OptiCo: Adaptive Distributed Training Optimization via Collaborative Agent Reasoning

Sheng Chen[1*], TangZhe[1*], Weixing Zhang[1], Fei Yang[1†], Yuanyuan.Wang[1], Tianlin Li[2], Yang Liu[3],

[1]Zhejiang Lab, [2]Beihang University, [3]Nanyang Technological University,

{scucs, tangzhe, wxzhang, yangf, wangyy2022}@zhejianglab.org

tianlin001@buaa.edu.cn, yangliu@ntu.edu.sg

Abstract

Optimizing distributed training strategies for large-scale deep learning models remains a critical challenge in both industry and academia, demanding extensive domain expertise and manual tuning. Existing automated distributed training frameworks are plagued by over-reliance on prior profiling, poor generalization across models/hardware, and scalability constraints stemming from vast search spaces, impeding real-world applicability. To address these challenges, we propose OptiCo, a model-driven multi-agent framework that leverages Large Language Models (LLMs) to enable automatic and explainable distributed training strategy configuration. OptiCo orchestrates a team of reasoning-driven agents, through a shared Global Message Pool facilitating persistent memory and coordination. By employing inception prompting and Chain-Of-Thought (CoT) reasoning, agents iteratively refine configurations, detect bottlenecks, analyze failures, and optimize resource utilization. Evaluated across 25+ configurations spanning diverse model architectures, GPU types and scales, OptiCo outperforms expert-designed strategies within 20 iterations, achieving an average performance improvement of 1.84%, with gains ranging from 0.08% to 8.65%. The source codes are available at https://github.com/TangZhe96/OptiCo-public.

1 Introduction

As large-scale deep learning models become the foundation of modern artificial intelligence applications, the efficiency of their training processes has emerged as a critical bottleneck. These models require distributed execution across multiple GPUs or nodes, pushing the limits of available hardware in terms of compute, memory, and interconnect resources. In such contexts, effective configuration of distributed training strategies, such as tensor parallelism (TP) (Shoeybi et al., 2019), pipeline parallelism (PP) (Huang et al., 2019), data parallelism (DP) (Sergeev and Del Balso, 2018), expert parallelism (EP) (Rajbhandari et al., 2022), etc., are essential to achieve high training throughput and model FLOPs utilization (MFU) (Chowdhery et al., 2023). However, achieving high MFU is non-trivial in practice due to the complex interference between model structure, parallel strategies, and hardware characteristics.

While recent systems, such as Alpa (Zheng et al., 2022), AMP (Li et al., 2022) and ACESO (Liu et al., 2024), have introduced automated parallel configuration frameworks to dynamically optimize configurations for higher MFU, they rely on static profiling, rule-based heuristics, or search algorithms to explore the vast configuration space, which is NP-hard to optimize in practice. However, they depend heavily on prior profiling, which incurs high overhead and limits adaptability to dynamic workloads. Their generalization is also limited as configurations tuned for one model or hardware setup often fail to transfer across settings.

Recent years have seen growing interest in multi-agent systems that decompose complex problems into collaborations among specialized agents. Prior work, including Agentic Reasoning (Wu et al., 2025), CAMEL (Li et al., 2023), and MetaGPT (Hong et al., 2023), demonstrates that assigning distinct roles, such as planning, tool invocation, and failure diagnosis, to individual agents enables scalable and modular solutions. These observations naturally raise the question of whether a multi-agent paradigm can be leveraged to address automated parallel configuration, while mitigating the limitations of existing approaches. Unlike prior automated parallelism frameworks, a multi-agent formulation enables the decomposition of the con-

[*]Equal contribution.

[†]Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 27847–27867

figuration problem into role-specific subtasks. By allowing agents to iteratively interact and exchange feedback, such a framework can progressively refine training strategies without exhaustive search or heavy offline profiling. This perspective motivates our design of a multi-agent system that jointly reasons over system behavior and training performance to optimize parallel configurations in a scalable and adaptive manner.

In this paper, we present OptiCo, a multi-agent framework for adaptive distributed training Optimization with Collaborative agent reasoning. OptiCo is composed of four agents, each assigned a specific role in the optimization process: Bottleneck Analysis, Strategy Generation, Strategy Validation, and Failure Analysis. All agents communicate and coordinate through a shared Global Message Pool, which records system configurations, performance metrics such as MFU, training parameters and failure logs throughout the optimization process. By structuring the optimization as an iterative interaction among agents, OptiCo mimics expert-driven workflows in the real world while enabling adaptive adjustment of training parameters, including TP, PP, DP, EP, micro batch size, overlap, recompute, and fusion. This design allows the framework to progressively refine strategies based on historical outcomes, improving computational efficiency without manual intervention.

We summarize our contributions as follows:

(i) We propose OptiCo, a collaborative multi-agent framework for distributed training strategy optimization. OptiCo automates the tuning of distributed training configurations without human intervention, and demonstrates potential in adapting to diverse model architectures.

(ii) We design a multi-turn collaborative refinement mechanism, where four specialized agents continuously analyze bottlenecks, generate strategies, validate outcomes, and diagnose failures. These agents interact through a shared memory module to iteratively improve training configurations and mitigate inefficient or invalid decisions.

(iii) We conduct extensive experiments across 25+ configurations with different models, GPU types, and parallel settings. OptiCo achieves an average MFU improvement of 1.84%, with gains reaching up to 8.65%, outperforming expert-tuned baselines within 20 iterations.

2 Methodology

2.1 OptiCo Framework

OptiCo is a collaborative multi-agent framework for automatic training strategy adjustment in large-scale deep learning. It aims to maximize resource utilization by refining training parameters.

Inspired by expert-driven tuning workflows adopted in practice, as shown in Figure 1, OptiCo incorporates four specialized agents: Bottleneck Analysis Agent ($\mathcal{A}_{\mathrm{a}}$), Strategy Generation Agent ($\mathcal{A}_{\mathrm{g}}$), Strategy Validation Agent ($\mathcal{A}_{\mathrm{v}}$), and Failure Analysis Agent ($\mathcal{A}_{\mathrm{f}}$). These agents operate iteratively in a line paradigm while exchanging information through a shared Global Message Pool ($\mathcal{M}_{\mathrm{pool}}$). The iterative optimization process of OptiCo at each iteration t can be specified as follows:

$$\mathcal{C}_t = \mathcal{A}_{\mathrm{g}}(\mathcal{M}^{\mathrm{t}}_{\mathrm{pool}},\ \mathcal{A}_{\mathrm{a}}(\mathcal{M}^{\mathrm{t}}_{\mathrm{pool}}),\ \textit{rules}),$$
$$\mathcal{M}^{\mathrm{t+1}}_{\mathrm{pool}} \leftarrow \mathcal{M}^{\mathrm{t}}_{\mathrm{pool}} \cup \begin{cases} (\mathcal{C}_t,\ \mathcal{A}_{\mathrm{v}}(\mathcal{C}_t)), & \text{success} \\ (\mathcal{C}_t,\ \mathcal{A}_{\mathrm{f}}(\mathcal{A}_{\mathrm{v}}(\mathcal{C}_t))), & \text{failure} \end{cases} \quad (1)$$

where $\mathcal{C}_t$ denotes the configuration of training strategy at iteration t. Besides, each agent in OptiCo is instantiated using *GPT-5.2*[3], selected for its strong general reasoning capabilities. A detailed workflow description of OptiCo is provided in Appendix D.

2.1.1 Bottleneck Analysis Agent.

The agent $\mathcal{A}_{\mathrm{a}}$ serves as the entry point of the optimization loop by identifying configuration parameters that are most likely to limit training efficiency. Specifically, $\mathcal{A}_{\mathrm{a}}$ retrieves historical training configurations and associated performance metrics from $\mathcal{M}_{\mathrm{pool}}$. If $\mathcal{M}_{\mathrm{pool}}$ is initially empty, a random configuration is executed once to populate it.

Based on the collected execution data, $\mathcal{A}_{\mathrm{a}}$ performs structured reasoning inspired by Chain-of-Thought (CoT) prompting over multiple configuration factors to assess the relative impact of different parameters on training performance.

$\mathcal{A}_{\mathrm{a}}$'s output is a ranked list of configuration parameters ordered by their estimated influence on system bottlenecks. This ranking provides explicit guidance to agent $\mathcal{A}_{\mathrm{g}}$, enabling efficient exploration of the configuration space in subsequent optimization stages. A detailed workflow description of $\mathcal{A}_{\mathrm{a}}$ is provided in Appendix D.2.

[3] https://platform.openai.com/docs/guides/latest-model

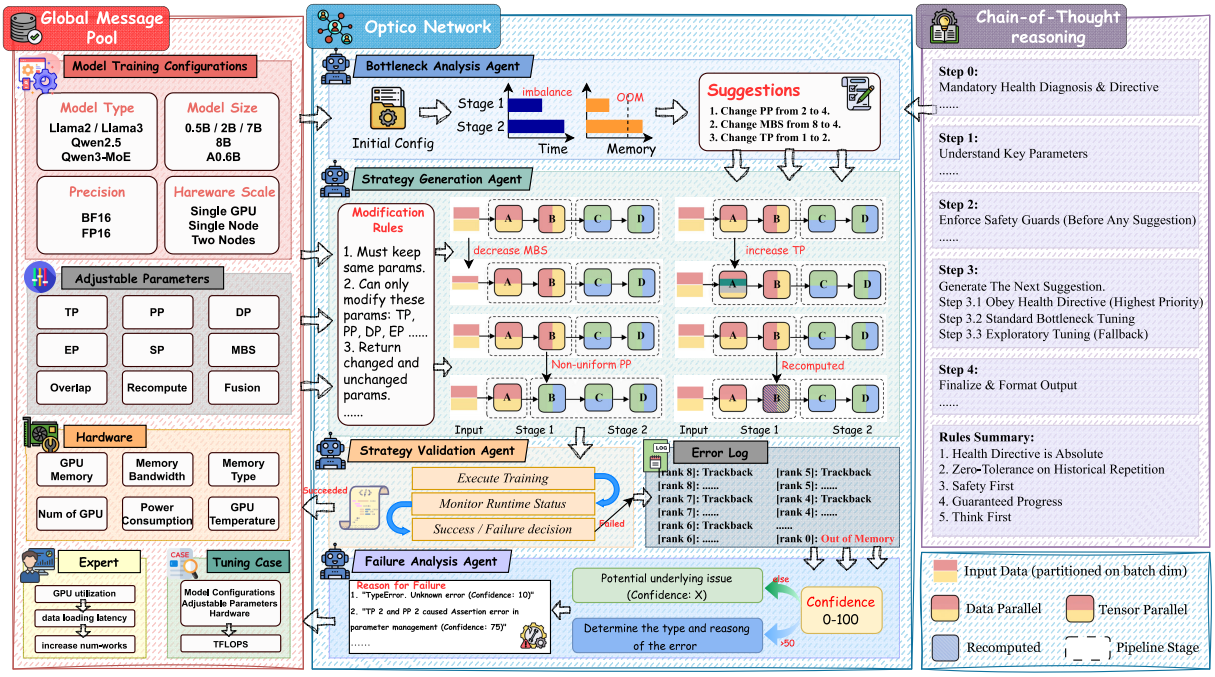


Figure 1: An overview of the OptiCo framework.

