AgentSlimming: Towards Efficient and Cost-Aware Multi-Agent Systems

Anonymous ACL submission

Abstract

Large Language Model-based Multi-Agent Systems (MAS) have demonstrated remarkable capabilities in complex tasks. However, manually designing optimal communication topologies is labor-intensive, while automated expansion methods often result in bloated structures with redundant agents, leading to excessive token consumption. To address this problem, we introduce **AgentSlimming**, a plug-and-play compression framework for graph-structured multi-agent workflows. Motivated by the AgentPruner and AgentQuant in neural networks, AgentSlimming compresses workflows by firstly estimiate the importance score of each agent with a hybrid mechanism, and then removing redundant agents or replacing them with low-cost ones, where each operation is then validated with a baseline-anchored acceptance rule to prevent performance collapse. Experiments show that AgentSlimming reduces average token cost by up to 78.9% with negligible performance degradation, and even somethimes improves accuracy, achieving a strong Pareto-optimal trade-off between cost and quality. *Our codes are available in the supplementary materials and will be released on Github.*

1 Introduction

The paradigm of Multi-Agent Systems (MAS) has shifted from manually crafting static prompts to orchestrating dynamic collaborations among specialized agents. Frameworks such as AutoGen (Wu et al., 2023) and ADAS (Hu et al., 2025) have shown that decomposing a complex problem into sub-tasks (Wei et al., 2022). Recent innovations, such as MetaGPT(Hong et al., 2024), AFlow (Zhang et al., 2025c), have revolutionized this field by reformulating workflow generation as a search problem. By leveraging Monte Carlo Tree Search (MCTS), AFlow can automatically navigate the vast space of agent interactions to discover highly effective reasoning topologies.

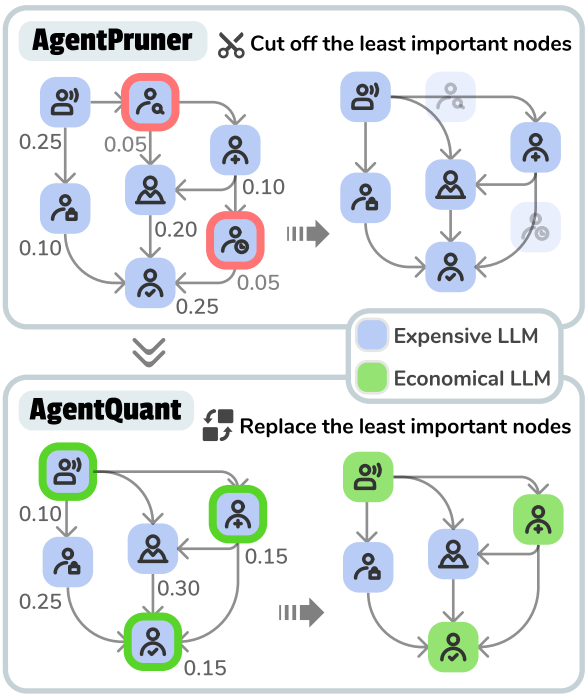

Figure 1: **Overview of the pruning and quantization concepts in AgentSlimming.** The process begins with an initially high-performance but computationally expensive workflow. Then, a *hybrid importance evaluation mechanism* is utilized to calculate each agent node's importance score (*i.e.,* the float number), which guides *pruning* and *quantization* in AgentSlimming.

However, this automated discovery process operates under an "unconstrained resource" assumption. The resulting workflows, while accurate, tend to be computationally bloated. They often feature dense, redundant connectivity and uniformly employ the most capable and expensive LLMs (e.g., GPT-4) as agent nodes for every sub-task (Li et al., 2023; Qian et al., 2024). This leads to two critical bottlenecks: **redundant communication**, where agents exchange repetitive or low-value information, and **excessive computational cost**, where simple sub-tasks consume high-value resources. As the number of agents and interaction turns grows, the quadratic increase in token consumption renders these systems difficult to scale.

To solve this problem, we propose AgentSlimming, a framework that can be directly applied to any MAS and workflows to reduce their execution costs. Motivated by the concepts of pruning, quantization, and finetuning in traditional neural network compression, AgentSlimming introduces AgentPruner, AgentQuant and AgentTuner as tools towards lightweight MAS. As shown in Figure 1, AgentPruner aims to identify the least important agent nodes in MAS and then remove them. In contrast, AgentQuant is designed as model substitution. When high-cost, high-precision models (e.g., GPT-4) are considered as not necessary for an unimportant role in MAS, they are dynamically replaced by cost-effective, lightweight surrogates (e.g., GPT-4o-mini) to reduce the inference costs. After AgentPruner and AgentQuant, AgentTuner is optionally employed to further optimize the MAS to recover the performance loss.

The proposition of AgentSlmming introduce a curcial problem for MAS compression, *i.e.,* how to determine the importance of each node in MAS. To tackle this challenge, we introduce a hybrid importance evaluation mechanism. Generally, a node's value is determined by both its *structural position* and *functional contribution*. Based on this observation, we consider the three following indicators: Degree Centrality (Freeman, 1978) and Betweenness Centrality (Brandes, 2001) for topological structure in the graph of MAS, and an Approximate Shapley value (Shapley, 1953; Lundberg and Lee, 2017) for functional contribution. Specifically, we adopt a Leave-One-Out (LOO) estimation strategy to calculate the marginal contribution of each node against the complete topology, serving as a computationally efficient proxy for the exact Shapley value. These rankings are fused via Reciprocal Rank Fusion (RRF) (Cormack et al., 2009) to generate a robust importance score.

Guided by this score, AgentSlimming employs an iterative greedy strategy: we progressively prune and quantize agent nodes starting from the least important ones. After each operation, we re-evaluate the workflow against a performance threshold and re-calculate the RRF scores, ensuring the optimization dynamically adapts to the changing graph structure until the efficiency limit is reached. While the pruned and quantized workflows typically achieve a superior Pareto frontier (Deb et al., 2002; Chen et al., 2024) between cost and accuracy, AgentSlimming further incorporates a MCTS fine-tuning mechanism that optimizes the streamlined graph rather than accepting the compressed policy as final. Here, we report the average execution cost in USD per problem. Empirically, the resulting workflows are consistently more cost-efficient than AFlow, often achieving strictly superior performance in both accuracy and cost. Reporting the average execution cost per problem (USD), AgentSlimming demonstrates significant gains across diverse benchmarks.

For instance, on GSM8K, AgentSlimming matches AFlow's score (95.5) while reducing the cost by **78.8%**. On code generation and complex reasoning tasks like MBPP and LiveCode, AgentSlimming achieves a "dual victory": it improves accuracy while reducing costs by **71.7%** and **78.9%**, respectively. Even in cases where AFlow retains a marginal score advantage (e.g., HotpotQA: 77.3 vs. 77.0), AgentSlimming achieves this at a substantially lower cost (a 27.4% reduction), indicating that our pipeline reliably locates the workflow at a more favorable operating point on the cost–quality Pareto frontier.
Our contributions can be summarized as follows:

- We introduce AgentSlimming, a training-free framework that integrates automated agentic workflow exploration with a novel pruning and semantic quantization pipeline.
- We first propose an iterative multi-metric optimization algorithm that integrates topological and functional importance via RRF to accurately identify redundant components.
- Extensive evaluations on eight benchmarks show that AgentSlimming achieves a striking reduction in token costs of up to **78.9%**.

2 Related Work

2.1 Agentic Workflows

LLM-based systems can be broadly characterized into two paradigms: *agentic workflows* and *autonomous agents*. The former executes tasks through predefined multi-step pipelines with repeated LLM invocations, whereas the latter emphasizes dynamic decision-making and planning under interaction and feedback. Recent work has made notable progress in language-driven decomposition and collaboration (Zhuge et al., 2025), data-science agents (Hong et al., 2025), tool-enabled mobile agent teams (Zhang et al., 2024), and open-ended embodied exploration (Wang et al., 2024a).

