Learning to Shard: RL for Co-optimizing the Parallelism Degrees and Per-operator Sharding Dimensions in Distributed LLM Inference

Ruokai Yin[*1], **Sattwik Deb Mishra**[2], **Xuan Zuo**[2], **Hokchhay Tann**[2], **Preyas Shah**[2], **Apala Guha**[2]
[1]Yale University, [2]Microsoft Azure

Abstract

Distributed LLM inference requires careful coordination of parallelization strategies across hundreds to thousands of NPUs to meet production SLOs. Current systems like Megatron-LM rely on static heuristics that separately configure parallelism degrees and per-operator sharding dimensions, leaving significant performance on the table as models scale and hardware topologies diversify. We introduce **Learn to Shard**, to our knowledge, the first RL-based approach to co-optimize both coarse-grained parallelism degrees and fine-grained per-operator sharding dimensions for distributed LLM inference. Our method employs an attention-based policy over an elite history that learns from high-performing strategies to efficiently navigate the vast combinatorial search space. Evaluated on H100 clusters with MoE models up to 1.6T parameters, Learn to Shard achieves up to 3.5× throughput improvement over metaheuristic baselines and 1.06× over Megatron heuristics.

1 Introduction

Large language models (LLMs), including LLaMA (Touvron et al., 2023) and mixture-of-experts (MoE) models (Liu et al., 2024), have achieved remarkable performance across a wide range of tasks. However, their massive scale—often exceeding hundreds of billions or even trillions of parameters—demands inference over a vast number of NPUs to deliver production-quality latency and throughput. As a result, *distributed inference* has become the default, and optimizing parallelism is critical for meeting service-level objectives (SLOs) under strict hardware and cost constraints.

Modern distributed inference relies on multiple types of parallelism, such as tensor parallelism (TP), expert parallelism (EP), and pipeline parallelism (PP), each offering distinct efficiency trade-offs. These strategies are not mutually exclusive: in fact, combining them into *mixed parallelism* often yields better throughput and efficiency than any single type alone. In practice, existing inference systems like Megatron-LM (Shoeybi et al., 2019) determine the parallelization degrees of different parallelization types using heuristics and hand-tuned rules. We refer to the selection of parallelism degrees as the *coarse-grained parallelization strategy*.

Beyond coarse-grained parallelization strategies, we emphasize a *fine-grained* and largely overlooked strategy design axis—the selection of the *per-operator sharding dimension*. The per-operator sharding dimension specifies which tensor dimensions individual operators are sharded along. Existing inference systems fix this choice through static heuristics. For instance, in Megatron's TP implementation of MLP operators, FFN1 and FFN2 are sharded along feedforward dimension (dim 1 and 0 respectively). While this minimizes the number of inter-NPU communications, it assumes fixed all-reduce operations that may be suboptimal on different interconnects or under strict latency constraints (e.g., all-gather often performs better). Exploring the sharding dimensions—especially together with parallelism degrees—remains a largely unexplored problem.

[*]Work performed during an internship at Microsoft Azure. Correspondence: ruokai.yin@yale.edu

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: ML for Systems.

With both the fine and coarse-grained parallelization strategies considered, the co-optimization space grows significantly. Specifically, as LLM models scale to trillions of parameters (NVIDIA Corporation, 2024), hundreds of NPUs are required just for storing weights and KV caches, while thousands may be needed to meet inference SLOs. Therefore, heuristic approaches become increasingly unscalable. While empirical optimization continues to discover new parallelization strategies beyond existing heuristics (Bhatia et al., 2025; Qin et al., 2025), the time required to discover effective strategies increases as the search space expands. This motivates the need for an automated and scalable approach for searching for new strategies.

To automate parallelization strategy optimization, prior work has explored automated search via integer programming (Lin et al., 2025) and metaheuristic algorithms (Raju et al., 2025), offering various trade-offs between scalability and optimality. Reinforcement learning (RL) has recently emerged as a powerful tool in automating compiler optimizations (Zhou et al., 2020), but its application to parallelization strategy remains largely unexplored. Furthermore, many other considerations such as intra-device tiling, collective algorithms, and model compression impact SLO and cost during inference. We explore whether RL-based agentic search is a feasible approach to this problem[2].