2.1.2 Strategy Generation Agent.

The agent $\mathcal{A}_{\mathrm{g}}$ is responsible for synthesizing candidate training configurations $\mathcal{C}_t$ through structured and rule-constrained parameter transformations. Rather than performing unconstrained search, $\mathcal{A}_{\mathrm{g}}$ applies directional parameter modifications based on the bottleneck prioritization provided by $\mathcal{A}_{\mathrm{a}}$.

Given an input configuration, $\mathcal{A}_{\mathrm{g}}$ selectively adjusts a restricted set of tunable parameters (e.g., MBS, TP, PP and recompute as shown in Figure 1) while strictly preserving the overall configuration schema. Each modification follows explicit modification rules that require all parameters to be returned in a consistent format, ensuring compatibility with the training system. A detailed workflow description of $\mathcal{A}_{\mathrm{g}}$ is provided in Appendix D.3.

2.1.3 Strategy Validation Agent.

The agent $\mathcal{A}_{\mathrm{v}}$ serves as a runtime feasibility checker that verifies whether a newly generated configuration from $\mathcal{A}_{\mathrm{g}}$ can be successfully executed in the target training environment.

For each candidate configuration, $\mathcal{A}_{\mathrm{v}}$ attempts to instantiate and launch a training run while monitoring execution status and runtime signals. If the execution completes successfully, the configuration with the collected runtime performance metrics is recorded in $\mathcal{M}_{\mathrm{pool}}$, enabling subsequent iterations to leverage validated historical observations. If the execution fails, $\mathcal{A}_{\mathrm{v}}$ captures a structured error log $\mathcal{E}$ that summarizes the failure context, including the modified parameters and the stage at which the failure occurs. This error log is then forwarded to the agent $\mathcal{A}_{\mathrm{f}}$ for diagnosis. By decoupling execution validation from failure interpretation, $\mathcal{A}_{\mathrm{v}}$ ensures that only practically executable configurations are admitted, while failure information is systematically preserved for informed strategy adjustment. A detailed workflow description of $\mathcal{A}_{\mathrm{v}}$ is provided in Appendix D.4.

2.1.4 Failure Analysis Agent.

The agent $\mathcal{A}_{\mathrm{f}}$ is triggered whenever the agent $\mathcal{A}_{\mathrm{v}}$ reports an unsuccessful training attempt. Its primary role is to transform raw execution failures into structured knowledge that can be reused in subsequent optimization iterations.

Upon receiving the error log $\mathcal{E}$, $\mathcal{A}_{\mathrm{f}}$ performs a structured analysis to identify the root cause of the failure. Specifically, it classifies the failure into the categories (*e.g.*, out of memory, invalid parallelism configuration, or framework-level constraints such as non-divisible attention heads under tensor parallelism), and associates each failure with the configuration parameters that are most likely responsible. Importantly, each diagnosis is accompanied by a confidence score that quantifies the agent's certainty regarding the inferred root cause. This confidence design allows the system to distinguish between high-confidence parameter-induced

Algorithm 1 Iterative Strategy Optimization in OptiCo

Require: Global Message Pool $\mathcal{M}_{\text{pool}}$
Ensure: Optimized training configuration $\mathcal{C}_{\text{best}}$

$t \leftarrow 0$
if $\mathcal{M}_{\text{pool}} = \varnothing$ **then**
    Generate random initial configuration $\mathcal{C}_0$
    Execute training with $\mathcal{C}_0$, obtain performance metrics $\mathcal{P}_0$
    Append $(\mathcal{C}_0, \mathcal{P}_0)$ to $\mathcal{M}_{\text{pool}}$
    $t \leftarrow t + 1$
end if
while $t < T_{\max}$ **and** not Converged($\mathcal{M}_{\text{pool}}$) **do**
    $\Delta_t \leftarrow \mathcal{A}_{\text{a}}(\mathcal{M}_{\text{pool}})$
    $\mathcal{C}_t \leftarrow \mathcal{A}_{\text{g}}(\mathcal{M}_{\text{pool}}, \Delta_t, rules)$
    Run training using configuration $\mathcal{C}_t$ to get result $\mathcal{P}_t$
    if training successful **then**
        Append $(\mathcal{C}_t, \mathcal{P}_t)$ to $\mathcal{M}_{\text{pool}}$
    else
        Extract error log $\mathcal{E}_t$ from failed run
        $\mathcal{R}_t \leftarrow \mathcal{A}_{\text{f}}(\mathcal{E}_t)$
        Append failure record $(\mathcal{C}_t, \mathcal{R}_t)$ to $\mathcal{M}_{\text{pool}}$
    end if
    $t \leftarrow t + 1$
end while

failures and low-confidence or ambiguous cases.

The extracted failure information is then appended to $\mathcal{M}_{\text{pool}}$. This provides explicit negative feedback to both $\mathcal{A}_{\text{a}}$ and $\mathcal{A}_{\text{g}}$, enabling them to avoid previously invalid configurations and to refine future parameter modifications. By accumulating and reusing failure knowledge, $\mathcal{A}_{\text{f}}$ effectively narrows the search space and accelerates convergence toward executable and high-performance training strategies. A detailed workflow description of $\mathcal{A}_{\text{f}}$ is provided in Appendix D.5.

2.2 Global Message Pool

Inspired by expert-driven manual tuning workflows, where configuration strategies are refined through accumulated trial-and-error experience, our framework emphasizes the importance of shared execution knowledge for coordinated decision-making across agents. Rather than relying on direct point-to-point communication, which often incurs high coordination overhead and limits scalability (Li et al., 2023; Zhang et al., 2023), we introduce a Global Message Pool $\mathcal{M}_{\text{pool}}$ as a persistent and structured communication hub for all agents.

$\mathcal{M}_{\text{pool}}$ serves as a centralized execution memory that stores structured messages, including configuration settings, runtime performance metrics, and diagnosed failure causes. Agents interact with $\mathcal{M}_{\text{pool}}$ according to their functional roles:

As interactions accumulate over multiple iterations, $\mathcal{M}_{\text{pool}}$ builds a growing repository of strategy–performance and strategy–failure mappings. This shared memory enables agents to reuse prior execution knowledge, avoid repeatedly exploring invalid or ineffective configurations, and progressively prune the search space.

2.3 Iterative Feedback-Driven Strategy Optimization

Building upon the four agents described above, we design an iterative and interactive optimization framework that progressively refines training configurations through execution-driven feedback. The framework operates as a closed-loop process in which agents collaboratively reason, execute, and adapt until a satisfactory configuration is identified.

This iterative process enables the framework to continuously incorporate both positive and negative execution feedback, allowing agents to avoid repeatedly exploring invalid configurations and to progressively narrow the search space. The optimization terminates when a predefined stopping criterion is met. The overall procedure is summarized in Algorithm 1.

2.4 Inception Prompting

We adopt a structured prompt design paradigm to control our agent-based system. Our system relies on inception prompting, which drives agent behavior from a compact set of initial instructions. Once the interaction begins, agents operate autonomously in a closed-loop manner without external prompting.

As shown in Figure 2, we present a prompt example of $\mathcal{A}_{\text{a}}$. This prompt demonstrates our layered prompt design logic that integrates duty workflow, context conditioning, input emphasis, output constraints and behavior nudging. These prompt taxonomies are critical to ensure the agent performs safe, non-redundant reasoning under automation constraints. Our design balances precision and flexibility to support robust multi-agent coordination in training optimization tasks.

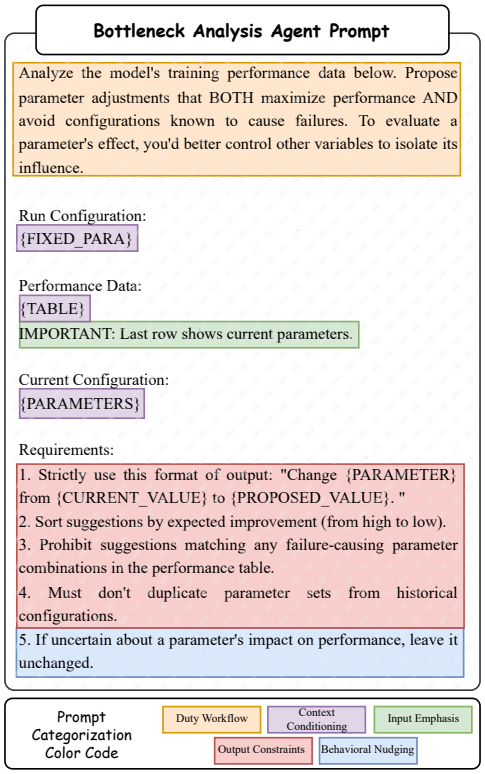


Figure 2: Inception prompt of $\mathcal{A}_{\text{a}}$.