Compared to autonomous agents that often require environment-specific action spaces and decision patterns, workflows can more easily incorporate human domain expertise and improve through iterative refinement, making them particularly amenable to automated construction and optimization.

2.2 Multi-Agent Evolving

Many effective workflows are still developed primarily through manual discovery and engineering practice. At the general level, common transferable reasoning recipes include chain-of-thought, self-consistency, self-refine and structured self-collaboration (Wei et al., 2022; Wang et al., 2023; Madaan et al., 2023b; Wang et al., 2024b). At the domain level, multi-step procedures are specialized in reusable pipelines, such as code generation and debugging (Hong et al., 2024; Ridnik et al., 2024; Zhong et al., 2024), data analysis and visualization (Xie et al., 2024; Ye et al., 2024; Li et al., 2024a; Zhou et al., 2023), mathematical reasoning (Xu et al., 2024), and planning-style search for question answering (Nori et al., 2023; Zhou et al., 2024).

However, manual design cannot cover the combinatorial space across domains, which motivates automated agentic optimization. One line of work optimizes local instructions or components within a fixed backbone (Fernando et al., 2024; Yüksekgönül et al., 2024; Yang et al., 2024; Khattab et al., 2024) or tunes inference-time strategies (Saad-Falcon et al., 2024). Another line goes further by optimizing the end-to-end workflow structure, including automatic generation of code-represented workflows (Li et al., 2024b), viewing agent systems as optimizable graphs (Zhuge et al., 2024), and improving system designs in code space via meta-agents (Hu et al., 2025). Zhang et al. (2025c) similarly adopts code-based representations, combining finer-grained abstractions (named nodes/operators) with MCTS to leverage execution feedback and tree-structured experience for efficient workflow structure search.

2.3 Graph-Structured Orchestration, Pruning, and Cost-Aware Compression

Beyond prompt engineering, many agentic systems are best viewed as executable graphs, where nodes represent specialized modules and edges encode information flow. Recent studies focus on topology learning and pruning for improved efficiency and robustness (Zhang et al., 2025a,b; Boyi et al., 2025). These approaches also explore pruning at different granularities, including message-level pruning under bandwidth constraints (Mao et al., 2020), dynamic agent elimination for token efficiency (Wang et al., 2025), and progressive pruning that blends heuristics with execution experience (Zhang et al., 2025d). Existing methods typically starting from (near) fully connected interactions and primarily pruning edges or communication channels. Action selection for pruning and replacement can draw on heuristic importance measures from network analysis (Freeman, 1978) or contribution-based formulations such as Shapley values (Shapley, 1953), which usually require approximation for tractability. In addition, model compression and quantization reduce inference cost while preserving quality (Frantar et al., 2022; Xiao et al., 2023; Lin et al., 2024), complementing structural optimization at the workflow level.

2.4 Search-Based Optimization for Workflows

Search provides a natural mechanism for exploring large workflow design spaces. MCTS (Coulom, 2006) and its variant UCT (Kocsis and Szepesvári, 2006) enables effective exploration through sampled evaluation and progressive expansion, and has been applied to planning and reasoning in language agents (Zhou et al., 2024) as well as workflow structure optimization (Zhang et al., 2025c). Following this trajectory, we focus on optimizing DAG workflows under dependency constraints, integrating structure search with node-level pruning and cost-driven node replacement (semantic quantization).

3 Methodology

3.1 Problem Formulation

We formulate the optimization of multi-agent systems as a discrete structural search problem over a directed graph, with an explicit trade-off between task performance and execution cost.

Workflow as a directed graph. A multi-agent workflow is represented as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ denotes an executable operator (e.g., an LLM call), and each edge $e \in \mathcal{E}$ specifies an information dependency. Given a validation set $\mathcal{D}_{\text{val}}$, executing $\mathcal{G}$ yields an average task score $S(\mathcal{G})$ and an average execution cost $C(\mathcal{G})$:

$$S(\mathcal{G}) = \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{x \in \mathcal{D}_{\text{val}}} \text{score}(\mathcal{G}, x),$$
$$C(\mathcal{G}) = \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{x \in \mathcal{D}_{\text{val}}} \text{cost}(\mathcal{G}, x). \quad (1)$$

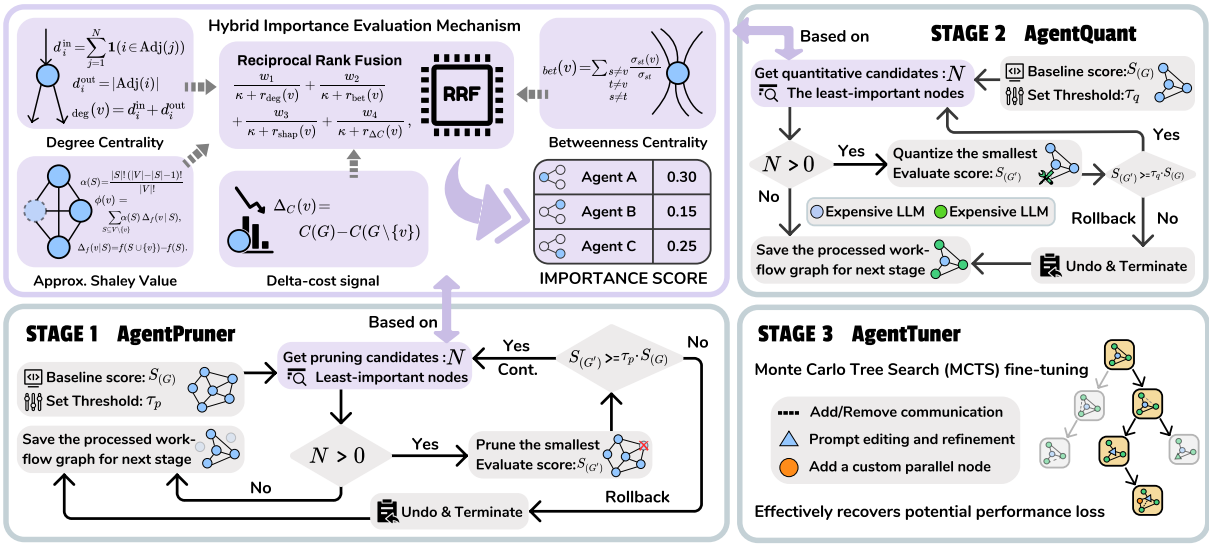

Figure 2: **Illustration of AgentSlimming.** To identify redundancy, AgentSlimming computes three distinct rankings of each agent node: *Degree Centrality*, *Betweenness Centrality*, *Costs Comparison* and *Approximate Shapley value*. Both pruning and quantization select candidate nodes based on the computed importance scores, ranking nodes from least to most important and optimizing the lowest-ranked candidate first. After each operation, a re-evaluation is performed. If the score drops below an acceptable threshold, a rollback mechanism is triggered to revoke the current operation and immediately terminate this phase, ensuring preserving superior performance.

Optimization Objective. Our goal is to maximize performance under a cost budget B. When multiple topologies achieve comparable performance, we prioritize the one with the minimal cost:

$$\max_{\mathcal{G}} \; S(\mathcal{G}) \quad \text{s.t.} \quad C(\mathcal{G}) \leq B. \tag{2}$$

3.2 The AgentSlimming Pipeline

As illustrated in Figure 2, our framework operates through a three-stage pipeline: *AgentPruner (Structural Pruning)*, *AgentQuant (Semantic Quantization)*, and *AgentTuner (MCTS Fine-tuning)*. This pipeline progressively compresses the workflow while maintaining its reasoning capabilities.

