FlowSteer: Towards Agents Designing Agentic Workflows via Reinforced Progressive Canvas Editing

Mingda Zhang[1,*], **Wenjin Liu**[2,*], **Tiesunlong Shen**[3,†], **Qika Lin**[4]
Rui Mao[2], **Erik Cambria**[2], **Xiaoying Tang**[5,6,7], **Haoran Luo**[2,†]

[1]School of Artificial Intelligence, The Chinese University of Hong Kong, Shenzhen, China
[2]College of Computing and Data Science, Nanyang Technological University, Singapore
[3]Centre for Biomedical Data Science, Duke-NUS Medical School, Singapore
[4]Saw Swee Hock School of Public Health, National University of Singapore, Singapore
[5]School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen, China
[6]Shenzhen Future Network of Intelligence Institute (FNii-Shenzhen), China
[7]Guangdong Provincial Key Laboratory of Future Networks of Intelligence, CUHK(SZ), China

Abstract

In recent years, agentic workflows have been widely applied to complex human tasks. However, existing workflow construction still faces key challenges, including human-dependent construction, the lack of graph-level execution feedback, and the inability to repair errors in-loop during long-horizon construction. To address these challenges, we propose FlowSteer, a new paradigm of Agent Designing Agentic Workflows—a single agent end-to-end designs the workflow that a downstream executor runs. To support this paradigm, we introduce the Workflow Canvas, an executable graph-state environment that returns syntax-checked execution feedback for every atomic edit. Built on the canvas, we propose Reinforced Progressive Canvas Editing, in which a lightweight policy agent issues one atomic edit per turn conditioned on real canvas feedback and is trained end-to-end via reinforcement learning. Moreover, FlowSteer provides a plug-and-play framework supporting diverse operator libraries and interchangeable LLM backends. Experiments on twelve datasets show that FlowSteer significantly outperforms baselines across various tasks. Our code is available at `https://github.com/SenticNet/FlowSteer`.

1 Introduction

In recent years, a variety of powerful agentic systems have been applied to solve a wide range of human problems [Ruan et al., 2023, Shen et al., 2023, Hong et al., 2023, Wu et al., 2025a, Peng et al., 2024], gradually moving beyond single-turn question answering (QA) toward executable end-to-end task completion.

In this process, workflow orchestration has become a key bridge from task goals to reproducible execution: by organizing operators into an executable workflow graph, systems can complete complex tasks with improved controllability, debuggability, and reusability [Zeng et al., 2023, Qian et al., 2024], as shown in Figure 1.

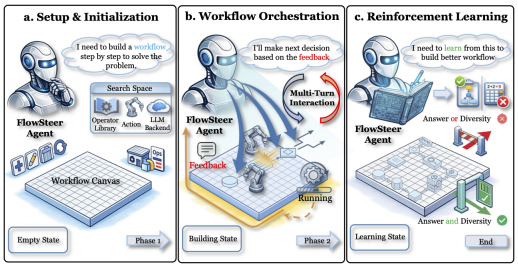


Figure 1: Overview of the FlowSteer framework. The agent initializes with the task, then through multi-turn interaction with the canvas iteratively builds and refines the workflow, finally learning from diversity-constrained rewards.

*Equal contribution.
†Co-corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

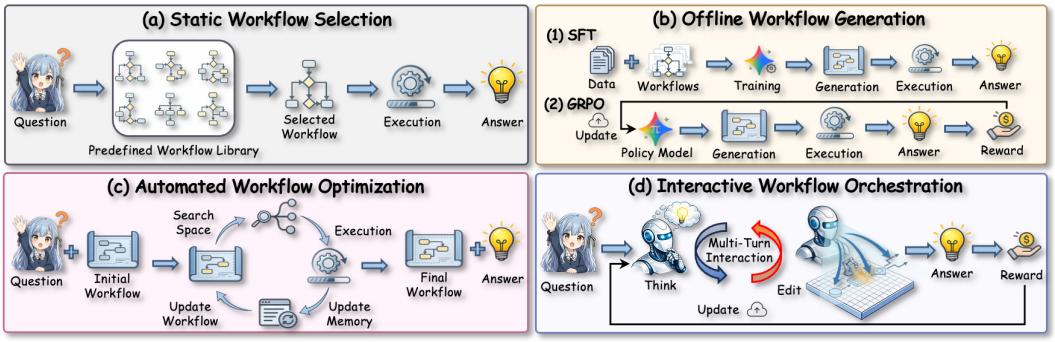


Figure 2: Comparison of workflow orchestration paradigms: static selection, offline generation, automated optimization, and our interactive orchestration framework FlowSteer.

However, in practice, workflow construction still heavily relies on manual drag-and-drop and rule-based configuration [Zhang et al., 2024, Wang et al., 2025], making it costly to transfer across new tasks, new operator libraries, new model backends, or different application domains.

To address these issues, three main paradigms of workflow orchestration have emerged, as shown in Figure 2. First, static workflow selection retrieves pre-defined workflows from a library based on task similarity [Wang et al., 2024]. Second, offline workflow generation trains policy models via supervised fine-tuning (SFT) or group relative policy optimization (GRPO) [Shao et al., 2024] to generate workflows. Third, automated workflow optimization—such as AFlow [Zhang et al., 2024], GPTSwarm [Zhuge et al., 2024], RobustFlow [Xu et al., 2025], and LATS [Zhou et al., 2023]—combines search and execution feedback to iteratively improve workflow structures.

However, these methods still face several challenges: **(i) Human-dependent workflow construction**—workflow graphs are still produced primarily through manual drag-and-drop, rule templates, or one-shot LLM prompting [Zhang et al., 2024, Wang et al., 2024], requiring continuous human effort to maintain and adapt across new tasks, which severely limits scalability and generalization; **(ii) Lack of graph-level execution feedback**—existing agent environments operate at the level of text generation or external tool calls [Yao et al., 2022, Schick et al., 2023], lacking an executable abstraction that can provide structured feedback on the workflow graph itself; without such feedback, the policy cannot diagnose or repair structural errors during construction; **(iii) Inability to repair errors in-loop during long-horizon construction**—one-shot generation or external search-based construction cannot iteratively repair errors as they emerge across many editing steps [Yu et al., 2025, Guo et al., 2025], leading to sparse, delayed learning signals, shortcut behaviors (e.g., premature termination, oversimplified graphs), and unstable long-horizon credit assignment.

To address these challenges, we propose FlowSteer, a new paradigm of *Agent Designing Agentic Workflows*, in which a single agent end-to-end designs the workflow that a downstream executor runs—replacing the manual-design loop. To support this paradigm, we introduce the *Workflow Canvas*, a novel executable graph-state environment that maintains the workflow graph and returns syntax-checked execution feedback for every atomic edit, lifting agent–environment interaction from prompt rewriting to graph-level operations. Built on this canvas, we further propose *Reinforced Progressive Canvas Editing*: a lightweight policy agent (Flow-Director) issues one atomic edit per turn—inserting, deleting, modifying, or configuring an operator node—conditioned on real canvas feedback, decomposing long-horizon workflow construction into checkable and repairable local decisions. The policy is trained end-to-end via reinforcement learning on canvas trajectories. Moreover, FlowSteer supports plug-and-play deployment across operator libraries and LLM backends.

We perform experiments on twelve datasets covering mathematical reasoning, question answering, and code generation, across six modern LLM backbones. Experimental results demonstrate that FlowSteer outperforms static-workflow, search-based, and agent-RL baselines in task accuracy, orchestration efficiency (token consumption and interaction turns), and cross-backbone transferability. As shown in Figure 2, the reinforced progressive canvas editing strategy guides the agent through iterative graph-level edits with real execution feedback, effectively bridging the gap between human-designed workflows and self-designing agentic systems. This work lays a foundation for building the next generation of agent-designing-agent systems for autonomous agentic-workflow construction.

2 Related Work

Agent Workflows. Before the rise of LLM agents, workflow automation was largely rule- or template-driven, with operators and control flow specified by hand. In the era of LLM agents, agent workflows improve long-horizon reliability via a plan–act–feedback loop [Erdogan et al., 2025, Li et al., 2025b, Hong et al., 2023]. Existing studies can be grouped into three lines: single-agent decision making, which models tool use as sequential decisions [Erdogan et al., 2025] or interleaved reasoning and acting [Yang et al., 2024]; orchestration, which uses LLM controllers for tool/model routing [Shang et al., 2024, Qian et al., 2025] or constrained API planning to ground intent [Wang et al., 2024]; and multi-agent collaboration, which relies on standard operating procedures (SOPs)/roles [Hong et al., 2023] or cross-team orchestration [Chen et al., 2023, Huang et al., 2026]. For sustained capability, agentic tool reasoning [Wu et al., 2025b], reusable workflow memory, and evolving skill libraries further improve efficiency [Wang et al., 2024, Han et al., 2025, Zheng et al., 2025, Wu et al., 2025a, Zhang et al., 2026a,d,e], while modular agent search automates compositional design [Li et al., 2026].

Reinforcement Learning for Agents. Recent agent RL frames interaction as a long-horizon Markov decision process (MDP), where coupling value learning with token-level policy learning eases delayed credit assignment in hierarchical multi-turn settings [Zhou et al., 2024, Wang and Ammanabrolu, 2025]. For tool-chain control, step-grained shaping [Yu et al., 2025, Liu et al., 2025] and outcome feedback improve tool selection and self-correction [Feng et al., 2025, Jiang et al., 2025]. When retrieval is treated as an action, search rollouts learn think-then-search behaviors [Jin et al., 2025]. End-to-end RL also trains a small model to prompt a larger one [Liu et al., 2026]. Moreover, large-scale RL motivates GRPO-style objectives based on verifiable or group-relative signals [Guo et al., 2025, Shao et al., 2024, Ji et al., 2025]. Concurrently, Workflow-R1 [Kong et al., 2026] trains a policy calling operators in a think–action–observation history and may answer directly, while FlowSteer edits an explicit executable graph and lets the executor answer (Appendix H.1).

3 Preliminaries

Definition 1: Workflow Graph. A workflow graph is a directed acyclic graph $\mathcal{G} = (V, E, \mathrm{attr})$, where $V = \{v_1, \ldots, v_n\}$ is a set of n operator nodes, $E \subseteq V \times V$ encodes data dependencies and execution order, and $\mathrm{attr}(v) = (\mathrm{op}(v), \mathrm{param}(v), \mathrm{prompt}(v))$ specifies the operator type from library $\mathcal{O}$, parameter configuration, and execution prompt for each node $v \in V$ in the workflow.

Definition 2: Orchestration Trajectory. An orchestration trajectory is a complete sequence of T interaction steps that uniquely determines a workflow graph $\mathcal{G}_\tau$:

$$\tau = \{(a_t^{\mathrm{think}}, a_t, o_t^{\mathrm{exec}})\}_{t=1}^{T} \quad \Rightarrow \quad \mathcal{G}_\tau, \tag{1}$$

where T is the total number of interaction steps before termination, a_t^{think} denotes the reasoning reflection at step t, $a_t = (\alpha_t, a_t^{\mathrm{out}})$ is the editing action with type $\alpha_t \in \mathcal{A}_{\mathrm{type}}$ and content a_t^{out}, and o_t^{exec} is the execution feedback from the canvas environment. Table 1 lists all operators and actions.

Problem Statement. Given a task q drawn from a task distribution $\mathcal{D}_Q$, an operator library $\mathcal{O}$, and a pluggable backend LLM $\mathcal{M}_{\mathrm{exec}}$, the workflow orchestration problem aims to learn a policy π_θ that generates trajectory τ. The corresponding workflow $\mathcal{G}_\tau$ is then executed to produce the answer:

$$y_q = \mathrm{Execute}(\mathcal{G}_\tau, q, \mathcal{M}_{\mathrm{exec}}). \tag{2}$$

The overall learning objective is to maximize expected reward over the task distribution:

$$\max_\theta \ \mathcal{J}(\theta) = \mathbb{E}_{q \sim \mathcal{D}_Q} \, \mathbb{E}_{\tau \sim P_\theta(\cdot|q)} \left[R(\tau) \right], \tag{3}$$

where $P_\theta(\tau \mid q)$ is the trajectory distribution induced by policy π_θ, and $R(\tau)$ is the trajectory-level reward measuring both structural quality and answer correctness.

