines.

Applications to large-scale training. Future work will explore applying TrainTailor to real-world, large-scale training tasks. These workloads introduce more complex communication and GPU memory-management requirements on larger multi-node clusters, including implementing more complex parallelism strategies, coordinating computation with communication, sharding model states, and managing activation memory.

6 Conclusion

We presented TrainTailor, a coding-agent system that synthesizes standalone training engines tailored to a model, training task, and hardware environment. It follows a two-stage pipeline that first builds and verifies an initial engine, then jointly optimizes its implementation and execution configuration. Experiments demonstrate numerical consistency with reference implementations and MFU improvements of approximately 30% over established training frameworks.

AI use statement

We used generative AI tools to assist with manuscript organization, drafting, and language editing. The authors are responsible for the final manuscript.

REFERENCES

Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. Optimus: Accelerating Large-Scale Multi-Modal LLM training by bubble exploitation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 161–177, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-48-9. URL https://www.usenix.org/conference/atc25/presentation/feng.

Qingfeng He, Zhui Zhu, Shangzhan Li, Yaojian Chen, Haojun Sun, Xu Chen, Leshan Li, Yifei Shen, Changjingxing Zhao, Mengyuan Fan, Wenyu Guan, Yiyun Zheng, Yuxuan Zuo, Zhen Li, Zhenghang Luo, Yuxuan Li, Xu Han, and Zhiyuan Liu. ForgeTrain: Forging production-grade training frameworks via harness-driven AI development, 2026. URL https://arxiv.org/abs/2609.13645.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, Yanning Chen, and Zhipeng Wang. Liger-Kernel: Efficient Triton kernels for LLM training. In *Championing Open-source DEvelopment in ML Workshop @ ICML25*, 2025. URL https://openreview.net/forum?id=36SjAIT42G.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In Dawn Song, Michael Carbin, and Tianqi Chen (eds.), *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*. mlsys.org, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/hash/80083951326cf5b35e5100260d64ed81-Abstract-mlsys2023.html.

Ruihang Lai, Hao Kang, Haozhan Tang, Akaash R. Parthasarathy, Zichun Yu, Junru Shao, Todd C. Mowry, Chenyan Xiong, and Tianqi Chen. PithTrain: A compact and agent-native MoE training system. *arXiv preprint arXiv:2605.31463*, 2026. URL https://arxiv.org/abs/2605.31463.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *Conference on Language Modeling (COLM)*, 2026. URL https://arxiv.org/abs/2603.28052.

Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In Ada Gavrilovska and Douglas B. Terry (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 347–363. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi.

Zewen Liu, Zhan Shi, Yisi Sang, Bing He, Minhua Lin, Tianxin Wei, Dakuo Wang, Benoit Dumoulin, Wei Jin, and Hanqing Lu. Adaptive Auto-Harness: Sustained self-improvement for agentic system deployment on open-ended task streams, 2026. URL https://arxiv.org/abs/2606.01770.

Nicholas Lourie, Kyunghyun Cho, Karen Ullrich, and Sanae Lotfi. Small-scale experiments: Are we there yet?, 2026. URL https://arxiv.org/abs/2608.11859.

Yinyi Luo, Wenwen Wang, Hayes Bai, Hongyu Zhu, Hao Chen, Pan He, Marios Savvides, Sharon Li, and Jindong Wang. TorchUMM: A unified multimodal model codebase for evaluation, analysis, and post-training. *arXiv preprint arXiv:2604.10784*, 2026. URL https://arxiv.org/abs/2604.10784.

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 Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (eds.),

International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021, pp. 58. ACM, 2021. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.8-Flash-Next: A new architecture, towards ultimate cost-efficiency, August 2026b. URL https://qwen.ai/blog?id=qwen3.8-flash-next.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for GPU kernel performance optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. URL https://arxiv.org/abs/2509.07506.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026a. URL https://arxiv.org/abs/2606.09498.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations, ICLR 2026*, 2026b. URL https://openreview.net/forum?id=pUpzQZTvGY.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, 2026c. URL https://arxiv.org/abs/2603.19461.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-demos.38. URL https://aclanthology.org/2024.acl-demos.38/.

A Empirical Validation of Numerical Tolerances

To assess the suitability of our numerical tolerance settings, we compare training losses for Qwen3.5-0.8B across LlamaFactory and Megatron-Bridge. We select seven substantially different configurations and train each for 30 steps with the same initialization, training task, and input sequence, using a global batch size of 16. We use the basic LlamaFactory configuration as the baseline.

The shaded band in Figure 5 spans the lowest and highest loss at each step across these configurations. Across all six remaining configurations, the largest first-step difference (0.000910 versus a bound of 0.002345) and the tightest per-step comparison (0.010689 versus 0.013374) both remain within the prescribed mixed tolerances with a margin.

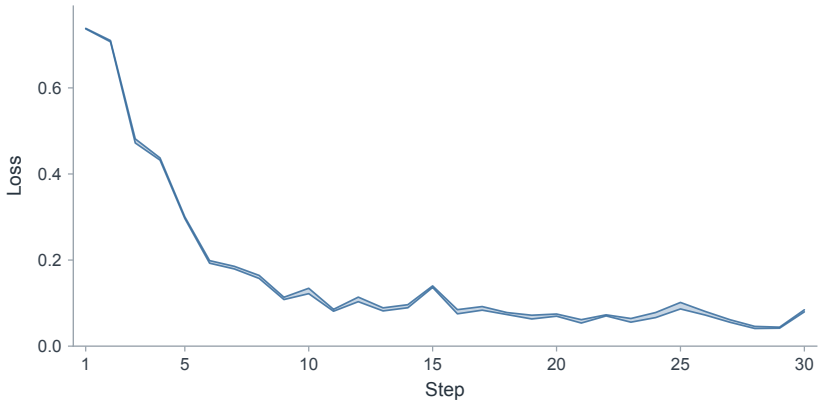


Figure 5: Loss variation across training frameworks.

B Additional Experimental Details

B.1 Model Specifications

For Qwen3.5, we use the released 0.8B model (Qwen Team, 2026a). Table 3 summarizes our reduced Qwen3.8-Flash-Next configuration. Relative to the released model (Qwen Team, 2026b), we reduce text depth, the number of routed experts, and the n-gram hash-table capacity. The attention configurations, gated residual streams, and vision encoder remain unchanged.

Table 3: Qwen3.8-Flash-Next experimental configuration.

Specification	Value
Approximate parameters	15B
Text layers / hidden dimension	24 / 2,560
Repeating attention pattern	3 GDN + 1 QSA
QSA heads (query / key-value)	24 / 2
GDN heads (key / value)	16 / 48
Routed experts / top-k	96 / 10
Expert intermediate dimension	640
Residual streams / bottleneck rank	4 / 320
N-gram hash vocabulary base	131,072