3.2.1 Hybrid Importance Evaluation

To guide both pruning and quantization, we introduce a hybrid importance evaluation mechanism. For any node $v \in \mathcal{V}$, we compute four complementary signals using a small probe dataset $\mathcal{D}_{\text{probe}} \subset \mathcal{D}_{\text{val}}$.

1. Degree and Betweenness-Based Signals (Topological Priors). We capture the structural significance of a node using graph centrality metrics.

- **Degree-based Signal (s_{deg}):** Derived from the node's local connectivity (in-degree and out-degree). Nodes with weaker structural connectivity are assigned larger pruning likelihood.
- **Betweenness Centrality (s_{bet}):** Measures the node's role as an information bridge. It is calculated as:

$$s_{\text{bet}}(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{3}$$

where σ_{st} is the total number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of those paths passing through v.

2. Approximate Shapley value signal (Functional Contribution). To quantify the semantic contribution of node v, we adopt the Shapley value concept from cooperative game theory. Since exact computation is NP-hard, we employ a efficient **Leave-One-Out (LOO)** approximation. The marginal contribution $\hat{\phi}(v)$ is estimated by the performance drop when v is removed (or replaced):

$$\hat{\phi}(v) \approx \Delta_S(v) = S(\mathcal{G}) - S(\mathcal{G} \setminus \{v\}), \tag{4}$$

where $\mathcal{G} \setminus \{v\}$ denotes the workflow after removing node v (and patching connections). A smaller $\hat{\phi}(v)$ indicates a lower contribution to task completion.

3. Delta-Cost signal (Economic Potential). We estimate the potential cost saving $\Delta_C(v)$ on $\mathcal{D}_{\text{probe}}$:

$$\Delta_C(v) = C(\mathcal{G}) - C(\mathcal{G} \setminus \{v\}). \tag{5}$$

Nodes with higher $\Delta_C(v)$ are prioritized for pruning or quantization to maximize efficiency gains.

Rank Fusion. To robustly combine these heterogeneous signals, we employ Reciprocal Rank Fusion (RRF). Let $r_m(v)$ be the rank of node v under metric $m \in \{\deg, \text{bet}, \text{shap}, \Delta C\}$. The fused score is calculated as:

$$\text{RRF}(v) = \sum_m \frac{w_m}{\kappa + r_m(v)}, \tag{6}$$

where κ is a smoothing constant (set to $\max\{10, |\mathcal{V}|\}$) and $\mathbf{w}$ balances the metrics. This fused score identifies nodes that are simultaneously structurally peripheral, functionally redundant, and computationally expensive.

3.2.2 Iterative Optimization Process

Stage 1: AgentPruner (Structural Pruning) Let $\mathcal{G}_{\text{base}}$ denote the initial high-performance workflow graph . We employ an iterative greedy strategy to prune nodes. In each iteration, we rank nodes by $\text{RRF}(v)$ and evaluate the top candidates. A pruning operation is accepted if the performance degradation is within a tolerance threshold τ_p:

$$S(\mathcal{G}') \geq \tau_p \cdot S(\mathcal{G}_{\text{base}}), \tag{7}$$

where $\mathcal{G}'$ is the pruned graph and $\tau_p \in (0, 1)$ is a predefined threshold. We evaluate the Top-1 candidate first and then the Top-k candidate if necessary. If neither candidate is accepted, the stage terminates and the workflow rolls back to the last accepted state. **Graph Surgery:** To preserve executability after removing node v, we perform edge patching to maintain the graph's connectivity. For every pair of predecessor $s \in \text{In}(v)$ and successor $t \in \text{Out}(v)$, we add a direct edge $(s \to t)$ if one does not already exist, self-loops and duplicate edges are disallowed. Each accepted pruning step is logged with the probe signals, fused ranks, and the resulting workflow artifact.

Stage 2: AgentQuant (Semantic Quantization) Let $\mathcal{G}_{\text{base}}$ now refer to the pruned workflow graph obtained from Stage 1. The quantization process operates on this sparse structure to further reduce computational costs. We define *Semantic Quantization* as a model substitution strategy. For a selected node v, we replace its high-cost LLM with a cost-effective surrogate $\pi(v)$, yielding a quantized graph $\mathcal{G}^{(v)}$. Candidate ranking follows the same RRF mechanism, where the Shapley signal is instantiated by evaluating the performance impact of the model substitution. We also employ the greedy strategy to quantize nodes, a quantization operation is accepted if the performance degradation is within a tolerance threshold τ_q:

$$S(\mathcal{G}') \geq \tau_q \cdot S(\mathcal{G}_{\text{base}}), \tag{8}$$

where $\mathcal{G}'$ is the quantized graph and $\tau_q \in (0, 1)$ is a predefined threshold.This step drastically reduces token costs for non-critical reasoning steps.

Stage 3: AgentTuner (Adaptive MCTS Fine-tuning) To address the variance in compression sensitivity across different tasks, we introduce an adaptive MCTS exploration on the compressed workflow. This phase is selectively applied and focuses on localized refinements, such as prompt adaptation and parameter tuning, to recover from potential degradation. Critically, this fine-tuned configuration is adopted only if it improves upon the initial compressed version, thereby ensuring constant and robust performance gains.

4 Experiments

4.1 Setup

Overview. We evaluate our method on eight public benchmarks. To ensure a fair comparison, we strictly adhere to AFlow's configuration, adopting its datasets, splits, and sampling methods where applicable. Additionally, we incorporate three challenging benchmarks to test robustness.

Datasets. Our evaluation suite consists of two categories: (1) Standard Benchmarks: We utilize the complete AFlow (Zhang et al., 2025c) suite, including GSM8K (Cobbe et al., 2021), and MBPP (Austin et al., 2021) (full sets), as well as HotpotQA (Yang et al., 2018) and DROP (Dua et al., 2019) (randomly sampled 1,000-instance subsets). For MATH (Hendrycks et al., 2021), we follow the specific subsetting of 617 Level-5 problems across four categories. These datasets employ a 1:4 validation/test split. (2) High-Difficulty Benchmarks: To assess performance on complex reasoning and coding tasks, we incorporate AIME (Mathematical Association of America; Art of Problem Solving) , MuSiQueAns (Trivedi et al., 2022), and LiveCode (Jain et al., 2025). These datasets are partitioned using a 3:7 validation/test split.

Baselines. We compare AgentSlimming against a diverse set of baselines: *(I) Manual Prompting Strategies:* Standard Chain-of-Thought (CoT) (Wei et al., 2022), Self-Consistency (SC-CoT)

Method	MATH		GSM8K		HotpotQA		MBPP		DROP	
	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)
Ours	73.9	6.88e-3 ↓44.2%	**95.5**	9.30e-4 ↓78.8%	77.0	5.55e-3 ↓27.4%	**77.9**	7.30e-4 ↓71.7%	**81.7**	9.50e-4 ↓38.7%
AFlow	**74.8**	1.20e-2	**95.5**	4.38e-3	**77.3**	7.64e-3	73.3	2.58e-3	78.7	1.55e-3
LLM-Debate	74.5	1.47e-2	93.5	3.57e-3	73.4	9.47e-3	75.1	6.22e-3	75.6	2.94e-3
ADAS	69.0	1.15e-2	94.9	4.79e-3	65.4	4.72e-2	75.4	4.31e-3	78.6	3.07e-3
CoT	62.9	6.03e-4	90.5	2.48e-4	57.5	7.11e-4	73.6	2.80e-4	75.9	3.75e-4
SC-CoT	65.3	3.67e-3	90.2	1.71e-3	61.7	4.48e-3	75.6	2.12e-3	75.3	2.37e-3
Self-Refine	65.5	1.01e-3	89.8	5.60e-4	57.9	1.50e-3	74.5	5.56e-4	75.0	7.91e-4

Table 1: **Performance and inference cost comparison across standard benchmarks.** Cost denotes the average API expense ($/prob.). **Bold** indicates the best results among **workflow-based methods** (excluding simple prompting baselines). Green percentages indicate the relative cost reduction compared to AFlow, highlighting how much inference budget is saved while maintaining competitive accuracy.

