

Optimize Once, Execute Fast: Latency-Aware Multi-Agent Workflow Learning for Recurrent Queries

Enpei Zhang¹ Feiyu Qu¹ Zheng Huang¹ Dawei Zhou² Elynn Chen³ Yujun Yan¹
¹Dartmouth College ²Virginia Tech ³New York University
 {enpei.zhang.gr, feiyu.qu.gr, zheng.huang.gr, yujun.yan}@dartmouth.edu
 zhou.d@vt.edu elynn.chen@stern.nyu.edu

Abstract

Large Language Model-based Multi-Agent Systems (LLM-based MAS) have shown remarkable success in solving complex tasks by coordinating specialized agents through multi-step workflows. To overcome the high cost and specificity of manually designed workflows, recent research has shifted toward learning them automatically. However, a key limitation remains: while agent calls introduce significant latency due to model inference time and service congestion, most automated methods optimize solely for downstream performance, ignoring execution time (makespan). This issue is amplified when workflows are reused across recurring queries, making latency accumulate over time. This limits deployment in recurrent applications such as AI customer support and AI cloud troubleshooting, where users expect fast responses. To bridge this gap, we introduce a new problem, termed latency-aware workflow generation, where the objective is to minimize workflow makespan while aiming to retain downstream performance. We propose **LAWA** (Latency-Aware Workflow OptimizAtion), a novel framework that reduces makespan via theoretically grounded graph edits targeting critical-path bottlenecks. By unifying a graph generator with this graph-editing procedure, LAWA supports both de novo workflow generation and refinement of existing ones. Empirical results across seven benchmarks demonstrate that LAWA significantly reduces makespan while achieving competitive task performance, supporting efficient reuse of multi-agent workflows for recurrent queries.

1 Introduction

Large Language Model-based Multi-Agent Systems (LLM-based MAS) have shown strong capabilities in complex tasks such as code generation and reasoning by coordinating specialized agents through structured, multi-step workflows [31, 12]. Meanwhile, manually designing such workflows requires substantial effort and domain expertise, motivating recent research on automated workflow generation [40, 16], which allows MAS to automatically customize its workflow for different tasks.

However, most prior workflow generation methods still optimize workflows primarily for downstream task performance, while largely overlooking execution time. This limitation is increasingly problematic in LLM-based MAS, where each agent call can incur considerable latency due to model inference and service congestion [5], and the overall execution time (makespan) of the workflow can grow rapidly with increasing workflow depth [37]. For workflows repeatedly invoked for recurrent queries, small per-query delays can accumulate into substantial latency costs. Such workloads arise

Correspondence to enpei.zhang.gr@dartmouth.edu.
 Our codes are available at <https://github.com/GraphmindDartmouth/LAWA>.

in AI customer support, automated software engineering, and cloud troubleshooting [26, 12, 7], where similar requests recur and users expect fast responses. Therefore, focusing solely on task performance may lead to workflows that are impractical for real-world deployment. To bridge this gap, we formally propose a new problem: latency-aware workflow generation, where the objective is to minimize makespan while preserving task performance.

To solve this problem, we propose **LAWA (Latency-Aware Workflow OptimizAtion)**, a theoretically grounded framework for latency-aware workflow optimization, designed to reduce makespan with monotonicity guarantees while preserving downstream task performance. LAWA is built on three core components: First, we devise a workflow optimization engine that performs makespan-guided beam search over graph edits with theoretical guarantees of monotonic makespan reduction. Second, we train a Graph Evaluator that estimates downstream performance of candidate workflows and guides edit selection toward edits that maintain downstream task performance. Third, we train a Graph Generator that constructs initial chain workflows when no initial workflow is given, enabling LAWA to generate new workflows from scratch. Empirical results across seven benchmarks confirm that LAWA significantly reduces makespan while maintaining competitive task performance.

Our contributions are summarized as follows:

- **New Problem:** We define latency-aware workflow generation, where makespan is treated as a primary optimization objective alongside task performance in time-sensitive multi-agent systems.
- **New Theoretically Grounded Framework:** We propose LAWA, a framework that applies theoretically grounded graph editing operations, providing formal guarantees on makespan reduction.
- **Empirical Validation:** We conduct comprehensive experiments across seven benchmarks, demonstrating that LAWA achieves significant makespan reduction compared to state-of-the-art baselines while maintaining competitive task accuracy.

2 Problem Formulation

In this section, we formalize the problem of latency-aware workflow optimization. We first introduce the notation for multi-agent workflows and the definition of workflow makespan, and then present our optimization objective.

Notations. We model a multi-agent workflow as a directed acyclic graph (DAG), $G = (\mathcal{V}, \mathbf{A}, \mathbf{t})$, where $\mathcal{V}$ is the node set, $\mathbf{A}$ is the adjacency matrix, and $\mathbf{t}$ is the vector of node latencies. Each node $v_k \in \mathcal{V}$ represents a single agent call with execution time $t_k > 0$. Different nodes may represent repeated calls to the same agent at different steps. The binary adjacency matrix $\mathbf{A} \in \{0, 1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ encodes directed information dependencies among agent calls: $\mathbf{A}_{ij} = 1$ indicates that the output of node v_i is provided as an input context to node v_j . We also introduce two special nodes, v_I and v_O , denoting workflow entry and exit. The entry node v_I has no incoming edges, the exit node v_O has no outgoing edges, and every other node lies on at least one path from v_I to v_O .

Makespan Definition. Under full parallelism, each node starts immediately after all its predecessors have completed, and the makespan $T(G)$ (i.e., end-to-end runtime of the workflow) is determined by the critical path (i.e., the path from v_I to v_O with the maximum cumulative latency). Formally,

$$T(G) = \max_{P \in \mathcal{P}_G(v_I, v_O)} \sum_{v_k \in P} t_k, \quad (1)$$

where $\mathcal{P}_G(v_I, v_O)$ denotes the set of all paths from the entry node v_I to the exit node v_O in G . In the main text, we only consider the above definition; an additional experiment with a limited number of executors is reported in Appendix F.4.

Optimization Goal. We define the optimization objective for latency-aware workflows as:

$$\min_G T(G), \quad \text{s.t. } S(G) \geq S(G_0), \quad (2)$$

where $S(G)$ denotes the downstream task performance of workflow G , and G_0 is the initial graph before modifications. The objective in Equation (2) minimizes the makespan $T(G)$, while the constraint enforces that the performance of the optimized workflow is no worse than that of the initial

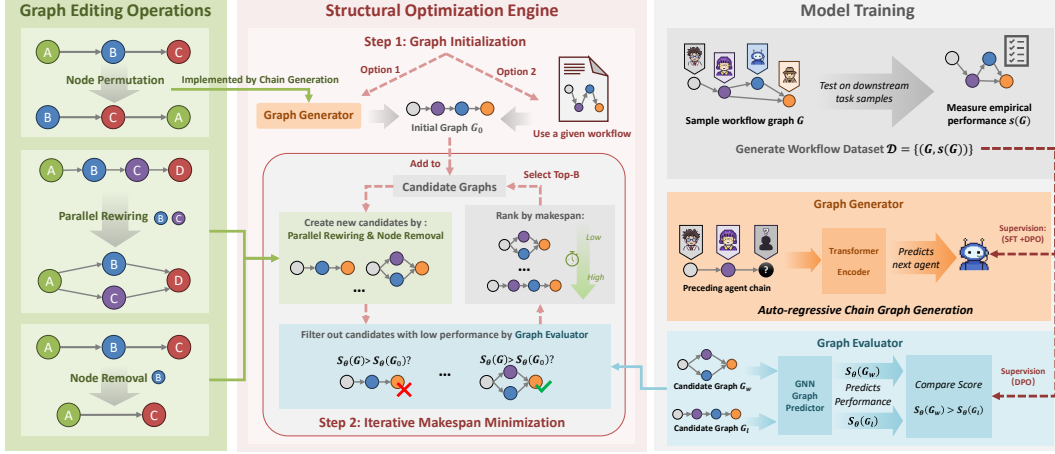


Figure 1: **Overview of LAWA.** Three main components of LAWA: (left) graph editing operations including node permutation, parallel rewiring, and node removal; (middle) a structural optimization engine that initializes a workflow, generates and ranks candidate graphs by makespan, and iteratively selects valid modifications; and (right) the training pipelines for the graph generator and graph evaluator based on empirical downstream task performance.

graph. In practice, the initial graph can be an existing workflow that meets performance requirements but is latency-inefficient, or a generated initial workflow.

3 Method

3.1 Overview

In this section, we introduce **LAWA**, as shown in Figure 1, a latency-aware workflow generation framework that constructs and optimizes multi-agent workflows through iterative graph modifications. At its core, LAWA defines a set of **theoretically grounded graph modification operations** that characterize admissible structural changes for reducing makespan under workflow execution constraints. Building on this formulation, LAWA consists of a structural optimization engine and two learnable modules. The structural optimization engine performs a makespan-guided search by iteratively applying valid graph modification operations to reduce workflow makespan. When no initial graph is provided, LAWA includes a **Graph Generator**, which constructs initial workflows and enables LAWA to operate in a de novo generation setting. To maintain downstream task performance during structural optimization, LAWA trains a **Graph Evaluator**, which estimates downstream task performance of candidate workflows and guides operation selection. In the following, we first introduce the graph editing operations and their theoretical properties, then present the detailed iterative structural optimization procedure, followed by the graph generator for workflow initialization and the graph evaluator used in the structural optimization.

3.2 Basic Graph Editing Operations

We first define a set of graph editing operations that increase parallelism, remove redundant agents, and reorder agent execution, respectively.

Definition 3.1 (Parallel Rewiring). Given two nodes i and j with $A_{ij} > 0$, the *parallel rewiring* operation removes the dependency from i to j and merges their outgoing and incoming edges to increase potential parallelism. Formally, we define the operator $\text{ParallelRewire}(G, i, j)$ as:

$$A_{ij} \leftarrow 0; \quad A_{ik} \leftarrow 1 \text{ for all } k \text{ that } A_{jk} = 1; \quad A_{kj} \leftarrow 1 \text{ for all } k \text{ that } A_{ki} = 1 \quad (3)$$

Definition 3.2 (Node Removal). The *node removal* operation eliminates a non-terminal node $k \in \mathcal{V} \setminus \{v_I, v_O\}$ from the workflow while preserving connectivity by linking its predecessors and

successors. Formally, we define the operator $\text{RemoveNode}(G, k)$ as:

$$\mathbf{A}_{ij} \leftarrow 1 \quad \text{for all } (i, j) \text{ that } \mathbf{A}_{ik} = 1 \text{ and } \mathbf{A}_{kj} = 1 ; \quad (4)$$

$$\mathbf{A}_{ik} \leftarrow 0 \quad \text{for all } i \text{ that } \mathbf{A}_{ik} = 1 ; \quad (5)$$

$$\mathbf{A}_{kj} \leftarrow 0 \quad \text{for all } j \text{ that } \mathbf{A}_{kj} = 1 . \quad (6)$$

Definition 3.3 (Node Permutation). Let $\sigma : \mathcal{V} \rightarrow \mathcal{V}$ be a permutation over the node set. We define the induced permutation operator Π_σ acting on adjacency matrix $\mathbf{A}$ as

$$(\Pi_\sigma(\mathbf{A}))_{\sigma(u), \sigma(v)} = \mathbf{A}_{u,v} \quad \forall u, v \in \mathcal{V}, \quad (7)$$

which reorders nodes in the workflow.