3 Experiments

3.1 Experimental Settings

3.1.1 Models and Baselines

In our experiments, we evaluate OptiCo on Llama (Dubey et al., 2024), Qwen (Qwen et al., 2025) and GPT (Brown et al., 2020) model families, including *Llama2-0.5B*, *Llama2-7B*, *Llama3-0.5B*, *Llama3-8B*, *Qwen2.5-0.5B*, *Qwen2.5-7B*, *GPT3-7B*, and *Qwen3-moe-A0.6B*. Besides, we compare OptiCo with widely used automatic parallelism frameworks (Alpa, AMP and Aceso) and expert-tuned baselines which are derived from previous expert-guided configuration efforts.

3.1.2 Environments

All experiments are conducted on a cloud environment, with each node equipped with 8 NVIDIA A100 GPUs (80GB) or 8 XPU accelerators (96GB), where XPU denotes a non-NVIDIA high-performance accelerator. We adopt the different patched version[4] of Megatron-LM (Shoeybi et al., 2019) as our training framework.

[4] https://github.com/alibaba/Pai-Megatron-Patch

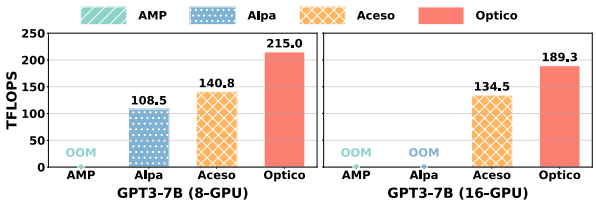


Figure 3: End-to-end training performance of GPT3-7B models with different approaches.

3.1.3 Metrics

We evaluate the performance of OptiCo using the following seven metrics: TFLOPS, MFU, TGS, Iter. $\geqslant$ Manual, Time $\geqslant$ Manual, Iter. (best), and Time (best). Detailed definitions for these metrics are provided in Appendix C.1.

3.2 Main Result

We conduct comprehensive evaluations of OptiCo on widely used models. Experiments are performed under two standard hardware settings: 8 GPUs and 16 GPUs. For all configurations, we limit the optimization process to at most 20 iterations.

As reported in Table 1, OptiCo consistently achieves better performance than expert manually designed strategies across multiple efficiency metrics. These expert-tuned baselines are derived from previous expert-guided configuration efforts. We selected the best-performing manual setting as the final reference, and a representative example of this manual tuning process is provided in Appendix C.2. Table 1 also reports the number of iterations and the corresponding time required for OptiCo to first surpass expert-tuned performance and to reach its best performance during optimization.

Besides, we compare OptiCo with representative automated parallelism frameworks, including AMP, Alpa, and Aceso, on end-to-end training performance of GPT3-7B with Megatron-LM Core v0.6.0. As shown in Figure 3, AMP fails to find valid configurations due to out-of-memory (OOM) errors with both 8 GPUs and 16 GPUs, while Alpa and Aceso exhibit limited scalability, with OOM failures or suboptimal throughput under larger device scale. In contrast, OptiCo consistently achieves strong performance across all evaluated settings. Notably, OptiCo maintains both high throughput and stable execution even when other methods fail, demonstrating its robustness and effectiveness in optimizing large-model training configurations. Moreover, the evaluation on XPUs is shown in Figure 4(k) and Figure 4(l).

Model	Config	⩾ Manual		Best		Manual			OptiCo		
		Iter.	Time(h)	Iter.	Time(h)	MFU	TGS	TFLOPS	MFU	TGS	TFLOPS
Llama2-0.5B	8-GPU	4	0.30	10	0.48	52.1%	54166.7	162.5	52.7%	54790.5	164.4
	16-GPU	1	0.05	1	0.05	48.0%	49900.0	149.7	48.2%	50107.9	150.5
Llama2-7B	8-GPU	20	3.48	20	3.48	67.5%	5011.9	210.5	67.7%	5026.8	211.2
	16-GPU	2	0.04	19	0.90	58.3%	4333.3	182.0	59.1%	4392.8	184.5
Llama3-0.5B	8-GPU	2	0.1	3	0.17	51.5%	53566.7	160.7	51.9%	53982.8	162.1
	16-GPU	11	0.49	12	0.50	45.7%	47533.3	142.6	47.3%	49197.5	147.4
Llama3-8B	8-GPU	15	3.25	15	3.25	67.9%	4416.7	212.0	68.2%	4436.2	212.7
	16-GPU	14	0.42	17	0.51	58.3%	3789.6	181.9	59.3%	3854.6	185.1
Qwen2.5-0.5B	8-GPU	1	0.14	5	0.30	45.4%	47266.7	141.8	45.8%	47683.1	142.8
	16-GPU	15	3.57	15	3.57	39.6%	41166.7	123.5	39.6%	41166.7	123.7
Qwen2.5-7B	8-GPU	8	1.71	8	1.71	67.3%	4997.6	209.9	67.9%	5042.2	211.7
	16-GPU	12	0.69	12	0.69	57.9%	4302.4	180.7	58.2%	4324.7	181.5
GPT3-7B	8-GPU	7	2.01	10	2.71	64.1%	4761.9	200.0	64.8%	4813.9	202.2
	16-GPU	10	0.20	10	0.20	56.1%	4164.3	174.9	57.2%	4246.0	178.5
Qwen3-MoE-A0.6B	8-GPU	7	0.79	7	0.79	19.2%	16666.7	60.0	19.5%	16927.1	60.8
	16-GPU	8	0.50	17	0.77	12.6%	10916.7	39.3	13.7%	11869.7	42.7

Table 1: Performance comparison of expert tuning and OptiCo across models and hardware configurations.

3.3 Optimization Process of OptiCo

As shown in Figure 4(a) to Figure 4(b) which illustrates the MFU optimization trajectories across two representative models, we observe that OptiCo consistently improves MFU within 20 iterations, surpassing manually-tuned baselines (blue dashed lines) on all settings. Notably, in Figure 4(a), a distinct upward shift occurs between iterations 14 and 15 (corresponding to the agent-generated suggestion: *Adjusted AC from full to none for performance gain*). Similarly, in Figure 4(b), a Marker $\times$ at iteration 12 indicates a training failure under this configuration (leading OptiCo to recommend *reducing the MBS from 2 to 1 to avoid OOM*). These changes align with expert intuition and demonstrate that OptiCo does not behave as a black-box optimizer. Instead, it performs step-by-step reasoning and iterative refinement in a manner comparable to human practitioners, with each decision traceable through interpretable agent outputs.

3.4 Ablation Experiments

We conduct extensive ablation experiments to assess the contribution of each component in the OptiCo. As shown in Table 2, the full OptiCo system outperforms all ablated variants in "Avg Performance Compared to Manual", and "% to Achieve Manual". Figure 4 illustrates MFU progression throughout the optimization process, highlighting OptiCo's superior convergence behavior and the effectiveness of its coordinated multi-agent design.

3.4.1 Analysis of Different Agent Models

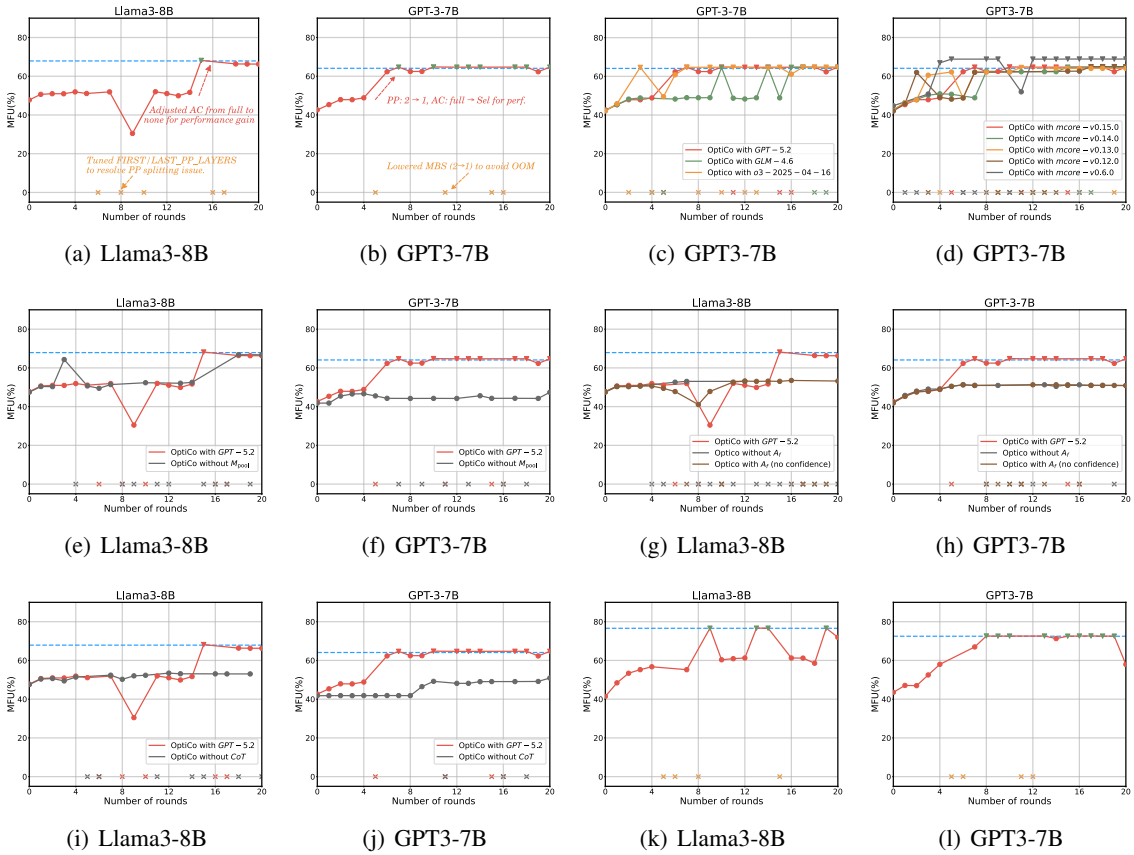
To investigate the flexibility of OptiCo with respect to the agent model, we conduct experiments using several widely adopted LLMs as agent backbones, including *GPT-5.2*, *OpenAI O3-2025-04-16*[5], and *GLM-4.6*[6]. This selection ensures that our evaluation covers diverse and representative models.