Method	AIME		LiveCode		MusiqueAns	
	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)
Ours	65.7	1.35e-2 ↓19.2%	61.7	2.47e-3 ↓78.9%	**89.3**	8.24e-3 ↓23.2%
AFlow	**67.1**	1.67e-2	55.3	1.17e-2	84.5	1.07e-2
LLM-Debate	56.8	2.86e-2	**67.8**	1.67e-2	45.3	1.62e-2
Cost-aware AFlow	61.8	1.20e-2	57.9	8.41e-3	83.6	1.04e-2
ADAS	62.3	1.42e-2	52.1	9.01e-3	77.2	1.62e-2
CoT	53.4	2.99e-3	46.4	1.25e-3	73.9	1.11e-3
Self-Consistency CoT	59.7	1.41e-2	47.2	7.93e-3	74.9	6.89e-3
Self-Refine	54.1	4.08e-3	49.1	2.30e-3	72.8	2.30e-3

Table 2: **Performance and inference cost comparison across high-difficulty benchmarks.** Cost denotes the average API expense ($/prob.). **Bold** indicates the best results among **workflow-based methods** (excluding simple prompting baselines). Green percentages indicate the relative cost reduction compared to AFlow.

(Wang et al., 2023), and Self-Refine (Madaan et al., 2023a). *(II) Automated Workflow Optimization:* ADAS (Hu et al., 2025) and AFlow (Zhang et al., 2025c). *(III) Cost-Aware AFlow:* We implement a modified AFlow variant. This baseline explicitly integrates cost comparisons into its selection mechanism during search phase, directly competing on cost efficiency.

Details. We employ GPT-4.1-mini (OpenAI, 2025) as the high-precision model and GPT-4.1-nano (OpenAI, 2025) as the cost-effective surrogate for quantization, both accessed via OpenAI API. For workflow-based optimization, we use GPT-4.1-mini to maintain computational efficiency. All models are accessed with temperature set to 0 to ensure reproducibility. Full experimental details are provided in Appendix B.

4.2 Main Results

Superior Cost-Performance Efficiency. The main experimental results are summarized in Table 1 and Table 2. Overall, AgentSlimming consistently discovers workflows that are more cost-efficient than AFlow. In many cases, our method achieves *strict dominance*, surpassing the baseline in both accuracy and execution cost. Here, we report the average cost in USD per problem.

On standard benchmarks, AgentSlimming maintains competitive performance while significantly reducing computational overhead. For instance, on GSM8K, it matches AFlow's accuracy (95.5) but reduces the average cost from 4.38×10^{-3} to 9.30×10^{-4}, a **78.8%** cost reduction. On MBPP, AgentSlimming improves the score from 73.3 to 77.9 while simultaneously lowering the cost from 2.58×10^{-3} to 7.30×10^{-4} (**71.7%** reduction). On datasets where AFlow attains a slightly higher raw score, such as HotpotQA , AgentSlimming achieves comparable performance at a substantially lower cost, reducing the average cost by 27.4%.

The advantage of AgentSlimming is even more pronounced on high-difficulty benchmarks. On LiveCode, it increases the score from 55.3 to 61.7 while reducing cost by nearly an order of magnitude (from 1.17×10^{-2} to 2.47×10^{-3}, a **78.9%** reduction). Similarly, on MuSiQueAns, AgentSlim-

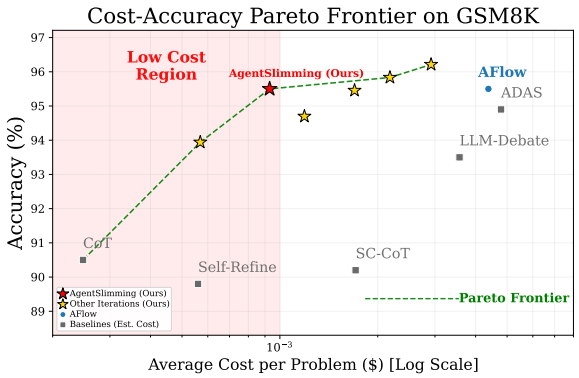

Figure 3: **Cost-Accuracy Pareto Frontier on the GSM8K dataset.** The x-axis represents the average inference cost per problem (USD), and the y-axis denotes accuracy. Data points for our method correspond to varying iteration rounds. These configurations consistently align with the Pareto frontier, demonstrating optimal cost-quality trade-offs compared to baselines.

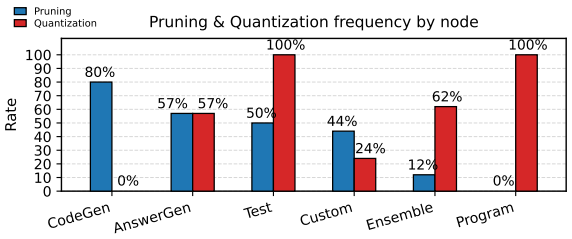

Figure 4: **Node-type specific compression rates.** We report the ratio of nodes processed by each compression strategy across six categories. Blue bars indicate structural removal (Pruning), while red bars indicate model substitution (Quantization). The variance across categories highlights the adaptive nature of our hybrid importance evaluation.

ming improves accuracy from 84.5 to 89.3 with a **23.2%** reduction. These results indicate that AgentSlimming reliably positions the discovered workflows at a more favorable operating points on the cost–quality Pareto frontier, particularly for complex reasoning and programming tasks.

4.3 Analysis

We conducted a comprehensive analysis to interpret the efficiency sources of AgentSlimming and validate its practical feasibility.

Denoising via Pruning In the initial heavy workflows, we observed redundant parallel generation components, particularly multiple `CodeGen` nodes and `AnswerGen` nodes. Our method consistently deprioritizes these redundant nodes due to their limited marginal benefit, pruning them to yield a simplified graph structure.

Strategic Heterogeneity When parallel structures remain after pruning, AgentSlimming applies non-uniform compression and retains a mixed-precision configuration. Specifically, it preserves a small number of high-capability `AnswerGen` nodes at full precision and quantizes the remaining parallel auxiliary nodes to lower-cost models.

Cost-Effective Arbitration via Functional Alignment The quantization frequencies indicate that AgentSlimming primarily compresses nodes whose functionality is procedure-oriented rather than generation-intensive. `ScEnsemble` and `Program` nodes are infrequently pruned but are frequently quantized, while `Test` nodes are consistently quantized. Although these nodes can be topologically central, they mainly implement selection, consensus aggregation, and execution-based validation. By selectively quantizing them while retaining high-capability generators, AgentSlimming preserves accuracy and strategically directs the computational budget toward core reasoning rather than auxiliary validation.

Total Evaluation Budget Analysis. To translate the structural efficiency gains into tangible economic implications, we estimate the total end-to-end evaluation budget for the test sets of each benchmark by multiplying the optimized per-problem cost by the effective sample size. Remarkably, the computational burden is minimal: standard benchmarks like MBPP and GSM8K require less than **$2.00** to evaluate the entire test set ($1.63 and $1.84, respectively). Even for computationally intensive, high-difficulty benchmarks such as AIME and HotpotQA, the total costs remain strictly affordable at $26.06 and $14.76.

Crucially, this cost-effectiveness extends to the optimization process itself. As a fully training-free framework, AgentSlimming avoids the prohibitive overhead of gradient-based updates and expensive parameter tuning. Furthermore, by leveraging topological priors (e.g., degree and betweenness centrality) for importance scoring, we ensure that the pruning and quantization decisions remain computationally efficient and lightweight.