The proposed graph operations are designed to satisfy three requirements. First, the operations preserve workflow connectivity. Second, the operations monotonically reduce workflow makespan. Finally, under a suitable node ordering, the operator set is expressive enough to transform a chain into any transitively reduced DAG considered in our formulation. The following theorems establish these guarantees.

Theorem 3.4 (Reachability). Let $u \rightsquigarrow_G v$ denote that there exists a directed path from u to v in G . Let G' denote the graph obtained from G after applying one of the operations: (i) Node Removal. Let $k \in \mathcal{V} \setminus \{v_I, v_O\}$ be a removed node. Then,

$$\forall u, v \in \mathcal{V} \setminus \{k\}, \quad u \rightsquigarrow_G v \iff u \rightsquigarrow_{G'} v. \quad (8)$$

(ii) Parallel Rewiring. Let (i, j) be the rewired edge. Then,

$$\forall u, v \in \mathcal{V} \setminus \{i, j\}, \quad u \rightsquigarrow_G v \iff u \rightsquigarrow_{G'} v. \quad (9)$$

Proof is presented in Appendix A.1.

Theorem 3.5 (Non-increasing Makespan). Let $G = (\mathcal{V}, \mathbf{A}, \mathbf{t})$ be a workflow DAG. Let G' denote the graph obtained from G by applying either the Node Removal operation to a node $k \in \mathcal{V} \setminus \{v_I, v_O\}$ or the Parallel Rewiring operation to an existing edge (i, j) . Then $T(G') \leq T(G)$.

Proof is presented in Appendix A.2.

Theorem 3.6 (Completeness). Let G_{chain} be an arbitrary chain DAG over all nodes in $\mathcal{V}$ with adjacency matrix $\mathbf{A}_{\text{chain}}$. For any target DAG $G^* = (\mathcal{V}, \mathbf{A}^*)$ that is transitively reduced, there exist a node permutation σ and a finite operation sequence $\mathcal{O}$ consisting only of Parallel Rewiring and Node Removal such that $\mathcal{O}(\Pi_\sigma(\mathbf{A}_{\text{chain}})) = \mathbf{A}^*$.

Proof is presented in Appendix A.3.

Together, Theorems 3.4 and 3.5 motivate Parallel Rewiring and Node Removal as graph-modification operators that preserve workflow connectivity and do not increase makespan. Theorem 3.6 further indicates that the edit operators are expressive enough, in principle, to transform a chain into any transitively reduced DAG under a suitable node ordering, and the Graph Generator (Section 3.4) is designed to learn initial chain workflows with such suitable node orderings.

3.3 Latency-Aware Structural Optimization

Based on the graph editing operations defined above, we now describe the procedure for latency-aware structural optimization in LAWA, which operates in two stages.

Workflow Initialization. LAWA initializes workflows in two ways: it either refines an existing workflow graph, or constructs an initial graph using the Graph Generator when no prior structure is available. The Graph Generator produces chain-structured graphs, motivated by the expressivity result in Theorem 3.6. These two modes allow LAWA to function both as a plug-in optimizer for existing workflows and as a de novo workflow generator.

Iterative Structural Optimization. In the second stage, LAWA refines workflow structure via a guided beam search to reduce makespan (Algorithm 1). Before the search, we first estimate node latencies t_k by calibration runs (e.g., each agent's P95 response time on a calibration set). At each

Algorithm 1 Latency-Aware Structural Optimization

Input: Optional initial graph G_0 , beam size B , max iterations K .

- 1: Initialize pretrained Graph Generator $\mathcal{G}_\phi$ and pretrained Graph Evaluator S_θ
- 2: Set $G_0 \leftarrow \mathcal{G}_\phi()$ if $G_0 = \emptyset$
- 3: Initialize candidate set $\mathcal{C} \leftarrow \{G_0\}$
- 4: **for** $k = 1$ to K **do**
- 5: **for all** $G \in \mathcal{C}$ **do**
- 6: Construct candidates by Node Removal: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\text{RemoveNode}(G, u) \mid u \in \mathcal{V}(G) \setminus \{v_I, v_O\}\}$
- 7: Compute $\mathbf{R}^{\text{rwt}}$ by Equation (15)
- 8: **for all** edge (u, v) with $\mathbf{R}_{u,v}^{\text{rwt}} \geq 0$ **do**
- 9: Construct candidates by Parallel rewiring: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\text{ParallelRewire}(G, u, v)\}$
- 10: **end for**
- 11: **end for**
- 12: Performance-Based filtering: $\mathcal{C} \leftarrow \{G' \in \mathcal{C} \mid S_\theta(G') \geq S_\theta(G_0)\}$
- 13: Candidate update by runtime-first ranking: $\mathcal{C} \leftarrow \text{Top-}B_{T(\cdot)}(\mathcal{C})$
- 14: **end for**
- 15: Select final solution: $G^* \leftarrow \arg \min_{G \in \mathcal{C}} (T(G))$

Output: Optimized workflow G^* .

beam-search iteration, LAWA follows a three-stage procedure. First, for each workflow $G \in \mathcal{C}$, LAWA generates new candidates by applying one step of Node Removal or Parallel Rewiring. Node Removal is applied to every non-terminal node in G excluding v_I and v_O . For Parallel Rewiring, to reduce the number of candidates, we apply it only to edges (u, v) with $\mathbf{R}_{u,v}^{\text{rwt}} \geq 0$, where $\mathbf{R}_{u,v}^{\text{rwt}}$ is the approximate change in downstream task performance induced by rewiring (u, v) . *Parallel Rewiring is disallowed for node pairs invoking the same agent, as such nodes cannot be executed in parallel in reality.* Second, LAWA evaluates each candidate workflow $G' \in \mathcal{C}$ using a Graph Evaluator $S_\theta(\cdot)$ and retains only those satisfying the baseline performance constraint $S_\theta(G') \geq S_\theta(G_0)$. Finally, LAWA keeps the Top- B retained workflows with the smallest makespan $T(G)$ as the candidate set $\mathcal{C}$ for the next iteration. To avoid repeatedly executing candidate workflows, the Graph Evaluator is used to predict downstream task performance; $\mathbf{R}_{u,v}^{\text{rwt}}$ is computed from its gradient information.

3.4 Graph Generator for Initialization

The Graph Generator provides an initial chain-structured workflow when no prior structure is available. For each agent node u , we first construct an embedding $\mathbf{e}_u$ by encoding the agent role and system prompt with a frozen Sentence-BERT encoder. We formulate workflow initialization as an auto-regressive sequence generation problem, where each token corresponds to the agent embedding $\mathbf{e}_u$, and the model predicts the next token (agent) a_t according to current sequence $a_{<t}$, producing an ordered agent sequence that defines a chain workflow.

Graph Generator Architecture. The Graph Generator adopts a transformer encoder $\mathcal{R}_\phi$ for auto-regressive sequence generation. Given a partial agent sequence $a_{<t}$, the encoder produces a contextual hidden representation $\mathbf{h}_t = \mathcal{R}_\phi(a_{<t}) \in \mathbb{R}^d$. The model predicts the next agent a_t by computing a similarity-based distribution between $\mathbf{h}_t$ and the agent embedding matrix $\mathbf{E}$:

$$P_\phi(a_t \mid a_{<t}) = \text{Softmax}\left(\frac{\mathbf{h}_t \cdot \mathbf{E}^\top}{\sqrt{d}}\right), \quad (10)$$

During generation, the entry node is used as the initial token, and generation terminates when the exit node is produced.

Graph Generator Training. We train $\mathcal{R}_\phi$ using a two-stage strategy consisting of supervised pre-training and preference alignment. We use a graph dataset $\mathcal{D} = \{(G, s(G))\}$, where $s(G)$ is the empirical performance of workflow G evaluated on the held-out training set for the corresponding task. From $\mathcal{D}$, we retain highest-scoring graphs $\mathcal{D}_{\text{high}}$ for supervised learning. For each $G \in \mathcal{D}_{\text{high}}$, we sample sequences $y = (a_1, \dots, a_L)$ such that for any indices $1 \leq i < j \leq L$, the corresponding node of a_i in G is not reachable from that of a_j , and minimize the negative log-likelihood:

$$\mathcal{L}_{\text{SFT}}(\phi) = -\mathbb{E}_{y \sim \mathcal{D}_{\text{high}}} \sum_{t=1}^L \log P_\phi(a_t \mid a_{<t}). \quad (11)$$

To further align generation with workflow performance, we also apply DPO. Preference pairs (y_w, y_l) are constructed from graph $G_w, G_l \in \mathcal{D}$ with $s(G_w) > s(G_l)$, and $\mathcal{R}_\phi$ is trained using the DPO loss:

$$\mathcal{L}_{\text{DPO}}(\phi) = -\mathbb{E}_{(y_w, y_l)} \left[\log \sigma \left(\beta \log \frac{P_\phi(y_w)}{P_{\text{ref}}(y_w)} - \beta \log \frac{P_\phi(y_l)}{P_{\text{ref}}(y_l)} \right) \right], \quad (12)$$

where P_{ref} is the frozen SFT model and β controls deviation from the reference policy.

3.5 Graph Evaluator for Performance Prediction

The Graph Evaluator is a supervised model that maps a workflow graph G to a scalar score $S_\theta(G) \in \mathbb{R}$ to estimate the downstream task performance. As only the relative performance of the workflow matters, we train it using preference ranking instead of regression.

Graph Evaluator Architecture. We implement the Graph Evaluator S_θ as a graph neural network (GNN) over the workflow DAG. Each node $u \in \mathcal{V}$ is initialized with the agent embedding $\mathbf{e}_u$ defined in Section 3.4. The node embeddings are propagated through GNN message-passing layers and then pooled to obtain a graph-level representation, which is mapped to a scalar score by an MLP head:

$$S_\theta(G) = \text{MLP}(\text{Pool}(\text{GNN}(\mathbf{A}, \{\mathbf{e}_u\}))). \quad (13)$$

Graph Evaluator Training. We train the Graph Evaluator using a preference-based objective. Specifically, we reuse the graph dataset $\mathcal{D}$ defined in Section 3.4. From $\mathcal{D}$, we construct preference pairs (G_w, G_l) such that $s(G_w) > s(G_l)$. We then train the Graph Evaluator S_θ using a pairwise preference objective inspired by Direct Preference Optimization (DPO) [23], which encourages higher scores for preferred workflows:

$$\mathcal{L}_{\text{pairwise}} = -\log \sigma(\beta(\Delta_\pi - \Delta_{\text{ref}})), \quad (14)$$

where $\sigma(\cdot)$ is the sigmoid function; $\beta \in \mathbb{R}$ is a hyperparameter; $\Delta_\pi = S_\theta(G_w) - S_\theta(G_l)$ denotes the predicted score difference; and Δ_{ref} is provided by a reference model (e.g., current best model on the validation set).

