MermaidFlow-CF: How Agentic Workflow Representation Governs Constraint-Faithful Control

Keyi Xiang[1], Boyuan Shi[2], Yueming Lyu[3], Jianda Chen[1], Ivor Tsang [3], Haiyan Yin[3]*

[1]Nanyang Technological University (NTU), Singapore
[2]National University of Singapore (NUS), Singapore
[3]CFAR and IHPC,Agency for Science, Technology and Research (A*STAR),Singapore
keyi003@e.ntu.edu.sg, e1597858@u.nus.edu, jianda001@ntu.edu.sg, {lyu_yueming, ivor_tsang, yin_haiyan}@a-star.edu.sg

Abstract

Agentic workflows coordinate LLM agents to autonomously generate, refine, and execute multi-step pipelines, yet maintaining reliable control over instruction-following behavior remains challenging, often resulting in cascading workflow failures. These failures frequently stem from *unconstrained* workflow synthesis, where structural drift and broken control flow accumulate over time. In this paper, we show that a key driver of this brittleness is *workflow representation*, which determines whether planning structure and control flow can be preserved during generation, evaluation, and execution. We introduce **MermaidFlow-CF**, a *constraint-faithful* workflow optimization framework that represents workflows in a symbolic graph DSL **Mermaid**[1], enabling symbolic control-flow syntax that renders planning structure explicit and supports interpretable and checkable workflow generation. Building on this formulation, we formalize the **constrained workflow optimization problem** and introduce a **structured taxonomy of workflow constraints** spanning *resource feasibility, executability, structural validity*, and *causal coherence*. We further develop an evaluation protocol to measure constraint violation and correction dynamics for the constraints. Across multi-step reasoning benchmarks, MermaidFlow-CF achieves significantly higher constraint fidelity and markedly fewer cascading failures than AFlow, a *Python*-based workflow optimization baseline. These results show that symbolic workflow representations in Mermaid provide a more reliable foundation for agentic pipelines than Python, and that constraints function not as barriers but as structural priors that shape optimization dynamics and enable more stable, higher-performance optimization in agentic workflow planning.

Introduction

Large Language Models (LLMs) have demonstrated strong reasoning and problem-solving capabilities across domains (OpenAI 2025; DeepSeek-AI et al. 2025; Anthropic 2025). Building on this progress, agentic AI extends LLMs beyond single-turn inference toward autonomous planning, tool use, and coordinated multi-agent interaction (Anonymous 2025; Wang et al. 2024a; Yao et al. 2022; Schick et al. 2023; Guo et al. 2024). As tasks become more compositional, spanning research automation, software synthesis, and multi-step mathematical reasoning, systems increasingly organize computation into **agentic workflows**: structured pipelines that decompose complex objectives into coordinated sub-processes handled by specialized agents (Chen et al. 2024; Wang et al. 2024b; Wu et al. 2025; Tran et al. 2025). These agentic workflows offer modularity, role specialization, and persistent decision structure, making them a core mechanism for scaling LLM-based autonomy. However, LLM-generated workflows are brittle: small reasoning slips cascade into broken steps and a loss of execution control.

Modern agentic systems build workflows through **LLM-driven code generation** to assemble and modify executable pipelines governing control flow, tool invocation, and inter-agent coordination (Hong et al. 2024; Zhang et al. 2025b; Hu, Lu, and Clune 2025). Conceptually, these workflows are often modeled as **directed graphs** of symbolic operations and message-passing steps, where nodes represent agent actions or tools and edges define information flow and execution constraints. A *workflow representation* is thus a programmatic encoding of this graph into a target language, mapping nodes, dependencies, and execution transitions into executable code. **Yet most systems instantiate workflows in imperative Python** (Yao et al. 2023; Schick et al. 2023), where control flow, state mutation, and execution semantics are intermixed, making structural dependencies **implicit, hard to inspect, and easy to break**. As a result, workflows become brittle: small textual deviations induce malformed graphs, dangling calls, circular transitions, or silent behavioral drift (Hong et al. 2024; Zhang et al. 2025b). Simply put, treating workflows as imperative scripts obscures structure, weakens guarantees, and undermines execution control.

In this work, we represent workflows using **Mermaid**, a graph domain-specific language (DSL), replacing LLM-generated imperative Python with explicit graph-structured programs. Instead of relying on brittle token-level script edits to preserve global logic (Hong et al. 2024; Zhang et al. 2025b; Hu, Lu, and Clune 2025; Yao et al. 2023; Schick et al. 2023), we encode workflow structure directly as symbolic graphs with named nodes, typed edges, and execution constraints. Mermaid surfaces workflow topology as first-class syntax, enabling **structural inspection, compiler val-**

*Corresponding author.

[1]https://mermaid.js.org/

idation, and localized graph edits. This shifts workflow construction from text-level imitation to structure-aware planning and repair, enabling explicit control over execution semantics. As a result, agentic pipelines become more controllable, auditable, and robust, with evolution grounded in verifiable structure rather than fragile token edits. This structural foundation is crucial for moving beyond static or handcrafted task graphs toward adaptive agent pipelines that must evolve while remaining valid and safe.

When workflows are represented as explicit graphs, the challenge shifts from merely generating workflows to **controlling how they evolve**. Agentic systems refine workflows **iteratively** as goals shift and errors surface, yet conventional systems treat refinement as **unstructured code search**, allowing edits to drift, violate invariants, and silently corrupt execution. Building on this, we show that a Mermaid-based representation enables **constraint-guided refinement**, where capability bounds and semantic constraints (Kapoor et al. 2025) shape updates before execution, improving stability and coherence during optimization. To operationalize this view, we introduce **MermaidFlow-CF**, a *constraint-faithful* workflow optimization framework evaluating stability, interpretability, and executability under iterative refinement. Our setting mirrors deployment: an LLM orchestrator edits workflows drawn from a fixed node library, and pipelines must remain comprehensible, editable, and runnable throughout evolution. We benchmark Python and Mermaid workflows, showing that representation directly governs constraint fidelity and stable iterative improvement.