As reported in Table 2 and Figure 4(c), OptiCo achieves competitive performance with all agent models, demonstrating its general adaptability. Among them, *GPT-5.2* consistently delivers the strongest overall performance. Although *GPT-5.2* does not always minimize the number of optimization iterations, it reliably converges to higher-quality configurations and achieves the best final performance. Based on its superior optimization outcomes and robustness, we adopt *GPT-5.2* as the default agent model.

3.4.2 Analysis of Different Megatron-LM Cores

To evaluate the robustness of OptiCo with training framework, we conduct a study across multiple versions of Megatron-LM Core. Specifically, we consider Megatron-LM Core v0.6.0, which is adopted by Alpa, AMP and Aceso (adapting these frameworks to newer Megatron releases would require non-trivial engineering effort due to evolving framework interfaces and execution semantics,

[5] https://platform.openai.com/docs/models/o3
[6] https://docs.z.ai/guides/llm/glm-4.6



(a) Llama3-8B (b) GPT3-7B (c) GPT3-7B (d) GPT3-7B

(e) Llama3-8B (f) GPT3-7B (g) Llama3-8B (h) GPT3-7B

(i) Llama3-8B (j) GPT3-7B (k) Llama3-8B (l) GPT3-7B

Figure 4: Comparison of optimization processes of Llama3-8B and GPT3-7B using OptiCo with 8 accelerators. Subfigures (a) and (b) illustrate the optimization process using OptiCo. Subfigure (c) compares the optimization process of different agent models. Subfigure (d) compares the optimization process of different Megatron-LM Cores. Subfigures (e) and (f) compare the effect of using the Global Message Pool in OptiCo. Subfigures (g) and (h) show whether the Failure Analysis Agent is used and whether it incorporates confidence estimation. Subfigures (i) and (j) compare the effect of using the CoT reasoning. Subfigures (k) and (l) compare the optimization process of different accelerators. Markers: ▾ denotes configurations surpassing manual tuning. • denotes configurations not surpassing manual tuning. × indicates training failure. The blue dashed line denotes the manually tuned baseline.

whereas OptiCo remains directly applicable without version-specific modifications), as well as several recent releases up to the latest Megatron-LM Core v0.15.0.

As shown in Figure 4(d), OptiCo consistently achieves substantial performance across all Megatron-LM Core versions, indicating that its optimization capability is insensitive to framework-level implementation changes. Notably, the highest achievable performance is observed on v0.6.0, instead of the most recent v0.15.0. This observation is not unexpected. Recent Megatron-LM releases increasingly prioritize engineering correctness, fault tolerance, and large-scale training stability. As a result, older versions can occasionally exhibit higher raw throughput, while newer releases are designed to better support large-scale, stable, and production-oriented training workloads. Given these considerations, we adopt Megatron-LM Core v0.15.0 for all remaining experiments.

3.4.3 Analysis of Global Message Pool

We analyze the role of the Global Message Pool by comparing OptiCo's optimization process, as illustrated in Figure 4(e) to Figure 4(f). When the Global Message Pool is disabled, a number of generated strategies across multiple iterations result in invalid configurations, leading to failed training attempts. In contrast, with the Global Message Pool enabled, agents avoid repeating past failures by reasoning over accumulated execution history. This effect is supported by the results in Table 2. Without Global Message Pool, average performance drops to 86.15% of the manually-tuned MFU, and none of configurations surpass expert baselines. These findings highlight the critical role of the Global

Config	Avg Performance Compared to Manual	% to Achieve Manual	Avg Num of Iters
OptiCo	**101.84%**	**100%**	9
with o3-2025-04-16	101.00%	**100%**	**3**
with GLM-4.6	100.90%	**100%**	10
w/o Global Message Pool	86.15%	0%	N.A.
w/o Failure Analysis Agent	79.22%	0%	N.A.
w/o CoT Reasoning	79.06%	0%	N.A.

Table 2: Ablation results comparing different variants of the OptiCo framework in 8-GPU settings. "Avg Performance Compared to Manual" reports the average ratio between OptiCo-generated configurations and expert-tuned configurations in terms of performance. "% to Achieve Manual" denotes the percentage of OptiCo-generated configurations whose performance exceeds that of the corresponding expert-tuned configuration. "Ave Num of Iters" denotes the average number of iterations required for the method to first achieve performance equal to or better than that of manually tuned expert configurations.

Message Pool in enabling memory-aware decision making, improving both stability and sample efficiency throughout the optimization process.

3.4.4 Analysis of Failure Analysis Agent

We evaluate the impact of the Failure Analysis Agent on optimization performance by comparing settings with and without it. As shown in Figure 4(g) to Figure 4(h), incorporating the Failure Analysis Agent yields a notable improvement in MFU. This improvement arises from the agent's capability to systematically diagnose failed configurations and encode the corresponding failure causes into the Global Message Pool. By maintaining a persistent record of failure reasons, the system effectively constrains the search space in subsequent iterations, reducing redundant failures and enabling more targeted strategy refinement. This observation is corroborated by the ablation results in Table 2, where removing the Failure Analysis Agent results in a drop in both average performance (79.22% vs. 101.84%) and none of configurations surpass expert baselines. These findings underscore the importance of failure-aware reasoning in improving optimization stability and success rate across diverse configurations.

Further, we conduct a prompt-level ablation to examine whether incorporating explicit confidence estimation improves MFU. Comparing prompt with and without confidence estimation as shown in Figure 4(g) to Figure 4(h), we observe that incorporating confidence scores leads to more stable convergence. This improvement is due to the ability of confidence-conditioned outputs to guide downstream agents toward high-priority failure causes, while reducing the influence of ambiguous or non-deterministic error signals.

3.4.5 Analysis of CoT Reasoning

To evaluate the impact of CoT reasoning in agent decision making, we compare optimization performance with and without CoT prompting in the Bottleneck Analysis Agent, as shown in Figure 4(i) to Figure 4(j). Across all models, CoT-enhanced agents achieve faster and more stable improvements in MFU.

This trend is further supported by quantitative results in Table 2, where disabling CoT leads to a dramatic decrease in rate that none of configurations without CoT surpass the manual tuning baseline. These results highlight the importance of structured reasoning in enabling effective parameter selection, especially in complex model settings with intricate performance bottlenecks.

3.5 Cost-Benefit Analysis

The API cost of LLM calls in OptiCo is negligible compared to GPU training cost. In the 16-GPU A100 GPT3-7B setting, OptiCo requires 32 LLM calls in total to reach the best TGS. Assuming an average of 3,000 input tokens and 200 output tokens per call, the cost per call is approximately $0.00805, resulting in a total LLM cost of $0.26 for the entire optimization process.

To quantify the benefit, we compare OptiCo with expert tuning under the same setting as shown in Table 1. Expert tuning achieves 4,164.3 tokens/s/GPU, while OptiCo achieves 4,246.0 tokens/s/GPU. For a training target of 1T tokens, this difference translates into a saving of approximately 80 training hours, equivalent to 1,280 A100 GPU-hours.

In terms of optimization overhead, OptiCo takes about 0.2 hours to find the best configuration, whereas manual expert tuning typically requires around 8 hours based on our empirical experience.

Therefore, OptiCo not only reduces overall training time but also lowers configuration effort. Given the substantially higher cost of GPU compute relative to LLM API calls ($0.26), the resulting savings far exceed the optimization overhead. Besides, large-scale foundation model training often involves trillions of tokens, where such improvements scale proportionally.

4 Conclusion

In this paper, we presented OptiCo, *the first* collaborative multi-agent framework that automates the optimization of distributed training strategies for large-scale deep learning. By coordinating specialized agents for bottleneck analysis, strategy generation, strategy validation, and failure analysis through a shared Global Message Pool, OptiCo emulates expert reasoning workflows and enables iterative and explainable strategy refinement. Leveraging large language models with CoT prompting and structured inception instructions, OptiCo demonstrates strong adaptability across diverse models, hardware settings, and parallel configurations. Extensive experiments show that OptiCo consistently outperforms expert-tuned baselines within 20 iterations, achieving an average performance improvement of 1.84%, with gains ranging from 0.08% to 8.65%. Ablation studies highlight the importance of memory sharing, failure-aware reasoning, and structured prompting in accelerating convergence and avoiding invalid configurations. These findings validate the effectiveness of multi-agent reasoning in distributed training and offer a generalizable solution for automated system optimization.

Limitations

Despite its effectiveness, OptiCo has several limitations that are important to consider:

(i) **Limited evaluation scope:** Due to resource constraints, our experimental evaluation was conducted only on 8-GPU and 16-GPU environments. While these settings provide initial validation of OptiCo's effectiveness, further experiments on larger-scale clusters are necessary to fully assess the framework's scalability and generalizability to more complex deployment scenarios.

(ii) **Handling of repeated failures and computational overhead:** OptiCo is designed to recover from training failures using automated analysis and strategy adjustment. However, repeated failed attempts can introduce additional computational overhead and latency. This suggests that, while the framework is fault-tolerant, efficiency may degrade under conditions with frequent or systemic failures, highlighting the need for cost-aware exploration or prioritization mechanisms in future iterations.

(iii) **Limited handling of multi-tenant interference:** OptiCo currently assumes that the training workload has exclusive access to cluster resources. In multi-tenant environments where multiple jobs share GPUs, CPUs, or network bandwidth, contention can affect profiling accuracy and strategy effectiveness, potentially leading to suboptimal scheduling or unexpected slowdowns.

5 Acknowledgements

We thank all reviewers for their insightful comments and feedback. This work is supported by the National Key Research and Development Program of China (No. 2023YFE0108600), National Natural Science Foundation of China Grant (No. U22A6001), the Key Research Project of Zhejiang Lab (No. 2025SSYS0005) and Zhejiang Provincial Natural Science Foundation of China (No. LQK26F020007).

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Jiaqi Chen, Yuxian Jiang, Jiachen Lu, and Li Zhang. 2024. S-agents: Self-organizing agents in open-ended environments. *arXiv preprint arXiv:2402.04578*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, and 1 others. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, and 1 others. 2021. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445.

Runsheng Benson Guo, Utkarsh Anand, Arthur Chen, and Khuzaima Daudjee. 2024. Cephalo: Harnessing heterogeneous gpu clusters for training transformer models. *arXiv preprint arXiv:2411.01075*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and 1 others. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13.

Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. 2022. Amp: Automatically finding model parallel strategies with heterogeneity awareness. *Advances in Neural Information Processing Systems*, 35:6630–6639.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. 2024. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 163–181.

Yat Long Lo, Christian Schroeder de Witt, Samuel Sokota, Jakob Nicolaus Foerster, and Shimon Whiteson. 2023. Cheap talk discovery and utilization in multi-agent reinforcement learning. *arXiv preprint arXiv:2303.10733*.

Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878*.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. 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.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

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

Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 86–100.

Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast automatic distributed training on heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 563–578.

Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17.

Yujie Wang, Youhe Jiang, Xupeng Miao, Fangcheng Fu, Shenhan Zhu, Xiaonan Nie, Yaofeng Tu, and Bin Cui. 2024. Improving automatic parallel training via balanced memory workload optimization. *IEEE Transactions on Knowledge and Data Engineering*, 36(8):3906–3920.

Junde Wu, Jiayuan Zhu, and Yuyuan Liu. 2025. Agentic reasoning: Reasoning llms with tools for the deep research. *arXiv preprint arXiv:2502.04644*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B Tenenbaum, Tianmin Shu, and Chuang Gan. 2023. Building cooperative embodied agents modularly with large language models. *arXiv preprint arXiv:2307.02485*.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, and 1 others. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578.

A Parallelism Strategies

A.1 Data Parallelism

Data parallelism is a foundational strategy in distributed training, where the training dataset is partitioned across multiple workers, each maintaining a replica of the model. During each iteration, workers compute gradients independently on their local data and then synchronize updates to ensure consistent model states. While data parallelism scales computation effectively, it requires the full model to fit in each device's memory, which limits its applicability for large models.

A.2 Tensor Parallelism

Tensor parallelism is a model parallel strategy that partitions the weight tensors of neural network layers across multiple devices. Instead of replicating full layers on each GPU, the parameters and corresponding computations are split into smaller shards, allowing different devices to collaboratively process portions of a layer during forward and backward passes. This approach reduces per-device memory consumption, making it suitable for training large models that exceed single-device capacity.

A.3 Pipeline Parallelism

Pipeline parallelism partitions a model into sequential stages, each assigned to a different device or group of devices. During training, the input mini-batch is divided into smaller micro-batches, which are processed in a staggered pipeline fashion across the stages. Pipeline parallelism reduces the memory footprint per device by limiting the scope of model layers held on each one.

A.4 3D Parallelism

3D parallelism integrates data parallelism, tensor parallelism, and pipeline parallelism to enable efficient training of large-scale models. By combining these strategies, 3D parallelism partitions both model parameters and computation across multiple devices while distributing the training data. The hierarchical design of 3D parallelism balances memory usage, computation load, and communication cost. Data parallelism ensures scalable throughput across replicas, tensor parallelism splits individual layer computations to fit large models, and pipeline parallelism sequences model stages across devices to further reduce memory footprint. Together, these methods form a flexible parallelization scheme that

maximizes hardware utilization for training models.

B Related Work

B.1 Conventional Parallel Strategy Configuration

Training large-scale deep learning models presents significant challenges due to their substantial computational and memory demands. Traditional distributed training systems predominantly depend on manual parallel strategies, which demand substantial domain expertise and extensive manual effort. To reduce manual tuning efforts, a range of automated parallel configuration frameworks have emerged (Fan et al., 2021; Jia et al., 2019; Narayanan et al., 2019; Sun et al., 2024; Wang et al., 2024; Li et al., 2022; Shoeybi et al., 2019; Wang et al., 2019; Zheng et al., 2022). Frameworks such as Galvatron (Miao et al., 2022) and Aceso (Liu et al., 2024) explore hybrid parallelism search spaces by leveraging decision trees, dynamic programming search algorithms or iterative bottleneck alleviation strategies. Nevertheless, the inherent combinatorial complexity of the optimization space remains a critical challenge. In addition, frameworks like CEPHALO (Guo et al., 2024) and Metis (Um et al., 2024) extend conventional methods by incorporating hardware heterogeneity into parallelism planning. Overall, existing automatic parallelism tuning solutions typically function with approximated profilers, cost models and optimizers, offering limited interpretability and flexibility. They often focus narrowly on throughput maximization, without providing insights into why certain configurations are selected or how they adapt to changing workloads. Therefore, in this area, our multi-agent, reasoning-driven framework OptiCo has the potential to offer explainable and adaptive parallel strategy configuration.

B.2 LLM-Based Multi-Agent Frameworks

The rapid development of LLMs has substantially advanced the field of agent-based systems. Leveraging LLMs as core reasoning engines, multi-agent frameworks (Li et al., 2023; Wu et al., 2024; Chen et al., 2024; Park et al., 2023; Schick et al., 2023; Lo et al., 2023) have emerged as a promising paradigm for solving complex tasks. Unlike single-agent systems, multi-agent systems can simulate diverse roles, collaborate dynamically, and tackle problems with a division of labor. Recent studies have explored various agentic designs for enhancing LLM capabilities. For instance, Agentic Reasoning (Wu et al., 2025) introduces a pipeline where LLMs interact with external tools such as web search engines and code execution modules during inference, enabling context-aware and tool-augmented reasoning. MetaGPT (Hong et al., 2023) adopts a human workflow, assigning specialized roles to agents and structuring communication protocols to prevent misunderstanding during multi-round interactions. Similarly, ChatDev (Qian et al., 2023) demonstrates a chat-driven multi-agent framework for software engineering tasks, where agents cooperate on design, coding, and testing phases. Different from the aforementioned frameworks, OptiCo is specifically designed for configuration and optimization of large-scale model training strategy. By coordinating multiple agents, OptiCo automates the exploration of training strategy configuration and performance tuning, substantially reducing the human effort and expertise required for model efficiency optimization.

C Supplementary Experimental Details and Results

C.1 Detailed Definitions of Evaluation Metrics

This section provides detailed definitions of the evaluation metrics used throughout the paper. These metrics are designed to assess training performance from the perspectives of computational efficiency, hardware utilization, throughput, and optimization convergence behavior.

- **TFLOPS.** This metric denotes the floating-point operations per second (in teraFLOPs) for each GPU during training.
- **MFU.** MFU measures the utilization efficiency of hardware compute capacity. It is defined as the ratio between the model's FLOPs per iteration and the peak available hardware FLOPs over the same period.
- **TGS.** TGS reflects token-level throughput and is defined as the number of tokens processed per second per GPU.
- **Iter.** or **Time** $\geqslant$ **Manual.** This metric indicates the number of iterations or the amount of time required for OptiCo to match or surpass the performance of manually tuned configurations by experts.

	Expert Tuning																	
	Model Type	Mode Size	Nnodes	Nproc	Precison	SEQ_LENGTH	GLOBAL_BATCH_SIZE	MICRO_BATCH_SIZE	TP	PP	Result	TFLOPS	MFU	MemAve	MemMax	MemLimit	Reasons for Failure
	Llama3	8B	1	8	fp16	8192	512	1	1	1	fail	NULL	NULL	NULL	NULL	NULL	OOM
	Llama3	8B	1	8	fp16	8192	512	1	1	2	success	210.86	68%	72478.72	77291.52	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	1	4	success	187.81	60%	62074.88	73717.76	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	1	8	fail	NULL	NULL	NULL	NULL	NULL	OOM
	Llama3	8B	1	8	fp16	8192	512	1	2	1	success	209.13	67%	54272	57026.56	81920	NULL
Modify TP and PP	Llama3	8B	1	8	fp16	8192	512	1	2	2	success	198.04	63%	50391.04	54865.92	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	2	4	success	177.71	57%	30812.16	44533.76	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	4	1	success	186.98	60%	35665.92	39526.4	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	4	2	success	182.6	59%	29214.72	36894.72	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	8	1	success	152.44	49%	26501.12	29962.24	81920	NULL
	Llama3	8B	1	8	fp16	8192	256	1	2	1	success	198.15	64%	62607.36	66191.36	81920	NULL
Modify GLOBAL BATCH SIZE	Llama3	8B	1	8	fp16	8192	1024	1	2	1	success	208.5	67%	53995.52	57128.96	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	1	1	1	fail	NULL	NULL	NULL	NULL	NULL	OOM
	Llama3	8B	1	8	fp16	8192	512	2	2	1	fail	NULL	NULL	NULL	NULL	NULL	OOM
Modify MICRO BATCH SIZE	Llama3	8B	1	8	fp16	8192	512	1	8	1	success	152.44	49%	26501.12	29962.24	81920	NULL
	Llama3	8B	1	8	fp16	8192	512	2	8	1	success	165.09	53%	36956.16	40744.96	81920	NULL

Figure 5: Example of Expert Tuning of Llama3-8B.

- **Iter.** or **Time (best).** This metric records the specific iteration or the amount of time in which OptiCo reaches its highest performance.

C.2 Example of Expert Tuning

Figure 5 presents a real-world example of manual tuning performed by a domain expert on the *Llama3-8B* model. As illustrated in Figure 5, the expert adopts an iterative process, adjusting one group of parameters at a time (e.g., only modifying MICRO_BATCH_SIZE) and monitoring hardware-level indicators such as memory usage after each change. This decomposed, step-by-step workflow mirrors the reasoning and decision-making pattern implemented in our multi-agent framework OptiCo.

Besides, expert tuning is inherently time-consuming and requires extensive prior knowledge of both system constraints and model behavior. In contrast, OptiCo automates this process through collaborative agent reasoning, achieving superior performance with fewer iterations and significantly less human effort.