Gradient-Based Approximation for Edge Editing. To filter out rewiring operations that are unlikely to improve downstream task performance, we approximate the performance change of Parallel Rewiring using first-order gradients of the Graph Evaluator with respect to the workflow adjacency matrix $\mathbf{A}$. Let $\mathbf{\Gamma} = \partial S_\theta / \partial \mathbf{A}$ denote the gradient of the Graph Evaluator with respect to $\mathbf{A}$. For each potential rewiring (i, j) , the induced score change is approximated by the matrix inner product $\langle \mathbf{\Gamma}, \Delta \mathbf{A} \rangle$, where $\Delta \mathbf{A}$ represents the change of $\mathbf{A}$ after the rewiring. The score change of all edges can be approximated by:

$$\mathbf{R}^{\text{rwr}} = -\mathbf{H} + \tilde{\mathbf{H}} \mathbf{A}^\top + \mathbf{A}^\top \tilde{\mathbf{H}}, \quad (15)$$

where $\mathbf{H} = \mathbf{\Gamma} \odot \mathbf{A}$, $\tilde{\mathbf{H}} = \mathbf{\Gamma} \odot (1 - \mathbf{A})$, $\odot$ denotes the Hadamard product, and $\mathbf{R}_{i,j}^{\text{rwr}}$ represents the estimated score change of rewiring edge (i, j) . The derivations are provided in Appendix B.

We further provide a theoretical analysis of a perturbed Graph Evaluator in Appendix E, showing that LAWA can still preserve performance with high probability under mild evaluator decision errors.

4 Experiments

In this section, we conduct extensive experiments to evaluate LAWA through four research questions. **(RQ1)** How does LAWA, as a workflow generation framework, compare to prior methods in generating workflows? **(RQ2)** Can LAWA, when used as a plug-in module, reduce workflow makespan while preserving performance? **(RQ3)** How robust is LAWA when Graph Evaluator decisions are inaccurate or perturbed? **(RQ4)** How do the individual components of LAWA contribute to its latency and performance? Additional experiments are provided in Appendix F.

Table 1: **Real wall-clock makespan (Seconds↓) per query on 7 benchmarks.** All methods are evaluated using agents powered by Qwen3-30B-A3B-Instruct-2507 with different roles. Each method is tested three times and results are mean $\pm$ std. The **first** and **second** best entries in each column are highlighted. LAWA achieves the lowest makespan on six of seven benchmarks and the best average rank.

Method	MBPP	HumanEval	AQuA	GSM8K	MATH500	SVAMP	MMLU	Avg. Rank
Rand. Graph	16.92 $\pm$ 1.53	19.80 $\pm$ 1.48	24.16 $\pm$ 0.42	17.13 $\pm$ 1.24	26.33 $\pm$ 1.41	17.24 $\pm$ 1.12	18.47 $\pm$ 0.43	8.86
Rand. Chain	22.67 $\pm$ 1.52	22.26 $\pm$ 0.94	31.34 $\pm$ 2.65	27.90 $\pm$ 0.08	36.86 $\pm$ 0.29	23.54 $\pm$ 0.25	24.06 $\pm$ 0.22	10.57
3-Round Disc.	17.52 $\pm$ 2.84	15.08 $\pm$ 0.71	17.04 $\pm$ 0.61	15.06 $\pm$ 0.10	21.04 $\pm$ 0.94	13.60 $\pm$ 0.76	13.93 $\pm$ 0.13	7.29
DyLAN	11.46 $\pm$ 1.34	11.29 $\pm$ 0.08	13.65 $\pm$ 1.10	8.09 $\pm$ 0.19	16.63 $\pm$ 0.23	9.38 $\pm$ 0.61	9.11 $\pm$ 0.26	3.14
G-Designer	25.27 $\pm$ 1.75	24.01 $\pm$ 1.28	15.56 $\pm$ 0.26	9.74 $\pm$ 0.55	21.34 $\pm$ 9.46	23.47 $\pm$ 4.08	25.32 $\pm$ 2.43	9.43
AgentDropout	13.92 $\pm$ 5.56	14.24 $\pm$ 3.73	13.85 $\pm$ 0.42	9.57 $\pm$ 0.95	21.54 $\pm$ 7.58	16.72 $\pm$ 2.80	17.31 $\pm$ 4.21	6.14
AgentPrune	20.18 $\pm$ 2.54	18.28 $\pm$ 1.24	14.56 $\pm$ 2.15	9.20 $\pm$ 0.08	19.25 $\pm$ 2.12	18.75 $\pm$ 0.69	19.01 $\pm$ 1.44	7.29
BiRouter	13.54 $\pm$ 0.85	17.40 $\pm$ 3.51	14.19 $\pm$ 2.15	9.95 $\pm$ 1.12	18.64 $\pm$ 1.20	8.93 $\pm$ 0.54	12.17 $\pm$ 1.67	5.57
AFlow	10.24 $\pm$ 1.79	14.49 $\pm$ 2.86	15.53 $\pm$ 1.27	9.51 $\pm$ 0.57	18.55 $\pm$ 1.03	8.63 $\pm$ 0.84	11.00 $\pm$ 1.29	4.43
LAMaS	8.82 $\pm$ 0.97	11.87 $\pm$ 1.38	9.92 $\pm$ 2.37	8.78 $\pm$ 1.80	11.85 $\pm$ 1.16	8.12 $\pm$ 0.88	8.39 $\pm$ 0.24	2.14
LAWA	8.84 $\pm$ 0.79	11.26 $\pm$ 1.03	9.86 $\pm$ 0.38	7.30 $\pm$ 0.10	11.80 $\pm$ 0.48	6.18 $\pm$ 0.56	7.86 $\pm$ 1.28	1.14

Table 2: **Comparison of task performance (Accuracy↑) on 7 Benchmarks.** All methods are evaluated using agents powered by Qwen3-30B-A3B-Instruct-2507 with different roles. Each method is tested three times and results are mean $\pm$ std. The **first** and **second** best entries in each column are bolded with colors. LAWA maintains comparable or superior accuracy to the baselines, achieving the best average rank.

Method	MBPP	HumanEval	AQuA	GSM8K	MATH500	SVAMP	MMLU	Avg. Rank
Rand. Graph	63.94 $\pm$ 0.49	92.07 $\pm$ 2.99	80.58 $\pm$ 0.67	93.23 $\pm$ 0.20	66.53 $\pm$ 0.41	93.67 $\pm$ 0.62	65.33 $\pm$ 4.94	7.36
Rand. Chain	64.07 $\pm$ 0.73	91.67 $\pm$ 0.76	80.31 $\pm$ 0.64	93.91 $\pm$ 1.15	66.47 $\pm$ 0.57	93.30 $\pm$ 0.37	64.11 $\pm$ 5.60	8.00
3-Round Disc.	64.33 $\pm$ 1.12	92.89 $\pm$ 0.76	72.57 $\pm$ 0.98	94.77 $\pm$ 0.21	63.80 $\pm$ 0.28	94.27 $\pm$ 0.29	65.66 $\pm$ 0.29	6.64
DyLAN	64.72 $\pm$ 1.02	93.29 $\pm$ 1.32	80.71 $\pm$ 0.96	95.80 $\pm$ 0.23	65.87 $\pm$ 0.41	93.67 $\pm$ 0.24	68.03 $\pm$ 0.09	4.79
G-Designer	63.81 $\pm$ 1.10	89.84 $\pm$ 0.76	76.77 $\pm$ 1.79	94.16 $\pm$ 0.33	66.13 $\pm$ 1.09	94.20 $\pm$ 0.29	67.71 $\pm$ 0.17	7.71
AgentDropout	64.59 $\pm$ 1.91	91.06 $\pm$ 2.24	80.84 $\pm$ 1.30	93.58 $\pm$ 0.83	66.47 $\pm$ 0.09	93.27 $\pm$ 0.65	65.97 $\pm$ 2.15	7.07
AgentPrune	64.46 $\pm$ 0.66	90.45 $\pm$ 2.24	79.13 $\pm$ 3.36	94.06 $\pm$ 0.43	67.33 $\pm$ 0.77	94.20 $\pm$ 0.57	72.12 $\pm$ 5.66	5.64
BiRouter	63.81 $\pm$ 0.84	92.48 $\pm$ 0.76	82.15 $\pm$ 1.94	95.05 $\pm$ 0.28	66.73 $\pm$ 0.38	93.90 $\pm$ 1.00	65.09 $\pm$ 0.64	5.93
AFlow	65.24 $\pm$ 0.22	93.70 $\pm$ 0.93	83.20 $\pm$ 0.23	96.06 $\pm$ 0.95	66.80 $\pm$ 0.69	92.13 $\pm$ 0.83	76.04 $\pm$ 1.48	3.14
LAMaS	64.98 $\pm$ 0.39	91.67 $\pm$ 0.93	79.53 $\pm$ 5.05	92.12 $\pm$ 1.14	65.33 $\pm$ 1.67	92.70 $\pm$ 2.17	66.99 $\pm$ 1.71	7.93
LAWA	66.28 $\pm$ 0.37	92.89 $\pm$ 1.04	83.46 $\pm$ 0.85	96.03 $\pm$ 0.20	67.00 $\pm$ 0.86	94.23 $\pm$ 0.21	77.77 $\pm$ 3.15	1.79

4.1 Experimental Setup

Benchmarks. We conduct evaluations across seven benchmarks spanning three domains. The code generation domain includes MBPP and HumanEval [3, 6]. The mathematical reasoning domain includes GSM8K, MATH500, SVAMP, and AQuA [8, 11, 21, 17]. The general knowledge domain is evaluated using MMLU [10]. For each benchmark, we reserve a small subset of samples for training the proposed method and baselines, and hold them out from evaluation (see Appendix C).

Baselines. We compare LAWA against state-of-the-art dynamic multi-agent frameworks, including DyLAN [18], G-Designer [39], AgentDropout [27], AgentPrune [38], and BiRouter [32], as well as the accuracy-oriented workflow generator AFlow [40], and the latency-aware framework LAMaS [25]. In addition, we include three basic workflow topologies: (1) **Random Chain**, which randomly permutes all available agents into a sequential structure; (2) **Random Graph**, which constructs a directed acyclic graph (DAG) using all agents, where edges are sampled with probability $p = 0.5$ while enforcing acyclicity; and (3) **3-Round Discussion**, which simulates a round-table discussion where all agents execute in parallel for three rounds, with the full context of the previous round broadcast to all agents in the subsequent round.

Agent Backbones. For a fair comparison, all methods use the same pool of agents built on Qwen3-30B-A3B-Instruct-2507 [30]. Each agent is configured with different roles and functionalities. Across task domains, each method can access 8 candidate agents for coding, 10 for general knowledge, and 11 for mathematical reasoning. Additional details are provided in Appendix C.

Wall-Clock Makespan Measurement. We report real end-to-end wall-clock makespan per query under the same computational budget; detailed settings are provided in Appendix C. For LAWA, the t_k used during optimization is estimated as each agent’s P95 response time on training-set queries.