Overall, our work introduces four key contributions:

1. We formulate agentic workflow construction as a **constrained optimization problem**, where the goal is not only to generate workflows but to iteratively refine them while preserving structural and execution integrity.
2. We introduce a principled **taxonomy of workflow constraints**, capturing four practical dimensions.
3. We provide an **evaluation protocol** with actionable metrics that measure both *structural violations* and the system's ability to *recover* during iterative refinement.
4. Experiments demonstrate that our Mermaid-based system significantly outperforms Python-based approaches in structural validity and reliable workflow evolution. Notably, **MermaidFlow-CF** achieves a lower overall constraint violation rate across three benchmarks compared to **AFlow**, with a maximum improvement exceeding **50%**. Furthermore, **MermaidFlow-CF** consistently shows fewer executability violations across all four benchmarks, highlighting the superior generative robustness of graphical workflow representations.

Related Work

Agentic Workflow Representations

The representation of agentic workflows specifies agents' inputs, outputs, operational sequences, and logical conditions, serving as a medium through which the orchestrator or meta-agent (Xiang et al. 2025) interprets and optimizes the workflow. Early natural language-based approaches, such as Chain of Thought (CoT) (Wei et al. 2023; Zhang et al. 2022), ReAct (Yao et al. 2023), and Debate (Liang et al. 2024), expressed agent capabilities and response schemas purely through prompting. Inspired by the notion of agentic functional blocks in programming languages, code-centric approaches have emerged to represent workflows. AFlow (Zhang et al. 2025b) encodes workflows directly in code and optimizes them via Monte Carlo Tree Search (MCTS), while ADAS (Hu, Lu, and Clune 2025) enables a meta-agent to discover new agents through iterative code modifications. Graph-based representations were subsequently explored, grounded in the notion that workflows are inherently graph-structured, with discrete agentic components as nodes and their interactions as edges. GPTSwarm (Zhuge et al. 2024) adopts a hierarchical optimization strategy that assembles composite graphs into larger computational structures, whereas MermaidFlow (Zheng et al. 2025) leverages the *Mermaid* graph language to formalize workflows and define modification operators alongside validation mechanisms. Building upon these diverse representation paradigms, our work aims to **compare approaches leveraging different representations under constraints to achieve more controllable workflow orchestration and optimization.**

Workflow Search and Optimization under Constraints

While static workflows yield moderate improvements over single-agent frameworks, researchers have investigated methods for identifying optimal workflows within task-specific search spaces. ADAS (Hu, Lu, and Clune 2025) introduces a meta-agent that iteratively generates new agents by modifying prompts, tool usage, and other parameters. The meta-agent functions within unconstrained code spaces, with its self-reflection process triggered only upon workflow failure. AFlow (Zhang et al. 2025b) uses a variant of MCTS to find optimal workflows, fixing key parameters while optimizing only code-defined edges and prompts.GPTSwarm (Zhuge et al. 2024) assembles atomic agents into composite graphs, enforcing DAG constraints during graph sampling to prevent cycles. MaAS (Zhang et al. 2025a) employs a layered search space, halting the search once the cumulative activation score of selected operators exceeds a set threshold. MermaidFlow (Zheng et al. 2025) models workflows with the *Mermaid* graph language, providing six graph-level operators and combining soft checks for code completeness with hard checks for executable validity. In all, the lack of a graphic validation foundation hinders the robustness of most workflow optimization systems. Accordingly, **MermaidFlow-CF formulates a constrained workflow optimization problem and systematically study the effect of representing workflow with the native graphic DSL Mermaid compared to Python.**

Methodology

In this section, we introduce **MermaidFlow-CF**, a novel constrained workflow optimization framework. We begin by formalizing workflow optimization as **a constrained optimization problem**, where structure and execution rules act

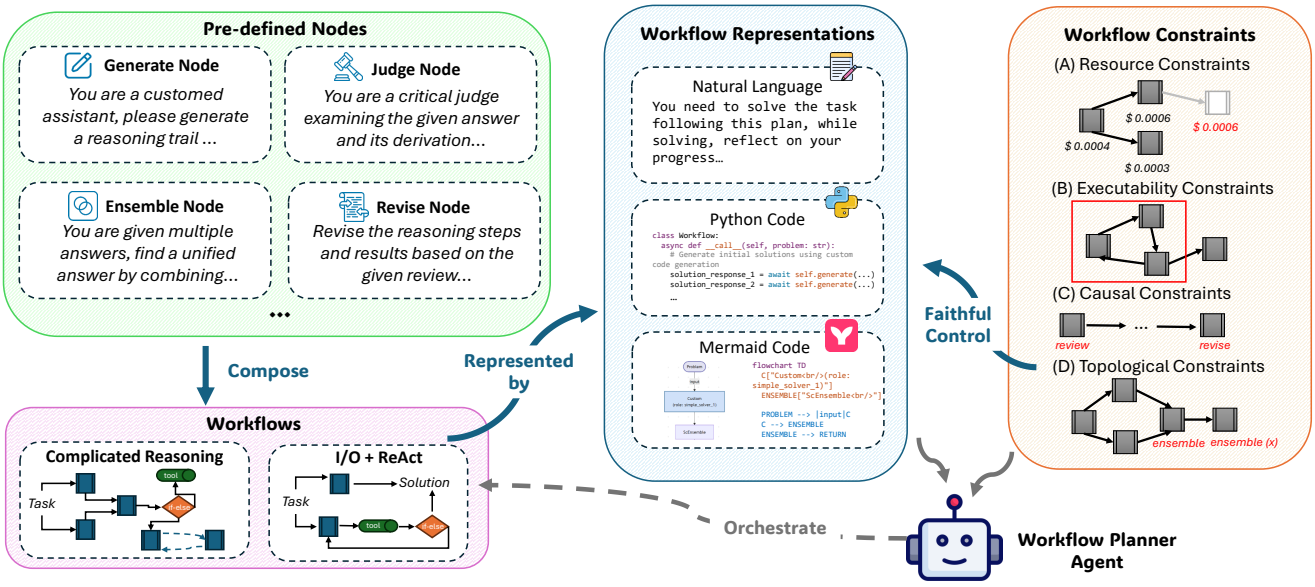

Figure 1: Overview of the **MermaidFlow-CF** framework, illustrating how **workflow representation governs constraint-faithful control**. **Left:** Examples of key pre-defined node primitives (generate, judge, ensemble). These nodes serve as basis for workflow formation. **Middle:** Workflow representations (*natural language*, *Python code*, *Mermaid programs*) with varying expressiveness and constraint adherence fidelity. The structured Mermaid representation not only provides more reliable constraint-following control, but also yields interpretable workflow visualizations. They provide visibility and serve as a controlling medium for constraints. **Right:** Four workflow constraint classes: *resource*, *executability*, *causal*, and *topological*, that shape feasible solution space. The *workflow planner agent* orchestrates new workflows conditioned on the imposed constraints and workflow representations.