4 Methodology: FlowSteer

As illustrated in Figure 3, FlowSteer comprises three components, each addressing one of the three challenges identified in the introduction: the canvas environment and operator/action interface that supply graph-level execution feedback (Section 4.1), the *Agent Designing Agentic Workflows* paradigm that replaces human-driven workflow construction with a designer agent (Section 4.2), and *Reinforced Progressive Canvas Editing* that turns long-horizon construction into a sequence of in-loop, repairable edits (Section 4.3).

Table 1: Operator library $\mathcal{O}$ and action space $\mathcal{A}$ in FlowSteer. Each row shows a category of operators, their outputs, action types, and the corresponding graph update operations.

Category	Operator $o \in \mathcal{O}$	Output	Type	Action	Graph Update
Planning	`Plan`, `Decompose`	strategy, sub-problems	Editing	`add`	$V_t \leftarrow V_{t-1} \cup \{v\}$
Solving	`Programmer`, `Custom`, `AnswerGen`	code, response, answer	Editing	`delete`	$V_t \leftarrow V_{t-1} \setminus \{v\}$
Verification	`Test`, `Review`, `Verify`	correct, feedback	Editing	`modify`	$\mathrm{op}(v) \leftarrow o'$
Revision	`Revise`	revised solution	Config	`set_prompt`	$\mathrm{prompt}(v) \leftarrow p$
Ensemble	`ScEnsemble`, `Aggregate`	voted, combined	Terminal	`finish`	$y_q = \mathrm{Execute}(\mathcal{G}_T, q)$
Formatting	`Format`	final answer y_q	Control	`parallel`, `cond`, `loop`	branch/merge in E_t
User-defined	$o^+ \in \mathcal{O}, \ldots$	*task-specific*	*Editing/Config*	*`add`, `modify`, `set_prompt`*	*(as above)*

4.1 Workflow Canvas and Agent Initialization

FlowSteer follows a ReAct-based agent paradigm [Yao et al., 2022]: a lightweight policy model (Flow-Director) interacts with an executable canvas environment (Workflow Canvas) to construct the workflow graph $\mathcal{G}$ (Definition 1). The canvas, operator library, action space, and state space below jointly fix the agent–environment interface that the rest of the section builds on.

Workflow Canvas $\mathcal{C}$. The Workflow Canvas is the environment that maintains the workflow graph state $\mathcal{G}_t$ and provides executable feedback at each orchestration step:

$$\mathcal{C} = (\mathcal{G}_t, \mathcal{O}, \mathcal{M}_{\text{exec}}, d^{\text{lib}}), \tag{4}$$

where $\mathcal{G}_t = (V_t, E_t, \text{attr})$ is the workflow graph at step t, $\mathcal{O} = \{o_1, \ldots, o_K\}$ is the operator library with K operators, $\mathcal{M}_{\text{exec}}$ is the pluggable LLM backend, and d^{lib} is the operator library description that enables the policy to learn available operators.

Operator Library $\mathcal{O}$ and Action Space $\mathcal{A}$. The 12 functional operators and the corresponding editing/control actions are summarized in Table 1. The library $\mathcal{O}$ is plug-and-play: operators can be freely added, removed, or replaced without affecting the framework, since the policy interacts with each operator only through its standardized action interface and structured output. At each step t, the agent emits a reflection a_t^{think} followed by an editing action $a_t = (\alpha_t, a_t^{\text{out}})$ with type $\alpha_t \in \mathcal{A}_{\text{type}}$; orchestration terminates when $\alpha_t = \texttt{finish}$.

State Space $\mathcal{S}$. The agent state $H_t \in \mathcal{S}$ is defined by its complete interaction history. Given task q, operator description d^{lib}, and template a^{tmpl}, the initial state is $H_0 = [q \oplus d^{\text{lib}} \oplus a^{\text{tmpl}}]$, where $\oplus$ denotes sequence concatenation. The state updates as:

$$H_t = H_{t-1} \oplus (a_t^{\text{think}}, a_t, o_t^{\text{exec}}). \tag{5}$$

Orchestration Target. The agent interacts with canvas $\mathcal{C}$ until reaching `finish` or maximum turns $T_{\max}$. Following Definition 2, the complete trajectory τ determines workflow $\mathcal{G}_\tau$, and the answer is obtained via execution (Eq. 2).

Proposition 1. *The operator-action space $(\mathcal{O}, \mathcal{A})$ covers diverse workflow patterns across task types through canvas-grounded orchestration (proof in Appendix B.1; evidence in Sections 5.2 and 5.3).*

4.2 Agent Designing Agentic Workflows

A defining feature of FlowSteer is the explicit separation between *workflow design* and *workflow execution*: the trained agent π_θ is responsible only for designing a workflow, while task execution is delegated to a separate backbone LLM operating on the designed workflow. We formalize this designer–executor paradigm below.

Designer–Executor Decoupling. Given a task $q \in \mathcal{D}_Q$, the policy π_θ (Flow-Director) samples an orchestration trajectory τ, which is decoded into a workflow graph $\mathcal{G}_\tau$; the answer is produced by an executor backbone $\mathcal{M}_{\text{exec}}$ running the graph,

$$y_q = \mathrm{Execute}(\mathcal{G}_\tau,\ q,\ \mathcal{M}_{\text{exec}}), \quad \mathcal{G}_\tau = \mathrm{Decode}(\tau), \quad \tau \sim \pi_\theta(\cdot \mid q), \tag{6}$$

where $\mathrm{Decode}(\cdot)$ deterministically extracts the final workflow graph from the editing trajectory τ by sequentially applying its atomic edits, and $\mathrm{Execute}(\cdot)$ runs the resulting graph $\mathcal{G}_\tau$ on the executor backbone $\mathcal{M}_{\text{exec}}$ to produce the answer y_q. Throughout training, π_θ is optimized only to produce τ, while $\mathcal{M}_{\text{exec}}$ is held fixed.

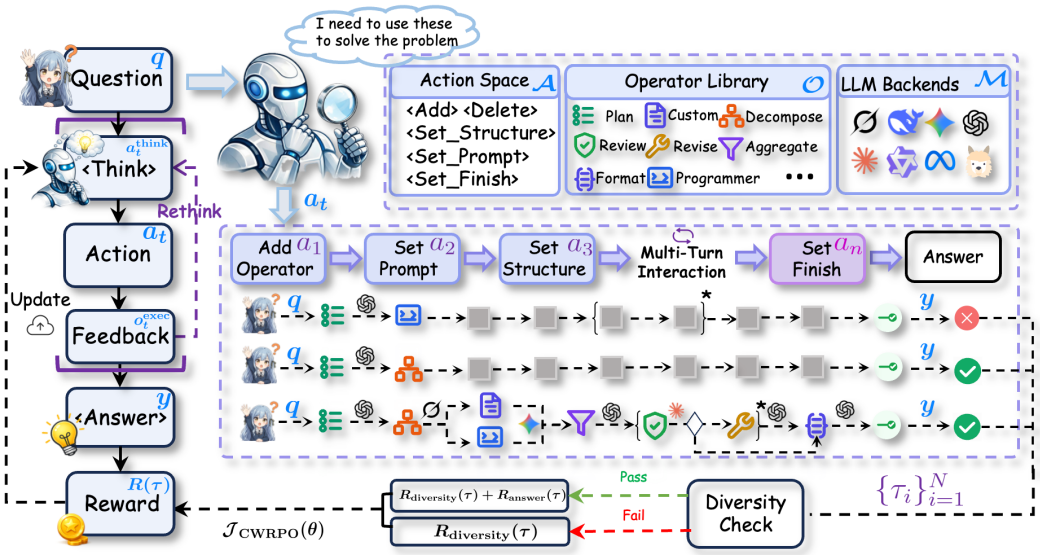


Figure 3: An overview of the FlowSteer framework. The designer agent progressively edits the workflow graph in Workflow Canvas through atomic actions, conditioned on canvas execution feedback, and is trained end-to-end with reinforcement learning over the trajectory-level outcome reward $R(\tau)$.

Multi-Agent Workflow Execution. The executor runs $\mathcal{G}_\tau$ as a multi-agent system: each node is an LLM-based agent on $\mathcal{M}_{\text{exec}}$ with its own role, prompt, and upstream inputs, directed edges pass results between agents, and `parallel`, `cond`, and `loop` structures schedule concurrent, branching, and iterative execution. Flow-Director only edits the graph and issues `finish`, while subproblem solving and the final answer come from these executor agents (e.g., the Decompose $\rightarrow$ [Custom$\times$3] $\rightarrow$ Verify $\rightarrow$ Aggregate workflow in Appendix I.2).

Marginal Likelihood over Workflow Designs. Equivalently, the answer distribution induced by FlowSteer marginalizes over all possible workflow designs sampled from π_θ,

$$P(y_q \mid q;\, \pi_\theta, \mathcal{M}_{\text{exec}}) = \mathbb{E}_{\tau\sim\pi_\theta(\cdot|q)}\Big[\, P_{\text{exec}}(y_q \mid \mathcal{G}_\tau, q, \mathcal{M}_{\text{exec}})\,\Big], \tag{7}$$

where $P_{\text{exec}}(y_q \mid \mathcal{G}_\tau, q, \mathcal{M}_{\text{exec}})$ is the executor's stochastic answer distribution when running workflow $\mathcal{G}_\tau$ on backbone $\mathcal{M}_{\text{exec}}$ for task q. The unit of decision is a workflow design τ at the level of operator composition; π_θ thereby learns orchestration strategies that transfer across operator implementations and backbone LLMs.

Design Space and Decoupled Gradient. For every node $v \in V_\tau$, a trajectory τ specifies the per-node attributes $\text{attr}_\tau(v) = (\text{op}(v),\, \text{param}(v),\, \text{prompt}(v))$ through the editing actions `add`, `modify`, and `set_prompt` (Table 1); $\text{prompt}(v)$ in particular can be rewritten by π_θ online during the editing loop. The training objective and its policy gradient are

$$\mathcal{J}(\theta) = \mathbb{E}_{q\sim\mathcal{D}_Q,\,\tau\sim\pi_\theta(\cdot|q)}\big[R(\tau)\big], \qquad \nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{q,\,\tau\sim\pi_\theta}\big[\nabla_\theta \log \pi_\theta(\tau \mid q)\cdot R(\tau)\big]. \tag{8}$$

The executor $\mathcal{M}_{\text{exec}}$ influences the gradient solely through its scalar contribution to $R(\tau)$ via the executed answer y_q. Consequently, swapping $\mathcal{M}_{\text{exec}}$, editing the library $\mathcal{O}$, or rewriting an operator's prompt preserves the form of Eq. 8 and only updates the realized reward of a given workflow design—this underlies the transferability of π_θ across operator libraries and backbones.

Proposition 2. *The designer–executor decoupling allows the trained policy π_θ to generalize across operator libraries $\mathcal{O}$ and backbone LLMs $\mathcal{M}_{\text{exec}}$ without retraining (proof in Appendix B.2; evidence with six target and three training executors in Section 5.4).*

4.3 Reinforced Progressive Canvas Editing

We instantiate the above paradigm by training π_θ via *progressive canvas editing*: an interaction loop where the agent commits one atomic edit per turn and the canvas returns execution feedback that conditions the next edit, following the prompt template in Table 2.

Table 2: The system prompt template utilized by Flow-Director to interact with the Workflow Canvas.

Build a workflow step by step to solve the problem. In each turn, output **EXACTLY ONE** XML action (`add/delete/modify/set_prompt/finish` or a structure action). Keep thinking brief; let operators do the computation. Format every turn as: **<think>**(reasoning, conditioned on the latest **<feedback>**...**</feedback>**)**</think>** **<action>**(one editing action that builds on previous edits)**</action>**. Let the workflow evolve naturally (e.g., plan → solve → verify → format). When the workflow is complete, output **<action>finish</action>**. Task: **{task}**.