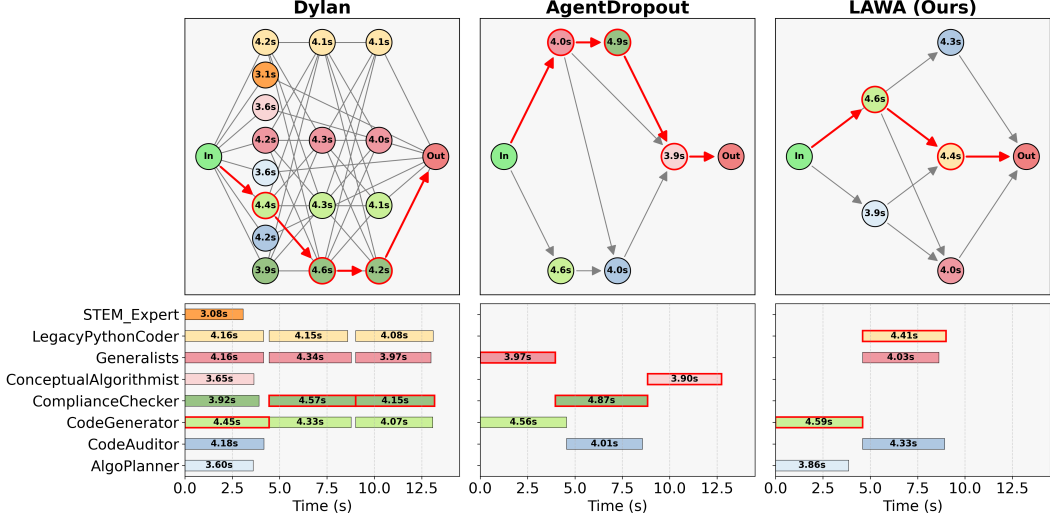


Figure 2: **Visualization of Workflow Topology and Execution Gantt Chart.** We visualize the workflow of three representative methods on one MBPP task. The top row shows the workflow graphs where red arrows denote the critical path, with the wall-clock latency labeled on each node. The bottom row shows the corresponding Gantt charts, with red-outlined blocks highlighting critical path tasks. LAWA generates the workflow with the shortest critical path to achieve maximal parallel execution and the shortest makespan.

4.2 RQ1: LAWA as a Workflow Generator

We first evaluate the effectiveness of LAWA as a workflow generator by comparing it with prior methods in terms of both execution efficiency and downstream task performance. In this setting, LAWA uses the Graph Generator to initialize workflows. To mitigate randomness, we execute all methods across 3 independent runs and report average results.

Makespan Analysis. Table 1 reports real wall-clock makespan of generated workflows across seven benchmarks. LAWA is the fastest method on six of seven benchmarks and is only 0.02s slower than LAMaS on MBPP, while outperforming LAMaS in task accuracy on all seven benchmarks (Table 2). LAWA is also efficient to obtain: since the search operates on graph structures without executing agents, its one-time search and evaluator-training cost is recovered within a few recurrent executions, and even with graph-dataset construction included, the full pipeline remains substantially faster than the on-policy AgentDropout (Appendix F.2).

To intuitively demonstrate why LAWA achieves lower makespan, Figure 2 visualizes three example workflow topologies and corresponding Gantt charts produced by representative methods on the MBPP benchmark. DyLAN adopts a layered, neural-network-like execution pattern that enables high parallelism within each round but enforces strict inter-round synchronization. This causes each round to wait for the slowest agent in the previous layer, leading to higher overall makespan. Another baseline, AgentPrune, simplifies workflows by pruning nodes and edges but does not explicitly target critical-path bottlenecks, often leaving a long critical path that dominates end-to-end makespan. In contrast, LAWA produces workflows with shorter critical paths. We include a case study on how LAWA reduces the critical path in Appendix D.

Performance Analysis. Table 2 summarizes task performance across all benchmarks. Despite aggressively optimizing workflow topology for makespan reduction, LAWA consistently maintains comparable or superior accuracy to the state-of-the-art baselines, achieving the best average rank among all methods, including the strong accuracy-oriented AFlow. These results show that, as a workflow generation method, LAWA can produce workflow graphs that achieve both low makespan and strong downstream performance. We also report total token consumption in Appendix F.1, where LAWA still remains token-efficient, and results with the frontier model Kimi-K2.6 in Appendix F.3.

Table 3: **LAWA as a Plug-in Module.** We apply LAWA to the workflow graphs generated by four baseline methods and report the resulting changes in accuracy and makespan (mean $\pm$ std) relative to the original baseline performance. LAWA consistently compresses the makespan of these diverse initial workflows with non-negative mean accuracy changes in the reported runs, validating its role as a flexible plug-in module.

Dataset	Random Graph		Random Chain		AgentPrune		AgentDropout	
	Acc (%)	Makespan (s)	Acc (%)	Makespan (s)	Acc (%)	Makespan (s)	Acc (%)	Makespan (s)
HumanEval	+1.63 $\pm$ 3.53	-4.40 $\pm$ 0.97	+0.20 $\pm$ 0.57	-7.61 $\pm$ 0.04	+1.42 $\pm$ 1.88	-2.79 $\pm$ 0.46	+0.81 $\pm$ 2.35	-2.19 $\pm$ 1.56
MBPP	+0.26 $\pm$ 0.49	-5.39 $\pm$ 1.49	+0.39 $\pm$ 0.84	-6.86 $\pm$ 0.06	+0.91 $\pm$ 1.20	-2.41 $\pm$ 0.72	+0.13 $\pm$ 1.63	-2.75 $\pm$ 0.73
AQuA	+0.52 $\pm$ 0.74	-4.92 $\pm$ 1.59	+0.52 $\pm$ 0.19	-8.05 $\pm$ 0.11	+1.84 $\pm$ 3.17	-3.65 $\pm$ 0.06	+0.79 $\pm$ 1.96	-3.21 $\pm$ 0.12
GSM8K	+0.25 $\pm$ 0.25	-6.16 $\pm$ 1.33	+0.08 $\pm$ 1.07	-8.17 $\pm$ 0.01	+0.08 $\pm$ 0.75	-3.06 $\pm$ 0.43	+1.26 $\pm$ 1.26	-2.78 $\pm$ 0.24
MATH500	+0.47 $\pm$ 1.57	-6.63 $\pm$ 0.57	+0.47 $\pm$ 0.57	-8.66 $\pm$ 0.16	+0.07 $\pm$ 0.90	-3.35 $\pm$ 0.42	+0.13 $\pm$ 0.52	-3.71 $\pm$ 0.13
SVAMP	+1.23 $\pm$ 0.78	-5.53 $\pm$ 1.43	+0.03 $\pm$ 0.21	-8.17 $\pm$ 0.01	+0.03 $\pm$ 1.01	-3.70 $\pm$ 0.33	+0.63 $\pm$ 0.69	-2.92 $\pm$ 0.73
MMLU	+1.94 $\pm$ 5.88	-4.61 $\pm$ 0.33	+3.26 $\pm$ 3.87	-8.62 $\pm$ 0.29	+0.02 $\pm$ 1.10	-3.14 $\pm$ 0.13	+0.11 $\pm$ 5.74	-3.48 $\pm$ 0.58

Table 4: **RQ3: Robustness under evaluator perturbations.** We flip the evaluator’s keep/reject decision with probability α , and report the task accuracy (%) of the generated workflows as mean $\pm$ std. Performance remains close to the unperturbed setting for small flip rates ($\alpha \leq 0.2$).

Dataset	$\alpha=0.00$	$\alpha=0.10$	$\alpha=0.20$
HumanEval	92.89 $\pm$ 1.04	92.48 $\pm$ 0.35	91.67 $\pm$ 1.53
MBPP	66.28 $\pm$ 0.37	66.28 $\pm$ 0.59	65.76 $\pm$ 0.67
AQuA	83.46 $\pm$ 0.85	83.34 $\pm$ 0.82	82.83 $\pm$ 1.38
MATH500	67.00 $\pm$ 0.86	66.73 $\pm$ 0.31	65.87 $\pm$ 0.23

Table 5: **RQ4: Component ablations.** Changes in accuracy (%) and makespan (s) relative to full LAWA. Removing any component reduces accuracy on all three benchmarks.

Ablation	AQuA		SVAMP		MMLU	
	Δ Acc.	Δ Time	Δ Acc.	Δ Time	Δ Acc.	Δ Time
w/o Node Removal	-0.65	+1.96	-4.71	+8.30	-3.36	+15.92
w/o Parallel Rewiring	-0.78	+1.64	-4.45	+4.97	-2.25	+18.67
w/o SFT	-2.75	-1.96	-0.68	+2.37	-1.66	+5.51
w/o DPO	-3.41	+7.26	-3.40	-0.36	-6.81	+13.08

4.3 RQ2: LAWA as a Plug-in Module

We further evaluate LAWA as a plug-in structural optimizer by applying it to workflows generated by different baseline methods, without modifying their original generation procedures. Specifically, we extract the workflow graphs produced by each baseline and use each graph as the initialization for LAWA. To mitigate randomness, each baseline is executed three times, and we report the average changes in performance and makespan before and after applying LAWA.

As shown in Table 3, LAWA consistently reduces the makespan of workflows produced by all baseline methods while showing no systematic degradation in mean task performance in the reported settings. This suggests that the proposed graph modification operators can compress makespan with limited impact on downstream accuracy. Notably, these gains are observed across diverse initial topologies, including both chain-structured and general DAG workflows. This indicates that LAWA’s optimization behavior does not rely on chain-structured initialization. Overall, the results demonstrate that LAWA can serve as a flexible plug-in module for latency-aware workflow optimization.

4.4 RQ3: Robustness under Inaccurate or Perturbed Evaluator Decisions

To evaluate robustness under an imperfect evaluator, we flip the evaluator’s keep/reject decision with probability α at the filtering step in Algorithm 1 (line 12) for each candidate graph during optimization and report final task performance under small perturbation rates $\alpha \in \{0.0, 0.1, 0.2\}$. As shown in Table 4, LAWA remains stable under mild evaluator perturbations, with performance close to the unperturbed setting for $\alpha \leq 0.2$. These results suggest that occasional evaluator mistakes have limited impact on final performance under mild perturbations.

4.5 RQ4: Component Ablations

To isolate the sources of LAWA’s latency and performance gains, we ablate the two graph-editing operations and the two training stages of the Graph Generator, and report the resulting changes relative to full LAWA in Table 5. Removing either Node Removal or Parallel Rewiring restricts the optimizer to a single type of edit; the resulting optimization terminates earlier and yields longer makespans. Both ablations also reduce accuracy on all three benchmarks, as the available structures are more limited. For Graph Generator training, removing either SFT or DPO reduces accuracy on all three benchmarks, with consistently larger degradation without DPO. Both training stages therefore contribute to performance, with DPO having the larger effect.

5 Related Works

Workflow Generation for LLM-MAS. Empowered by Large Language Models, LLM-based multi-agent systems leverage multiple specialized agents and structured coordination to solve complex tasks across domains [31, 15, 12]. Early systems often rely on manually specified protocols, static interaction topologies, or SOP-style workflows [28, 22, 36]. To reduce manual design effort, recent work explores automated workflow generation and orchestration, including LLM-generated or modular workflow synthesis [40, 29, 20], parallel or broadcast-style collaboration [18, 37], routing-based agent selection [33, 32, 35], and graph-based topology learning or pruning [38, 27, 39, 16]. Despite these advances, most methods primarily optimize task performance, communication efficiency, or token cost, rather than explicitly minimizing workflow makespan.