as inductive priors rather than post-hoc filters. We then establish **a constraint taxonomy**, identifying four fundamental classes that govern valid workflow behavior and shape the feasible workflow search space.

Constrained Agentic Workflow Optimization Problem

We formulate agentic workflow optimization as a constrained generation problem: the model must produce an executable workflow that maximizes task performance while respecting structural and execution constraints. Rather than treating constraints as filters applied after generation, we elevate them to first-class inductive priors that shape the search space and guide workflow construction from the start.

Formally, each workflow is represented as a directed acyclic graph (DAG):

$$\mathcal{G} = \{\mathcal{V}_{[\tau,\alpha]}, \mathcal{E}_{[\rho]}\}, \quad \mathcal{V} \subset \mathbf{O}, \quad \mathcal{E} \subset \mathcal{V} \times \mathcal{V}, \tag{1}$$

where $\mathcal{V}$ denotes operator nodes with type τ and configuration α (e.g., model backbone, tool choices), and $\mathcal{E}$ denotes directed edges annotated by data schema ρ. Each node is drawn from the operator library $\mathbf{O}$, and edges encode information flow between operators.

For agentic workflow optimization, a workflow is ultimately a program synthesized by an LLM. We therefore represent them in a programmatic language λ (*e.g.*, Python, Mermaid, natural-language plans):

$$f_{\text{rep}}(\mathcal{G}; \lambda); \mapsto; \text{program expressed in language } \lambda, \tag{2}$$

where λ defines how the workflow graph $\mathcal{G}$ is serialized and executed. In **MermaidFlow-CF**, we set λ = Mermaid, a graph-native DSL that encodes nodes, edges, and dataflow directly in the language syntax. Unlike Python, where control and data flow are implicit in code, Mermaid exposes structure explicitly, yielding interpretable and deterministic execution traces.

Workflow optimization proceeds via iterative refinement (e.g., Monte Carlo Tree Search), where each step proposes a new workflow consistent with constraints:

$$f_{\text{rep}}(\mathcal{G}_n; \lambda) = \mathcal{P}_\theta\Big(f_{\text{rep}}(\mathcal{G}_t; \lambda), f_{\text{score}}\big(f_{\text{rep}}(\mathcal{G}_t; \lambda)\big), \mathbf{C}\Big), \quad 0 < t < n. \tag{3}$$

where the workflow planner $\mathcal{P}$ parameterized by θ generates workflows in represented language λ, $f_{\text{score}}(\cdot)$ evaluates the workflow's execution quality under the same representation interface, and $\mathbf{C}$ supplies constraints that act as inductive priors guiding generation.

The constraints ensure that generated workflows remain structurally valid and executable, defining the feasible search region. For each constraint $c \in \mathbf{C}$,

$$\mathcal{G}^c = \{\, \mathcal{G} \mid \mathcal{G} \models c \,\}, \qquad \mathbf{G}^{\mathbf{C}} = \bigcap_{c \in \mathbf{C}} \mathcal{G}^c, \tag{4}$$

where $\mathbf{G}^{\mathbf{C}}$ denotes a feasible set, and $\mathcal{G} \models \mathbf{C} \iff \mathcal{G} \in \mathbf{G}^{\mathbf{C}}$. In practice, feasibility is verified by a structural checker:

$$\mathcal{G}_n \models \mathbf{C} \iff f_{\text{check}}\left(f_{\text{rep}}(\mathcal{G}_n; \lambda), \mathbf{C}\right) = 1, \tag{5}$$

Table 1: Constraint taxonomy in **MermaidFlow-CF**, covering four workflow constraint classes with representative examples and explanations.

Constraint Type	Examples	Explanation
Resource Constraints	$\sum_i \text{token}_i \le B_{\text{token}}$, $\sum_i \text{cost}_i \le B_{\text{cost}}$	Total token usage $\sum_i \text{token}_i$ and total API cost $\sum_i \text{cost}_i$ must not exceed budgets B_{token} and B_{cost}.
Executability Constraints	$\texttt{RETURN} \in \mathcal{V}$; $G \in \text{DAG}(\mathcal{V}, \mathcal{E})$	The workflow must contain a `RETURN` node, and the graph structure must form a directed acyclic graph (DAG).
Topological Constraints	$\lvert\mathcal{V}\rvert \le \text{Node}_{\max}$; $\lvert\mathcal{E}\rvert \le \text{Edge}_{\max}$	The number of nodes and edges must not exceed the respective limits $\text{Node}_{\max}$ and $\text{Edge}_{\max}$.
Causal Constraints	$(v_i, v_j) \notin \mathcal{E}$ if $v_i, v_j \in \mathcal{V}$; $v_j \in \mathcal{V} \Rightarrow (v_i, v_j) \in \mathcal{E},\ v_i \in \mathcal{V}$	Edges must follow causal rules. If v_j is used, its parent v_i must appear upstream in the workflow.

where f_{check} returns 1 only if the serialized workflow satisfies all constraints.