In this work, we propose **Learn to Shard**, an RL-based agent that co-optimizes both the coarse-grained (parallelism degrees) and the fine-grained (per-operator sharding dimensions) parallelization strategy for distributed LLM inference. Our key insight is that high-performing parallelization strategies exhibit patterns that can be learned from the agent's explorations. Consequently, an RL-based approach becomes beneficial through its sampling efficiency. We develop an attention-based elite-context policy that learns from previously discovered strategy configurations to guide exploration in the vast and combinatorial optimization space—dramatically reducing the time-to-discovery compared to non-learning baselines.

We evaluate **Learn to Shard** on real-world workloads on H100 GPU clusters with up to thousands of devices. Our method achieves up to 3.5× throughput improvement over metaheuristic baselines, and outperforms Megatron-LM's heuristic strategies by up to 1.06×, requiring only 4000 search rounds out of 10^9 possible configurations.

2 Background and Related Work

Parallelization types and strategies. In tensor parallelism (TP), weight matrices for each layer are sharded and distributed across different NPUs, as shown in Figure 1. We only consider 1-D TP. Pipeline parallelism (PP) can be viewed as task parallelism at layer level, operator level, etc. In this work, we focus on layer level parallelism. In PP, different task subsets are placed on different NPUs, with adjacent pipeline stages transferring activations between devices. Expert parallelism (EP) is designed for modern MoE models, where different experts are placed on different NPUs and input tokens are routed to appropriate devices based on gating scores. Other parallel types also exist such as sequence parallelism and data parallelism which are complementary to our search framework and optimization space.

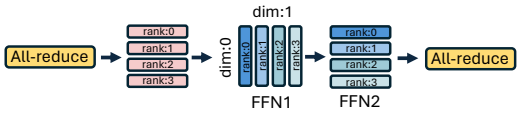

Figure 1: An illustrative example of the standard TP parallelism strategy.

For agentic search, we need a formal representation of the sharding strategy. We define a parallelization strategy with two parts: the coarse-grained and the fine-grained strategy. The coarse-grained strategy refers to the parallelization degrees, which refer to the number of NPUs that each specific parallelization type requires. In the TP-only example of Figure 1, the coarse-grained strategy can be described as {TP=4, EP=1, PP=1}. The fine-grained strategy determines which dimension each operator's output tensor is sharded along. In the provided example, the fine-grained strategy for the loading operators of FFN1 and FFN2 is {dim=1, dim=0}. In practice, the output tensor can be sharded on any dimension as long as correct communication collectives ensure dimension matching between producer and consumer operations. This representation is flexible and can capture arbitrary[3] sharding strategies, for example,

[2]In this work, we start the exploration by applying the RL-based agentic search to a subset of the problem (parallelization strategy)

[3]Prior empirically discovered strategies are representable under this framework and thus discoverable through agentic search

sequence parallelism (Li et al., 2021) shards the attention operators on the context length dimension[4]. Under this representation, the sharding strategy can be effectively modeled into a single set of integer values that can be seamlessly integrated into our agentic search flow.

Related work. Extensive prior work has focused on automatically determining parallelism strategies for large model training and inference. Systems such as MEGATRON-LM (Shoeybi et al., 2019), TensorFlow-XLA (OpenXLA Project, 2025), and GSPMD (Xu et al., 2021) provide heuristic or rule-based parallelization strategies. ALPA (Zheng et al., 2022) extends this with cost-model-based search over coarse-grained parallelism configurations, and FLEXFLOW (Jia et al., 2019) applies simulation-guided MCMC search for training graphs. More recent inference-oriented frameworks, such as ALPASERVE (Yu et al., 2023) and SEESAW (Su et al., 2025), optimize serving throughput or dynamically switch sharding strategies between prefill and decoding. TAPAS (Shi et al., 2025) identifies the large sharding search space and prunes the computation graph to accelerate the search. DFmodel (Ko et al., 2024) and UniAP (Lin et al., 2025) automate the parallelism strategy optimization through integer programming and COSMIC (Raju et al., 2025) leverages metaheuristic algorithms like simulated annealing. However, these automated optimization methods only target the coarse-grained strategy, neglecting the importance of the fine-grained strategy (i.e., per-operator sharding dimension). To our knowledge, our work is (i) the first to explore RL-based parallelization strategy optimization with SLO constraints and (ii) the first to co-optimize both the coarse and fine-grained strategies.