Makespan Optimization for LLM-MAS. Latency is an increasingly important bottleneck for LLM-MAS, since multi-step agent workflows introduce repeated LLM calls, synchronization barriers, and service-side congestion [5, 37]. Existing latency-oriented studies mainly address the LLM serving layer [5, 24, 9, 19, 2, 14], while optimization of the workflow topology itself remains relatively underexplored. Recent topology learning methods such as LAMaS [25] and AgentBalance [4] begin to incorporate latency into MAS topology design; however, LAMaS focuses on predefined layer-wise parallel orchestration, while AgentBalance mainly treats latency as a budget constraint without a theoretical analysis of topology-level makespan reduction. LAWA fills this gap by optimizing arbitrary DAG-structured LLM-MAS workflows and providing theoretical characterizations of graph edits in terms of reachability preservation, non-increasing makespan, and expressivity.

6 Conclusion

In this paper, we present LAWA, a latency-aware framework for optimizing Large Language Model-based multi-agent workflows. LAWA uses makespan-guided graph editing to shorten critical paths, while a learned evaluator guides the search toward workflows that preserve downstream task performance. Across seven benchmarks, LAWA produces lower-latency execution graphs and can also serve as a plug-in optimizer for existing workflows. We hope this work encourages further study of workflow-level latency optimization in multi-agent systems. Additional discussion of limitations is provided in Appendix G.

References

- [1] Thomas L Adam, K. Mani Chandy, and JR Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.
- [2] Saurabh Agarwal, Marco Laju, Donghyun Son, Nitin Kedia, Myungjin Lee, Jayanth Srinivasa, and Aditya Akella. Nalar: Workflow-aware management of agentic applications. *arXiv preprint arXiv:2601.05109*, 2026.
- [3] Jacob Austin, Augustus Odena, Maxwell Nye, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [4] Shuowei Cai, Yansong Ning, and Hao Liu. AgentBalance: Backbone-then-topology design for cost-effective multi-agent systems under budget constraints. *arXiv preprint arXiv:2512.11426*, 2025.
- [5] Jinyuan Chen, Jiuchen Shi, Quan Chen, and Minyi Guo. Kairos: Low-latency multi-agent serving with shared llms and excessive loads in the public cloud. *arXiv preprint arXiv:2508.06948*, 2025.
- [6] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [7] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *arXiv preprint arXiv:2501.06706*, 2025.
- [8] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- [9] Yinwei Dai, Zhuofu Chen, Anand Iyer, and Ravi Netravali. Aragog: Just-in-time model routing for scalable serving of agentic workflows. *arXiv preprint arXiv:2511.20975*, 2025.
- [10] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.
- [11] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.
- [12] Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2024.
- [13] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.
- [14] Hao Kang, Ziyang Li, Xinyu Yang, Weili Xu, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. ThunderAgent: A simple, fast and program-aware agentic inference system. *arXiv preprint arXiv:2602.13692*, 2026.
- [15] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.
- [16] Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. *arXiv preprint arXiv:2507.18224*, 2025.
- [17] Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation : Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*, 2017.

- [18] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024.
- [19] Kangqi Ni, Wenyue Hua, Xiaoxiang Shi, Jiang Guo, Shiyu Chang, and Tianlong Chen. Chimera: Latency- and performance-aware multi-agent serving for heterogeneous llms. *arXiv preprint arXiv:2603.22206*, 2026.
- [20] Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. Flow: Modularized agentic workflow automation. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [21] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems? *arXiv preprint arXiv:2103.07191*, 2021.
- [22] Chen Qian, Wei Liu, Hongzhang Liu, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024.
- [23] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.
- [24] Junyi Shen, Noppanat Wadlom, and Yao Lu. Batch query processing and optimization for agentic workflows. *arXiv preprint arXiv:2509.02121*, 2026.
- [25] Xi Shi, Mengxin Zheng, and Qian Lou. Learning latency-aware orchestration for parallel multi-agent systems. *arXiv preprint arXiv:2601.10560*, 2026.
- [26] Haoxin Wang, Xianhan Peng, Huang Cheng, Yizhe Huang, Ming Gong, Chenghan Yang, Yang Liu, and Jiang Lin. ECom-bench: Can LLM agent resolve real-world E-commerce customer support issues? In Saloni Potdar, Lina Rojas-Barahona, and Sebastien Montella, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 276–284, Suzhou (China), November 2025. Association for Computational Linguistics.
- [27] Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. Agentdropout: Dynamic agent elimination for token-efficient and high-performance llm-based multi-agent collaboration. *arXiv preprint arXiv:2503.18891*, 2025.
- [28] Qingyun Wu, Gagan Bansal, Jieyu Zhang, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.
- [29] Shengxiang Xu, Jiayi Zhang, Shimin Di, Yuyu Luo, Liang Yao, Hanmo Liu, Jia Zhu, Fan Liu, and Min-Ling Zhang. Robustflow: Towards robust agentic workflow generation. *arXiv preprint arXiv:2509.21834*, 2025.
- [30] An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [31] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.
- [32] Qingwen Yang, Feiyu Qu, Tiezheng Guo, Yanyi Liu, and Yingyou Wen. Augmented runtime collaboration for self-organizing multi-agent systems: A hybrid bi-criteria routing approach. *arXiv preprint arXiv:2512.00740*, 2025.
- [33] Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, and Weinan Zhang. Agentnet: Decentralized evolutionary coordination for llm-based multi-agent systems. *arXiv preprint arXiv:2504.00587*, 2025.
- [34] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, Mark Boyer, and Tie-Yan Liu. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021.

- [35] Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. Masrouter: Learning to route llms for multi-agent systems. *arXiv preprint arXiv:2502.11133*, 2025.
- [36] Ceyao Zhang, Kaijie Yang, Siyi Hu, et al. Proagent: building proactive cooperative agents with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17591–17599, 2024.
- [37] Enhao Zhang, Erkang Zhu, Gagan Bansal, Adam Fourney, Hussein Mozannar, and Jack Gerrits. Optimizing sequential multi-step tasks with parallel llm agents. *arXiv preprint arXiv:2507.08944*, 2025.
- [38] Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. *ICLR*, 2025.
- [39] Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*, 2025.
- [40] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, et al. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025.

A Proofs of Theoretical Properties

Here we provide formal proofs for the properties in the main text.

A.1 Proof of Theorem 3.4

Proof. We analyze the two operations separately:

1. Node Removal (of k): Let $k \in \mathcal{V} \setminus \{v_I, v_O\}$ and $\mathcal{V}' = \mathcal{V} \setminus \{k\}$. We consider any $u, v \in \mathcal{V}'$.

($\implies$) *Forward Direction:* Assume $u \rightsquigarrow_G v$. Let P be a path from u to v in G .

- If $k \notin P$, then P consists entirely of nodes in $\mathcal{V}'$ and edges unchanged by the operation. Thus, P exists in G' .
- If $k \in P$, the path contains a segment $x \rightarrow k \rightarrow y$. The removal operation explicitly adds the shortcut edge $x \rightarrow y$ to G' whenever $x \rightarrow k$ and $k \rightarrow y$ exist in G . We can replace every occurrence of $x \rightarrow k \rightarrow y$ with the single edge $x \rightarrow y$, yielding a valid path in G' .

($\impliedby$) *Reverse Direction:* Assume $u \rightsquigarrow_{G'} v$. Let P' be a path in G' . Edges in G' are either original edges from G or newly created shortcut edges.

- If an edge $(x, y) \in P'$ exists in G , it represents the direct connection $x \rightarrow y$.
- If an edge $(x, y) \in P'$ is a new shortcut not in G , it implies that x was a predecessor of k and y was a successor of k in G . Thus, this edge corresponds to the path segment $x \rightarrow k \rightarrow y$ in G .

2. Parallel Rewiring (of edge $i \rightarrow j$): We consider $u, v \in \mathcal{V} \setminus \{i, j\}$. Note that the operation removes $i \rightarrow j$ and updates connectivity such that j inherits i 's predecessors (incoming merge) and i inherits j 's successors (outgoing merge).

($\implies$) *Forward Direction:* Assume $u \rightsquigarrow_G v$. Let P be a path in G .

- If $(i, j) \notin P$, the path from u to v remains intact in G' since only edge $i \rightarrow j$ is removed.
- If $(i, j) \in P$, the path from u to v contains a segment $x \rightarrow i \rightarrow j \rightarrow y$. In G' , the operation sets $\mathbf{A}'_{xj} \leftarrow 1$ (since $x \rightarrow i$) and $\mathbf{A}'_{iy} \leftarrow 1$ (since $j \rightarrow y$). We can bypass the removed edge by using the new edge $x \rightarrow j$, creating a path $\dots \rightarrow x \rightarrow j \rightarrow y \rightarrow \dots$ in G' .

($\impliedby$) *Reverse Direction:* Assume $u \rightsquigarrow_{G'} v$. Let P' be a path in G' . Any edge in P' is either an original edge or a newly formed edge due to merging neighborhoods.

- A new edge of type $x \rightarrow j$ exists in G' only if $x \rightarrow i$ existed in G . Thus, $x \rightarrow j$ in P' corresponds to the segment $x \rightarrow i \rightarrow j$ in G .
- A new edge of type $i \rightarrow y$ exists in G' only if $j \rightarrow y$ existed in G . Thus, $i \rightarrow y$ in P' corresponds to the segment $i \rightarrow j \rightarrow y$ in G .

Since $u, v \neq i, j$, the path must start and end outside the rewired edge. By expanding any new edges back to their original segments involving $i \rightarrow j$, we recover a valid path in G . $\square$

A.2 Proof of Theorem 3.5

Proof. Let $\mathcal{P}_G(v_I, v_O)$ denote the set of all directed execution paths from the entry node v_I to the exit node v_O in graph G . It suffices to show that for every path $P' \in \mathcal{P}_{G'}(v_I, v_O)$ in the edited graph, there exists a corresponding path $P \in \mathcal{P}_G(v_I, v_O)$ in the original graph such that

$$\sum_{v \in P'} t_v \leq \sum_{v \in P} t_v.$$

(i) Node Removal. Suppose a non-terminal node $k \in \mathcal{V} \setminus \{v_I, v_O\}$ is removed. For any path $P' \in \mathcal{P}_{G'}(v_I, v_O)$:

- If P' uses only original edges from G , then the same path exists in G , and the path duration is identical.
- If P' uses a shortcut edge (u, v) introduced by removing k , then G contains the segment $u \rightarrow k \rightarrow v$. Replacing every such shortcut in P' by its original two-edge segment yields a valid path P in G , with

$$\sum_{v \in P'} t_v \leq \sum_{v \in P} t_v,$$

since $t_k > 0$.