Overall, given task distribution $(x, y) \sim \mathcal{T}$, the objective is formulatd as:

$$\max_{\mathcal{G}_n} \mathbb{E}_{(x,y)\sim\mathcal{T}}\Big[f_{\text{score}}\big(f_{\text{rep}}(\mathcal{G}_n;\lambda);\, x, y\big)\Big] \quad \text{s.t. } \mathcal{G}_n \in \mathbf{G}^{\mathbf{C}},$$
$$f_{\text{rep}}(\mathcal{G}_n;\lambda) \;=\; \mathcal{P}_\theta\Big(f_{\text{rep}}(\mathcal{G}_t;\lambda),\, f_{\text{score}}\big(f_{\text{rep}}(\mathcal{G}_t;\lambda)\big),\, \mathbf{C}\Big),$$
$$0 < t < n. \tag{6}$$

The proposed constrained workflow optimization task sheds light on **workflow representation** as a shaping factor for generation and evaluation under constraints. The framework also brings up a largely overlooked control surface, where constraints are mainly perceived as a post hoc correctness filters or repair triggers during optimization in Python-based planners such as AFlow. In contrast, Mermaid represents workflows as explicit symbolic graphs that disentangle structure from execution semantics, enabling **constraint-faithful** optimization where workflows are valid by construction and remain aligned with constraints throughout refinement. Representation, therefore, is not a mere serialization choice, it defines the *geometry of the search space*. **Constraint-faithful control** can only be made possible by *systematically eradicating infeasible graphs*, which emerges as an intrinsic property of representation itself, rather than an externally enforced mechanism.

A Taxonomy of Workflow Constraints

Constraints define the feasible workflow manifold, ensuring the search remains within executable and semantically coherent plans. Crucially, when constraints are coupled to the workflow representation, they shape generation itself rather than acting as post-hoc rejection rules. When expressed as inductive priors over the workflow space, constraints prevent the planner from drifting into non-executable or logically inconsistent trajectories. We formalize four core classes of constraints: *resource, executability, topology*, and *causal*. The detailed specifications and examples are presented in Table 1.

Resource Constraints. Resource constraints capture the practical limitations arising from workflow execution, such as token consumption and monetary cost. These quantities typically exhibit a cumulative nature, aggregating usage across executed workflow nodes. Given a token budget B_{token} and a cost budget B_{cost}, the constraints can be formally expressed as:

$$\sum_i \text{token}_i \le B_{\text{token}}, \quad \sum_i \text{cost}_i \le B_{\text{cost}}. \tag{7}$$

Executability Constraints. Once a workflow is constructed, it is essential to verify its correct execution, as flawed workflows may lead to errors such as incompatible input handling, premature termination without producing output, or infinite loops. To mitigate this issue, we established the following checklist for validating workflow executability.

Workflow Executability Checklist

1. **Input Validity:** All required input nodes exist in the workflow graph:
   $$\forall v \in \mathcal{V}_{input},\ v \in \mathcal{V}_{\mathcal{G}}$$
   where $\mathcal{V}_{input}$ is the set of expected input nodes and $\mathcal{V}_{\mathcal{G}}$ is the set of all nodes in $\mathcal{G}$.
2. **Output Validity:** The workflow graph contains at least one output node:
   $$\exists v \in \mathcal{V}_{\mathcal{G}} \text{ s.t. } v \in \mathcal{V}_{output}$$
   where $\mathcal{V}_{output}$ denotes the set of valid output nodes.
3. **Type-Compatible Execution:** Every node $v \in \mathcal{V}_{\mathcal{G}}$ has compatible inputs and outputs:
   $$\forall v \in \mathcal{V}_{\mathcal{G}},\ \text{type}(v_{in}) \subseteq \text{type}(v_{expected\_in})$$
4. **Acyclicity:** The workflow graph does not contain cycles that prevent termination:
   $$\mathcal{G} \text{ is acyclic, i.e., } \mathcal{G} \in \text{DAG}(\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}})$$
   where $\mathcal{E}_{\mathcal{G}}$ is the set of directed edges in the workflow graph.

Topological Constraints. Topological constraints define restrictions on the composition of workflow graphs, including the total number of nodes, the number of nodes of specific types, the number of edges, and the allowable ranges of node types. These constraints are selected based on empirical observations and practical requirements of the tasks, ensuring that the workflows remain expressive yet computationally manageable.

Specifically, the total number of nodes in a workflow graph is bounded as:

$$N_{min} \leq |\mathcal{V}_{\mathcal{G}}| \leq N_{max}. \tag{8}$$

Edge counts are limited in conjunction with node counts to maintain a manageable level of connectivity.

For type-specific nodes, the number of nodes of a given type t is restricted by

$$N_t^{min} \leq |\{v \in \mathcal{V}_{\mathcal{G}} : \tau(v) = t\}| \leq N_t^{max}, \tag{9}$$

allowing, for instance, a limitation on programming nodes when solving math reasoning tasks. Collectively, these topological constraints regulate the complexity of workflow graphs, promoting computational efficiency, interpretability, and suitability for the intended tasks.

Causal Constraints. Causal constraints restrict the permissible orderings of nodes within a workflow. For instance, certain sequences (e.g., *ensemble* $\rightarrow$ *review*) may be disallowed, or a node of a specific type may be required to follow one or more nodes of other types (e.g., *codeGen* $\rightarrow$ *test*). Formally, we categorize causal constraints as follows:

1. **Forbidden Precedence:**

$$(v_i, v_j) \notin \mathcal{E}, \quad v_i \in \mathcal{V}_i,\, v_j \in \mathcal{V}_j$$

2. **Required Precedence:**

$$v_j \in \mathcal{V} \implies (v_i, v_j) \in \mathcal{E}, \quad v_i \in \mathcal{V}$$

Constraint-Grounded Workflow Execution In **Mermaid-CF**, the constraint-grounded workflow optimization tasks are treated differently in terms of operational units: we shift the atomic basis of workflow feature targeting and search space filtering to native graph features. Detailed pipelines are shown in Fig. 2. When workflows are represented in a graphic language like *Mermaid*, the system can directly target structural features such as **node count** and **topological dependencies**, yielding a filtered search space $\widehat{\mathcal{S}}_{\text{struct}}$ building on top of **concrete graph structures**. In contrast, code-based workflows only allow targeting **code snippets**, producing a search space $\widehat{\mathcal{S}}_{\text{logic}}$ that is *only confined by logical conditions and offers limited structural awareness to the planner agent*.