Progressive Graph Construction. Each turn submits one atomic editing action, decomposing long-horizon design into checkable and repairable local decisions. Starting from the empty initial graph $\mathcal{G}_0 = (\emptyset, \emptyset, \emptyset)$, after T atomic edits the workflow is the cumulative composition

$$\mathcal{G}_T = \mathcal{G}_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_T, \qquad \oplus : \mathcal{G} \times \mathcal{A} \to \mathcal{G}, \tag{9}$$

where $\oplus$ denotes the graph-update operator instantiated by Table 1 (i.e. `add`, `delete`, `modify`, `set_prompt`, control structures), which maps a graph and an atomic edit to a new graph.

Hierarchical Editing Policy. At each step t, Flow-Director factorizes the editing action into a reflection, an action type, and a structured output, conditioned on the interaction history H_{t-1}:

$$\log \pi_\theta(a_t^{\text{think}}, a_t \mid H_{t-1}) = \begin{cases} \log \pi_\theta(a_t^{\text{think}} \mid H_{t-1}) & \alpha_t \neq \texttt{finish}, \\ \quad + \log \pi_\theta(\alpha_t \mid a_t^{\text{think}}, H_{t-1}) & \\ \quad + \log \pi_\theta(a_t^{\text{out}} \mid \alpha_t, a_t^{\text{think}}, H_{t-1}), & \\ \log \pi_\theta(a_t^{\text{think}} \mid H_{t-1}) & \alpha_t = \texttt{finish}. \\ \quad + \log \pi_\theta(\texttt{finish} \mid a_t^{\text{think}}, H_{t-1}), & \end{cases} \tag{10}$$

Canvas-Feedback-Driven Trajectory. Given action a_t, Workflow Canvas executes it and returns feedback o_t^{exec}, forming a closed-loop "diagnose–edit–verify": *(i)* the canvas performs syntax parsing and constraint checking, sampling $o_t^{\text{exec}} \sim \mathcal{C}_{\text{exec}}(\cdot \mid \mathcal{G}_{t-1}, a_t)$, where $\mathcal{C}_{\text{exec}}$ denotes the canvas's stochastic feedback distribution conditioned on the current graph and edit, carrying success status, failure reasons, and repair suggestions; *(ii)* the graph updates as $\mathcal{G}_t = \text{Update}(\mathcal{G}_{t-1}, a_t, o_t^{\text{exec}})$ and the state evolves via Eq. 5; *(iii)* Flow-Director continues until $\alpha_t = \texttt{finish}$ or $t = T_{\max}$, the maximum number of editing turns. The full trajectory distribution factorizes as

$$P_\theta(\tau \mid q) = \underbrace{\pi_\theta(a_1^{\text{think}}, a_1 \mid H_0)}_{\text{first edit}} \cdot \prod_{t=2}^{T} \underbrace{\Big(\mathcal{C}_{\text{exec}}(o_{t-1}^{\text{exec}} \mid \mathcal{G}_{t-2}, a_{t-1}) \cdot \pi_\theta(a_t^{\text{think}}, a_t \mid H_{t-1}) \Big)}_{\text{feedback-conditioned subsequent edits}}, \tag{11}$$

where only π_θ is optimized.

End-to-End Reinforcement Learning. We adopt the standard GRPO objective [Shao et al., 2024], modified by one canvas-specific adaptation: a *token-level mask* $\text{mask}_t^{(i)} \in \{0, 1\}$ that restricts the policy gradient to tokens generated by Flow-Director, excluding canvas-feedback tokens that share the same context window. Given N rollouts $\{\tau_i\}_{i=1}^N$ for task q sampled under behavior policy $\pi_{\theta_{\text{old}}}$, the masked objective is

$$\mathcal{J}(\theta) = \mathbb{E}_{q \sim \mathcal{D}_Q,\, \{\tau_i\} \sim \pi_{\theta_{\text{old}}}} \left[\frac{1}{N} \sum_{i=1}^{N} \frac{1}{|\tau_i|_{\text{mask}}} \sum_{t=1}^{|\tau_i|} \text{mask}_t^{(i)} \cdot \min\Big(\rho_\theta^{(i,t)} \hat{A}_i,\ \text{clip}\big(\rho_\theta^{(i,t)},\, 1 \pm \epsilon\big)\, \hat{A}_i \Big) - \beta\, D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}) \right], \tag{12}$$

where $|\tau_i|_{\text{mask}} = \sum_t \text{mask}_t^{(i)}$ counts policy-generated tokens in τ_i, and the importance ratio $\rho_\theta^{(i,t)}$, the group-standardized trajectory-level advantage $\hat{A}_i$ computed from the trajectory-level reward $R(\tau_i)$, the clip range ϵ, the KL strength β, the reference policy π_{ref}, and the behavior policy $\pi_{\theta_{\text{old}}}$ all follow the standard GRPO formulation [Shao et al., 2024]. The outcome reward $R(\tau)$ combines a structural reward over the workflow graph and an answer reward that is released only when the structural reward is satisfied; full forms are deferred to Appendix C.1.

Proposition 3. *End-to-end training of the canvas-feedback-driven multi-turn loop yields stable orchestration policies, suppresses shortcut behaviors, and improves long-horizon credit assignment (proof in Appendix B.3; evidence in Sections 5.5 and 5.6).*

Table 3: Main results on twelve IID and OOD benchmarks. SFT and GRPO are applied on Qwen3-8B; all workflow methods use GPT-4o-mini as the evaluation executor (Flow-Director is trained with GPT-OSS-120B). $\Delta \uparrow$ is the gap to GPT-4o-mini.

		Baseline		SFT	GRPO	AFlow	Agent+RL (4o-mini)			Ours (4o-mini)
Dataset	**Metric**	**Qwen3-8B**	**4o-mini**	**Qwen3-8B**	**Qwen3-8B**	**4o-mini**	**Agentflow**	**Router-R1**	**Orchestrator**	**FlowSteer ($\Delta \uparrow$)**
Part I: In-Distribution (IID) Benchmarks										
HotPotQA	**EM**	67.19	63.28	70.31	59.38	68.75	67.19	72.00	67.97	**78.12** (+14.84)
	F1	74.05	73.03	75.25	64.95	77.90	77.88	79.84	75.61	**84.98** (+11.95)
SQuAD v2	**EM**	54.69	47.66	73.44	66.41	73.44	64.06	59.84	70.34	**78.12** (+30.46)
	F1	61.54	59.42	77.31	72.00	82.41	72.45	65.29	75.24	**83.67** (+24.25)
GSM8K	**Acc.**	91.41	92.97	92.19	92.97	94.53	93.75	94.01	93.94	**96.09** (+3.12)
MATH	**Acc.**	66.41	60.94	61.72	68.75	70.31	71.87	76.56	72.26	**81.25** (+20.31)
MBPP	**Pass@1**	63.28	64.84	57.03	77.34	83.20	79.69	73.43	74.22	**84.38** (+19.54)
HumanEval	**Pass@1**	81.25	82.81	61.72	86.72	90.62	87.50	85.15	89.06	**92.96** (+10.15)
Avg. (IID)	**EM**	60.94	55.47	71.88	62.90	71.10	65.63	65.92	69.16	**78.12** (+22.65)
	F1	67.80	66.23	76.28	68.48	80.16	75.17	72.57	75.43	**84.33** (+18.10)
	Acc./Pass	75.59	75.39	68.17	81.45	84.67	83.20	82.29	82.37	**88.67** (+13.28)
Part II: Out-of-Distribution (OOD) Benchmarks										
TriviaQA	**EM**	60.16	71.09	60.94	59.38	73.44	75.00	75.78	77.36	**79.69** (+8.60)
	F1	69.17	81.40	69.88	69.23	82.50	81.47	80.43	83.23	**84.11** (+2.71)
NaturalQuestions	**EM**	39.84	39.84	46.09	43.75	42.97	45.70	49.22	50.00	**54.69** (+14.85)
	F1	50.75	51.42	53.40	53.24	49.92	55.98	52.79	55.41	**62.56** (+11.14)
MathQA	**Acc.**	75.00	79.69	61.71	60.15	83.59	82.81	80.47	82.03	**88.67** (+8.98)
AIME 2025	**Acc.**	16.66	10.00	0.00	8.33	13.33	10.00	10.00	20.00	**26.67** (+16.67)
APPS	**Pass@1**	39.84	40.62	26.56	34.38	42.97	41.41	42.95	44.53	**49.21** (+8.59)
DS-1000	**Pass@1**	34.38	45.31	25.78	38.28	53.91	46.88	42.97	51.56	**58.59** (+13.28)
Avg. (OOD)	**EM**	50.00	55.47	53.52	51.57	58.21	60.35	62.50	63.68	**67.19** (+11.72)
	F1	59.96	66.41	61.64	61.24	66.21	68.73	66.61	69.32	**73.34** (+6.93)
	Acc./Pass	41.47	43.91	28.51	35.29	48.45	45.28	44.10	49.53	**55.79** (+11.88)

5 Experiments

We evaluate FlowSteer through the following research questions (RQs): **RQ1:** Can FlowSteer outperform existing workflow orchestration methods? **RQ2:** How does FlowSteer generalize to out-of-distribution benchmarks? **RQ3:** How transferable is FlowSteer across different LLM backends? **RQ4:** What are the contributions of core components such as the multi-turn interaction paradigm and RL modules? **RQ5:** How does our canvas-aware RL objective compare against other RL algorithms (GRPO, DAPO)?

5.1 Experimental Setup

We evaluate FlowSteer on diverse in-distribution (IID) and out-of-distribution (OOD) benchmarks, compare against strong baselines, and report standard task-specific metrics.

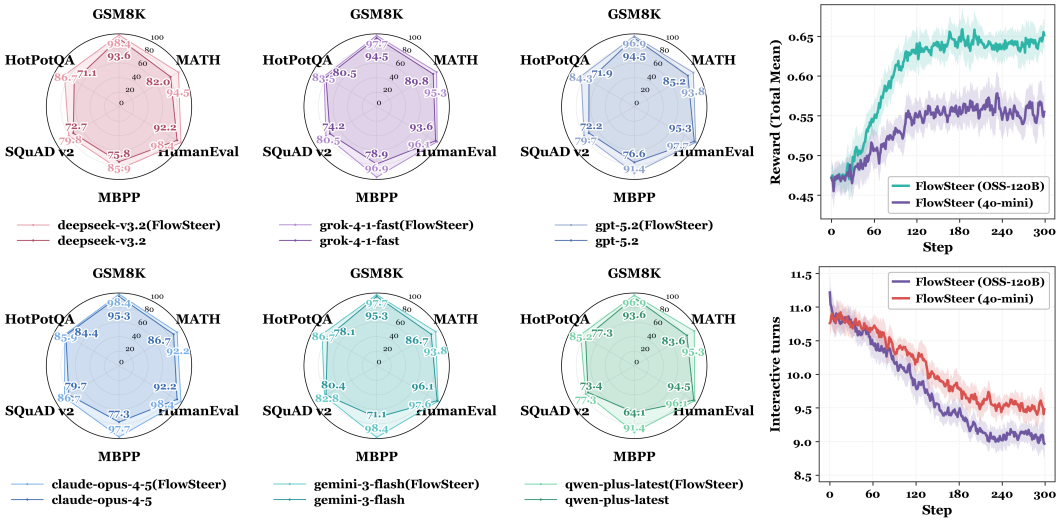
Datasets. We use six IID benchmarks covering three task categories: *(i)* mathematical reasoning (GSM8K [Cobbe et al., 2021], MATH [Hendrycks et al., 2021b]), *(ii)* question answering (HotPotQA [Yang et al., 2018], SQuAD v2 [Rajpurkar et al., 2018]), and *(iii)* code generation (MBPP [Austin et al., 2021], HumanEval [Chen et al., 2021]). To assess generalization, we also test on six OOD benchmarks: TriviaQA [Joshi et al., 2017], NaturalQuestions [Kwiatkowski et al., 2019], MathQA [Amini et al., 2019], AIME 2025, APPS [Hendrycks et al., 2021a], and DS-1000 [Lai et al., 2023]. All methods are evaluated on the same fixed set of 1,438 test instances, with 128 instances sampled once from each dataset and all 30 AIME 2025 problems. More details in Appendix D.