(ii) Parallel Rewiring. Suppose the edge (i, j) is rewired. For any path $P' \in \mathcal{P}_{G'}(v_I, v_O)$:

- If P' uses only original edges from G , then the same path exists in G .
- Otherwise, every edge newly introduced by the rewiring is either of the form $x \rightarrow j$ where $x \rightarrow i$ existed in G , or of the form $i \rightarrow y$ where $j \rightarrow y$ existed in G . We can replace each new edge $x \rightarrow j$ by the segment $x \rightarrow i \rightarrow j$, and each new edge $i \rightarrow y$ by the segment $i \rightarrow j \rightarrow y$. This recovers a valid path P in G and only inserts nodes with positive latency. Hence $\sum_{v \in P'} t_v \leq \sum_{v \in P} t_v$.

Thus, every execution path in G' is no longer than a corresponding path in G . Taking the maximum over paths in G' yields

$$T(G') = \max_{P' \in \mathcal{P}_{G'}(v_I, v_O)} \sum_{v \in P'} t_v \leq \max_{P \in \mathcal{P}_G(v_I, v_O)} \sum_{v \in P} t_v = T(G).$$

□

A.3 Proof of Theorem 3.6

Proof. The transformation from an arbitrary chain to the target graph G^* is achieved via a three-stage procedure. We prove the validity of each stage sequentially.

Stage 1: Topological Alignment (via Permutation).

Let $\sigma : \mathcal{V} \rightarrow \mathcal{V}$ be a permutation such that applying the operator Π_σ to the initial chain yields a sequence where, for every directed edge (u, v) in G^* , node u precedes v . This condition ensures that the linear ordering defined by the permuted chain is consistent with the **topological sort** of G^* [13].

Stage 2: Node Alignment (via Removal).

If the initial chain contains nodes not required by G^* , we apply *Node Removal* to eliminate all extraneous nodes. By reachability preservation, removing nodes does not alter the relative reachability among the remaining nodes. After this stage, the graph is a chain defined exactly on the target node set $\mathcal{V}$, with an ordering consistent with G^* . Therefore, the permuted chain after the node permutation is the topological sort of G^* . Without loss of generality, we assign node indices according to this ordered chain, starting from v_1 to v_n .

Stage 3: Exact Edge Reconstruction (via Rewiring)

With the nodes correctly aligned and ordered, we employ a *Reverse Sweeping* strategy to recover the exact edge structure. We first define the construction, then prove its correctness.

I. The Construction. We execute the following iterative procedure to refine the graph structure:

For each node v_i iterating from v_{n-1} down to v_1 (reverse from end to start):

- Let $\mathcal{N}'_{out}(v_i)$ denote the current set of immediate successors of v_i .
- **While** there exists a neighbor $v_k \in \mathcal{N}'_{out}(v_i)$ such that (v_i, v_k) not in G^* :
 - **Do:** Apply *Parallel Rewiring* to the edge (v_i, v_k) , which updates:

$$\mathcal{N}'_{out}(v_i) \leftarrow (\mathcal{N}'_{out}(v_i) \setminus \{v_k\}) \cup \mathcal{N}'_{out}(v_k)$$

- **End While** (All remaining immediate successors are now valid target neighbors).

End For

II. Correctness Verification. We prove that this process yields $G' \cong G^*$ using structural induction on the node index i (descending from n to 1).

Inductive Hypothesis: Assume that for all nodes v_j in the downstream of v_i ($j > i$), the outgoing edges are already identical to the target graph: $\mathcal{N}'_{out}(v_j) = \mathcal{N}^*_{out}(v_j)$. (Base case v_n is trivial).

Inductive Step: Consider node v_i . We verify the final state of $\mathcal{N}'_{out}(v_i)$ when the rewiring loop for v_i terminates.

1. **No False Positives:** The loop condition explicitly mandates the removal of any edge (v_i, v_k) that does not exist in G^* . Thus, $\mathcal{N}'_{out}(v_i) \subseteq \mathcal{N}^*_{out}(v_i)$.
2. **No False Negatives:** Consider any valid target neighbor $v_m \in \mathcal{N}^*_{out}(v_i)$. For the sake of contradiction, suppose that v_m is **absent** from the current immediate successor set $\mathcal{N}'_{out}(v_i)$.
 - Since no Parallel Rewiring operation has been applied to (v_i, v_m) , the reachability is strictly preserved (Theorem 3.4), the absence of a direct link implies that v_m must be reachable via some intermediate neighbor $v_k \in \mathcal{N}'_{out}(v_i)$.
 - Because the loop has terminated, v_k is confirmed to be a valid target neighbor (i.e., (v_i, v_k) exists in G^*).
 - Furthermore, since $k > i$, the inductive hypothesis guarantees that the path segment $v_k \rightarrow \dots \rightarrow v_m$ exists in G^* .
 - Consequently, G^* contains both the direct edge (v_i, v_m) and the indirect path $v_i \rightarrow v_k \rightarrow \dots \rightarrow v_m$. This structure violates the **transitive reduction** property of G^* .
 - Thus, the assumption is false. No such blocking node v_k can exist, and v_m must be present in $\mathcal{N}'_{out}(v_i)$.

Thus, $\mathcal{N}'_{out}(v_i) = \mathcal{N}^*_{out}(v_i)$. By induction, after the loop terminates at v_1 , the graph G' is exactly isomorphic to G^* . $\square$

B Derivation of the Parallel Rewiring Score

Let $\mathbf{A} \in \{0, 1\}^{n \times n}$ denote the adjacency matrix of the workflow graph, and let $\mathbf{\Gamma} = \frac{\partial S_\theta}{\partial \mathbf{A}} \in \mathbb{R}^{n \times n}$ be the gradient of the Graph Evaluator score S_θ with respect to $\mathbf{A}$. For a graph edit that induces a perturbation $\Delta \mathbf{A}$, a first-order Taylor expansion yields

$$\Delta S \approx \langle \mathbf{\Gamma}, \Delta \mathbf{A} \rangle = \sum_{u,v} \mathbf{\Gamma}_{uv} \Delta \mathbf{A}_{uv}. \quad (16)$$

Rewiring an edge (i, j) removes the original edge $i \rightarrow j$ and merges the outgoing neighbors of j into i , as well as the incoming neighbors of i into j . Accordingly, the perturbation $\Delta \mathbf{A}$ consists of three parts:

- Removal of the edge (i, j) , giving $\Delta \mathbf{A}_{ij} = -\mathbf{A}_{ij}$;
- Addition of edges (i, k) for all k such that $\mathbf{A}_{jk} = 1$ and $\mathbf{A}_{ik} = 0$;
- Addition of edges (k, j) for all k such that $\mathbf{A}_{ki} = 1$ and $\mathbf{A}_{kj} = 0$.

Substituting these changes into (16) gives

$$\begin{aligned} \Delta S(i, j) &\approx -\mathbf{\Gamma}_{ij} \mathbf{A}_{ij} \\ &\quad + \sum_k \mathbf{\Gamma}_{ik} \mathbf{A}_{jk} (1 - \mathbf{A}_{ik}) \\ &\quad + \sum_k \mathbf{\Gamma}_{kj} \mathbf{A}_{ki} (1 - \mathbf{A}_{kj}), \end{aligned} \quad (17)$$

which is the stated first-order estimate of the score change induced by rewiring (i, j) .

For compactness, define $\mathbf{H} = \mathbf{\Gamma} \odot \mathbf{A}$ and $\tilde{\mathbf{H}} = \mathbf{\Gamma} \odot (1 - \mathbf{A})$. The three terms in (17) can then be written in matrix form, yielding the parallel-rewiring score matrix

$$\mathbf{R}^{\text{rwr}} = -\mathbf{H} + \tilde{\mathbf{H}} \mathbf{A}^\top + \mathbf{A}^\top \tilde{\mathbf{H}}, \quad (18)$$

where each entry $\mathbf{R}^{\text{rwr}}_{i,j}$ equals $\Delta S(i, j)$ and provides the first-order estimated score change for rewiring edge (i, j) .

C Details in Experiment Settings

Agent Settings. We define specific agent pools for each domain to support specialized reasoning.

For the **Code Generation** domain (MBPP, HumanEval), the available agents include: AlgoPlanner, CodeGenerator, CodeAuditor, ConceptualAlgorithmist, LegacyPythonCoder, ComplianceChecker, Generalists and STEM_Expert. For the **Mathematical Reasoning** domain (GSM8K, MATH500, SVAMP, AQuA), the candidate agents are: MathReasoner, Calculator, STEM_Expert, ProblemSimplifier, HeuristicMathSolver, Generalists, ConsensusBuilder, ExecutiveArbitrator, ContrarianThinker, Verifier, and LateralThinkingAnalyst. For the **General Knowledge** domain (MMLU), the agent pool consists of: LateralThinkingAnalyst, SocialScientist, Generalists, STEM_Expert, MathReasoner, TheoreticalScientist, CriticalTheorist, HeuristicMathSolver, ContrarianThinker, and ConsensusBuilder.

Serving Environment. For wall-clock makespan measurements, each run uses a freshly launched and warmed-up SGLang server on an NVIDIA RTX PRO 6000 Blackwell Max-Q Workstation Edition GPU (96GB memory), with exclusive GPU allocation during measurement.

Graph Dataset Construction. To train the Graph Evaluator and the Graph Generator for each task, we construct benchmark-specific graph datasets. For each benchmark, we generate a fixed pool of $N = 2048$ random graphs using the agent candidates from the corresponding task domain. To obtain the ground-truth performance labels (scores) for each graph G , we evaluate G on a subset of task samples from that benchmark’s training set. Specifically, we use $k = 50$ samples for mathematical reasoning datasets (GSM8K, MATH500, SVAMP, AQuA) and $k = 30$ samples for code generation (HumanEval, MBPP) and general knowledge (MMLU) datasets. Since HumanEval and MATH500 do not provide official training splits, we use equivalent training proxies: for HumanEval, we build HumanEval-compatible training samples from the MBPP train split; for MATH500, we sample from the MATH train split (levels 4–5) while excluding instances that overlap with the MATH500 test set. The average accuracy across these k samples serves as the target label for training. We also use the same samples for the training-based baselines.

LAWA Parameters. For the optimization phases, we use fixed beam size of $B = 32$. The maximum number of optimization steps (iterations) is set to $K = 11$ for the generation experiments and $K = 4$ for the plugin-experiments due to reduced workflow depth.

Our implementation and scripts for reproducing the experiments are available at <https://github.com/GraphmindDartmouth/LAWA>.

D Case Study: Optimization Trajectory on MBPP.

To illustrate the step-by-step optimization behavior of LAWA, we start from a chain-structured workflow on MBPP and apply LAWA with beam size $B = 1$, so that it will only modify the single best candidate graph of the last iteration. Figure 3 illustrates the optimization trajectory. The original workflow achieves an accuracy of 63.9% and exhibits a fully sequential dependency pattern, resulting in a long critical path.

In the first step, LAWA applies parallel rewiring between Generalists and AlgoPlanner, exposing latent parallelism without altering the agent set. This change shortens the critical path and yields a slight accuracy improvement to 64.1%.

In the second step, LAWA removes the second ComplianceChecker node. This node contributes redundant computation along the critical path, and its removal further reduces the makespan while increasing accuracy to 65.6%.

In the third step, LAWA performs parallel rewiring between ComplianceChecker and CodeAuditor. While this modification slightly reduces accuracy to 65.2%, it eliminates a remaining sequential dependency and reduces the runtime contribution of ComplianceChecker.