$$(\mathcal{G}_t^{\text{mermaid}}, \mathbf{C}) \xrightarrow[\text{graph features}]{\text{targeting}} \underbrace{\text{nodes, edges, motifs}}_{\text{structural form}} \xrightarrow[\text{search space}]{\text{filtering}} \widehat{\mathcal{S}}_{\text{struct}}$$

$$(\mathcal{G}_t^{\text{code}}, \mathbf{C}) \xrightarrow[\text{code snippets}]{\text{targeting}} \underbrace{\text{functions, logic units}}_{\text{syntactic form}} \xrightarrow[\text{search space}]{\text{filtering}} \widehat{\mathcal{S}}_{\text{logic}}$$

Figure 2: Comparison of constraint-grounded workflow execution pipelines with different representations

Experiments

Experiment Setup

Benchmarks. Our experiments were conducted on two math reasoning benchmarks, **MATH** (Hendrycks et al. 2021) and **GSM8K** (Cobbe et al. 2021), and two code generation benchmarks, **HumanEval** (Chen et al. 2021) and **MBPP** (Austin et al. 2021).

Baselines. We adopt AFlow (Zhang et al. 2025b), which represents workflows directly as Python code, and MermaidFlow (Zheng et al. 2025), which uses Mermaid graph representations as baselines. In each iteration, the orchestrator agent modifies the workflow code directly in AFlow, whereas in MermaidFlow, it edits the corresponding Mermaid graph description. The updated Mermaid graph is then processed by a converter and a validation module before being transformed into executable code.

Implementation Details. We utilized **GPT-4o-mini** as the LLM backbone. The maximum number of iterations was set to 20 for all experiments. The tested constraint settings are detailed in Supplementary Materials.

Evaluation Metrics

We present our novel evaluation protocol on two different aspects: (1) whether the orchestrated workflows remain within prescribed constraint boundaries, and (2) whether the orchestrator agent can effectively revise the workflow through its underlying representation upon constraint violation. Accordingly, we define the following two metrics to assess these core capabilities:

Violation Rate. We define the Violation Rate (VR) as the ratio of the number of violations and the maximum number of iteration rounds as (10):

$$\text{VR} = \frac{1}{N_{\text{iter}}} \sum_{i=1}^{N_{\text{iter}}} \mathbf{1}\big[\mathcal{G}_i \notin \mathbf{G}_{\mathbf{C}}\big], \tag{10}$$

where $\mathcal{G}_i$ is the orchestrated workflow at the i-th round, $\mathbf{G}_{\mathbf{C}}$ is the set of valid workflow graphs subject to constraint set $\mathbf{C}$, and N_{iter} is the maximum number of iterations.

Correction Rate. We define the Correction Rate (CR) as the fraction of detected constraint violations that are successfully corrected in subsequent iterations. In our selected baselines, each iteration round i begins by sampling a father round $i_t (i > i_t)$, formally, it can be expressed as (11):

$$\text{CR} = \frac{\sum_{i=2}^{N_{\text{iter}}} \mathbf{1}\big[\mathcal{G}_{i_t} \notin \mathbf{G}_{\mathbf{C}} \ \wedge \ \mathcal{G}_i \in \mathbf{G}_{\mathbf{C}}\big]}{\sum_{n=1}^{N_{\text{iter}}} \mathbb{I}\big[\mathcal{G}_n \notin \mathbf{G}_{\mathbf{C}}\big]}. \tag{11}$$

Evaluation Pipeline

To assess the expressiveness and robustness of different workflow graph representations under varying constraints, we exploit the iterative training paradigm of existing workflow design baselines. At the start of each iteration, the task description and designated constraint are jointly provided as

Table 2: Constraint-specific **violation** (VR↓) and **correction** (CR↑) rates across four constraint families: resource, topological, causal, and executability. Better values are highlighted in **bold**. Note that CR is omitted for *executability constraints*, since non-executable workflows cannot be used as modifiable baselines for correction.

Framework	MATH		GSM8K		HumanEval		MBPP	
	VR↓	CR↑	VR↓	CR↑	VR↓	CR↑	VR↓	CR↑
Resource Constraints								
AFlow	53.2	16.0	**7.9**	0.0	**18.5**	10.0	20.4	**70.0**
MermaidFlow-CF	**43.5**	**65.0**	36.2	**9.5**	28.3	**29.4**	**3.6**	0.0
Topological Constraints								
AFlow	74.6	17.0	52.4	33.3	**23.8**	**86.7**	60.3	26.3
MermaidFlow-CF	**19.6**	**90.9**	**28.6**	**81.2**	32.1	50.0	**40.0**	**45.5**
Causal Constraints								
AFlow	67.9	5.6	85.7	16.7	61.9	**28.2**	**67.7**	**19.1**
MermaidFlow-CF	**61.7**	**20.7**	**59.6**	**29.0**	**57.4**	25.8	67.9	5.3
Executability Constraints								
AFlow	48.2	-	44.9	-	25.4	-	25.9	-
MermaidFlow-CF	**30.2**	-	**18.0**	-	**21.2**	-	**19.1**	-

prompts, after which the generated workflow is evaluated by embedded checker modules to determine its compliance with the specified constraints. The validation process by checkers occurs at two timestamps: *post-orchestration* for topological, causal, and executability constraints, and *post-execution* for resource constraints.

Evaluation Results

The evaluation results are presented as in Table 2. From the results, we draw the following key observations:

Resource Constraints. *AFlow adheres more closely to resource constraints, whereas MermaidFlow-CF exhibits a more pronounced trend of correction.* **AFlow** achieves lower VRs on the GSM8K, HumanEval, and MBPP benchmarks, while **MermaidFlow-CF** incurs fewer resource violations on the MATH benchmark and attains consistently higher CRs across all four tasks. This pattern likely arises because *MermaidFlow-CF's graphical workflow representations* tend to produce more complex initial graphs during early iterations, which have structures that are more prone to exceeding cost limits when the orchestrator agent lacks constraint awareness. Conversely, it is precisely the *integrated visual nature* of MermaidFlow-CF that enables the orchestrator to rapidly develop sensitivity to node-level costs and to correct violations more effectively over subsequent rounds.