Baselines. We compare FlowSteer against the following baselines: *(i)* direct LLM baselines (Qwen3-8B [Yang et al., 2025], GPT-4o-mini [Hurst et al., 2024]), *(ii)* standard fine-tuning methods (SFT, GRPO [Shao et al., 2024]), *(iii)* workflow-based methods (AFlow [Zhang et al., 2024]), and *(iv)* agent with RL methods (AgentFlow [Li et al., 2025a], Router-R1 [Zhang et al., 2025], Orchestrator [Dang et al., 2025]). Unless noted otherwise, Flow-Director is trained with GPT-OSS-120B, and all workflow methods are evaluated with the GPT-4o-mini executor. More details are provided in Appendix E.

Evaluation Metrics. Following standard practice, we use Exact Match (EM) and F1 score for question answering tasks, Accuracy (Acc.) for mathematical reasoning tasks, and Pass@1 for code generation tasks. More details of the evaluation metrics are shown in Appendix F.

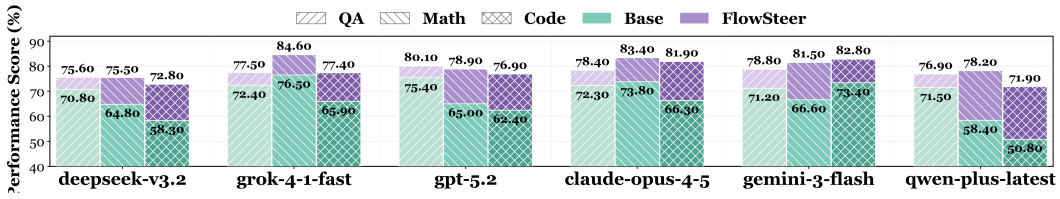
Table 4: Ablation of FlowSteer on twelve datasets (GPT-4o-mini backend). Component-level rows remove the agent, multi-turn interaction, canvas, or RL; RL-objective rows remove the token mask, diversity-constrained reward, or conditional release.

Method	IID								OOD							
	GSM8K	MATH	HotPotQA		SQuAD v2		MBPP	HEval	TriviaQA		NQ		MathQA	AIME	APPS	DS-1k
	Acc.	Acc.	EM	F1	EM	F1	Pass@1	Pass@1	EM	F1	EM	F1	Acc.	Acc.	Pass@1	Pass@1
Component-level ablations																
w/o Agent	94.53	70.31	72.66	78.91	66.41	70.63	60.94	83.59	69.53	75.08	53.12	57.43	80.47	23.33	41.40	40.63
w/o Multi-turn	91.41	75.00	72.66	81.09	56.25	66.48	57.81	88.28	63.28	68.28	43.75	55.00	78.13	20.00	38.84	41.41
w/o Canvas	94.53	73.44	70.31	79.53	59.38	69.53	63.28	85.16	57.81	65.63	53.13	61.48	78.91	10.00	42.96	45.31
w/o RL	91.41	71.81	72.66	82.19	59.38	67.66	64.06	89.06	64.84	71.80	53.13	60.72	80.47	20.00	38.28	49.22
RL-objective ablations																
w/o Token Mask	93.75	67.97	70.31	76.32	75.78	80.23	81.25	89.84	71.88	76.31	53.12	59.94	78.12	23.33	39.06	46.88
w/o Div. Reward	94.53	70.31	69.53	75.44	75.00	79.43	82.81	88.28	71.09	75.05	51.56	55.43	78.91	16.67	37.50	46.09
w/o Cond. Rel.	91.41	74.21	72.66	78.79	75.00	79.69	82.03	89.84	70.31	75.40	46.88	50.69	83.59	23.33	42.19	50.78
FlowSteer (Full)	**96.09**	**81.25**	**78.12**	**84.98**	**78.12**	**83.67**	**84.38**	**92.96**	**79.69**	**84.11**	**54.69**	**62.56**	**88.67**	**26.67**	**49.21**	**58.59**



(a) Radar charts on different backends

(b) Training dynamics



(c) Aggregated performance by task type

Figure 4: Transferability analysis of FlowSteer across six LLM backends (RQ3), using one frozen Flow-Director trained with GPT-OSS-120B. (a) Per-backend radar comparison on six IID benchmarks. (b) Training dynamics with GPT-4o-mini vs. GPT-OSS-120B as training executor. (c) Aggregated performance by task type (QA, math, and code), Base versus FlowSteer.

5.2 Main Results (RQ1)

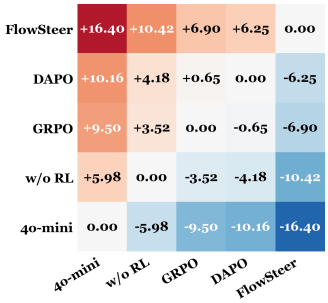
As illustrated in Table 3, FlowSteer achieves consistent improvements over GPT-4o-mini across six IID benchmarks and outperforms multiple baseline categories. We attribute these gains to three interlocking design choices. First, the Workflow Canvas exposes graph-level execution feedback after every atomic edit, so structural errors are repaired *in-loop* rather than only at the end of generation, which drives the largest gain on long-horizon mathematical reasoning. Second, the Flow-Director–executor decoupling lets a lightweight policy focus on transferable orchestration while the pluggable backend handles task-specific computation, allowing a small policy to *amplify* a strong backend without backend fine-tuning. Third, progressive canvas editing decomposes long construction into

Table 5: Transfer across training and target executors, averaged over the twelve datasets. Each row is a frozen Flow-Director trained with the listed executor; Base runs the target executor without FlowSteer; † marks matched training and target executors.

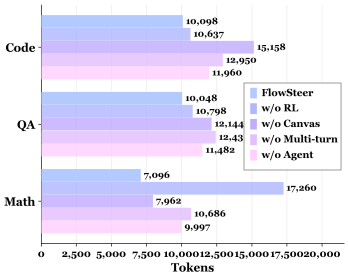
Train ↓ / Target →	**DS-V3.2**	**Opus-4.5**	**Gemini-3-F**	**OSS-120B**	**4o-mini**	**Qwen3.5-9B**	**Avg.**
Base (no FlowSteer)	72.93	78.37	77.51	66.42	58.25	60.56	69.01
GPT-OSS-120B	82.63	87.23	86.93	**80.60**†	72.37	76.81	81.10
DeepSeek-V3.2	**85.77**†	89.86	87.25	79.98	73.06	77.32	82.21
Claude-Opus-4.5	83.59	**91.98**†	**89.46**	80.11	**73.54**	**77.58**	**82.71**

Group	**Setting**	**GSM8K**	**MATH**	**HotPot**	**SQuAD2**	**MBPP**	**HEval**	**TrivQA**	**NQ**	**MathQA**	**AIME**	**APPS**	**DS1K**
Full	—	**96.09**	**81.25**	**78.12**	**78.12**	**84.38**	**92.96**	**79.69**	**54.69**	**88.67**	**26.67**	**49.21**	**58.59**
Removal	−Review/Revise	96.09	81.25	71.09	73.44	84.38	92.96	79.69	50.78	88.67	26.67	49.21	58.59
	−ScEnsem/Aggr	96.09	81.25	75.78	78.12	84.38	92.96	79.69	53.12	88.67	20.00	49.21	58.59
	−Verify/Test	94.53	70.31	78.12	75.78	78.91	86.72	79.69	54.69	85.16	20.00	35.16	38.28
	Minimal (4 ops)	90.62	68.75	65.63	71.09	78.91	86.72	78.12	48.44	85.16	16.67	35.16	38.28
Subst.	Prog→Jupyter	96.09	84.38	78.12	78.12	85.94	92.96	79.69	54.69	88.67	30.00	52.34	62.50
	Custom→GenSkills	96.09	81.25	84.38	78.91	84.38	92.96	79.69	57.03	88.67	26.67	49.21	58.59
	Review→SelfCrit	96.09	81.25	71.88	77.34	84.38	92.96	73.44	50.00	88.67	26.67	49.21	58.59
Addition	+Search	96.09	81.25	71.88	77.34	84.38	92.96	85.16	63.28	88.67	26.67	49.21	58.59
	+Calculator	96.09	80.47	78.12	78.12	84.38	92.96	79.69	54.69	88.28	30.00	49.21	58.59
	+Debugger	96.09	81.25	78.12	78.12	84.38	92.96	79.69	54.69	88.67	33.33	51.56	62.50

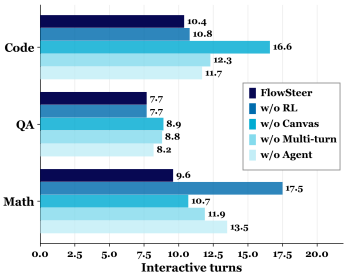
(a) Operator-level ablation across 12 datasets



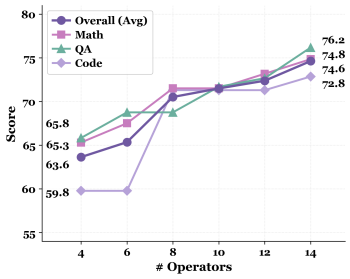
(b) Overall pairwise



(c) Token consumption



(d) Interactive turns



(e) Performance scaling

Figure 5: Ablation, scaling, and RL algorithm analysis. (a) Operator-level ablation. (b) Pairwise comparison across RL algorithms (averaged across IID and OOD benchmarks; row − col). (c–d) Token consumption and interaction turns. (e) Performance scaling with operator-library size.

checkable, repairable local decisions, yielding cleaner credit assignment and consistent Pass@1 gains on code generation.

5.3 Generalization Results (RQ2)

As shown in Table 3, FlowSteer maintains its superiority over baselines on six OOD benchmarks, with stable improvements in question answering and mathematical reasoning. The OOD generalization stems from the same decoupled design that drives the IID gains: because the Flow-Director learns to compose task-agnostic cognitive operators rather than task-specific solution patterns, the orchestration strategy fitted on IID training carries over naturally to novel task families such as AIME 2025 and DS-1000. In contrast, large-scale LLM backends show strong zero-shot capability but lack transferable orchestration, while search-based workflows are inefficient in OOD settings. These results show that FlowSteer achieves robust cross-task and cross-distribution gains without task-specific fine-tuning of the backend LLM.

5.4 Transferability across LLM Backends (RQ3)

As shown in Figure 4, a single Flow-Director trained with GPT-OSS-120B and then frozen yields consistent improvements across all six target backends on IID benchmarks (Figure 4(a)), with weaker baselines benefiting more, and stable gains across math, QA, and code on the aggregated OOD evaluation (Figure 4(c)). The transfer holds because the frozen policy conditions every edit on

canvas feedback returned by the target executor (Eq. 11), so it carries over orchestration patterns such as decomposition, verification, and revision, and rebuilds the graph for the new executor rather than replaying a fixed one. Table 5 further varies the training executor itself: every frozen Flow-Director improves every target executor, including transfer from the strongest Claude-Opus-4.5 to the weaker GPT-4o-mini and Qwen3.5-9B. A stronger training executor raises the overall ceiling because its more reliable execution and verification feedback separates the effects of different edits more clearly, and a matched training and target executor adds a further gain (†). The training dynamics (Figure 4(b)) further show that locally deployed GPT-OSS-120B reaches higher reward with more stable optimization than GPT-4o-mini while converging to similar interaction-turn counts, indicating that lower-cost local executors can effectively replace API backbones for training, trailing a Claude-Opus-4.5-trained Flow-Director by only 1.6 points on average (Table 5; details in Appendix H.3).