4.4 Ablation Study

We investigate the impact of the node-importance ranking strategy used in pruning and quantization. Our full method employs a weighted Reciprocal

Method	DROP		MATH		MBPP	
	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)	Acc.	Cost ($/prob.)
Degree-only	66.5 ↓15.2%	3.2e-4 ↓28.9%	73.1 ↓2.3%	5.28e-3 ↑107.9%	73.3 ↓8.6%	1.56e-3 ↑119.7%
Betweenness-only	77.8 ↓0.8%	4.5e-4 ↓0.0%	74.0 ↓1.1%	4.17e-3 ↑64.2%	79.1 ↓1.4%	7.1e-4 ↓0.0%
Shapley-only	65.7 ↓16.2%	3.1e-4 ↓31.1%	71.4 ↓4.5%	5.25e-3 ↑106.7%	74.4 ↓7.2%	1.50e-3 ↑111.3%
RRF	**78.4**	4.5e-4	**74.8**	2.54e-3	**80.2**	7.1e-4

Table 3: **Ablation study of the importance ranking strategies.** We compare topology-based signals (degree-only, betweenness-only) and a functional signal (Shapley-only), as well as their fusion via Reciprocal Rank Fusion (RRF), on DROP, MATH, and MBPP. Results report task accuracy (Acc) and average inference cost per problem . **Bold** denotes the best accuracy on each dataset, while underline highlights the lowest average cost.

Rank Fusion (RRF) scheme to synthesize multiple signals into a unified candidate ranking. In contrast, we evaluate ablated variants that rely on a single metric for ranking: (i) Degree-only, (ii) Betweenness-only, and (iii) Shapley-only. Experimental settings are consistent across stages: for pruning, all variants initialize from the original workflow. For quantization, each variant inherits the best pruned graph derived from its corresponding pruning strategy and applies the same metric for node selection. Both stages maintain identical validation subset sizes and baseline-anchored acceptance thresholds.This design isolates the effect of ranking quality from confounders such as search budget, initialization, and stopping criteria. Table 3 details the specific performance metrics recorded after both stages.

We observe that single-metric strategies exhibit divergent cost–quality trade-offs across tasks. These complementary behaviors further underscore the necessity of multi-signal fusion via RRF, which consistently achieves a more robust balance between effectiveness and efficiency.

4.5 Case Study

We analyze the pruning process of the workflow on MATH to illustrate structural simplification. Starting from a complex graph with parallel reasoning paths and refinement steps, AgentSlimming identified redundancy in the dual `Answer Generate` nodes and the marginal utility of the `Code Refine` node. Concretely, probe-based importance ranking suggests that one generator path consistently dominates in contribution, while refinement rarely corrects errors relative to its token cost.Notably, pruning the parallel path diminished the utility of the downstream `ScEnsemble` node, leading to its removal. The transformation of this stage is detailed in Figure 5. The result is a streamlined pipeline

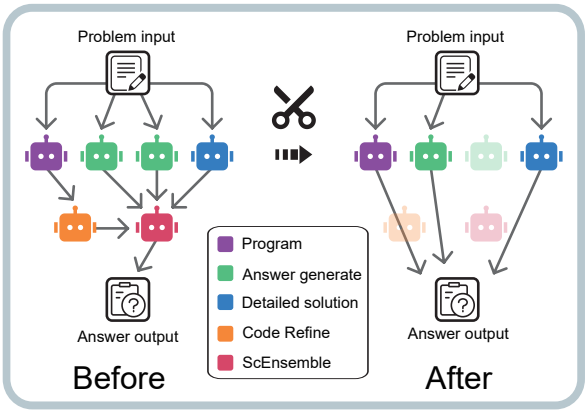

Figure 5: **Visualizing the pruning process of MATH workflow.** (*Before*) The initial workflow features complex parallel reasoning paths and an ensemble mechanism. (*After*) The streamlined workflow after AgentSlimming's pruning stage. It effectively identifies the core reasoning backbone, pruning redundant parallel paths and refinement steps to reduce cost.

that preserves core reasoning logic while achieving substantial cost reductions, empirically validating our topo-functional optimization strategy. The complete details of this case study is provided in Appendix C.

5 Conclusion

We introduced AgentSlimming, a unified framework that optimizes multi-agent workflows via structural pruning and semantic quantization. Guided by a novel hybrid metric combining topological and functional signals, AgentSlimming achieves up to **79.8%** cost reduction across diverse benchmarks. Importantly, this efficiency is attained with negligible impact on reasoning quality, matching or even exceeding their performance. Our method shows that sparse, heterogeneous topologies can effectively replace computationally redundant dense multi-agent systems, establishing a new Pareto frontier for scalable agentic collaboration.

Limitations

We only evaluate GPT-4.1-series models, and do not benchmark newer frontier models such as OpenAI GPT-5.2, Anthropic Claude 4.5, Google Gemini 3 Pro, Meta Llama 4.

Ethics Statement

We acknowledge that all authors are informed about and adhere to the ACL ARR Code of Ethics and the Code of Conduct.

Risks Our benchmarks are sourced from publicly available datasets. We cannot guarantee that they are free of socially harmful, biased, or toxic content. In addition, AgentSlimming optimizes cost by pruning and replacing nodes, which may alter the behavior of the original workflow in unexpected ways; when applied to safety-critical domains, such changes could amplify errors or reduce reliability. We use ChatGPT to correct grammatical errors in this paper.

References

Art of Problem Solving. Aime problems and solutions. https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Li Boyi, Zhonghan Zhao, Der-Horng Lee, and Gaoang Wang. 2025. Adaptive graph pruning for multi-agent communication. *CoRR*, abs/2506.02951.

Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*.

Lingjiao Chen, Matei Zaharia, and James Zou. 2024. Frugalgpt: How to use large language models while reducing cost and improving performance. *Trans. Mach. Learn. Res.*

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, Boston, MA, USA, July 19-23, 2009*.

Rémi Coulom. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer.

Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 2368–2378. Association for Computational Linguistics.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323.

Linton C. Freeman. 1978. Centrality in social networks conceptual clarification. *Social Networks*, 1.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Sirui Hong, Yizhang Lin, Bang Liu, and 1 others. 2025. Data interpreter: An LLM agent for data science. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 19796–19821. Association for Computational Linguistics.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, and 1 others. 2024. Metagpt: Meta programming for A multi-agent collaborative framework.

Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, and 1 others. 2024. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to SQL: are we fully ready? [experiment, analysis & benchmark]. *Proc. VLDB Endow.*, 17(11):3318–3331.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Zelong Li, Shuyuan Xu, Kai Mei, and 1 others. 2024b. Autoflow: Automated workflow generation for large language model agents. *CoRR*, abs/2407.12821.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Guangxuan Xiao, and Song Han. 2024. AWQ: activation-aware weight quantization for on-device LLM compression and acceleration. *GetMobile Mob. Comput. Commun.*, 28(4):12–17.

Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, and 1 others. 2023a. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Aman Madaan, Niket Tandon, Prakhar Gupta, and 1 others. 2023b. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Hangyu Mao, Zhengchao Zhang, Zhen Xiao, Zhibo Gong, and Yan Ni. 2020. Learning agent communication under limited bandwidth by message pruning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5142–5149. AAAI Press.

Mathematical Association of America. Maa invitational competitions: American invitational mathematics examination (aime). `https://maa.org/maa-invitational-competitions/`.

Harsha Nori, Yin Tat Lee, Sheng Zhang, and 1 others. 2023. Can generalist foundation models outcompete special-purpose tuning? case study in medicine. *CoRR*, abs/2311.16452.

OpenAI. 2025. Introducing gpt-4.1 in the api: A new series of gpt models featuring major improvements on coding, instruction following, and long context—plus our first-ever nano model.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *CoRR*, abs/2401.08500.

Jon Saad-Falcon and 1 others. 2024. Archon: An architecture search framework for inference-time techniques. *CoRR*, abs/2409.15254.