Topological Constraints. *MermaidFlow-CF adhered to substantially more topological constraints and demonstrated a pronounced correction trend.* Across three benchmarks, **MermaidFlow-CF** exhibited markedly fewer structural violations, most notably on the MATH benchmark, where it outperformed by an impressive margin of *55%*. It is also noteworthy that **MermaidFlow-CF** achieved an over **80%** correction rate on three benchmarks. This suggests that *MermaidFlow-CF possesses a strong and consistent capability to detect node-limit violations and rectify them effectively.* Furthermore, **AFlow** demonstrated considerably higher correction rates under topological constraints than under resource constraints, implying that node-related topological constraints are inherently easier to detect, and that both baselines exhibit a tangible degree of instruction-following competence.

Causal Constraints. *MermaidFlow-CF has significantly lower violations when given causal constraints.* Across all four benchmarks, MermaidFlow-CF achieved consistently lower VRs than AFlow, especially on GSM8K benchmark where it has an advantage of **54.9%**. Moreover, **MermaidFlow-CF** also gains more correction rate significantly on three benchmarks. This implies that MermaidFlow-CF 's directed-graph representation provides the orchestrator with clearer supervision over the operational sequence of nodes, allowing it to better preserve causal dependencies such as "Node A must always precede Node B." By visually encoding upstream–downstream relationships, MermaidFlow-CF reduces the ambiguity in causal ordering that can arise in code-based representations, where such dependencies are often implicit within nested functions or variable scopes. The graphical abstraction thus enhances global causal coherence without requiring line-by-line program analysis.

Executability Constraints. *MermaidFlow-CF generates more executable workflows throughout the iterations.* When subjected to the same validity checking procedures, **MermaidFlow-CF** consistently produced a higher proportion of executable (valid) workflows. The visual form of representation grants **MermaidFlow-CF** more intuitive control over essential graph properties, such as ensuring acyclicity, verifying the presence of terminal nodes, and maintaining complete dataflow paths. For instance, detecting dead loops or missing return nodes can be trivially accomplished by inspecting the adjacency of edges in a mermaid diagram,

Table 3: Constraint adherence results of AFlow and MermaidFlow-CF over 20 iterations on the **MATH** benchmark under the **causal constraint** `require(Custom, Programmer)`.

Methods	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**	**#13**	**#14**	**#15**	**#16**	**#17**	**#18**	**#19**	**#20**
AFlow	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓
MermaidFlow-CF	✗	✗	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓

whereas **AFlow** must rely on dynamic code tracing or symbolic parsing. Consequently, *the graphical abstraction not only reduces the barrier for constraint validation but also enhances the interpretability of workflow corrections during orchestration.*

Discussions. To conclude, **MermaidFlow-CF demonstrates clear advantages derived from its graphic representation, offering superior constraint awareness and enforcement efficiency compared to code-based AFlow.** While AFlow's programmatic formulation ensures deterministic execution and internal logic consistency, it often obscures higher-level topological and causal relationships that are immediately visible in a graph. MermaidFlow-CF, on the other hand, makes these relationships explicit through visual topology, allowing the orchestrator to reason over node dependencies, causal flows, and constraint satisfaction more intuitively. This graphical transparency leads to lower violation rates and higher correction efficiency across diverse constraint types. Overall, the results suggest that *workflow representations grounded in visual graph structures provide a more interpretable and constraint-sensitive foundation for agentic orchestration than naïve code representations*.

Case Study: Optimization Dynamics under Causal Constraints

In this section, we present a case study showing a central insight: **causal constraints are not optional, but foundational drivers of workflow optimization performance**. We evaluate MermaidFlow and AFlow on the MATH dataset under two regimes: **CF-Req**, where a causal constraint `require(Custom, Programmer)` is enforced, meaning that every `Programmer` node must be preceded by a *Custom* node, and **CF-Forb**, where the same node sequence is explicitly prohibited. This bidirectional setup tests whether improvement arises from internalizing causal structure or from exploiting unconstrained search flexibility. Results are shown in Fig. 3.

We observe a clear divergence between the two regimes: both MermaidFlow and AFlow steadily improve when the causal constraint is enforced (**CF-Req**), whereas performance plateaus substantially lower when the constraint is forbidden (**CF-Forb**). MermaidFlow shows the strongest effect, with **CF-Req climbs and stabilizes near 0.59**, while CF-Forb stalls near 0.51, yielding an **∼8% absolute improvement**. AFlow exhibits the same pattern, demonstrating that the effect reflects a general property of agentic workflow optimization rather than a model-specific artifact.

Furthermore, MermaidFlow-CF exhibits a substantially larger gap between CF-Req and CF-Forb than AFlow, indicating that its structured workflow representation is inherently **constraint-faithful**, turning causal structure into reliable control signals that support higher attainable performance. This points to a deeper principle for agentic systems: *causal constraints are not restrictions, but a source of optimization energy*. By shaping the feasible search space and guiding credit assignment, they induce learning dynamics that are **more stable, more directed, and capable of higher performance ceilings**. This reframes causal constraints from external rules into internal inductive priors, pointing toward agentic systems that succeed not by avoiding structure, but by learning through privileged access to the right structure.

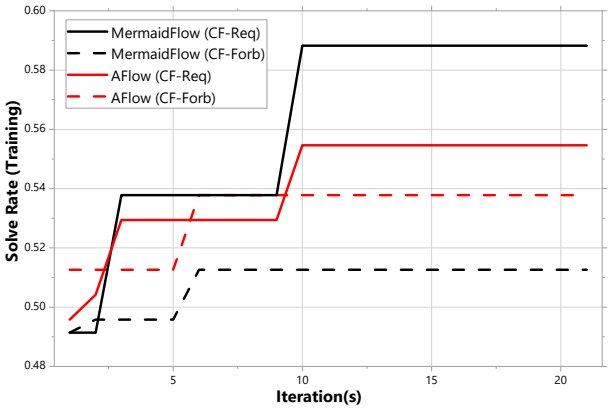

Figure 3: Training curves for AFlow and MermaidFlow on MATH with the causal constraint `require(Custom, Programmer)` enforced (CF-Req) vs. forbidden (CF-Forb). Surprisingly, enforcing the causal constraint boosts performance of MermaidFlow by ∼8%.