5.5 Ablation Study (RQ4)

As shown in Table 4 and Figure 5, removing any component (agent, multi-turn interaction, canvas, or RL) degrades performance across all twelve datasets, with the largest drops on math/code (RL) and QA (canvas feedback); the three RL-objective adaptations (token mask, diversity-constrained reward, conditional release) are likewise complementary and each removal substantially hurts one or more task families. The control-structure check R_{control} serves as a training-time exploration signal rather than a template: without it, FlowSteer still exceeds both the fixed graph (w/o Agent) and selection among predefined templates on average, the full model is best on all twelve datasets (Appendix H.2), and the Flow-Director still builds plain sequential graphs for simple QA at inference (Appendix I.4). Operator-level ablations (Figure 5(a)) show that adding or substituting operators maintains or improves performance while removing verification operators or shrinking the library to four operators causes the largest drops, so the Flow-Director composes operators by function rather than by memorized names; scaling the operator library from 4 to 14 operators steadily lifts Overall score from 63.6 to 74.6 (Figure 5(e)). The full model also reduces token consumption and interaction turns relative to all ablation variants (Figure 5(c–d)) while maintaining accuracy. Since w/o Agent and w/o Multi-turn correspond to static workflows and single-shot workflow generation, progressive editing adds short orchestration turns without raising total token cost, and workflow execution rather than Flow-Director orchestration accounts for most wall-clock time on every dataset (Appendix H.4).

5.6 Comparison of RL Algorithms (RQ5)

As shown in Figure 5(b), FlowSteer outperforms GRPO and DAPO under identical training settings. The canvas-aware token mask removes biased gradient signals from environment-feedback tokens, and the structural-then-answer reward gate suppresses shortcut behaviors — two failure modes GRPO and DAPO inherit from treating the trajectory as a structurally opaque sequence.

6 Conclusion

In this work, we propose FlowSteer, an end-to-end reinforcement learning framework for interactive workflow orchestration. By training a lightweight policy model to interact with an executable Workflow Canvas, FlowSteer learns transferable orchestration strategies from real execution feedback. Recent work points to further directions for such orchestrators, including evolving skill libraries, learned agent collaboration, self-play self-improvement, and continual self-evolution of science agents [Zhang et al., 2026d, Huang et al., 2026, Zhang et al., 2026b,c].

Acknowledgments and Disclosure of Funding

This work is supported in part by the Guangdong Basic and Applied Basic Research Foundation under Grant No. 2025A1515012968, in part by the Shenzhen Science and Technology Program under Grant No. JCYJ20240813113502004, in part by the National Natural Science Foundation of China under Grant Nos. 62001412 and 62366057, in part by the National Social Science Fund of China (NSSFC) under Grant No. 25CFX009, in part by Shenzhen Stability Science Program 2023, in part by the Guangdong Provincial Key Laboratory of Future Networks of Intelligence (Grant No. 2022B1212010001), in part by the Shenzhen Key Lab of Crowd Intelligence Empowered Low-Carbon Energy Network (Grant No. ZDSYS20220606100601002), in part by the Key Research and Development Program of Yunnan Province under Grant Nos. 202502AD080017, 202502AD080013,

and 202402AA310056, and in part by the Special Fund for the Central Government to Guide Local Scientific and Technological Development under Grant No. 202407AB110003. The authors declare no competing interests.

References

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2357–2367, 2019.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, et al. Multi-agent collaboration via evolving orchestration. *arXiv preprint arXiv:2505.19591*, 2025.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. *arXiv preprint arXiv:2503.09572*, 2025.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent llm systems for workflow automation. *arXiv preprint arXiv:2510.04851*, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021a.

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. *arXiv preprint arXiv:2103.03874*, 2021b.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Xiao Huang, Mingda Zhang, Junming Zhang, Qiang Huang, Hanwen Zhang, Yue Dai, Zijia Wang, and Xiaoying Tang. Collabflow: Recursive self-improvement of agent collaboration. *arXiv preprint arXiv:2609.38662*, 2026.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Yuxiang Ji, Ziyu Ma, Yong Wang, Guanhua Chen, Xiangxiang Chu, and Liaoni Wu. Tree search for llm agent reinforcement learning. *arXiv preprint arXiv:2509.21240*, 2025.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, et al. Verltool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*, 2025.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.

Mingze Kong, Zikun Qu, Zhongquan Zhou, Pengyu Liang, Xiang Li, Zhiwei Shang, Zhi Hong, Kaiyu Huang, Zhiyong Wang, and Zhongxiang Dai. Workflow-r1: Group sub-sequence policy optimization for multi-turn workflow construction. *arXiv preprint arXiv:2602.01202*, 2026.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, and Fengli Xu. Agentswift: Efficient llm agent design via value-guided hierarchical search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 31843–31851, 2026.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. In-the-flow agentic system optimization for effective planning and tool use. *arXiv preprint arXiv:2510.05592*, 2025a.

Ziyue Li, Yuan Chang, Gaihong Yu, and Xiaoqiu Le. Hiplan: Hierarchical planning for llm-based agents with adaptive global-local guidance. *arXiv preprint arXiv:2508.19076*, 2025b.

Wenjin Liu, Haoran Luo, Xueyuan Lin, Haoming Liu, Tiesunlong Shen, Jiapu Wang, Rui Mao, and Erik Cambria. Prompt-r1: Collaborative automatic prompting framework via end-to-end reinforcement learning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 16260–16280, 2026.

Xiaoqian Liu, Ke Wang, Yuchuan Wu, Fei Huang, Yongbin Li, Junge Zhang, and Jianbin Jiao. Agentic reinforcement learning with implicit step rewards. *arXiv preprint arXiv:2509.19199*, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Yujia Peng, Jiaheng Han, Zhenliang Zhang, Lifeng Fan, Tengyu Liu, Siyuan Qi, Xue Feng, Yuxi Ma, Yizhou Wang, and Song-Chun Zhu. The tong test: Evaluating artificial general intelligence through dynamic embodied physical and social interactions. *Engineering*, 34:12–22, 2024.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, 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.

Cheng Qian, Zuxin Liu, Shirley Kokane, Akshara Prabhakar, Jielin Qiu, Haolin Chen, Zhiwei Liu, Heng Ji, Weiran Yao, Shelby Heinecke, et al. xrouter: Training cost-aware llms orchestration system via reinforcement learning. *arXiv preprint arXiv:2510.08439*, 2025.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 784–789, 2018.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, et al. Tptu: Task planning and tool usage of large language model-based ai agents. In *NeurIPS 2023 foundation models for decision making workshop*, 2023.

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

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*, 2024.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.

Ruiyi Wang and Prithviraj Ammanabrolu. A practitioner's guide to multi-turn agentic reinforcement learning. *arXiv preprint arXiv:2510.01132*, 2025.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Fei Wu, Tao Shen, Thomas Bäck, Jingyuan Chen, Gang Huang, Yaochu Jin, Kun Kuang, Mengze Li, Cewu Lu, Jiaxu Miao, et al. Knowledge-empowered, collaborative, and co-evolving ai models: The post-llm roadmap. *Engineering*, 44:87–100, 2025a.

Junde Wu, Jiayuan Zhu, Yuyuan Liu, Min Xu, and Yueming Jin. Agentic reasoning: A streamlined framework for enhancing llm reasoning with agentic tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 28489–28503, 2025b.

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.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzefa Rangwala. Agentoccam: A simple yet strong baseline for llm-based web agents. *arXiv preprint arXiv:2410.13825*, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 2369–2380, 2018.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yuanqing Yu, Zhefan Wang, Weizhi Ma, Shuai Wang, Chuhan Wu, Zhiqiang Guo, and Min Zhang. Steptool: Enhancing multi-step tool usage in llms via step-grained reinforcement learning. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, pages 3952–3962, 2025.

Zhen Zeng, William Watson, Nicole Cho, Saba Rahimi, Shayleen Reynolds, Tucker Balch, and Manuela Veloso. Flowmind: automatic workflow generation with llms. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, pages 73–81, 2023.

Haozhen Zhang, Tao Feng, and Jiaxuan You. Router-r1: Teaching llms multi-round routing and aggregation via reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2024.

Mingda Zhang, Qiang Huang, Yanjin Li, Zijia Wang, Qika Lin, Xiaoying Tang, and Tiesunlong Shen. R^2 flow: Recursive self-improvement via recursive skill evolution. *arXiv preprint arXiv:2609.33867*, 2026a.

Mingda Zhang, Yun Huang, Qiang Huang, Yanjin Li, Rui Qian, Qichao Zhao, Tiesunlong Shen, Na Zhao, Jian Wang, and Xiaoying Tang. R^2orch: Recursive self-improvement for agent orchestration via self-play within a skill-internalization loop. *arXiv preprint arXiv:2610.22740*, 2026b.

Mingda Zhang, Wenjin Liu, Tiesunlong Shen, Zikai Xiao, Zhenghong Lin, Qing Xu, Erik Cambria, Xiaoying Tang, and Haoran Luo. Scienceclaw: Benchmarking continual self-evolution of ai-for-science agents across the natural and social sciences. *arXiv preprint arXiv:2610.08691*, 2026c.

Mingda Zhang, Tiesunlong Shen, Haoran Luo, Wenjin Liu, Zikai Xiao, Erik Cambria, and Xiaoying Tang. Skillflow: Flow-driven recursive skill evolution for agentic orchestration. *arXiv preprint arXiv:2605.14089*, 2026d.

Mingda Zhang, Hanwen Zhang, Qiang Huang, Zijia Wang, Pengfei Guo, Yuchen Zhang, Jionghao Zhu, and Xiaoying Tang. Evosteer: Online self-evolving graph orchestration via reference-anchored credit assignment. *arXiv preprint arXiv:2609.38661*, 2026e.

Qinghua Zheng, Huan Liu, Xiaoqing Zhang, Caixia Yan, Xiangyong Cao, Tieliang Gong, Yong-Jin Liu, Bin Shi, Zhen Peng, Xiaocen Fan, et al. Machine memory intelligence: Inspired by human memory mechanisms. *Engineering*, 2025.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. Archer: Training language model agents via hierarchical multi-turn rl. *arXiv preprint arXiv:2402.19446*, 2024.

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

System Prompt for Flow-Director

You are building a workflow step by step to solve the problem. In each turn, output EXACTLY ONE XML action (add/delete/modify/set_prompt/finish or a structure add). The goal is to build a reliable workflow, not a single-shot answer. Keep your thinking brief (under 200 words) and focus on choosing the next action, not solving the whole problem yourself. Let the operators do the computation. If you use <think>...</think>, you MUST output an <action> tag AFTER it.

Available Operators (12 total). Programmer: write and execute Python code. Plan: create solution strategy. Custom: natural language reasoning. Decompose: break into sub-problems. Test: run test cases. Review: evaluate quality. Verify: double-check result. Revise: fix issues. ScEnsemble: multiple solutions voting. Aggregate: combine results. AnswerGenerate: format final answer. Format: extract concise answer.

Actions (8 types). The system supports 8 action types: add (add a single operator), finish (complete workflow building), parallel (add parallel branches), conditional (add conditional branch), loop (add loop structure), delete (remove a node), modify (change operator type), and set_prompt (set custom prompt for an operator). All actions use XML format for reliable parsing.

Finish Policy. Always add Format as the last step before finishing to extract concise answer. Before finishing, add Format to extract concise answer from the solution. When Format has extracted the answer and you are satisfied, output <action>finish</action>. When the result is wrong or needs improvement, add more operators.

Figure 6: The system prompt template for Flow-Director. The prompt instructs the agent to build workflows step by step through multi-turn interactions with the Workflow Canvas.

Table 6: Summary of the 12 operators in $\mathcal{O}$ with input/output specifications.