Lloyd S. Shapley. 1953. A value for n-person games. In *Contributions to the Theory of Games II*, pages 307–317. Princeton University Press.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. 9835 musique: Multi-hop questions via single-hop question composition. *Trans. Assoc. Comput. Linguistics*, 10:539–554.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, and 1 others. 2024b. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-personal self-collaboration. In *Proceedings of NAACL-HLT*.

Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. 2025. Agentdropout: Dynamic agent elimination for token-efficient and high-performance LLM-based multi-agent collaboration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24013–24035. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Peng, and 1 others. 2023. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155.

Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR.

Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. Haichart: Human and AI paired visualization system. *Proc. VLDB Endow.*, 17(11):3178–3191.

Yiheng Xu and 1 others. 2024. Lemur: Harmonizing natural language and code for language agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 2369–2380. Association for Computational Linguistics.

Yilin Ye, Jianing Hao, Yihan Hou, Zhan Wang, Shishi Xiao, Yuyu Luo, and Wei Zeng. 2024. Generative AI for visualization: State of the art and future directions. *Vis. Informatics*, 8(1):43–66.

Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic "differentiation" via text. *CoRR*, abs/2406.07496.

Guibin Zhang, Yanwei Yue, Zhixun Li, and 1 others. 2025a. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, and 1 others. 2025b. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025c. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*.

Jiayi Zhang, Chuang Zhao, Yihan Zhao, Zhaoyang Yu, Ming He, and Jianping Fan. 2024. Mobileexperts: A dynamic tool-enabled agent team in mobile devices. *CoRR*, abs/2407.03913.

Ruijia Zhang, Xinyan Zhao, Ruixiang Wang, Sigen Chen, Guibin Zhang, An Zhang, Kun Wang, and Qingsong Wen. 2025d. Safesieve: From heuristics to experience in progressive pruning for llm-based multi-agent communication. *CoRR*, abs/2508.11733.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. pages 851–870.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. 2023. LLM as DBA. *CoRR*, abs/2308.05481.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, and 1 others. 2025. Mindstorms in natural language-based societies of mind. *Comput. Vis. Media*, 11(1):29–81.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Appendix

A Algorithm

Algorithm 1 Prune–Quantize–Fine-tune Pipeline

Input: initial graph G_0, validation set D_{val}, probe size m, budget B.
  Top-k size k, thresholds τ, τ_q.
  RRF weights $\mathbf{w}$, low-cost mapping π.
Output: optimized graph $G^\star$.
Step 1: AgentPruner.
Set baseline $S_0 \leftarrow S(G)$.
while pruning budget remains **do**
  Rank nodes by probe signals; fuse by weighted RRF; get Top-k $\{v_1, v_2, ..., vk\}$.
  for $v \in \{v_1, v_2, ..., vk\}$ **do**
    $G' \leftarrow \text{Surgery}(G, v)$; evaluate $S(G')$.
    if $S(G') \geq \tau \cdot S_0$ **then**
      $G \leftarrow G'$; **break**
    end if
  end for
  Stop if neither candidate is accepted.
end while
Step 2: AgentQuant.
Set baseline $S_0 \leftarrow S(G)$.
while quantization budget remains **do**
  Rank LLM nodes by probe replacement signals; fuse by weighted RRF; get Top-k $\{v_1, v_2, ..., vk\}$.
  for $v \in \{v_1, v_2, ..., vk\}$ **do**
    $G^{(v)} \leftarrow \text{Replace}(G, v, \pi(v))$; evaluate $S(G^{(v)})$.
    if $S(G^{(v)}) \geq \tau_q \cdot S_0$ **then**
      $G \leftarrow G^{(v)}$; **break**
    end if
  end for
  Stop if neither candidate is accepted.
end while
Step 3: AgentTuner.
Run MCTS on G with atomic edits; update G.
return $G^\star \leftarrow \arg\max_{H \in \mathcal{K}}(S(H), -C(H))$.
  s.t. $C(G^\star) \leq B$.

B Experimental Details

We employ different models for workflow optimization and execution. For execution, we primarily use a two-tier configuration with GPT-4.1-mini (OpenAI, 2025) and GPT-4.1-nano (OpenAI, 2025). Specifically, we employ GPT-4.1-mini by default during the pruning stage, while adopting the lightweight GPT-4.1-nano for quantized nodes to minimize inference costs. To ensure a fair comparison, GPT-4.1-mini serves as the optimizer for both our method and all workflow-based baselines (AFlow (Zhang et al., 2025c), ADAS (Hu et al., 2025)). All models are accessed via APIs with a temperature of 0. We set the iteration rounds to 10 for AFlow and 15 for ADAS. Notably, as iteration rounds increase, workflows become more complex—a regime in which our method demonstrates superior cost reduction.

C Case Study

C.1 Workflow Overview and Legend Mapping

Figure 5 visualizes the pruning process on the MATH (Hendrycks et al., 2021) workflow. To match the legend used in the figure, we map the implementation node IDs to the paper terms as follows: (1) `GenerateSolutionA/GenerateSolutionB` correspond to the dual `Answer Generate` nodes. (2) `RefineWithCode` corresponds to the `Code Refine` node. (3) `ScEnsembler` corresponds to the `ScEnsemble` aggregation node. (4) `AnswerFormatter` formats the final output to satisfy benchmark constraints.

C.2 Pruning Rationale and Structural Simplification

The original workflow (Figure 5, left) contains multiple parallel reasoning paths: a program-execution path (`Programmer` → `RefineWithCode`), a detailed chain-of-thought path (`DetailedSolution`), and two additional solution generation paths (`GenerateSolutionA` and `GenerateSolutionB`). All candidate answers are routed into `ScEnsembler` for self-consistency selection before being formatted by `AnswerFormatter`.

AgentSlimming identifies two main sources of redundancy: (1) The two `Answer Generate` branches are highly overlapping; keeping one branch provides most of the diversity gain. (2) The marginal utility of the `Code Refine` step is limited for this workflow; its benefits do not justify the extra LLM calls.

C.3 Complete Code for Reproducibility

We provide the complete executable code of the original MATH workflow and its pruned counterpart below.

```python
from typing import Literal, Dict, Any
from src.core.graphflow import GraphFlow
from src.core.edge import Edge
from src.core.nodes.custom_node import CustomNode
from src.core.nodes.input_node import InputNode
from src.core.nodes.program_node import ProgramNode
from src.core.nodes.sc_ensemble_node import ScEnsembleNode
from src.core.executor import GraphExecutor
from src.utils.cost_tracker import LLMCostTracker
from src.core.nodes.answer_format_node import AnswerFormatNode
from . import prompt as prompt_custom

DatasetType = Literal["MATH"]

class Workflow:

    def __init__(self, name: str, dataset: DatasetType, llm_config: Dict[str, Any] = None) -> None:
        self.name = name
        self.dataset = dataset

        if isinstance(llm_config, dict):
            self.llm_config = llm_config
        elif llm_config is None:
            self.llm_config = {"model": "gpt-4.1-mini", "temperature": 0.0}
        else:
            self.llm_config = {
                "model": getattr(llm_config, "model", "gpt-4.1-mini"),
                "temperature": getattr(llm_config, "temperature", 0.0),
                }

        self.llm_cost_tracker = LLMCostTracker()

        Input = InputNode(
            node_id="Input",
            node_llm_config=self.llm_config,
            node_description="Graph input entry"
        )

        Programmer = ProgramNode(
            node_id="Programmer",
            node_llm_config=self.llm_config,
            node_description="Generate and execute Python code for the problem"
        )

        RefineWithCode = CustomNode(
            node_id="RefineWithCode",
            node_prompt=prompt_custom.REFINE_ANSWER_PROMPT,
            node_llm_config=self.llm_config,
            node_description="Refine and format the solution using program output"
        )

        DetailedSolution = CustomNode(
            node_id="DetailedSolution",
            node_prompt=prompt_custom.DETAILED_SOLUTION_PROMPT,
            node_llm_config=self.llm_config,
            node_description="Provide a comprehensive, step-by-step solution"
        )
```