Conclusion

We introduced MermaidFlow-CF, a constrained agentic workflow optimization framework, along with a structured taxonomy of workflow constraints. Under identical operator libraries and tasks, symbolic Mermaid workflows consistently maintain structure, satisfy constraints, and avoid cascading failures, whereas Python-based pipelines degrade under iterative refinement. These results highlight that scalable agentic systems depend not only on model capability, but on representations that make control flow explicit and enforceable, enabling workflows to evolve reliably under constraint. Looking forward, we see opportunities to extend constraint-faithful workflow learning to richer planning domains, adaptive constraint discovery, and tighter integration with real-world tool chains and execution substrates.

Acknowledgments

This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-NMLP-2024-003), and the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech Funding Initiative, Career Development Fund (CDF) of the Agency for Science, Technology and Research (A*STAR) (No: C233312007, No: C243512014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, the Agency for Science, Technology and Research, or the Infocomm Media Development Authority.

References

Anonymous. 2025. FlowSearcher: Synthesizing Memory-Guided Agentic Workflows for Web Information Seeking. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Anthropic. 2025. Claude Opus 4.1. https://www.anthropic.com/claude/opus. Accessed: 2025-10-20.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Kaplan, J.; et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Chen, W.; You, Z.; Li, R.; Guan, Y.; Qian, C.; Zhao, C.; Yang, C.; Xie, R.; Liu, Z.; and Sun, M. 2024. Internet of Agents: Weaving a Web of Heterogeneous Agents for Collaborative Intelligence. arXiv:2407.07061.

Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; Hesse, C.; and Schulman, J. 2021. Training Verifiers to Solve Math Word Problems. arXiv:2110.14168.

DeepSeek-AI; Guo, D.; Yang, D.; Zhang, H.; et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

Guo, T.; Chen, X.; Wang, Y.; Chang, R.; Pei, S.; Chawla, N. V.; Wiest, O.; and Zhang, X. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. arXiv:2402.01680.

Hendrycks, D.; Burns, C.; Kadavath, S.; Arora, A.; Basart, S.; Tang, E.; Song, D.; and Steinhardt, J. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. arXiv:2103.03874.

Hong, S.; Zhuge, M.; Chen, J.; Zheng, X.; Cheng, Y.; Zhang, C.; Wang, J.; Wang, Z.; Yau, S. K. S.; Lin, Z.; Zhou, L.; Ran, C.; Xiao, L.; Wu, C.; and Schmidhuber, J. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. arXiv:2308.00352.

Hu, S.; Lu, C.; and Clune, J. 2025. Automated Design of Agentic Systems. arXiv:2408.08435.

Kapoor, S.; Stroebl, B.; Kirgis, P.; Nadgir, N.; Siegel, Z. S.; Wei, B.; Xue, T.; Chen, Z.; Chen, F.; Utpala, S.; Ndzomga, F.; Oruganty, D.; Luskin, S.; Liu, K.; Yu, B.; Arora, A.; Hahm, D.; Trivedi, H.; Sun, H.; Lee, J.; Jin, T.; Mai, Y.; Zhou, Y.; Zhu, Y.; Bommasani, R.; Kang, D.; Song, D.; Henderson, P.; Su, Y.; Liang, P.; and Narayanan, A. 2025. Holistic Agent Leaderboard: The Missing Infrastructure for AI Agent Evaluation. arXiv:2510.11977.

Liang, T.; He, Z.; Jiao, W.; Wang, X.; Wang, Y.; Wang, R.; Yang, Y.; Shi, S.; and Tu, Z. 2024. Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate. arXiv:2305.19118.

OpenAI. 2025. GPT-5 is here. https://openai.com/gpt-5/. Accessed October 15, 2025.

Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761.

Tran, K.-T.; Dao, D.; Nguyen, M.-D.; Pham, Q.-V.; O'Sullivan, B.; and Nguyen, H. D. 2025. Multi-Agent Collaboration Mechanisms: A Survey of LLMs. arXiv:2501.06322.

Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; Zhao, W. X.; Wei, Z.; and Wen, J. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Wang, Q.; Wang, Z.; Su, Y.; Tong, H.; and Song, Y. 2024b. Rethinking the Bounds of LLM Reasoning: Are Multi-Agent Discussions the Key? arXiv:2402.18272.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q.; and Zhou, D. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903.

Wu, S.; Galley, M.; Peng, B.; Cheng, H.; Li, G.; Dou, Y.; Cai, W.; Zou, J.; Leskovec, J.; and Gao, J. 2025. CollabLLM: From Passive Responders to Active Collaborators. In *Forty-second International Conference on Machine Learning*.

Xiang, V.; Snell, C.; Gandhi, K.; Albalak, A.; Singh, A.; Blagden, C.; Phung, D.; Rafailov, R.; Lile, N.; Mahan, D.; Castricato, L.; Franken, J.-P.; Haber, N.; and Finn, C. 2025. Towards System 2 Reasoning in LLMs: Learning How to Think With Meta Chain-of-Thought. arXiv:2501.04682.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*. Accessed: 2025-10-20.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629.

Zhang, G.; Niu, L.; Fang, J.; Wang, K.; Bai, L.; and Wang, X. 2025a. Multi-agent Architecture Search via Agentic Supernet. arXiv:2502.04180.

Zhang, J.; Xiang, J.; Yu, Z.; Teng, F.; Chen, X.; Chen, J.; Zhuge, M.; Cheng, X.; Hong, S.; Wang, J.; Zheng, B.; Liu, B.; Luo, Y.; and Wu, C. 2025b. AFlow: Automating Agentic Workflow Generation. arXiv:2410.10762.

Zhang, Z.; Zhang, A.; Li, M.; and Smola, A. 2022. Automatic Chain of Thought Prompting in Large Language Models. arXiv:2210.03493.

Zheng, C.; Chen, J.; Lyu, Y.; Ng, W. Z. T.; Zhang, H.; Ong, Y.-S.; Tsang, I.; and Yin, H. 2025. MermaidFlow: Redefining Agentic Workflow Generation via Safety-Constrained Evolutionary Programming. arXiv:2505.22967.