C.3 Supplementary Evaluation on Additional Models

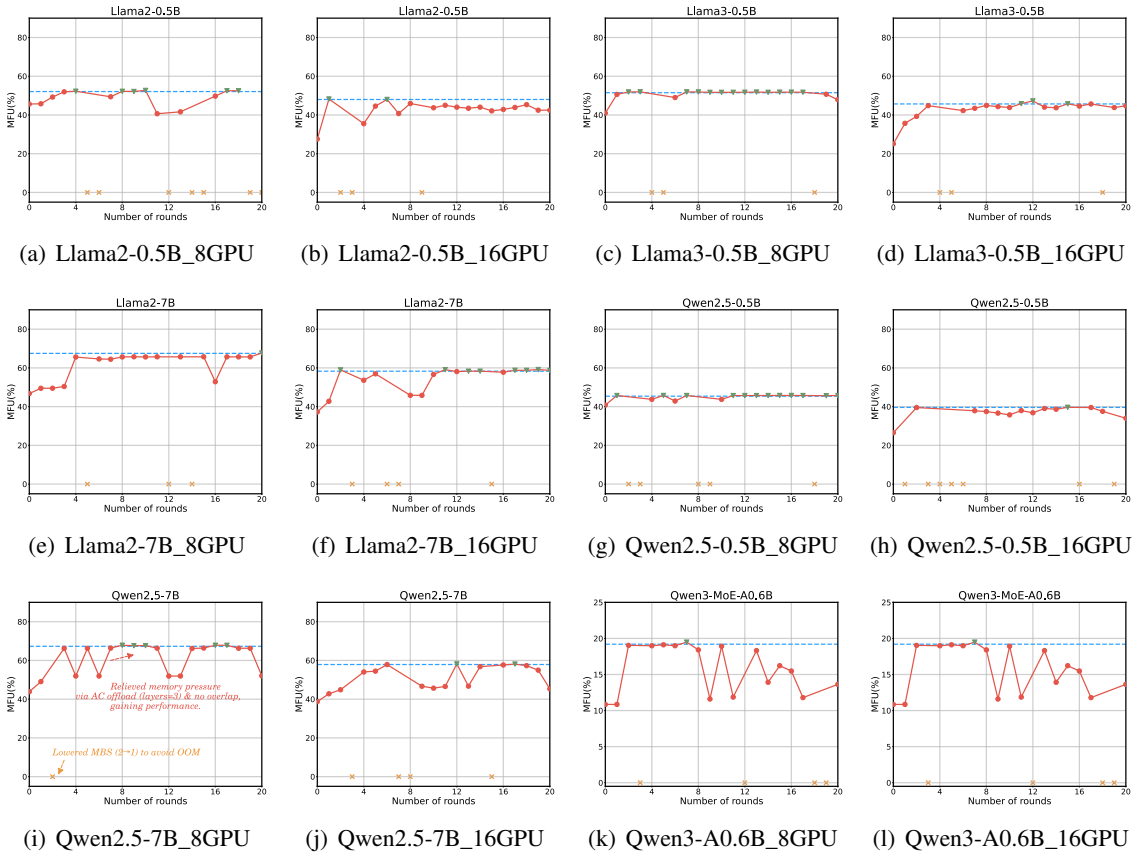
In the main paper, we focus on *Llama3-8B* and *GPT3-7B* with 8 accelerators as representative models to present the core experimental results. To further examine the generality of our approach, we additionally evaluate it on a broader set of model architectures in this appendix.

Specifically, we conduct experiments on *Llama2-0.5B*, *Llama3-0.5B*, *Llama2-7B*, *Qwen2.5-0.5B*, *Qwen2.5-7B* and *Qwen3-MoE-A0.6B* with both 8 GPUs and 16 GPUs, covering models with different parameter scales and architectural characteristics (MoE architecture, *Qwen3-MoE-A0.6B*). As shown in Figure 6, the proposed method exhibits performance trends that are consistent with the observations in the main paper. These results further support that our approach is not tailored to a specific model, but can generalize across different backbone architectures.

D Implementation Details

D.1 Initial Input

The optimization process in OptiCo begins with the Global Message Pool ($\mathcal{M}_{\text{pool}}$), which serves as the central input to all agents. Figure 7 illustrates the contents of the Global Message Pool in both scenarios: empty initialization and populated state after the first iteration of execution. If $\mathcal{M}_{\text{pool}}$ is initially empty (top of Figure 7), the system executes a randomly generated configuration once to collect the necessary performance statistics and populate the pool. Otherwise (bottom of Figure 7), the historical information available in $\mathcal{M}_{\text{pool}}$ is directly used as input to guide subsequent agent decisions.



(a) Llama2-0.5B_8GPU (b) Llama2-0.5B_16GPU (c) Llama3-0.5B_8GPU (d) Llama3-0.5B_16GPU

(e) Llama2-7B_8GPU (f) Llama2-7B_16GPU (g) Qwen2.5-0.5B_8GPU (h) Qwen2.5-0.5B_16GPU

(i) Qwen2.5-7B_8GPU (j) Qwen2.5-7B_16GPU (k) Qwen3-A0.6B_8GPU (l) Qwen3-A0.6B_16GPU

Figure 6: Comparison of optimization processes of *Llama2-0.5B*, *Llama3-0.5B*, *Llama2-7B*, *Qwen2.5-0.5B*, *Qwen2.5-7B* and *Qwen3-MoE-A0.6B* using OptiCo in different configurations (8GPU/16GPU).

Global Message Pool (Empty Pool)

Model Type	Mode Size	Nnodes	Nproc	Precison	SEQ_LENGTH	GLOBAL_BATCH_SIZE	MICRO_BATCH_SIZE	TP	PP	Result	TFLOPS	MFU	MemAve	MemMax	MemLimit	Log Director	Reasons for Failure
/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/

Global Message Pool (Historical Data)

Model Type	Mode Size	Nnodes	Nproc	Precison	SEQ_LENGTH	GLOBAL_BATCH_SIZE	MICRO_BATCH_SIZE	TP	PP	Result	TFLOPS	MFU	MemAve	MemMax	MemLimit	Log Director	Reasons for Failure
Llama3	0.5B	1	8	fp16	1024	128	1	1	1	success	31.7	0.101603	3966.12	8142	81920	1	NULL
Llama3	0.5B	1	8	fp16	1024	192	2	2	1	success	24.4667	0.0784189	3423.29	7304	81920	2	NULL
Llama3	0.5B	1	8	fp16	2048	192	3	2	1	success	71.4	0.228846	5566.33	12454	81920	3	NULL
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Llama3	0.5B	1	8	fp16	4096	1536	4	4	2	fail	NULL	NULL	NULL	NULL	NULL	22	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

TP 4 and num_attention_heads 14 caused ValueError due to incompatibility (Confidence: 95)

Figure 7: Initial Input of Global Message Pool.

D.2 Bottleneck Analysis Agent

The Bottleneck Analysis Agent $\mathcal{A}_{\text{a}}$ is designed to identify configuration parameters that most critically affect training efficiency.

Rather than presenting the agent behavior as a monolithic black box, we explicitly decompose its workflow into three structured components: (i) the constrained prompt specification, (ii) the agent input, (iii) the internal CoT reasoning process.

Specifically, $\mathcal{A}_{\text{a}}$ takes as input a historical performance table, where each row corresponds to a previously evaluated training configuration annotated with throughput and failure status. Based on this input, $\mathcal{A}_{\text{a}}$ operates under a strictly constrained prompt that defines the admissible action space, forbids failure-prone configurations, and enforces output formatting rules.

$\mathcal{A}_{\text{a}}$ then performs reasoning to isolate the im-

pact of individual parameters while controlling for confounding factors. Importantly, the reasoning process is guided by explicit constraints that discourage blind increases in tensor or pipeline parallelism, unless empirical evidence suggests performance gains.

Finally, $\mathcal{A}_a$ outputs a ranked list of parameter modification actions, sorted by their expected performance improvement and guaranteed to avoid historically observed failure cases.

For clarity and reproducibility, we present the agent's prompt specification, input format, and CoT reasoning template using structured blocks, as illustrated in the following boxes.

Prompt of Bottleneck Analysis Agent

Analyze the model's training performance data below. Propose parameter adjustments that BOTH maximize performance AND avoid configurations known to cause failures. To evaluate a parameter's effect, you'd better control other variables to isolate its influence.
Run Configuration:
{FIXED_PARA}
Performance Data:
{TABLE}.
IMPORTANT:
Last row shows current parameters.
Current Configuration:
{PARAMETERS}.
Requirements:
1. Strictly use this format of output: "Change {PARAMETER} from {CURRENT_VALUE} to {PROPOSED_VALUE}. "
2. Sort suggestions by expected improvement (from high to low).
3. Prohibit suggestions matching any failure-causing parameter combinations in the performance table.
4. Must do not duplicate parameter sets from historical configurations.
5. If uncertain about a parameter's impact on performance, leave it unchanged.

Input of Bottleneck Analysis Agent

You are an expert in distributed deep learning performance tuning. Your task is to analyze performance data and propose hyperparameter adjustments to maximize Model FLOPs Utilization (MFU).

CRITICAL FORMAT REQUIREMENT:
You MUST ALWAYS provide your reasoning inside '<think>' tags first, then provide your final suggestions outside the think tags. Your final output must strictly follow this format:
'<think>'
[Your step-by-step reasoning goes here]
'</think>'
The suggestions are:
[Your formatted suggestions here]
(Chain of Thought: Performance Tuning Suggestions)
Please answer the following open-ended question. You should think step by step to solve it.
Strictly provide your final answer in the format:
'<think>'
[Your step-by-step reasoning goes here]
'<think>'
The suggestions are:
[Your formatted suggestions here]
Now, here is the task:
Question:
Analyze the model's training performance data below. Propose parameter adjustments that BOTH maximize performance AND avoid configurations known to cause failures. To evaluate a parameter's effect, you'd better control other variables to isolate its influence.
Run Configuration:
Model Type: gpt3
Model Size: 7B
NUM_LAYERS: 32
Precison: fp16
Global Batch Size: 1024
Sequence Length: 4096
Number of Nodes: 2
Number of GPUs: 16
GPU memory limit per card: 81920
Performance Data:
(Global Message Pool)
IMPORTANT:
Last row shows current parameters.
Current Configuration:
MICRO_BATCH_SIZE: 2
TP: 1
PP: 1
OVERLAP_GRAD_REDUCE: true
OVERLAP_PARAM_GATHER: true

AC: full
AC_LAYERS: 2
FUSED_CROSS_ENTROPY: true
FIRST_PP_NUM_LAYERS: None
LAST_PP_NUM_LAYERS: None
Requirements:
1. Strictly use this format of output: "Change {PARAMETER} from {CURRENT_VALUE} to {PROPOSED_VALUE}. "
2. Sort suggestions by expected improvement (from high to low).
3. Prohibit suggestions matching any failure-causing parameter combinations in the performance table.
4. Must don't duplicate parameter sets from historical configurations.
5. If uncertain about a parameter's impact on performance, leave it unchanged.