Figure 6: Code of initial workflow on MATH

```python
        GenerateSolutionA = CustomNode(
            node_id="GenerateSolutionA",
            node_prompt=prompt_custom.GENERATE_SOLUTION_PROMPT,
            node_llm_config=self.llm_config,
            node_description="Generate an additional solution (A)"
        )

        GenerateSolutionB = CustomNode(
            node_id="GenerateSolutionB",
            node_prompt=prompt_custom.GENERATE_SOLUTION_PROMPT,
            node_llm_config=self.llm_config,
            node_description="Generate an additional solution (B)"
        )

        ScEnsembler = ScEnsembleNode(
            node_id="ScEnsembler",
            node_llm_config=self.llm_config,
            node_description="Select the best solution using self-consistency",
        )

        AnswerFormatter = AnswerFormatNode(
            node_id="AnswerFormatter",
            dataset=self.dataset,
            node_llm_config=self.llm_config,
            node_description=f"Format the final answer for {self.dataset} benchmark compatibility.",
        )

        self.workflow_graph = GraphFlow(
            nodes=[Input, Programmer, RefineWithCode, DetailedSolution,
                   GenerateSolutionA, GenerateSolutionB, ScEnsembler, AnswerFormatter],
            edges=[
                Edge(source="Input", target="Programmer"),
                Edge(source="Input", target="DetailedSolution"),
                Edge(source="Input", target="GenerateSolutionA"),
                Edge(source="Input", target="GenerateSolutionB"),
                Edge(source="Programmer", target="RefineWithCode"),
                Edge(source="RefineWithCode", target="ScEnsembler"),
                Edge(source="DetailedSolution", target="ScEnsembler"),
                Edge(source="GenerateSolutionA", target="ScEnsembler"),
                Edge(source="GenerateSolutionB", target="ScEnsembler"),
                Edge(source="ScEnsembler", target="AnswerFormatter")
            ],
            entry_node_ids=["Input"],
            final_node_id=AnswerFormatter.node_id,
            description="Workflow for MATH problem solving"
        )

        self.executor = GraphExecutor(workflow_graph=self.workflow_graph)

    async def __call__(self, problem: str):

        result, execution_time, cost_summary = await self.executor.run(problem=problem)

        final_output = (
            result.get("output")
            or str(result)
        )

        total_cost = cost_summary.get("total_cost_usd", 0.0)

        return final_output, total_cost
```

Figure 7: Code of initial workflow on MATH (cont.)

```python
from typing import Literal, Dict, Any
from src.core.graphflow import GraphFlow
from src.core.edge import Edge
from src.core.nodes.custom_node import CustomNode
from src.core.nodes.input_node import InputNode
from src.core.nodes.program_node import ProgramNode
from src.core.nodes.sc_ensemble_node import ScEnsembleNode
from src.core.executor import GraphExecutor
from src.utils.cost_tracker import LLMCostTracker
from src.core.nodes.answer_format_node import AnswerFormatNode
from . import prompt as prompt_custom

DatasetType = Literal["MATH"]

class Workflow:

    def __init__(self, name: str, dataset: DatasetType, llm_config: Dict[str, Any] = None) -> None:
        self.name = name
        self.dataset = dataset

        if isinstance(llm_config, dict):
            self.llm_config = llm_config
        elif llm_config is None:
            self.llm_config = {"model": "gpt-4.1-mini", "temperature": 0.0}
        else:
            self.llm_config = {
                "model": getattr(llm_config, "model", "gpt-4.1-mini"),
                "temperature": getattr(llm_config, "temperature", 0.0),
            }

        self.llm_cost_tracker = LLMCostTracker()

        Input = InputNode(
            node_id="Input",
            node_llm_config=self.llm_config,
            node_description="Graph input entry"
        )

        Programmer = ProgramNode(
            node_id="Programmer",
            node_llm_config=self.llm_config,
            node_description="Generate and execute Python code for the problem"
        )

        DetailedSolution = CustomNode(
            node_id="DetailedSolution",
            node_prompt=prompt_custom.DETAILED_SOLUTION_PROMPT,
            node_llm_config=self.llm_config,
            node_description="Provide a comprehensive, step-by-step solution"
        )
```

Figure 8: Code of pruned workflow on MATH

```python
        GenerateSolutionA = CustomNode(
            node_id="GenerateSolutionA",
            node_prompt=prompt_custom.GENERATE_SOLUTION_PROMPT,
            node_llm_config=self.llm_config,
            node_description="Generate an additional solution "
        )

        AnswerFormatter = AnswerFormatNode(
            node_id="AnswerFormatter",
            dataset=self.dataset,
            node_llm_config=self.llm_config,
            node_description=f"Format the final answer for {self.dataset} benchmark compatibility.",
        )
        # Graph
        self.workflow_graph = GraphFlow(
            nodes=[Input, Programmer, RefineWithCode, DetailedSolution,
                   GenerateSolutionA, GenerateSolutionB, ScEnsembler, AnswerFormatter],
            edges=[
                Edge(source="Input", target="Programmer"),
                Edge(source="Input", target="DetailedSolution"),
                Edge(source="Input", target="GenerateSolutionA"),
                Edge(source="Programmer", target="AnswerFormatter"),
                Edge(source="DetailedSolution", target="AnswerFormatter"),
                Edge(source="GenerateSolutionA", target="AnswerFormatter"),
            ],
            entry_node_ids=["Input"],
            final_node_id=AnswerFormatter.node_id,
            description="Workflow for MATH problem solving"
        )

        self.executor = GraphExecutor(workflow_graph=self.workflow_graph)

    async def __call__(self, problem: str):

        result, execution_time, cost_summary = await self.executor.run(problem=problem)

        final_output = (
            result.get("output")
            or str(result)
        )

        total_cost = cost_summary.get("total_cost_usd", 0.0)

        return final_output, total_cost
```

Figure 9: Code of pruned workflow on MATH (cont.)

```python
from typing import List, Dict
from collections import defaultdict
from src.core.pruners.base import Pruner
from src.core.graphflow import GraphFlow
from typing import Tuple
from src.utils.logs import logger

class DegreePruner(Pruner):

    def get_pruned(self) -> Tuple[Dict[str, GraphFlow], GraphFlow]:

        pruning_candidates = self.identify_pruning_candidates()
        one_pruned_graphs: Dict[str, GraphFlow] = {}

        for node_to_remove in pruning_candidates.keys():
            new_nodes = [
                node for node in self.original_graph.nodes
                if node.node_id != node_to_remove
            ]
            new_edges = [
                edge for edge in self.original_graph.edges
                if edge.source != node_to_remove and edge.target != node_to_remove
            ]
            new_graph = GraphFlow(
                nodes=new_nodes,
                edges=new_edges,
                entry_node_ids=self.original_graph.entry_node_ids,
                final_node_id=self.original_graph.final_node_id,
                description=f"Pruned from {self.original_graph.description} (removed: {node_to_remove})"
            )

            one_pruned_graphs[str(node_to_remove)] = new_graph

        new_nodes = [
            node for node in self.original_graph.nodes
            if node.node_id not in pruning_candidates.keys()
        ]
        new_edges = [
            edge for edge in self.original_graph.edges
            if edge.source not in pruning_candidates.keys() and edge.target not in pruning_candidates.keys()
        ]

        all_pruned_graph = GraphFlow(
            nodes=new_nodes,
            edges=new_edges,
            entry_node_ids=self.original_graph.entry_node_ids,
            final_node_id=self.original_graph.final_node_id,
            description=f"Pruned from {self.original_graph.description} (Pruned all candidates)"
        )

        return one_pruned_graphs, all_pruned_graph
```