Among these steps, LAWA incrementally compresses the critical path while maintaining stable task performance.

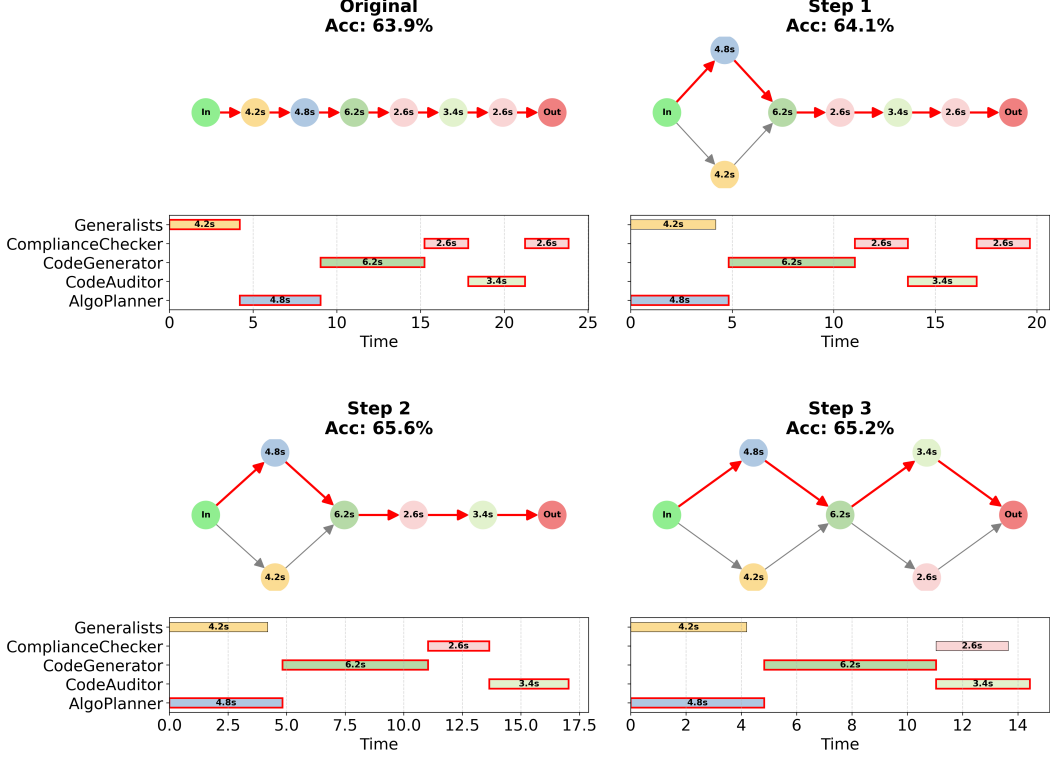


Figure 3: **Example of iterative workflow optimization on MBPP.** We visualize the original chain-structured workflow (top-left) and the resulting workflows after 1, 2, and 3 optimization steps (top-right, bottom-left, bottom-right, respectively). For each panel, the upper plot shows the workflow graph, where red arrows denote the critical path, and the lower plot shows the corresponding execution Gantt chart with critical-path agent executions highlighted (both using estimated t_k). As LAWA applies successive structural modifications, the workflow topology departs from the initial chain, progressively shortening the critical path and reducing makespan.

E Probabilistic Robustness Analysis under Evaluator Perturbations

We provide a probabilistic justification for the robustness behavior observed in RQ3 under evaluator perturbations. Let $\mathcal{M}$ denote the set of valid candidate graphs generated during optimization, where each $G \in \mathcal{M}$ satisfies the true constraint $S(G) \geq S(G_0)$, and let $\mathcal{U}$ denote the set of invalid candidates with $S(G) < S(G_0)$. Define the top-tier valid set as

$$\mathcal{M}^* = \{v \in \mathcal{M} \mid T(v) < \min_{u \in \mathcal{U}} T(u)\},$$

which contains valid candidates whose makespan is smaller than any invalid candidate. The optimization procedure outputs

$$G^* = \arg \min_{G \in \mathcal{W}} T(G),$$

where

$$\mathcal{W} = \{G \in \mathcal{M} \cup \mathcal{U} \mid S_\theta(G) \geq S_\theta(G_0)\}$$

is the retained set under the Graph Evaluator. Assume an otherwise accurate evaluator whose keep/reject decision at Algorithm 1 (line 12) is independently perturbed (flipped) with probability α . Then two events must simultaneously happen for an invalid final output: (i) at least one invalid candidate is falsely kept, and (ii) all top-tier valid candidates are falsely rejected. Conditioned on

Table 6: **Behavior of LAWA under limited executors.** Here m is the number of available executors; runtime improvement is the relative reduction versus the initial workflow; operation counts are total rewiring/node-removal edits aggregated over all generated instances (7 benchmarks $\times$ 3 runs).

m	Runtime Improvement (%)	Operation Counts (rewire/remove)
1	43.67	19/64
2	65.38	83/61
3	67.74	91/59
4	67.89	92/59

Table 7: **Accuracy robustness under rewiring-score approximation on three benchmarks.** We compare LAWA with and without first-order approximation when filtering rewiring edits. Values are accuracy (%), mean $_{\pm\text{std}}$ over 3 runs. Results show that the approximation does not introduce a noticeable drop in accuracy.

Setting	MATH500	HumanEval	AQuA
w/o approximation	67.00 $\pm$ 0.71	93.29 $\pm$ 0.00	83.86 $\pm$ 0.56
w/ approximation	67.00 $\pm$ 0.86	92.89 $\pm$ 1.04	83.46 $\pm$ 0.85

$(\mathcal{U}, \mathcal{M}^*)$, we have

$$\begin{aligned}
\Pr(G^* \in \mathcal{U} \mid \mathcal{U}, \mathcal{M}^*) &\leq \Pr(\exists u \in \mathcal{U}, u \in \mathcal{W}) \cdot \Pr(\forall v \in \mathcal{M}^*, v \notin \mathcal{W}) \\
&= \left(1 - (1 - \alpha)^{|\mathcal{U}|}\right) \alpha^{|\mathcal{M}^*|} \\
&\leq \left(1 - (1 - \alpha)^{\bar{N}}\right) \alpha^{|\mathcal{M}^*|},
\end{aligned} \tag{19}$$

where $\bar{N}$ is an upper bound on the number of candidates.

Let $p_0 = \Pr(|\mathcal{M}^*| = 0)$. Marginalizing over candidate sets yields

$$\begin{aligned}
\Pr(G^* \in \mathcal{U}) &= \sum_{\mathcal{U}, \mathcal{M}^*} \Pr(G^* \in \mathcal{U} \mid \mathcal{U}, \mathcal{M}^*) \Pr(\mathcal{U}, \mathcal{M}^*) \\
&\leq \left(1 - (1 - \alpha)^{\bar{N}}\right) \sum_{\mathcal{M}^*} \alpha^{|\mathcal{M}^*|} \Pr(\mathcal{M}^*) \\
&= \left(1 - (1 - \alpha)^{\bar{N}}\right) \left(\sum_{k=0}^{\infty} \Pr(|\mathcal{M}^*| = k) \alpha^k\right) \\
&\leq \left(1 - (1 - \alpha)^{\bar{N}}\right) (p_0 + (1 - p_0)\alpha).
\end{aligned} \tag{20}$$

Therefore, the probability of an invalid final output is explicitly determined by α . As $\alpha \rightarrow 0$, the upper bound also approaches 0, consistent with RQ3, where performance remains stable under mild perturbations.

F More Experiment Results

F.1 Token Consumption

To examine whether parallel rewiring increases context-broadcasting overhead, we report total token consumption (input + output) of the generated workflows for all methods under the same final setting. As shown in Table 8, LAWA consistently remains token-efficient. The token consumption is substantially lower than heavily parallel baselines such as DyLAN. This is because total token usage is largely determined by the number of agent invocations, and LAWA removes redundant nodes during optimization.

F.2 Optimization Runtime

To better characterize computational overhead of LAWA, we report both (i) wall-clock time to generate one workflow and (ii) Graph Evaluator training time. Unless otherwise stated, runtime measurements in this section are conducted on a machine with an AMD EPYC 9655 96-Core Processor CPU. As

Table 8: **Total token consumption per query (Tokens↓) on seven benchmarks.** We report total token consumption (input + output) for all methods. Values are mean $\pm$ std over three runs, and the best value in each benchmark is **bolded**.

Method	MBPP	HumanEval	AQuA	GSM8K	MATH500	SVAMP	MMLU
Rand. Graph	22768 $\pm$ 1949	34403 $\pm$ 4326	64483 $\pm$ 2350	40409 $\pm$ 8685	57252 $\pm$ 8018	40825 $\pm$ 1684	48050 $\pm$ 6204
Rand. Chain	40289 $\pm$ 4229	49315 $\pm$ 3638	99064 $\pm$ 11753	83317 $\pm$ 446	103547 $\pm$ 5895	71156 $\pm$ 1235	73758 $\pm$ 821
3-Round Disc.	219103 $\pm$ 7310	243745 $\pm$ 3320	595536 $\pm$ 12769	500286 $\pm$ 2419	646475 $\pm$ 21625	445061 $\pm$ 7785	470002 $\pm$ 1510
DyLAN	55388 $\pm$ 3921	72015 $\pm$ 5500	85396 $\pm$ 9330	41915 $\pm$ 20638	98495 $\pm$ 3777	62078 $\pm$ 2553	81276 $\pm$ 1301
G-Designer	37128 $\pm$ 2550	44954 $\pm$ 1157	88371 $\pm$ 4124	80719 $\pm$ 3347	99698 $\pm$ 4556	70241 $\pm$ 5089	67340 $\pm$ 2960
AgentDropout	19870 $\pm$ 9307	27288 $\pm$ 10278	54616 $\pm$ 5328	29668 $\pm$ 7334	46292 $\pm$ 7966	41654 $\pm$ 4011	46553 $\pm$ 10920
AgentPrune	34834 $\pm$ 5788	46360 $\pm$ 222	88365 $\pm$ 3343	75535 $\pm$ 1079	99848 $\pm$ 3995	59406 $\pm$ 5280	62790 $\pm$ 5324
BiRouter	11338 $\pm$ 2789	16150 $\pm$ 3886	14843 $\pm$ 3639	9344 $\pm$ 929	15100 $\pm$ 3514	8448 $\pm$ 227	14763 $\pm$ 3666
AFlow	11865 $\pm$ 3037	16532 $\pm$ 3825	35135 $\pm$ 4172	23594 $\pm$ 2120	35469 $\pm$ 4828	22166 $\pm$ 2675	24833 $\pm$ 4578
LAMaS	7898 $\pm$ 150	11705 $\pm$ 240	14047 $\pm$ 3976	15841 $\pm$ 6533	17849 $\pm$ 2544	13766 $\pm$ 2070	15012 $\pm$ 103
LAWA	6516 $\pm$ 30	11401 $\pm$ 4099	10796 $\pm$ 4121	10461 $\pm$ 1862	12207 $\pm$ 3152	7615 $\pm$ 2304	15077 $\pm$ 2897

Table 9: **Wall-clock time to generate a single workflow (Seconds↓).**

Benchmark	Time (s)
MBPP	2.2736 $\pm$ 0.2287
HumanEval	1.1563 $\pm$ 0.4062
AQuA	1.6917 $\pm$ 0.7676
GSM8K	2.2471 $\pm$ 0.1613
MATH500	1.7981 $\pm$ 0.1430
SVAMP	2.2039 $\pm$ 0.2768
MMLU	2.1496 $\pm$ 0.1832

shown in Table 9, workflow generation typically takes only around 1–2.3 seconds because the search operates only on graph structure without executing real agent calls. Compared with high-latency baseline workflows in Table 1, this search overhead is small relative to recurrent execution costs.