CoT reasoning of Bottleneck Analysis Agent

Step 0: Mandatory Health Diagnosis & Directive
Before any analysis, determine the system's health from the current run's metrics.
'gpu_util': GPU utilization (%).
'mem_usage': Peak memory usage (%).
Health States & Directives:
(python)
if gpu_util < 50 and mem_usage < 70:
health_state = "SEVERELY_UNDERUTILIZED"
PRIMARY_DIRECTIVE = "REDUCE_PARALLELISM"
reasoning = "Low GPU and memory utilization indicates gross inefficiency from excessive parallelism overhead."
elif PP > 1 and MFU < 0.5:
health_state = "HIGH_PP_OVERHEAD"
PRIMARY_DIRECTIVE = "REDUCE_PIPELINE_DEPTH"
reasoning = "High PP value creates significant pipeline bubble overhead, leading to low MFU."
elif mem_usage > 95:
health_state = "MEMORY_BOUND"
PRIMARY_DIRECTIVE = "RESOLVE_MEMORY_PRESSURE"
else:
health_state = "PERFORMANCE_TUNE"
# No overriding directive; proceed with standard bottleneck analysis.
Step 1: Understand Key Parameters
1. MICRO_BATCH_SIZE (MBS): Increases throughput, may cause OOM.
2. TP & PP: Parallelism strategies. High values increase communication overhead.
3. AC (Activation Checkpointing): Trades compute for memory. 'none' (fast, high memory) → 'sel' → 'full' → 'offload' (slow, low memory).
4. OVERLAP_GRAD_REDUCE / OVERLAP_PARAM_GATHER: Generally enable 'true' to hide communication cost, unless 'AC=offload'.
5. FUSED_CROSS_ENTROPY: Generally enable 'true' for speed.
Step 2: Enforce Safety Guards (Before Any Suggestion)
Duplicate Check: NEVER suggest a parameter set identical to a past run's signature. This check is absolute and non-negotiable.
Failure Check: NEVER suggest a combination that previously caused OOM or errors.
Constraint Check: MUST validate (e.g., 'GLOBAL_BATCH_SIZE % MBS == 0', 'TPPP <= GPUs').
Step 3: Generate The Next Suggestion
Follow this algorithm strictly:
3.1 Obey Health Directive (Highest Priority)
If 'health_state == "SEVERELY_UNDERUTILIZED"' or 'health_state == "HIGH_PP_OVERHEAD"':
Process:
1. Generate Candidates: Prioritize reducing PP if PP > 1. Create an ordered list of candidate values (e.g., [PP//2, 1]). If PP reduction is not possible or all candidates fail, generate a similar list for TP reduction.
2. Evaluate & Check: For each candidate value in order:
Construct the complete candidate configuration.
Perform the Constraint Check.
Perform the Duplicate Check against the full history.
Perform the Failure Check against known bad configurations.
3. Select: The first candidate that passes all checks is selected as the suggestion.

4. Fallback: If all candidates for reducing parallelism result in a duplicate configuration, violate constraints, or match a known failure, proceed directly to Step 3.3.
Reason: The PRIMARY_DIRECTIVE is 'REDUCE_PARALLELISM' or 'REDUCE_PIPELINE_DEPTH'.
Output: If a suggestion is found, proceed to Step 4.

3.2 Standard Bottleneck Tuning

If 'health_state' is 'MEMORY_BOUND' or 'PERFORMANCE_TUNE':
Process:
1. Diagnose & Generate Candidate List: Based on the metrics, create a prioritized list. For example:
If OOM or 'mem_usage > 90%': Candidates = [("AC", "sel"), ("MBS", 'max(1, MBS//2)')]
If 'gpu_util > 80%' and memory headroom exists: Candidates = [("MBS", next higher safe divisor)]
If overlaps are disabled and 'AC != 'offload'': Candidates = [("OVERLAP_GRAD_REDUCE", True), ("OVERLAP_PARAM_GATHER", True)]
If FUSED_CROSS_ENTROPY is disabled: Candidates = [("FUSED_CROSS_ENTROPY", True)]
2. Evaluate & Check in Order: For each '(parameter, new_value)' in the prioritized list:
Construct the complete candidate configuration.
Perform the Constraint Check, Duplicate Check, and Failure Check.
3. Select: The first candidate that passes all checks is selected.
4. Fallback: If all candidates result in a duplicate, violate constraints, or match a known failure, proceed to Step 3.3.

3.3 Exploratory Tuning (Fallback)

Trigger: This step is invoked only if all targeted changes in Step 3.1 or Step 3.2 failed because they would produce a duplicate, unsafe, or invalid configuration.
Purpose: To suggest a minimal, safe change to a less-utilized parameter, ensuring progress without repetition.
Process: Generate a list of candidate changes targeting parameters not primarily tuned for performance (e.g., toggling 'FUSED_CROSS_ENTROPY', adjusting 'AC_LAYERS', setting 'FIRST_PP_NUM_LAYERS' or 'LAST_PP_NUM_LAYERS' to a small integer like 4). Evaluate each candidate with the same strict Constraint, Duplicate, and Failure Checks. The first valid candidate is selected. If none are found, output an error stating that all plausible configurations have been tried.

Step 4: Finalize & Format Output

1. Final Verification: Confirm the selected suggestion has passed all Step 2 Safety Guards during the evaluation in Step 3.
2. Output Reasoning: Place the complete reasoning chain (health diagnosis, candidate generation, check results, final selection logic) inside '<think>' tags.
3. Output Suggestion: After '</think>', present the clear suggestion (e.g., "Set PP=4") or the error message from Step 3.3.

RULES SUMMARY

1. Health Directive is Absolute: A 'SEVERELY_UNDERUTILIZED' or 'HIGH_PP_OVERHEAD' diagnosis must first attempt to reduce parallelism .
2. Zero-Tolerance on Historical Repetition: A suggested configuration must be verified as novel against the entire historical record. This is enforced by the structured evaluation loop in Step 3.
3. Safety First: Never suggest configurations known to cause failures or violate constraints.
4. Guaranteed Progress: If standard tuning leads to duplicates or failures, the system must attempt exploratory changes (Step 3.3) to propose a new, valid configuration.
5. Think First: Always output reasoning within '<think>' tags before the final answer.

Output of Bottleneck Analysis Agent

Health Diagnosis & Directive

- Current run (last row): MFU=0.440064, GpuUtilAve=73.4184%, MemMax=44402 MiB (54% of 80GB).
- Health-state checks:
- gpu_util < 50 and mem_usage < 70? => false (gpu_util 73%).
- PP > 1 and MFU < 0.5? => false (PP=1).

- mem_usage > 95? => false (54%).
- health_state = PERFORMANCE_TUNE (no overriding directive).
Safety Guards: constraints, duplicates, failures
- Constraints: GLOBAL_BATCH_SIZE=1024 must be divisible by MBS; with MBS=2 it is valid. TP*PP must be <= 16; current 1*1 valid; proposed must remain valid.
- Known failure combinations to avoid (exact matches from table):
- (MBS=4, TP=1, PP=1, OVERLAPs=True, FUSED_CE=True) with AC=sel, AC_LAYERS=1 => OOM (run 9)
- (MBS=8, TP=1, PP=1, OVERLAPs=True, FUSED_CE=True) with AC=sel, AC_LAYERS=1 => OOM (run 8)
- (MBS=4, TP=1, PP=1, OVERLAPs=True, FUSED_CE=True) with AC=none, AC_LAYERS=0 => OOM (run 14)
- (MBS=4, TP=1, PP=1, OVERLAPs=True, FUSED_CE=True) with AC=full, AC_LAYERS=0 => OOM (run 15)
- (MBS=2, TP=1, PP=1, OVERLAPs=True, FUSED_CE=True) with AC=full, AC_LAYERS=0, FIRST_PP_NUM_LAYERS=32 => prior "fail/unknown" (run 17) => avoid
- Duplicate check: any suggested full config must not exactly match any successful historical signature. We will only change one parameter while keeping others at current values to isolate impact.
Standard Bottleneck Tuning
- Observations: Large memory headroom (MemMax 54%) suggests AC is likely adding extra compute overhead; also history shows much higher MFU at MBS=2, TP=1, PP=1 with AC_LAYERS=0 and AC in full, none.
- Candidate list (prioritized by expected MFU gain while staying safe/novel):
1) Reduce AC_LAYERS from 2 -> 0 while keeping AC=full (minimizes recompute) and MBS=2 (safe). This is novel vs history because (AC=full, AC_LAYERS=0, MBS=2, TP=1, PP=1, overlaps=true, fused_ce=true) exists (runs 11/12/16) BUT those runs had AC_LAYERS=0 and match all other current params except AC_LAYERS; once we set AC_LAYERS=0 it WOULD duplicate those signatures (duplicate => reject).
2) Switch AC from full -> none while keeping AC_LAYERS=2 (AC_LAYERS becomes irrelevant but config signature changes). Must ensure not duplicating: history has (AC=none, AC_LAYERS=0, MBS=2, TP=1, PP=1, overlaps=true, fused_ce=true) (runs 13/18/19/20) but NOT with AC_LAYERS=2, so this is novel and should be safe given memory headroom.
3) Change MBS 2 -> 4 is disallowed because multiple MBS=4, TP=1, PP=1 OOM failures exist (runs 14/15/9), and we must not propose known-failure combos.
- Select first valid candidate: Change AC full -> none (novel, passes constraints, not a known failure, isolates AC impact with MBS/TP/PP unchanged).
Final verification:
- Proposed config differs from all historical runs (AC=none with AC_LAYERS=2 does not exist). - Does not match any known failure signature. - Keeps constraints satisfied (TP*PP=1, GBS%MBS=0).
The suggestions are:
Change AC from full to none.