Operator	Category	Input	Output	Description
Plan	Planning	problem	approach, plan	Creates high-level solution strategy by identifying the overall approach and breaking it into actionable steps.
Decompose	Planning	problem	sub_problems	Breaks complex problems into smaller, independent sub-problems that can be solved separately.
Programmer	Solving	problem, analysis	code, output	Writes and executes Python code to compute answers; used for mathematical calculations and code generation tasks.
Custom	Solving	input, context	response	Performs natural language reasoning without code execution; used for QA, analysis, and explanations.
AnswerGenerate	Solving	input	thought, answer	Generates structured answers with explicit reasoning chains; similar to Custom but with formatted output.
Test	Verification	code, tests	pass/fail, solution	Executes unit tests against generated code and triggers automatic revision if tests fail; code-specific.
Review	Verification	solution	is_correct, feedback	Evaluates solution quality through critique and provides detailed feedback for potential revision.
Verify	Verification	answer	is_correct, answer	Independently re-solves the problem to verify correctness; does not execute code, uses logical reasoning.
Revise	Revision	solution, feedback	revised_solution	Fixes issues in solutions based on feedback from Review or Test operators.
ScEnsemble	Ensemble	solutions	selected_solution	Implements majority voting across multiple candidate solutions to select the most consistent answer.
Aggregate	Ensemble	sub_answers	aggregated	Combines results from parallel branches or sub-problems into a unified answer.
Format	Formatting	solution	final_answer	Extracts concise final answer from verbose solution; required as the last step before finishing.

A Prompts and Operator Library in FlowSteer

A.1 System Prompt for Flow-Director

As shown in Figure 6, we adopt the system prompt template for the Flow-Director, where the agent alternates between internal reasoning in `<think>` and communicates with the Workflow Canvas by `<action>`. The prompt instructs the agent to output exactly one XML action per turn, keeping thinking brief and focusing on action selection rather than problem-solving. The workflow evolves step by step until finalized with a `finish` action.

A.2 Operator Library $\mathcal{O}$

As shown in Table 6, the operator library $\mathcal{O}$ includes 12 operators in six categories: planning (Plan, Decompose), solving (Programmer, Custom, AnswerGenerate), verification (Test, Review, Verify),

revision (Revise), ensemble (ScEnsemble, Aggregate), and formatting (Format). The Programmer operator executes Python code in a sandboxed environment with scientific libraries under a 30-second timeout.

Multi-turn Interaction Example

Problem: "Find the sum of all positive integers n such that $n^2 + 12n - 2007$ is a perfect square."

Round 1 (BUILDING → AWAITING_PROMPT)
Flow-Director:

```
<think>This is a number theory problem. Start with Plan.</think>
<action>add</action><operator>Plan</operator>
```

Canvas Feedback: "Added Plan as node_1. State: AWAITING_PROMPT."

Round 2 (AWAITING_PROMPT → BUILDING)
Flow-Director:

```
<action>set_prompt</action><target>node_1</target>
<prompt>Analyze the equation and identify the approach.</prompt>
```

Canvas Feedback: "Prompt set. State: BUILDING. DSL: Plan"

Round 3-4: Add Programmer operator with prompt for algebraic computation.

Round 5-6: Add Verify operator with prompt for result validation.

Round 7-8: Add Format operator with prompt for answer extraction.

Round 9 (BUILDING → Finished)
Flow-Director:

```
<think>Workflow complete. Ready to finish.</think>
<action>finish</action>
```

Canvas Feedback: "Workflow finished. Final DSL: Plan -> Programmer -> Verify -> Format. Answer: 89"

Figure 7: A complete multi-turn interaction example showing the two-step mechanism (add + set_prompt) for workflow construction.

A.3 Multi-turn Interaction Process

The Flow-Director builds workflows through multi-turn interactions with the Workflow Canvas. Each interaction round consists of the Flow-Director generating an action and the Canvas providing execution feedback.

A.3.1 Environment State Machine

The Workflow Canvas maintains a finite state machine with two states:

- **BUILDING**: The normal state where the Flow-Director can execute `add`, `delete`, `modify`, or `finish` actions.
- **AWAITING_PROMPT**: After adding an operator, the Canvas transitions to this state, requiring the Flow-Director to specify a custom prompt via `set_prompt` before returning to BUILDING.

This two-step mechanism reduces the cognitive load on the policy model by separating structural decisions (which operator to add) from content decisions (what prompt to use), improving the effectiveness of small-scale models.

A.3.2 Action Types

The 8 action types available to the Flow-Director are summarized in Table 7. The basic actions (add, finish, delete, modify, set_prompt) enable sequential workflow construction, where the Flow-Director progressively builds the operator graph through iterative additions and modifications. The advanced actions (parallel, conditional, loop) extend the expressiveness to complex control structures,

allowing the Flow-Director to construct workflows with branching, conditional execution, and iterative refinement patterns.

Table 7: Action types in FlowSteer.

Action	Description
`add`	Add a single operator
`finish`	Complete workflow building
`delete`	Remove a node
`modify`	Change operator type
`set_prompt`	Set custom prompt
`parallel`	Add parallel branches
`conditional`	Add conditional branch
`loop`	Add loop structure

A.3.3 Complete Interaction Example

A complete multi-turn interaction between the Flow-Director (small-scale policy model) and the Workflow Canvas (large-scale LLM backend) is illustrated in Figure 7. The process begins with the Flow-Director analyzing the problem and selecting an appropriate initial operator, which establishes the reasoning strategy for subsequent workflow construction. The Flow-Director then engages in iterative exchanges: after each canvas feedback, it reflects on the current workflow state, selects the next operator, and specifies task-specific prompts. Through successive rounds of construction and verification, the workflow gradually takes shape, and the Canvas executes the completed workflow to produce the final answer.

B Theoretical Proofs

In this appendix, we provide detailed proofs for Propositions 1–3 stated in the main text. We first introduce the notation and assumptions, then present each proof in turn.

Notation. Given a task q, a workflow is represented as a directed graph $\mathcal{G} = (V, E, \mathrm{attr})$, where each node $v \in V$ is bound to an operator $\mathrm{op}(v) \in \mathcal{O}$ with attributes $\mathrm{attr}(v) = (\mathrm{param}(v), \mathrm{prompt}(v), \dots)$. The executor schedules nodes according to dependency edges E and produces output $y = \mathrm{Execute}(\mathcal{G}, q, \mathcal{M}_{\mathrm{exec}})$. During multi-turn orchestration, the policy model (Flow-Director) interacts with the canvas (Workflow Canvas), forming a trajectory

$$\tau = \{(a_t^{\mathrm{think}}, a_t, o_t^{\mathrm{exec}})\}_{t=1}^{T}, \tag{13}$$

where a_t^{think} denotes the reflection text that summarizes the current state and identifies potential issues, a_t denotes the edit action comprising an action type and its content, and o_t^{exec} denotes the execution and validation feedback returned by the canvas environment. The operator library consists of 12 functional operators:

$$\begin{aligned}\mathcal{O} = \{&\texttt{Plan}, \texttt{Decompose}, \texttt{Programmer}, \texttt{Custom}, \texttt{AnswerGenerate}, \texttt{Test}, \\ &\texttt{Review}, \texttt{Verify}, \texttt{Revise}, \texttt{ScEnsemble}, \texttt{Aggregate}, \texttt{Format}\},\end{aligned} \tag{14}$$

where each operator implements a specific cognitive function in the problem-solving process. The action type set consists of 8 types:

$$\mathcal{A}_{\mathrm{type}} = \{\texttt{add}, \texttt{delete}, \texttt{modify}, \texttt{set_prompt}, \texttt{finish}, \texttt{parallel}, \texttt{conditional}, \texttt{loop}\}, \tag{15}$$

where the first five types can be understood as local editing actions, and the last three types are used to explicitly generate control structures (parallel, conditional, loop) that form long-range control flows.

Assumptions. To obtain formal proofs, we introduce three mild and interpretable assumptions. They do not require the real system to be perfect, but are sufficient to support the theoretical justification of why the propositions hold.

Assumption 1 (Cognitive Decomposability). For the task families considered in this work (multi-hop QA, standard QA, mathematical reasoning, code generation), there exists a goal-directed problem-solving procedure that can be decomposed into a finite combination of cognitive primitives (defined below), and can be represented as a directed graph with conditionals and loops.

Assumption 2 (Informative Canvas Feedback). The canvas feedback o_t^{exec} is informative about whether the current workflow is executable, whether it satisfies structural constraints, and which local modifications can fix errors. Formally, there exists non-zero probability that the feedback changes the agent's posterior distribution over the optimal next edit, i.e., $I(Z; o_t^{\text{exec}} \mid H_{t-1}) > 0$, where Z denotes the latent variable related to the solution (e.g., the correct answer or a feasible workflow).

Assumption 3 (Repairability). When a workflow is in a non-executable or constraint-violating state, the canvas can provide sufficiently localized failure reasons or repair suggestions, such that there exist editing actions that can push it toward a more feasible state. Formally, there exists a potential function $\Phi(\mathcal{G})$ measuring the distance to the feasible set, and there exist actions such that $\mathbb{E}[\Phi(\mathcal{G}_t) \mid H_{t-1}] < \Phi(\mathcal{G}_{t-1})$.

B.1 Proof of Proposition 1

Proposition 1 (Operator-Action Cognitive Completeness)**.** *Let the operator library $\mathcal{O}$ and action type set $\mathcal{A}_{\text{type}}$ be defined as above. Under Assumption 1, for any task q, there exists a finite-length action sequence $\{a_t\}_{t=1}^{T}$ with each action type belonging to $\mathcal{A}_{\text{type}}$, such that starting from an empty canvas $\mathcal{G}_0$, the iterative updates $\mathcal{G}_t = \text{Update}(\mathcal{G}_{t-1}, a_t, o_t^{\text{exec}})$ can construct a terminal workflow $\mathcal{G}_T$ that implements a complete cognitive control loop, thereby covering the key problem-solving steps required for the task types considered in this work (mathematics, QA, code generation).*

Proof. This proposition requires proving two things: (i) functional completeness, showing that the operator library covers the core cognitive modules required for goal-directed problem solving; and (ii) structural completeness, showing that the action space can organize these modules into control flow graphs with sufficient expressive power (sequential, branching, looping, parallel), thereby constructing executable workflow programs. We divide the proof into three parts: first defining the cognitive primitive set $\mathcal{C}$, then proving that $\mathcal{O}$ covers $\mathcal{C}$, and finally proving that $\mathcal{A}_{\text{type}}$ can construct any structured workflow graph composed of these modules.

(i) Cognitive primitives and functional coverage. We first define a set of cognitive primitives consistent with goal-directed problem solving. Let the cognitive primitive set be

$$\mathcal{C} = \{\mathbf{P}, \mathbf{D}, \mathbf{E}, \mathbf{M}, \mathbf{R}, \mathbf{I}, \mathbf{O}\}, \tag{16}$$

where each element represents a fundamental cognitive process in problem solving. Specifically, $\mathbf{P}$ (Planning) forms goals, strategies, and resource budgets, determining what to do first, what to do later, and to what extent; $\mathbf{D}$ (Decomposition) breaks the overall task into operable subgoals with explicit dependency relationships; $\mathbf{E}$ (Execution) performs solving, reasoning, or externalized computation on a subgoal, including symbolic computation, code generation and execution, and other operations; $\mathbf{M}$ (Monitoring) verifies, reviews, unit tests, and checks constraints on intermediate products, determining whether to continue, backtrack, or redo; $\mathbf{R}$ (Revision) performs repairs based on monitoring results, including rewriting, supplementing, replacing, and adjusting prompts or parameters; $\mathbf{I}$ (Integration) fuses, votes, disambiguates, and ensures consistency of results from multiple branches or subproblems; and $\mathbf{O}$ (Output) outputs the final result in the format required by the task, including answer extraction and structured presentation.