Zhuge, M.; Wang, W.; Kirsch, L.; Faccio, F.; Khizbullin, D.; and Schmidhuber, J. 2024. GPTSwarm: Language Agents as Optimizable Graphs. In *Forty-first International Conference on Machine Learning*.

Supplementary Materials

A. Constraints

For each constraint type, we define three levels, where Level 1 represents the most lenient setting and Level 3 imposes the strictest limitations. The evaluated constraint categories are described as follows:

Resource Constraints. Based on an analysis of the average costs of execution logs and pricing schemes of the adopted APIs, we establish four groups of cost limits corresponding to different benchmarks. The cost estimation module is adapted from the baseline implementations, following a unified calculation methodology. The detailed cost limits for each benchmark are provided in Table 4. **All cost limits are for a single workflow execution.**

Table 4: Cost limits of different LLM backbones under varying resource constraint levels.

Benchmark	Cost Limits		
	Level 1	Level 2	Level 3
MATH	0.003	0.002	0.001
GSM8K	0.0012	0.0009	0.0007
HumanEval	0.0015	0.001	0.0005
MBPP	0.0009	0.0007	0.0005

Topological Constraints. We define topological constraints primarily as permissible intervals for the number of nodes, taking into account the characteristics of different task types (mathematical reasoning and code generation). For mathematical reasoning benchmarks (MATH and GSM8K), both baselines (AFlow and MermaidFlow) employ a Programmer operator to ensure precise computation. For code generation tasks (HumanEval and MBPP), both baselines include Code Generation and Test nodes to enable purpose-driven code synthesis and verification. The specific constraint settings for both task types are detailed in Table 5.

Table 5: Constrained node number intervals across task types and difficulty levels.

Task Type	Node Number Intervals		
	Level 1	Level 2	Level 3
Math Reasoning	(3,6)	(4,6)	(4,5)
Code Generation	(2,6)	(3,6)	(4,5)

Causal Constraints. Given the distinct predefined operator sets associated with the two task types, we design task-specific constraints on forbidden and mandatory node sequences, as summarized in Table 6. `req(A,B)` represents that any `B`-typed nodes within the workflow must be prefixed by `A`-typed nodes, while `forb(A,B)` represents that any `A`-typed nodes shouldn't appear in the upstream of `B`-typed nodes.

Table 6: Causal constraints across task types and levels.

Task Type	Level	Sequence Constraint
Math	Level 1	`forb(Programmer,Custom)`
	Level 2	`req(Custom,Programmer)`
	Level 3	`req(Ensemble,Programmer)`
Code	Level 1	`req(Custom,CodeGenerate)`
	Level 2	`forb(Test,Custom)`
	Level 3	`req(Test,Ensemble)`

B. Prompt Details

To ensure fair comparisons, we first constructed identical textual snippets containing both the constraint descriptions and corresponding instructions. Specifically, we provided a general instruction prompting the orchestrator agent to adhere to the listed guidelines, as shown below:

Workflow Constraint Instructions

<constraint>
The optimized or selected workflow graph MUST satisfy the following constraints (if the given workflow to be optimized doesn't satisfy, you should try your best to modify it to satisfy the constraints):
{constraints}
</constraint>

As shown above, a placeholder `constraints` is embedded within the instruction template, indicating where the constraint list is inserted. For clarity and consistency, we categorize the constraint explanations according to their respective types, as detailed below.

For resource constraints, we first retrieve the cost statistics of the sampled parent round from the recorded results. This procedure is identical for both baselines. We then integrate these retrieved statistics with the textual descriptions of the resource constraints as follows:

Prompt for Resource Constraints

- workflow should cost no more than **{max_cost}** USD for a single execution. The given workflow costs **{cost}** USD for reference.

For topological constraints, we instruct the models to ensure that the generated or selected workflows contain a number of nodes greater than the specified minimum and less than the specified maximum. In the case of *MermaidFlow*, since the model operates on Mermaid representations that inherently include the `PROBLEM` and `RETURN` nodes, we explicitly instruct the model to exclude these nodes from consideration. The complete prompt is presented as follows:

Prompt for Topological Constraints

- The workflow should have at least **{min_node}** nodes and at most **{max_node}** nodes. **(Exclude PROBLEM and RETURN nodes)**

For causal constraints, in addition to providing the constraint descriptions, we include a simple procedure to verify whether a candidate workflow satisfies the specified constraints. The corresponding prompts for forbidden and required sequences are given as follows:

Prompt for Causal Constraints with Forbidden Sequence

- In your optimized or selected workflow, **the execution of {node_2}-typed nodes should NOT be dependent on {node_1}-typed nodes**, that is, **within all sequential chains of the workflow containing {node_2}, {node_1} should NOT appear before {node_2}**.
- To make sure this, you first need to extract all dependency chains from start to end, then check for every chain containing {node_1}-typed nodes and {node_2}-typed nodes, whether {node_1} appears before {node_2} or not. If yes, the workflow violates this constraint.

Prompt for Causal Constraints with Required Sequence

- In your optimized or selected workflow, **the execution of {node_2}-typed nodes should NOT be dependent on {node_1}-typed nodes**, that is, **within all sequential chains of the workflow containing {node_2}, {node_1} should NOT appear before {node_2}**.
- To make sure this, you first need to extract all dependency chains from start to end, then check for every chain containing {node_1}-typed nodes and {node_2}-typed nodes, whether {node_1} appears before {node_2} or not. If yes, the workflow violates this constraint.

C. Constraint Checks

For a fair comparison, we employ a unified validation pipeline for both baselines. Since *AFlow* lacks a native graphical representation, we first integrate a parse-and-draw module to convert the workflow code produced by *AFlow* into a Mermaid graph.

For topological and causal constraints, we utilize a Mermaid-based analysis pipeline that provides a comprehensive examination of each workflow graph, including the number of nodes and edges, node types, and the corresponding dependency chains. These extracted features are then used to verify whether the generated workflow satisfies the specified topological and causal constraints.