D.3 Strategy Generation Agent

The Strategy Generation Agent $\mathcal{A}_g$ is responsible for synthesizing new training configurations by applying targeted modifications to the parameters identified as bottlenecks by the analysis stage.

Rather than generating configurations in an unconstrained manner, $\mathcal{A}_g$ operates under a set of explicit structural and system-level constraints to ensure validity and reproducibility. Concretely, the agent receives as input the current training configuration together with a ranked list of parameter modification suggestions. Based on these inputs, it performs controlled configuration updates while strictly preserving the original parameter schema, including parameter names, ordering, and formatting.

To prevent invalid or redundant configurations, $\mathcal{A}_g$ enforces multiple constraints during generation: (i) only a predefined subset of tunable parameters is allowed to be modified; (ii) the resulting configuration must not replicate any previously evaluated configuration in the historical performance table; and (iii) all unchanged parameters must be explicitly retained in the output.

This design allows $\mathcal{A}_g$ to systematically explore

the configuration space while maintaining compatibility with the underlying training system and avoiding failure-prone or duplicate configurations. For clarity, we present the prompt specification, input format, and output structure of the Strategy Generation Agent using structured blocks below.

Prompt of Strategy Generation Agent

Please update the configuration according to these guidelines:
Current Configuration:
{PARAMETERS}
Modification Suggestions:
{SUGGESTIONS}
Modification Rules:
1. Must keep same parameter names/count/format.
2. You can only modify these parameters: {PARAMETER_NAME}
3. Return both changed and unchanged parameters.
4. Must not replicate any complete parameter set in the table.
Reference Table:
{TABLE}
Example Response:
MICRO_BATCH_SIZE:4
AC:sel
PP:1

Input of Strategy Generation Agent

You are a reasoning assistant with the ability to modify the parameter configuration according to the user's requirements. Please answer the following question. You should think step by step to solve it.
Strictly provide your final answer in the format:
The modified parameters are:
YOUR_ANSWER
Question: Please update the configuration according to these guidelines:
Current Configuration:
MICRO_BATCH_SIZE: 2
TP: 1
PP: 1
OVERLAP_GRAD_REDUCE: true
OVERLAP_PARAM_GATHER: true
AC: full
AC_LAYERS: 2
FUSED_CROSS_ENTROPY: true
FIRST_PP_NUM_LAYERS: None
LAST_PP_NUM_LAYERS: None
Modification Suggestions:
Change AC from full to none.
Modification Rules:
1. Must keep same parameter names/count/format.
2. You can only modify these parameters:
MICRO_BATCH_SIZE,
TP,
PP,
OVERLAP_GRAD_REDUCE,
OVERLAP_PARAM_GATHER,
AC,
AC_LAYERS,
FUSED_CROSS_ENTROPY,
FIRST_PP_NUM_LAYERS,
LAST_PP_NUM_LAYERS
3. Return both changed and unchanged parameters.
4. Must not replicate any complete parameter set in the table.
Reference Table:
(Global Message Pool)
Example Response:
MICRO_BATCH_SIZE:4
AC:sel
PP:1

Output of Strategy Generation Agent

The modified parameters are:
MICRO_BATCH_SIZE: 2
TP: 1
PP: 1
OVERLAP_GRAD_REDUCE: true
OVERLAP_PARAM_GATHER: true
AC: none
AC_LAYERS: 2
FUSED_CROSS_ENTROPY: true
FIRST_PP_NUM_LAYERS: None
LAST_PP_NUM_LAYERS: None

D.4 Strategy Validation Agent

The Strategy Validation Agent $\mathcal{A}_v$ serves as the execution and feedback component of the multi-agent optimization loop. Its primary responsibility is to validate the feasibility and effectiveness of the training configurations generated by the Strategy Generation Agent under real system conditions.

Given a concrete execution context, including

model type, model size, world size, numerical precision, and the generated training configuration, $\mathcal{A}$v launches a training run and monitors its runtime behavior. Upon successful execution, the agent collects a comprehensive set of performance and resource utilization metrics, such as achieved TFLOPS, MFU, GPU utilization statistics, and peak memory consumption. These metrics, together with the corresponding configuration, are then recorded in the Global Message Pool ($\mathcal{M}$pool) as empirical evidence for subsequent analysis and strategy refinement.

In the event of a training failure, $\mathcal{A}$v captures the associated error logs and execution context, which are forwarded to the Failure Analysis Agent for root-cause diagnosis. By providing reliable execution feedback and failure signals, $\mathcal{A}$v effectively closes the optimization loop and enables iterative, data-driven improvement of training strategies.

Input of Strategy Validation Agent

1. MODEL_TYPE: qwen2.5 / llama2 / llama3 / gpt3 / qwen3-moe
2. MODEL_SIZE: 0.5B / 7B / 8B / A0.6B
3. WORLD_SIZE: 1 / 2 / 4 / ...
4. parameters(the output of Ag)
5. PR: fp16 / bf16

Output of Strategy Validation Agent

1. TFLOPS, MFU, MemAve, MemMax, GpuUtilAve, GpuUtilMax, Result (if success), update to Global Message Pool.
2. Get Error Log(if fail).

D.5 Failure Analysis Agent

The Failure Analysis Agent $\mathcal{A}_\text{f}$ is activated upon the detection of failed training executions and is responsible for identifying the most probable root cause of the failure. Its primary role is to transform unstructured runtime error logs and execution traces into structured diagnostic signals that can be utilized by subsequent optimization stages.

Given the error logs and the corresponding training configuration, $\mathcal{A}_\text{f}$ performs fault diagnosis by prioritizing the earliest observed error and analyzing whether the failure is attributable to improper parameter settings or to external system factors. Special attention is paid to timeout-related failures, which are treated as potential symptoms rather than definitive root causes and are further examined for deadlocks, resource starvation, network bottlenecks, and dependency failures.

To explicitly model diagnostic uncertainty, $\mathcal{A}_\text{f}$ associates each diagnosis with a confidence score, indicating the estimated reliability of the inferred root cause. High-confidence diagnoses are used to derive hard constraints for future configuration generation, while low-confidence diagnoses signal the need for further exploration or repeated validation. Both the diagnostic result and the associated confidence score are recorded in the Global Message Pool to inform subsequent iterations. For clarity, we present the prompt, input and output of $\mathcal{A}_\text{f}$ using structured blocks below.

Prompt of Failure Analysis Agent

Analyze this fault to determine if it is caused by improper parameter settings. It is necessary to analyze the underlying causes, especially timeout failures.
Logs:
{LOG}
Parameters:
{PARAMETERS}
Output rules:
1. [ERROR_SUMMARY]
2. Confidence: [0-100]
3. If Confidence <= 50: "[ERROR_SUMMARY]. Potential underlying issue (Confidence: X)"
4. If Confidence >50:
Parameter-related: "[PARAM(S)] [VALUE(S)] caused [ERROR_SUMMARY] (Confidence: X)"
Unrelated: "[ERROR_SUMMARY]. No parameter issue. RETRY (Confidence: X)"
5. For timeout errors, MUST check for:
- Deadlocks/race conditions
- Resource starvation (CPU/GPU/Memory)
- Network bottlenecks
- Dependency failures
6. Show exact confidence score
7. No explanations, pure diagnosis
8. Strict 30-word limit
Critical Notes:
- Prioritize the first error to occur
- Timeouts may be symptoms, not root causes
- Low confidence indicates need for deeper investigation
- Always suggest most probable root cause

Example Output:
1. "TypeError. Unknown error (Confidence: 10)"
2. "TP 2 and PP 2 caused Assertion error in parameter management (Confidence: 75)"
3. "Process hang. No parameter issue. RETRY (Confidence: 80)"

Input of Failure Analysis Agent

You are a reasoning assistant with the ability to locate the cause of the error from the logs. Please answer the following open-ended question. You should think step by step to solve it.
Strictly provide your final answer in the format: The conclusion is:
YOUR_ANSWER
Question:
Analyze this fault to determine if it is caused by improper parameter settings. It is necessary to analyze the underlying causes, especially timeout failures.
Logs:
{Logs}
Parameters:
MICRO_BATCH_SIZE: 4
TP: 1
PP: 1
OVERLAP_GRAD_REDUCE: true
OVERLAP_PARAM_GATHER: true
AC: none
AC_LAYERS: 0
FUSED_CROSS_ENTROPY: true
FIRST_PP_NUM_LAYERS: None
LAST_PP_NUM_LAYERS: None
Output rules:
1. [ERROR_SUMMARY]
2. Confidence: [0-100]
3. If Confidence <= 50:
"[ERROR_SUMMARY]. Potential underlying issue (Confidence: X)"
4. If Confidence >50:
Parameter-related: "[PARAM(S)] [VALUE(S)] caused [ERROR_SUMMARY] (Confidence: X)"
Unrelated: "[ERROR_SUMMARY]. No parameter issue. RETRY (Confidence: X)"
5. For timeout errors, MUST check for:
- Deadlocks/race conditions
- Resource starvation (CPU/GPU/Memory)
- Network bottlenecks
- Dependency failures
6. Show exact confidence score
7. No explanations, pure diagnosis
8. Strict 30-word limit
Critical Notes:
- Prioritize the first error to occur
- Timeouts may be symptoms, not root causes
- Low confidence indicates need for deeper investigation
- Always suggest most probable root cause

Output of Failure Analysis Agent

The conclusion is: MICRO_BATCH_SIZE 4 caused CUDA OutOfMemoryError during forward GEMM; each GPU had 53–65MiB free, indicating memory overcommit/resource starvation. Not a timeout/deadlock. (Confidence: 92)

D.6 Chain-of-Thought Prompting in Bottleneck Analysis Agent.

To enhance interpretability and decision-making within the agent $\mathcal{A}_{\text{a}}$, we incorporate CoT prompting, as shown in Figure 1 (right), to guide the generation of tuning suggestions from $\mathcal{M}_{\text{pool}}$. This prompting paradigm decomposes the reasoning process into structured steps, enabling the agent to interpret the meaning of configuration parameters, isolate their performance impact via controlled comparisons, and generate safe and impactful modifications. Besides, the structured prompting reduces the risk of arbitrary parameter changes and ensures the generated strategies remain both interpretable and effective.