We now construct a surjective mapping $\psi : \mathcal{O} \to \mathcal{C}$ to show that for each cognitive primitive $\mathbf{c} \in \mathcal{C}$, there exists at least one operator $o \in \mathcal{O}$ such that $\psi(o) = \mathbf{c}$. Consider the following correspondence:

For the planning primitive $\mathbf{P}$, the operator `Plan` directly implements high-level plan generation, including strategy formulation, step sequencing, budget allocation, and stopping condition specification. Therefore, we have

$$\psi(\texttt{Plan}) = \mathbf{P}. \tag{17}$$

For the decomposition primitive $\mathbf{D}$, the operator `Decompose` expresses task decomposition in the form of subtask lists or subproblem graphs, explicitly representing the dependency structure among subtasks. Therefore, we have

$$\psi(\texttt{Decompose}) = \mathbf{D}. \tag{18}$$

For the execution primitive $\mathbf{E}$, execution includes not only natural language reasoning but also externalized solving. The operator `Custom` covers general reasoning and retrieval-based processing, capable of hosting various tools and prompt templates according to implementation; the operator

`Programmer` covers executable code generation and execution, typically handling symbolic computation, scripted reasoning, and data processing; the operator `AnswerGenerate` covers generating final natural language answers from obtained key evidence or intermediate conclusions, which can be viewed as decoding or expression. Therefore, we have

$$\psi(\texttt{Custom}) = \psi(\texttt{Programmer}) = \psi(\texttt{AnswerGenerate}) = \mathbf{E}. \tag{19}$$

For the monitoring primitive $\mathbf{M}$, the operators `Verify` and `Review` perform consistency checking and critical evaluation on intermediate results; the operator `Test` performs executable testing on code solutions. These are all typical error monitoring and quality assessment processes. Therefore, we have

$$\psi(\texttt{Verify}) = \psi(\texttt{Review}) = \psi(\texttt{Test}) = \mathbf{M}. \tag{20}$$

For the revision primitive $\mathbf{R}$, the operator `Revise` explicitly implements error correction and rewriting in a feedback-based repair manner. Therefore, we have

$$\psi(\texttt{Revise}) = \mathbf{R}. \tag{21}$$

For the integration primitive $\mathbf{I}$, when there are multiple branches, multiple candidates, or multiple subproblems, fusion is needed. The operator `Aggregate` handles aggregation and consistency ensuring; the operator `ScEnsemble` handles selection and ensemble of diverse candidates. Therefore, we have

$$\psi(\texttt{Aggregate}) = \psi(\texttt{ScEnsemble}) = \mathbf{I}. \tag{22}$$

For the output primitive $\mathbf{O}$, the operator `Format` extracts, structures, and presents results in the target format. Therefore, we have

$$\psi(\texttt{Format}) = \mathbf{O}. \tag{23}$$

For each $\mathbf{c} \in \mathcal{C}$, the above construction provides at least one $o \in \mathcal{O}$ such that $\psi(o) = \mathbf{c}$. Hence ψ is surjective onto $\mathcal{C}$, establishing that $\mathcal{O}$ achieves functional coverage of $\mathcal{C}$.

(ii) Structural completeness of the action space. Let $\mathfrak{G}(\mathcal{O})$ denote the set of all finite workflow graphs with nodes labeled by operators from $\mathcal{O}$ and composed using structured control constructs (sequential, conditional, loop, parallel). We show that for any target graph $\mathcal{G}^\star \in \mathfrak{G}(\mathcal{O})$, there exists a finite-length action sequence $\{a_t\}_{t=1}^{T}$ with action types belonging to $\mathcal{A}_{\text{type}}$, such that starting from the empty graph $\mathcal{G}_0$, the iterative updates yield $\mathcal{G}_T = \mathcal{G}^\star$. We provide a constructive proof, which is equivalent to showing that these editing actions can assemble any target graph.

Step 1: Node construction and attribute assignment (basic editing closure). For any target graph $\mathcal{G}^\star$ with node set $V^\star$, we traverse $v \in V^\star$ in any order. For each node, we use the action `add` to create a new node in the current canvas and specify its operator type $\text{op}(v) \in \mathcal{O}$. We then use the action `set_prompt` to set prompts and constraints for that node. If necessary, we use the action `modify` to write in parameters. If there are redundant or erroneous nodes, we use the action `delete` to remove them. Since `add`, `delete`, `modify`, and `set_prompt` allow arbitrary finite discrete modifications to the node set and attributes, we can construct within finite steps a node set and attribute annotation isomorphic to $\mathcal{G}^\star$.

Step 2: Control structure construction (structured control closure). Control structures in the target graph can be divided into three categories: conditional branching, looping, and parallel execution. If $\mathcal{G}^\star$ contains conditional structures (if/else or selective execution), we use the action `conditional` to introduce conditional gate nodes or conditional edges in the graph, and write in condition predicates via `set_prompt` or `modify`. The condition can be a Boolean signal output by some checking or verification node. If $\mathcal{G}^\star$ contains loop structures (repeated execution until some criterion is met), we use the action `loop` to introduce iteration blocks. The stopping condition can similarly be produced by verification or test nodes, with the loop termination rule written in via `set_prompt` or `modify`. If $\mathcal{G}^\star$ contains parallel structures (multiple subproblems or candidates expanded simultaneously), we use the action `parallel` to organize several branches into concurrent subgraphs, with subsequent integration via operators such as `Aggregate` or `ScEnsemble` to merge branch results.

Step 3: Termination (completing construction). When the current canvas graph aligns with $\mathcal{G}^\star$ in nodes, attributes, and control structures, we use the action `finish` to terminate editing. Since $\mathcal{G}^\star$ is a finite graph, the above process is finite, hence there exists finite T such that $\mathcal{G}_T = \mathcal{G}^\star$.

(iii) Coverage of task types. We now show that the operator library and action space can cover the task types considered in this work by demonstrating the existence of constructible cognitive program templates for each task type.

For mathematical reasoning tasks, a typical cognitive program follows the pattern: `Plan` $\rightarrow$ `Decompose` $\rightarrow$ (`Custom` or `Programmer`) $\rightarrow$ `Verify` $\rightarrow$ (`Revise` + `loop`) $\rightarrow$ `Format`. This template first formulates a solution strategy, decomposes the problem into solvable steps, executes reasoning or symbolic computation, verifies intermediate results, revises if errors are detected (potentially looping), and finally formats the answer.

For question answering tasks including multi-hop QA, a typical cognitive program follows the pattern: `Plan` $\rightarrow$ `Decompose` $\rightarrow$ `parallel`(`Custom`/`AnswerGenerate`) $\rightarrow$ `Aggregate` $\rightarrow$ `Review`/`Verify` $\rightarrow$ `Format`. This template formulates a retrieval and reasoning strategy, decomposes into sub-questions that can be processed in parallel, aggregates evidence from multiple sources, reviews for consistency, and formats the final answer.

For code generation tasks, a typical cognitive program follows the pattern: `Plan` $\rightarrow$ `Decompose` $\rightarrow$ `Programmer` $\rightarrow$ `Test` $\rightarrow$ (`Revise` + `loop`) $\rightarrow$ `Format`. This template plans the code structure, decomposes into implementable modules, generates code, tests for correctness, revises based on test feedback (looping until tests pass), and formats the output.

These templates use only operators from $\mathcal{O}$ and control structures from $\mathcal{A}_{\text{type}}$. Therefore, under Assumption 1 (tasks can be decomposed into combinations of cognitive primitives), for any task q, there exists some $\mathcal{G}^\star$ implementing the corresponding cognitive program. By structural completeness established above, there exists an action sequence constructing $\mathcal{G}^\star$.

In summary, functional coverage ensures that the operator library implements all cognitive primitives required for goal-directed problem solving, while structural completeness ensures that the action space can construct any workflow graph composed of these primitives with arbitrary control structures. Together, they establish that for any task decomposable into cognitive primitive combinations, there exists an action sequence constructing a workflow that implements the complete cognitive control loop covering planning, decomposition, execution, monitoring, revision, integration, and output. □

B.2 Proof of Proposition 2

Proposition 2 (Decoupled Transfer with Monotonic Improvement)**.** *Let π_θ be trained on a canvas $\mathcal{C} = (\mathcal{G}_t, \mathcal{O}, \mathcal{M}_{\text{exec}}, d^{\text{lib}})$ and deployed without parameter updates on a target canvas $\mathcal{C}' = (\mathcal{G}_t, \mathcal{O}', \mathcal{M}'_{\text{exec}}, d'^{\text{lib}})$ that shares the action interface $\mathcal{A}_{\text{type}}$ and whose feedback satisfies Assumptions 2–3. Then π_θ is a valid editing policy on $\mathcal{C}'$, and its multi-turn orchestration possesses the monotonic improvement property in expectation: as the number of turns t increases, (1) the agent's uncertainty about the correct output or feasible workflow does not increase; (2) consequently, the success probability of generating an executable and correct workflow (and final answer) does not decrease. Furthermore, multi-turn interaction under the same budget is at least as good as single-turn open-loop generation.*

Proof. We first show that the decoupling makes the trained policy well defined on the target canvas, and then characterize the value of multi-turn interaction on that canvas using a Bayesian risk potential function: each turn obtains canvas feedback o_t^{exec} that provides information about the hidden correct solution, thereby concentrating the posterior distribution. The monotonic decrease of the potential function in expectation corresponds to the monotonic increase of reliability and success rate.

(0) Validity of the frozen policy on the target canvas. By Eq. 6, the executor enters the interaction only through the feedback o_t^{exec} appended to H_t, and the operator library enters only through the description d^{lib} in H_0; by Eq. 8, both enter training only through the scalar reward $R(\tau)$. The policy therefore maps any history H'_t produced on $\mathcal{C}'$ to a distribution over the same action space $\mathcal{A}_{\text{type}}$, and each sampled edit is applied by the same graph-update operator. Swapping $\mathcal{M}_{\text{exec}}$ for $\mathcal{M}'_{\text{exec}}$ or $\mathcal{O}$ for $\mathcal{O}'$ thus changes the feedback the policy conditions on, while the policy itself, its editing interface, and the form of its gradient stay unchanged. Steps (i)–(v) below use only Bayes' rule and Assumptions 2–3 on the feedback, so they apply verbatim to the histories generated on $\mathcal{C}'$.

(i) Posterior distribution and Bayesian risk potential. Let Z denote the latent variable related to the task solution. According to the problem setting, Z can be taken as the correct answer $y_q^\star$, or as a representative element from the set of feasible workflows that lead to the correct answer. At

interaction turn t, the history is defined as

$$H_t = \{(a_1^{\text{think}}, a_1, o_1^{\text{exec}}), \ldots, (a_t^{\text{think}}, a_t, o_t^{\text{exec}})\}, \tag{24}$$

where H_t encodes all reflection texts, edit actions, and canvas feedback up to turn t. The initial history $H_0 = [q \oplus d^{\text{lib}} \oplus a^{\text{tmpl}}]$ holds the task input and contains no interaction step. Given history H_t, we define the posterior distribution over Z as

$$\pi_t(z) \triangleq \mathbb{P}(Z = z \mid H_t), \tag{25}$$

where $\pi_t(z)$ represents the probability that the true solution is z given the interaction history up to turn t. We define the Bayes accuracy function as the maximum posterior probability:

$$A(H_t) \triangleq \max_z \pi_t(z), \tag{26}$$

where $A(H_t)$ represents how concentrated the posterior is on the most likely solution. We then define the Bayes risk potential function as

$$V(H_t) \triangleq 1 - A(H_t) = 1 - \max_z \pi_t(z), \tag{27}$$

where $V(H_t)$ measures the remaining uncertainty. Smaller $V(H_t)$ indicates more concentrated posterior and higher probability of the most likely correct solution.

(ii) Supermartingale property of the risk potential. We now show that the expected risk potential does not increase across turns, i.e., for any $t \geq 1$,

$$\mathbb{E}[V(H_t) \mid H_{t-1}] \leq V(H_{t-1}), \tag{28}$$

with strict inequality when the feedback at turn t provides non-zero information gain about Z.

The key observations are twofold. First, the posterior vector satisfies the martingale property. Since $H_t = H_{t-1} \oplus (a_t^{\text{think}}, a_t, o_t^{\text{exec}})$ is obtained by appending new observations to H_{t-1}, by Bayes' rule, for any z we have

$$\mathbb{E}[\pi_t(z) \mid \mathcal{F}_{t-1}] = \pi_{t-1}(z), \tag{29}$$

where $\mathcal{F}_{t-1} = \sigma(H_{t-1})$ is the natural filtration generated by the interaction history up to turn $t-1$. This martingale property states that the expected posterior at turn t, conditioned on information at turn $t-1$, equals the posterior at turn $t-1$.