3 Methodology

Search Framework Overview. We formulate the problem of optimizing the parallelization strategy as an RL search over a MULTIDISCRETE action space. At each step, the agent proposes a joint parallelization strategy $\boldsymbol{a} = \big(a_{\text{TP}}, a_{\text{EP}}, a_{\text{PP}}, a_{\text{B}}, \{a_{\text{dim}}^{(\ell)}\}_{\ell=1}^{L}\big)$, where a_{TP}, a_{EP}, and a_{PP} denote the degrees of tensor, expert, and pipeline parallelism, a_{B} controls the batch size, and $a_{\text{dim}}^{(\ell)} \in \{0, 1, \varnothing\}$ selects the per-operator sharding dimension for fused op ℓ (shard along dimension 0, dimension 1, or no sharding). Here L is the number of fused operations considered.

The environment evaluates the strategy $\boldsymbol{a}$ in our performance simulator and returns a scalar raw throughput (token/s/chip) $\text{raw}(\boldsymbol{a})$. We define the learning reward signal $r(a)$ as

$$r(\boldsymbol{a}) \;=\; \alpha\,\text{raw}(\boldsymbol{a}) \;+\; \beta\,\big(\text{raw}(\boldsymbol{a}) - b\big), \tag{1}$$

where b denotes the best raw throughput observed so far and $\alpha, \beta > 0$ are scaling factors. Thus $r(\boldsymbol{a})$ provides an improvement bonus when $\text{raw}(\boldsymbol{a}) > b$ and penalizes underperforming strategies. The simulator identifies invalid configurations (e.g., SLO violations) with large negative rewards. We run for a fixed simulator call budget B and select the final strategy as the highest reward configuration encountered.

Policy Network. As shown in Fig. 2, the observation X to the policy is a fixed-length history of elite strategies. We maintain an elite history buffer (deque) of length T containing the top-T strategies, sorted by performance (magnitude of $r(\boldsymbol{a})$). At each step we form $X \in \mathbb{R}^{T \times A}$ by concatenating the T strategy records (each an A-dimensional vector). Each record is embedded via a linear layer ($\mathbb{R}^A \to \mathbb{R}^d$), stacked, and passed through a Transformer encoder block, producing $Z \in \mathbb{R}^{T \times d}$. We apply mean pooling along the history dimension, $h = \frac{1}{T}\sum_{t=1}^{T} Z_{t,:} \in \mathbb{R}^d$, and project h with a small MLP to obtain the logits for each sub-strategy head. We optimize the policy with PPO to maximize expected reward of sampled strategies:

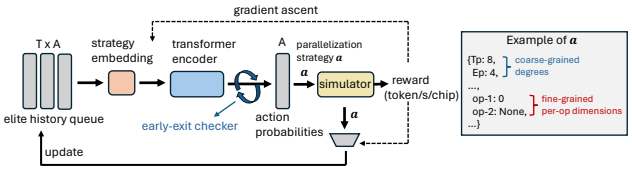

Figure 2: Overall search loop and the policy network design.

$$L_\pi(\theta) = \mathbb{E}_{a \sim \pi_\theta(\cdot|X)}\big[r(a)\big],$$

where the expectation is over on-policy rollouts with a sampled from $\pi_\theta(\cdot \mid X)$ given the current observation. Every time a valid strategy is found, we update the elite history buffer if the new strategy improves upon the worst elite in the buffer.