Figure 10: Code of Degree Centrality Algorithm

```
    def identify_pruning_candidates(self) -> Dict[str, float]:

        pruning_candidates = {}
        in_degree = defaultdict(int)
        out_degree = defaultdict(int)

        for node in self.original_graph.nodes:
            in_degree[node.node_id] = 0
            out_degree[node.node_id] = 0

        for edge in self.original_graph.edges:
            out_degree[edge.source] += 1
            in_degree[edge.target] += 1

        for node in self.original_graph.nodes:
            node_id = node.node_id
            in_deg = in_degree[node_id]
            out_deg = out_degree[node_id]

            if node_id in self.original_graph.entry_node_ids:
                continue

            if node_id == self.original_graph.final_node_id:
                continue

            if is_low_importance_node(in_deg, out_deg):
                pruning_candidates[node.node_id] = 1.0

        return pruning_candidates


def is_low_importance_node(in_degree: int, out_degree: int) -> bool:

    if in_degree + out_degree ⩽ 2:
        return True

    return False
```

Figure 11: Code of Degree Centrality Algorithm (cont.)

```python
from typing import List, Dict, Tuple
from collections import defaultdict, deque
from src.core.pruners.base import Pruner
from src.core.graphflow import GraphFlow
from src.utils.logs import logger


class BetweennessPruner(Pruner):

    def __init__(self, original_graph: GraphFlow):

        super().__init__(original_graph)
        self.betweenness_scores: Dict[str, float] = {}

    def compute_betweenness_centrality(self) -> Dict[str, float]:

        betweenness = defaultdict(float)

        for node in self.original_graph.nodes:
            if node.node_id not in self.original_graph.entry_node_ids and node.node_id ≠ self.original_graph.final_node_id:
                betweenness[node.node_id] = 0.0

        graph_adj = defaultdict(list)
        for edge in self.original_graph.edges:
            graph_adj[edge.source].append(edge.target)

        all_nodes = [node.node_id for node in self.original_graph.nodes]

        for source in all_nodes:

            stack = []
            paths = defaultdict(list)
            sigma = defaultdict(int)
            dist = defaultdict(lambda: -1)
            delta = defaultdict(float)

            sigma[source] = 1
            dist[source] = 0
            queue = deque([source])

            while queue:
                v = queue.popleft()
                stack.append(v)

                for w in graph_adj[v]:
                    if dist[w] < 0:
                        queue.append(w)
                        dist[w] = dist[v] + 1
                    if dist[w] == dist[v] + 1:
                        sigma[w] += sigma[v]
                        paths[w].append(v)

            while stack:
                w = stack.pop()
                for v in paths[w]:
                    delta[v] += (sigma[v] / sigma[w]) * (1 + delta[w])

                if w ≠ source and w not in self.original_graph.entry_node_ids and w ≠ self.original_graph.final_node_id:
                    betweenness[w] += delta[w]

        n = len(all_nodes)
        if n > 2:
            normalization = (n - 1) * (n - 2)
            for node_id in betweenness:
                betweenness[node_id] /= normalization

        return dict(betweenness)
```

Figure 12: Code of Betweenness Centraility Algorithm

```python
    def identify_pruning_candidates(self) -> Dict[str, float]:

        self.betweenness_scores = self.compute_betweenness_centrality()

        pruning_candidates = {}
        for node in self.original_graph.nodes:
            node_id = node.node_id

            if node_id in self.original_graph.entry_node_ids:
                continue

            if node_id == self.original_graph.final_node_id:
                continue

            pruning_candidates[node_id] = self.betweenness_scores.get(node_id, 0)

        return pruning_candidates

    def get_pruned(self) -> Tuple[Dict[str, GraphFlow], GraphFlow]:

        pruning_candidates = self.identify_pruning_candidates()

        one_pruned_graphs: Dict[str, GraphFlow] = {}

        for node_to_remove in pruning_candidates.keys():

            new_nodes = [
                node for node in self.original_graph.nodes
                if node.node_id != node_to_remove
            ]
            new_edges = [
                edge for edge in self.original_graph.edges
                if edge.source != node_to_remove and edge.target != node_to_remove
            ]

            new_graph = GraphFlow(
                nodes=new_nodes,
                edges=new_edges,
                entry_node_ids=self.original_graph.entry_node_ids,
                final_node_id=self.original_graph.final_node_id,
                description=f"Pruned from {self.original_graph.description} (removed: {node_to_remove},
betweenness: {self.betweenness_scores.get(node_to_remove, 0):.4f})"
            )

            one_pruned_graphs[str(node_to_remove)] = new_graph

        new_nodes = [
            node for node in self.original_graph.nodes
            if node.node_id not in pruning_candidates.keys()
        ]
        new_edges = [
            edge for edge in self.original_graph.edges
            if edge.source not in pruning_candidates.keys() and edge.target not in pruning_candidates.keys()
        ]

        all_pruned_graph = GraphFlow(
            nodes=new_nodes,
            edges=new_edges,
            entry_node_ids=self.original_graph.entry_node_ids,
            final_node_id=self.original_graph.final_node_id,
            description=f"Pruned all candidates"
        )

        return one_pruned_graphs, all_pruned_graph
```

Figure 13: Code of Betweenness Centraility Algorithm (cont.)

```python
from typing import Tuple, Dict, List
from copy import copy
from src.core.graphflow import GraphFlow
from src.utils.logs import logger
from src.core.pruners.base import Pruner
from src.core.graphflow import Edge

class ShapleyPruner(Pruner):

    def __init__(self, original_graph: GraphFlow):
        super().__init__(original_graph)

    def identify_pruning_candidates(self) -> Dict[str, float]:
        pruning_candidates = {}
        for node in self.original_graph.nodes:
            if node.node_id != self.original_graph.final_node_id and node.node_id not in
self.original_graph.entry_node_ids:
                pruning_candidates[node.node_id] = 1.0

        return pruning_candidates

    def get_pruned(self) -> Tuple[Dict[str, GraphFlow], GraphFlow]:

        pruning_candidates = self.identify_pruning_candidates()
        pruned_graphs: Dict[str, GraphFlow] = {}

        for node_id in pruning_candidates.keys():
            pruned_graph = self._prune_node(self.original_graph, node_id)
            if pruned_graph:
                pruned_graphs[node_id] = pruned_graph

        all_pruned_graph = None

        return pruned_graphs, all_pruned_graph

    def prune_node(self, graph: GraphFlow, node_id: str):

        new_nodes = [copy(n) for n in graph.nodes if n.node_id != node_id]
        new_edges = [copy(e) for e in graph.edges if e.source != node_id and e.target != node_id]

        predecessors = [e.source for e in graph.edges if e.target == node_id]
        successors = [e.target for e in graph.edges if e.source == node_id]
        existing_edges = {(e.source, e.target) for e in new_edges}

        for p in predecessors:
            for s in successors:
                if (p, s) not in existing_edges and p != s:
                    old_edge = next((e for e in graph.edges if e.source == p and e.target == node_id), None)
                    if old_edge:
                        new_edge = copy(old_edge)
                        new_edge.source = p
                        new_edge.target = s
                    else:
                        new_edge = Edge(source=p, target=s)
                    new_edges.append(new_edge)
                    existing_edges.add((p, s))

        try:
            new_graph = GraphFlow(
                nodes=new_nodes,
                edges=new_edges,
                entry_node_ids=[id for id in graph.entry_node_ids if id != node_id],
                final_node_id=graph.final_node_id,
                description=f"Graph which has pruned node:{node_id} ",
            )
            return new_graph
        except Exception as e:
            logger.error(f"Fail to prune node {node_id} failed: {e}")
            return None
```

Figure 14: Code of Approximate Shapley value Algorithm