As shown in Table 10, Graph Evaluator training takes 21–95 seconds per benchmark and is performed once per benchmark as an offline preprocessing step. Therefore, when LAWa is used repeatedly to replace high-latency workflows, this one-time overhead can be easily amortized over subsequent executions. For example, using Random Chain as a representative high-latency initial workflow, the one-time evaluator-training and search cost is offset after roughly 2–7 recurrent executions across benchmarks.

Cost of graph-dataset construction. On MATH500, constructing the full graph dataset (evaluating 2048 randomly generated workflows on 50 training samples) takes 38.7 minutes of wall-clock time on 8 NVIDIA RTX PRO 6000 GPUs. This is a one-time, fully parallelizable offline stage: unlike on-policy baselines, whose data collection is sequential and repeated during optimization, LAWa’s workflow evaluations have no sequential dependency. On the same server, the complete LAWa pipeline (dataset generation, training, and workflow generation) takes 41.2 minutes, whereas AgentDropout requires 129.6 minutes under the same 2048-rollout budget (64 iterations $\times$ batch size 32). The gap arises because evaluating 2048 workflows in one pass fully utilizes the inference server: it costs only about 20 $\times$ the time of evaluating 32 workflows, despite processing 64 $\times$ as many workflows. Thus, even after including graph-dataset construction, LAWa takes about one third of the wall-clock time of this on-policy baseline.

F.3 Frontier Backbone Model

We additionally evaluate the officially hosted frontier model Kimi-K2.6 on MBPP, MATH500, and MMLU, comparing LAWa with DyLAN and AgentDropout. For LAWa, we reuse the original Graph Evaluator, Graph Generator, and estimated t_k values, without retraining the learned modules or recalibrating the latency estimates. As shown in Table 11, LAWa’s performance advantage is preserved with the stronger backbone: it remains best on MBPP and MMLU, and is on par with DyLAN on MATH500. A stronger backbone makes each individual agent more capable, so a compact workflow can attain high accuracy with fewer agents, which favors LAWa’s ability to remove redundant agents. LAWa also remains the fastest method on all three benchmarks.

Table 10: Wall-clock time to train the Graph Evaluator (Seconds $\downarrow$).

Benchmark	Train Time (s)
MBPP	44.45
HumanEval	21.32
AQuA	63.34
GSM8K	34.65
MATH500	46.47
SVAMP	23.32
MMLU	95.08

Table 11: Accuracy (% $\uparrow$) and real wall-clock makespan (s $\downarrow$) with Kimi-K2.6. Values are mean $\pm$ std over three workflow graphs.

Method	MBPP		MATH500		MMLU	
	Acc.	Time	Acc.	Time	Acc.	Time
DyLAN	73.41 $\pm$ 0.22	40.27 $\pm$ 5.92	81.20 $\pm$ 1.25	32.01 $\pm$ 9.53	80.73 $\pm$ 1.80	33.07 $\pm$ 1.03
AgentDropout	72.24 $\pm$ 0.59	43.66 $\pm$ 2.37	79.27 $\pm$ 2.89	43.34 $\pm$ 1.98	79.82 $\pm$ 3.61	52.66 $\pm$ 13.15
LAWA	73.80 $\pm$ 0.81	14.19 $\pm$ 1.21	81.27 $\pm$ 1.22	18.43 $\pm$ 2.99	81.74 $\pm$ 1.77	19.75 $\pm$ 2.10

F.4 Finite-Resource Scheduling

We study how LAWAs editing policy changes when the available parallel resources are constrained. During graph editing, we replace the default critical-path estimator (Equation (1)) with an HLFET list-scheduling runtime estimator [1] and vary the number of executors $m \in \{1, 2, 3, 4\}$. Runtime improvement is measured as the relative reduction against the initial workflow, and operation counts are aggregated over all generated instances (7 benchmarks $\times$ 3 runs). As reported in Table 6, the optimization behavior shows a clear transition: under single-executor constraints, LAWAs mainly shortens the critical chain through node removal, while with more executors it increasingly prefers rewiring to unlock parallel branches; correspondingly, runtime gains improve quickly at first and then gradually plateau as available parallelism approaches saturation. This indicates that, even under limited executors, LAWAs can adapt its editing strategy to the available resources.

F.5 Robustness under First-Order Approximation

We examine whether the first-order gradient approximation used to filter rewiring operations harms final optimization quality. We compare default LAWAs (with approximation) against a variant that disables approximation and computes rewiring-score changes exactly during candidate generation, then evaluate final accuracy on MATH500, HumanEval, and AQuA. As shown in Table 7, the performance differences are minor across benchmarks, indicating that the approximation does not introduce a noticeable drop in final accuracy while helping reduce the number of candidate graphs.

F.6 Effectiveness of the Graph Evaluator

We conduct an ablation study to examine whether the Graph Evaluator stabilizes LAWAs iterative graph editing process. Starting from the same randomly initialized chain workflows, we compare the default LAWAs against a variant without evaluator-based filtering, which keeps candidates only according to makespan. Table 12 reports the downstream-performance change across optimization steps on two representative benchmarks. With the Graph Evaluator, LAWAs yields positive mean performance changes across optimization steps, whereas removing the evaluator leads to less stable performance and can result in degradation. These results suggest that the Graph Evaluator helps retain downstream task quality while the structural optimizer reduces latency.

F.7 Graph Evaluator Architecture Sensitivity

We evaluate how sensitive the Graph Evaluator is to GNN architecture choices and justify the default backbone used in LAWAs. On HumanEval, we run a full ablation over GNN type (gcn, gat, transformer), depth (1–4 layers), and hidden size (256/512), while keeping all other training settings fixed and reporting train/test accuracy over 3 seeds. Table 13 indicates that GCN performs

Table 12: **Performance change across optimization steps with and without the Graph Evaluator.** Values are downstream-performance changes (% , mean $\pm$ std) relative to the initial workflow, computed over eight different random initial graphs.

Dataset	Method	Step 1	Step 2	Step 3	Step 4	Step 5
HumanEval	LAWA (w/ Evaluator)	+1.68 ± 1.99	+2.21 ± 2.97	+2.29 ± 3.22	+2.44 ± 3.37	+1.75 ± 2.54
HumanEval	w/o Evaluator	-0.30 ± 2.14	-0.15 ± 2.82	-0.07 ± 2.26	-0.53 ± 2.12	-2.74 ± 1.72
MATH500	LAWA (w/ Evaluator)	+0.95 ± 0.60	+1.65 ± 1.09	+1.25 ± 1.17	+1.65 ± 1.14	+1.35 ± 1.23
MATH500	w/o Evaluator	-0.20 ± 0.63	-0.35 ± 1.19	-0.45 ± 1.19	-0.63 ± 1.34	-1.72 ± 1.51

most consistently across configurations, while the performance of GAT and Transformer varies more with depth and hidden size. The 3-layer Transformer with hidden dimension 512 achieves both train and test accuracy above 95%, and Graph Transformers are generally more suited to capture the long-range interactions that may arise from multi-hop dependencies in workflow graphs [34]. Therefore, we use a 3-layer Transformer with hidden dimension 512 as the default Graph Evaluator.

G Limitations and Future Works

This work considers a static agent pool and a fixed set of downstream tasks, a standard setting that is widely adopted in prior work. In practice, agent availability and task distributions may evolve over time. Extending the framework to more dynamic scenarios is a promising direction for future work.

In dynamic environments, the set of available agents may change over time, leading to changes in the feasible workflows and the associated makespan. Extending the framework to accommodate a dynamic agent pool would require the workflow to adapt to such changes while maintaining efficiency. Similarly, downstream tasks may evolve over time, potentially altering the desirable workflow patterns. Developing mechanisms that allow the evaluator and workflow to adapt to such shifts in a data-efficient and incremental manner, rather than being rebuilt from scratch, is another important direction for future work.

Table 13: **GNN architecture ablation on HumanEval for Graph Evaluator.** We vary GNN type (gcn, gat, transformer), number of layers ($\{1,2,3,4\}$), and hidden dimension ($\{256,512\}$), while keeping other training settings fixed. Values are train/test accuracy (% , mean $\pm$ std) over 3 seeds. Rows where both train and test mean exceed 95 are bolded.

Hidden Dim	GNN Type	#Layers	Train Acc (%)	Test Acc (%)
256	gcn	1	94.68 $\pm$ 0.68	94.79 $\pm$ 3.35
256	gcn	2	94.63 $\pm$ 0.49	95.17 $\pm$ 4.28
256	gcn	3	95.05 $\pm$ 0.44	95.14 $\pm$ 2.58
256	gcn	4	94.93 $\pm$ 0.20	95.77 $\pm$ 3.30
256	gat	1	94.97 $\pm$ 0.73	94.47 $\pm$ 3.27
256	gat	2	95.38 $\pm$ 1.27	94.00 $\pm$ 3.74
256	gat	3	94.22 $\pm$ 1.63	93.57 $\pm$ 3.74
256	gat	4	95.00 $\pm$ 0.77	92.44 $\pm$ 3.87
256	transformer	1	94.18 $\pm$ 0.58	93.47 $\pm$ 4.36
256	transformer	2	95.35 $\pm$ 1.47	94.43 $\pm$ 4.35
256	transformer	3	94.75 $\pm$ 1.70	94.03 $\pm$ 4.14
256	transformer	4	93.60 $\pm$ 1.58	94.00 $\pm$ 2.90
512	gcn	1	94.92 $\pm$ 1.39	95.11 $\pm$ 3.80
512	gcn	2	95.43 $\pm$ 0.68	95.45 $\pm$ 2.94
512	gcn	3	95.07 $\pm$ 0.80	94.17 $\pm$ 4.03
512	gcn	4	94.85 $\pm$ 0.58	95.28 $\pm$ 1.95
512	gat	1	95.00 $\pm$ 1.54	94.90 $\pm$ 2.21
512	gat	2	95.13 $\pm$ 1.16	93.40 $\pm$ 4.35
512	gat	3	95.13 $\pm$ 1.09	93.82 $\pm$ 2.16
512	gat	4	95.70 $\pm$ 0.82	93.25 $\pm$ 3.89
512	transformer	1	93.12 $\pm$ 1.45	91.67 $\pm$ 2.64
512	transformer	2	93.23 $\pm$ 0.35	89.93 $\pm$ 6.86
512	transformer	3	95.97 $\pm$ 1.29	95.11 $\pm$ 4.39
512	transformer	4	95.28 $\pm$ 3.68	94.68 $\pm$ 4.06