[4]In this work, we allow sharding of attention operators on the head and hidden dimensions, and the sharding of MLP operators on hidden and feed-forward dimensions. We leave exploring context length sharding as a future work

Optimization and early exit. To save search budget and avoid local optima, we use a confidence-based early exit. Formally, let $p_m = \pi_\theta^{(m)}(\cdot \mid X)$ denote the categorical distribution for sub-strategy head m. We define the confidence of head m as $\mathrm{CS}_m = \max(p_m)$ and trigger an early exit if $\mathrm{CS}_m \geq \tau$ for all heads m (e.g., $\tau = 0.95$), at which point the policy is effectively deterministic about the strategy. To fully utilize the saved budget, we partition the total budget B into smaller chunks (typically 5 chunks of equal size); upon an early exit we restart the agent by reinitializing network parameters while inheriting the prior agent's b and elite history buffer.

4 Evaluation

Setups. We evaluate parallelization strategy performance using an in-house roofline-based simulator[5]. The hardware system used in the evaluation is NVIDIA H100. We evaluate two synthetic MoE models (1.2T and 1.6T parameters) (NVIDIA Corporation, 2024) and consider $L = 12$ fused ops; the device budget is up to 24k GPUs. This yields a search space of $\sim 10^9$ joint configurations. Exhaustive evaluation would require ∼100 days. For all searches we allow a budget of 4000 agent forward + simulator calls (including invalid configurations), which takes less than 10 minutes. The detailed setups can be found in Appendix A.

Workloads	SA	Ours
GPT-MoE 1.2T (16k)	1.19×	**2.76**×
GPT-MoE 1.2T (32k)	0.95×	**2.17**×
GPT-MoE 1.2T (64k)	0.71×	**2.04**×
GPT-MoE 1.6T (16k)	0.54×	**1.65**×
GPT-MoE 1.6T (32k)	0.66×	**2.31**×
GPT-MoE 1.6T (64k)	0.89×	**1.92**×

Table 1: Normalized throughput improvement over random walk (RW) across models and context-lengths on H100 (mean over 10 runs; higher is better).

Main Results. We compare the search quality of our proposed PPO-based agent with the other metaheuristic search algorithms, including simulated annealing (SA) and random walk (RW), which are widely used in prior works (Zhou et al., 2020; Raju et al., 2025), across different decoding context lengths. As shown in Table 1, our method consistently outperforms SA and RW under the same evaluation budget, achieving up to 2.76× throughput improvement over the RW. In Appendix C, we provide a qualitative comparison with inter programming based methods.

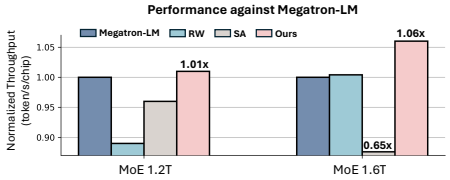

Figure 3: Normalized throughput of jointly optimized sharding vs. Megatron-LM heuristics (best of 10 runs; decoding, 16k context).

We further measure the improvement of the agent-found parallelization strategies over the heuristic ones. We set the heuristic strategy baseline to use the standard per-operator sharding dimensions from Megatron-LM (Shoeybi et al., 2019). We then fix Megatron's per-operator sharding dimensions and exhaustively search only over the parallelization degrees (TP/EP/PP/batch), which is tractable, to produce a strong heuristic baseline. The performance is measured for the decoding stage with the context length of 16k. As shown in Figure 3, our method consistently outperforms the Megatron-LM based heuristic parallelization strategies with up to 1.06× throughput improvement. In contrast, other metaheuristic-based agents fail to find a strategy that is better than the heuristic one. One example of the sharding strategy found by our agent is shown in Appendix B.

5 Conclusion and Takeaways

Learn to Shard jointly optimizes parallelism degrees and per-operator sharding dimensions via a lightweight PPO-trained RL policy, achieving up to 3.5× improvement over simulated annealing and 1.06× over Megatron heuristics on H100 clusters with MoE models up to 1.6T parameters. Key takeaways include: (i) Learning-based search outperforms metaheuristics in large, sparse reward spaces under realistic constraints (HBM, interconnect, SLOs). (ii) Co-optimizing parallelization degrees and per-operator sharding enables discovery of non-standard strategies that exceed Megatron heuristics. Future work includes extending the search space to include, for example, multi-dimensional sharding.

[5]Validated against Megatron-LM heuristic parallelization strategy. See Appendix A and C for a discussion on generality and simulator integration.

References

Bhatia, N., More, A., Borkar, R., Mitra, T., Matas, R., Zhao, R., Golub, M., Mudigere, D., Pharris, B., and Rouhani, B. D. Helix parallelism: Rethinking sharding strategies for interactive multi-million-token llm decoding. *arXiv preprint arXiv:2507.07120*, 2025.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems (MLSys)*, 2019.

Ko, S., Zhang, N., Hsu, O., Pedram, A., and Olukotun, K. Dfmodel: Design space optimization of large-scale systems exploiting dataflow mappings. *arXiv preprint arXiv:2412.16432*, 2024.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.

Lin, H., Wu, K., Li, J., Li, J., and Li, W.-J. Uniap: Unifying inter-and intra-layer automatic parallelism by mixed integer quadratic programming. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 20947–20957, 2025.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

NVIDIA Corporation. GTC 2024 Presentation Slides. Conference presentation at NVIDIA GTC 2024, March 2024. URL https://www.youtube.com/watch?v=f8DKD78BrQA.

OpenXLA Project. XLA: A Machine Learning Compiler. https://github.com/openxla/xla, 2025.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Qin, L., Cui, J., Cai, W., and Huang, J. Chimera: Communication fusion for hybrid parallelism in large language models. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pp. 498–513, 2025.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research*, 22(268): 1–8, 2021.

Raju, A., Ni, J., Won, W., Man, C., Krishnan, S., Sridharan, S., Yazdanbakhsh, A., Krishna, T., and Reddi, V. J. Cosmic: Enabling full-stack co-design and optimization of distributed machine learning systems. *arXiv preprint arXiv:2505.15020*, 2025.

Shi, Z., Jiang, L., Wang, A., Zhang, J., Wu, C., Li, Y., Xiao, X., Lin, W., and Li, J. TAPAS: Fast and automatic derivation of tensor parallel strategies for large neural networks. In *Proceedings of the 54th International Conference on Parallel Processing*, ICPP '25, New York, NY, USA, Aug 2025. ACM. URL https://arxiv.org/abs/2302.00247. Accepted for publication.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–14, 2019.

Su, Q., Zhao, W., Li, X., Andoorveedu, M., Jiang, C., Zhu, Z., Song, K., Giannoula, C., and Pekhimenko, G. Seesaw: High-throughput llm inference via model re-sharding. *arXiv preprint arXiv:2503.06433*, 2025.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Xu, S., Chen, X., Xu, Y., Johnson, D., Krikun, M., Chen, K., Liu, Y., Zhao, K., Jouppi, N., Laudon, J., et al. Gspmd: General and scalable parallelization for ml computation graphs. In *Proceedings of Machine Learning and Systems (MLSys)*, 2021.

Yu, C., Yan, Y., Zheng, L., Zhang, W., Chen, M., Xing, E., Gonzalez, J. E., and Stoica, I. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2023.

Zheng, L., Yu, C., Yan, Y., Xu, Z., Zhang, H., Gonzalez, J. E., Stoica, I., Jin, X., Xing, E., Chen, Q. H., et al. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

Zhou, Y., Roy, S., Abdolrashidi, A., Wong, D., Ma, P., Xu, Q., Liu, H., Phothilimtha, P., Wang, S., Goldie, A., et al. Transferable graph optimizers for ml compilers. *Advances in Neural Information Processing Systems*, 33:13844–13855, 2020.

A Experimental Setup

We use cosine annealing for the PPO learning rate and SA temperature (initial temperature = 100). Unless noted otherwise, the elite history buffer size is T=3, the confidence threshold is τ=0.95, and the search budget is split into 5 equal chunks. We use the PPO algorithm from StableBaseline3 (Raffin et al., 2021). Step size and the rollout buffer size are both set to 2. We use 2 gradient ascents steps per update. The learning rate for PPO learning starts at $1e^{-3}$. We use a single encoder transformer block (Paszke et al., 2019) in our policy network. Hidden size and strategy-embedding size are both 256.

While our evaluation focuses on large-scale MoE models, our framework readily generalizes to dense models. The search problem for dense models is, in fact, simpler, as it removes the Expert Parallelism (EP) dimension from the co-optimization space.

Furthermore, our method is not specific to one hardware platform. It can generalize to any hardware (e.g., different GPUs, custom NPUs) as long as its performance can be faithfully characterized by the user's performance simulator. As our agent treats the simulator as a "black-box" oracle (see Appendix C), it can be applied to diverse simulators, provided their performance outputs exhibit learnable patterns in response to different parallelization strategies.

It is important to clarify that the RL agent's training process is the search process. Consequently, the agent is re-trained for each new model and hardware combination, as it learns the optimal strategy for that specific problem instance. This process is highly efficient: as discussed in Section 3, the policy network is small, and our proposed early-exit mechanism significantly reduces the search cost.

B Different Strategy Example Found by Agent

Figure 4 illustrates the difference between Megatron-LM's heuristic (Shoeybi et al., 2019) and one example parallelization strategy found by our agent. The main difference is that instead of sharding FFN2 operator on the feedforward dimension, the agent shards on the hidden dimension. This requires an extra all-gather communication between FFN1 and FFN2 but simplifies the original all-reduce operations into all-gather operations.

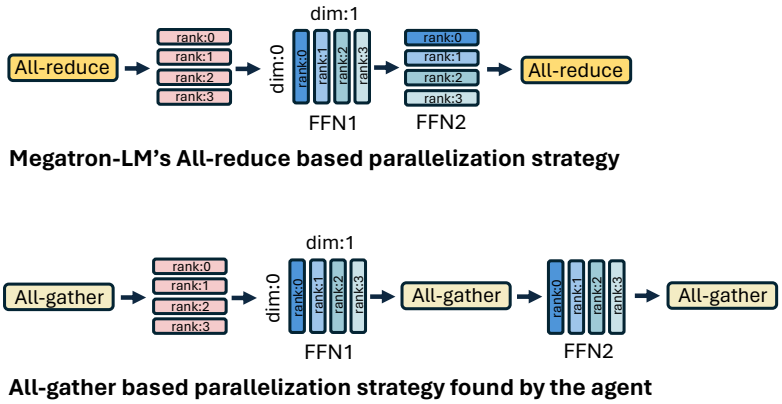

Figure 4: Illustration of the all-gather based sharding strategy found by our **Learn to Shard** agent vs. Megatron-LM's all-reduce based heuristic sharding strategy. (We only show the MLP operators).

In Figure 4, we only show the MLP part of the fine-grained parallelization strategy. For illustration purposes, we have a dummy coarse-grained parallelization strategy of {TP=4, EP=1, PP=1}.

C Qualitative Comparison with Integer-programming Methods

Our work introduces two novelties: an expanded parallelism strategy search space and an RL-agent-based search method. A natural question is what advantage our RL-based search offers over standard

integer-programming (IP) methods (Lin et al., 2025; Ko et al., 2024). The key difference lies in the "white-box" versus "black-box" nature of the optimization.

IP methods, such as DFModel (Ko et al., 2024) or UniAP (Lin et al., 2025), are "white-box" approaches. They are fundamentally constrained by their solver, requiring the entire complex performance landscape to be simplified and formulated as a set of (often linear) mathematical equations. This is an extremely rigid structure that cannot capture many complex, non-linear, or procedural dynamics, such as cache contention and experts selected pattern in MoE models.

Our RL-based framework, by contrast, treats the performance simulator as a "black-box" oracle. The agent only requires a scalar reward (i.e., throughput) in response to a proposed strategy. This decoupling of the search agent from the simulator provides critical, practical advantages. First, our agent can be integrated with any existing, high-fidelity production simulator without the massive engineering cost of refactoring it to be "solver-friendly." Second, this modularity allows new features to be added directly to the simulator without the intractable burden of deriving new mathematical constraints for an IP solver.