Second, $V(\cdot)$ is a concave function over distributions. To see this, consider the potential function over the probability simplex:

$$\phi(p) \triangleq 1 - \max_z p_z, \quad p \in \Delta_{|Z|-1}, \tag{30}$$

where p is a probability vector over the solution space, and $\Delta_{|Z|-1}$ is the probability simplex of such vectors. Since $\max_z p_z$ is a convex function of p (as the pointwise maximum of linear functions), its negation $-\max_z p_z$ is concave, and hence $\phi(p) = 1 - \max_z p_z$ is also concave.

Applying Jensen's inequality for concave functions, we obtain the contraction of expected risk:

$$\mathbb{E}[V(\pi_t) \mid \mathcal{F}_{t-1}] \leq V(\mathbb{E}[\pi_t \mid \mathcal{F}_{t-1}]) = V(\pi_{t-1}), \tag{31}$$

where the equality uses the martingale property established above. This yields

$$\mathbb{E}[V(H_t) \mid H_{t-1}] \leq V(H_{t-1}). \tag{32}$$

When the feedback provides non-zero information gain about Z (i.e., under Assumption 2), the posterior π_t undergoes strict contraction with non-zero probability, meaning it does not merely preserve its mean but actually becomes more concentrated. In this case, Jensen's inequality becomes strict in expectation, yielding

$$\mathbb{E}[V(H_t) \mid H_{t-1}] < V(H_{t-1}) \tag{33}$$

with positive probability, leading to strict improvement.

(iii) Monotonic improvement over multiple turns. Taking unconditional expectation and iterating the supermartingale relation yields:

$$\mathbb{E}[V(H_t)] \leq \mathbb{E}[V(H_{t-1})] \leq \cdots \leq \mathbb{E}[V(H_0)]. \tag{34}$$

This establishes that the expected Bayes risk monotonically decreases (or stays constant) as the number of interaction turns increases.

To quantify the improvement, we define the accuracy gain at turn t as

$$\Delta_t \triangleq \mathbb{E}[V(H_{t-1})] - \mathbb{E}[V(H_t)] \geq 0, \tag{35}$$

where Δ_t represents the expected one-step reduction of the Bayes risk potential at turn t. Summing over all turns, the expected Bayes risk after t turns satisfies:

$$\mathbb{E}[V(H_t)] = \mathbb{E}[V(H_0)] - \sum_{s=1}^{t} \Delta_s, \tag{36}$$

where each Δ_s accumulates the expected risk decrease at turn s. Substituting into the definition of accuracy, we obtain:

$$\mathbb{E}[A(H_t)] = 1 - \mathbb{E}[V(H_0)] + \sum_{s=1}^{t} \Delta_s, \tag{37}$$

where $\mathbb{E}[A(H_t)]$ represents the expected Bayes accuracy after t turns. This shows that expected accuracy monotonically increases with the number of turns.

(iv) From uncertainty reduction to error probability reduction. Let $\hat{z}_t = \arg\max_z \pi_t(z)$ be the Bayes optimal estimate at turn t, i.e., the solution with maximum posterior probability. Under 0-1 loss (where we incur loss 1 if wrong and 0 if correct), the Bayes optimal decision rule minimizes expected loss by choosing $\hat{z}_t$. The error probability of this estimator satisfies

$$\mathbb{P}(\hat{z}_t \neq Z \mid H_t) = 1 - \max_z \pi_t(z) = V(H_t). \tag{38}$$

This equality follows because the probability of error under the Bayes optimal rule equals one minus the probability of the chosen class, which is precisely $1 - \max_z \pi_t(z)$.

Therefore, monotonically non-increasing $\mathbb{E}[V(H_t)]$ is equivalent to monotonically non-increasing expected error rate, or equivalently, monotonically non-decreasing expected correctness rate. This proves conclusions (1) and (2) of the proposition.

(v) Comparison with single-turn generation. A single-turn strategy makes only one decision and directly outputs the final workflow or answer, which is equivalent to setting $T = 1$. A multi-turn strategy with $T > 1$ can still choose to execute the `finish` action at the first turn, thereby terminating immediately. Therefore, the strategy space of multi-turn interaction contains the single-turn strategy as a special case.

By this inclusion relationship of strategy spaces, the optimal value achievable by multi-turn interaction is no worse than that of single-turn interaction:

$$\max_{\pi \in \Pi_{\text{multi-turn}}} \mathbb{E}[A(H_T)] \geq \max_{\pi \in \Pi_{\text{single-turn}}} \mathbb{E}[A(H_1)], \tag{39}$$

where $\Pi_{\text{multi-turn}} \supseteq \Pi_{\text{single-turn}}$ denotes the respective policy spaces. This establishes that multi-turn is at least as good as single-turn under the same budget.

Furthermore, under Assumptions 2 (informative feedback) and 3 (repairability), multi-turn interaction can achieve strict improvement: in some turns, the posterior strictly contracts, leading to $\mathbb{E}[V(H_t)] < \mathbb{E}[V(H_{t-1})]$ on the training and inference distribution. This manifests as higher execution success rate, lower structural error rate, and higher final correctness rate.

In conclusion, multi-turn canvas-based interaction monotonically decreases the Bayes risk potential whenever feedback is informative, consequently increasing expected accuracy. The agent can progressively refine the workflow based on accumulated observations, each turn potentially reducing uncertainty about the correct solution. This iterative refinement process achieves higher reliability and success probability than single-turn open-loop generation, where errors cannot be detected or corrected. Combined with step (0), the property holds for the frozen policy on any executor and operator library whose canvas feedback is informative and repairable. Because every edit conditions on the target feedback, the policy rebuilds the workflow for $\mathcal{M}'_{\text{exec}}$ instead of replaying the graph built under $\mathcal{M}_{\text{exec}}$, consistent with the cross-executor gains in Section 5.4. □

B.3 Proof of Proposition 3

Proposition 3 (Structural Constraints, Conditional Release, and Mask Effectiveness)**.** *Let the trajectory-level reward be defined as*

$$R(\tau) = -1 + R_{\text{diversity}}(\tau) + \mathbb{I}\{R_{\text{diversity}}(\tau) = 1\} \cdot R_{\text{answer}}(\tau), \tag{40}$$

where $R_{\text{diversity}}(\tau) \in [0, 1]$ is composed of structural check items and capped at 1, and $R_{\text{answer}}(\tau) \geq 0$ measures the match between the final execution output and the ground truth. In advantage-based policy optimization such as the canvas-masked GRPO objective in Section 4.3, this design possesses three properties: (a) separable feasibility learning, where trajectories not satisfying structural diversity constraints necessarily receive non-positive returns and are systematically suppressed by gradient updates; (b) shortcut and collapse suppression, where answer rewards are unlocked only after learning to construct qualified skeletons, thereby avoiding shortcuts like skipping workflows to answer directly; (c) gradient correctness of mask, where token-level mask backpropagates gradients only for policy-generated tokens, maintaining unbiased policy gradient estimates and significantly reducing variance and noise from environment feedback tokens, thereby stabilizing training.

Proof. The essence of this proposition is that the reward design achieves numerical separation and gating between structural feasibility and answer correctness, making the optimization process naturally staged. Meanwhile, the mask ensures that gradients act only on policy-controllable parts, and the clipping mechanism with KL regularization bounds the policy update magnitude, keeping gradient signals clean and stable.

(i) Feasible skeleton set and sign separation of rewards. We first define the feasible skeleton trajectory set as

$$\mathcal{F} = \{\tau : R_{\text{diversity}}(\tau) = 1\}, \tag{41}$$

where $\mathcal{F}$ represents the set of trajectories with necessary structural skeleton, including verification steps, formatting operators, sufficient operator count, and appropriate control structures. The complement set is defined as

$$\mathcal{F}^c = \{\tau : R_{\text{diversity}}(\tau) < 1\}, \tag{42}$$

where $\mathcal{F}^c$ represents trajectories that fail to meet one or more structural requirements.

We now show that the reward is strictly separable on $\mathcal{F}$ and $\mathcal{F}^c$ with a sign gap. For any trajectory $\tau \in \mathcal{F}^c$, we have $R_{\text{diversity}}(\tau) < 1$, so the indicator function $\mathbb{I}\{R_{\text{diversity}}(\tau) = 1\}$ equals 0. Substituting into the reward formula:

$$R(\tau) = -1 + R_{\text{diversity}}(\tau) + 0 \cdot R_{\text{answer}}(\tau) = -1 + R_{\text{diversity}}(\tau). \tag{43}$$

Since $R_{\text{diversity}}(\tau) \in [0, 1)$ for $\tau \in \mathcal{F}^c$, we have

$$R(\tau) \in [-1, 0) \quad \text{for all } \tau \in \mathcal{F}^c. \tag{44}$$

This means all structurally non-compliant trajectories receive strictly negative rewards.

For any trajectory $\tau \in \mathcal{F}$, we have $R_{\text{diversity}}(\tau) = 1$, so the indicator function equals 1. Substituting into the reward formula:

$$R(\tau) = -1 + 1 + 1 \cdot R_{\text{answer}}(\tau) = R_{\text{answer}}(\tau). \tag{45}$$

Since $R_{\text{answer}}(\tau) \geq 0$ by definition, we have

$$R(\tau) \geq 0 \quad \text{for all } \tau \in \mathcal{F}. \tag{46}$$

This means all structurally compliant trajectories receive non-negative rewards. The sign separation is therefore complete: $\mathcal{F}^c$ trajectories are strictly negative while $\mathcal{F}$ trajectories are non-negative, achieving strong constraint separation at the numerical level.

(ii) Two-stage optimization via conditional release and policy gradient. We now analyze how the conditional release mechanism creates a natural two-stage optimization process through the lens of policy gradient theory. Let π_θ denote the policy parameterized by θ, and let π_{ref} denote the reference policy (typically the initial supervised fine-tuned model). Our RL objective can be written as

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right] - \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}), \tag{47}$$

where $\beta > 0$ is the KL penalty coefficient that prevents the policy from deviating too far from the reference distribution.

By the policy gradient theorem, the gradient of the expected reward with respect to θ is given by

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(\tau)\Big], \tag{48}$$

where $A(\tau)$ is the advantage function and the sum is over all time steps in the trajectory. In practice, the advantage is computed using group normalization within each sampled batch:

$$\hat{A}(\tau) = \frac{R(\tau) - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \epsilon}, \tag{49}$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are the mean and standard deviation of rewards in batch $\mathcal{B}$, and ϵ is a small constant for numerical stability.

From the sign separation established in (i), when both $\mathcal{F}$ and $\mathcal{F}^c$ samples exist in a batch, the normalized advantages satisfy

$$\mathbb{E}[\hat{A}(\tau) \mid \tau \in \mathcal{F}] > 0 > \mathbb{E}[\hat{A}(\tau) \mid \tau \in \mathcal{F}^c]. \tag{50}$$

This inequality holds because feasible trajectories have non-negative rewards while non-feasible ones have strictly negative rewards, so after mean-centering, feasible trajectories lie above the mean and non-feasible ones lie below.

The policy gradient update therefore increases the log-probability of actions in feasible trajectories and decreases the log-probability of actions in non-feasible trajectories:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta), \tag{51}$$

where α is the learning rate. This systematically shifts probability mass from $\mathcal{F}^c$ to $\mathcal{F}$.

To quantify this effect, define the feasibility probability $p_\theta = \mathbb{P}_{\tau\sim\pi_\theta}(\tau \in \mathcal{F})$. Under mild regularity conditions, the gradient update increases p_θ whenever $p_\theta < 1$:

$$\frac{dp_\theta}{d\theta} \cdot \nabla_\theta J(\theta) > 0 \quad \text{when } 0 < p_\theta < 1. \tag{52}$$

This follows because the expected advantage is positive for $\mathcal{F}$ and negative for $\mathcal{F}^c$, so the gradient points in the direction of increasing p_θ.

During early training when p_θ is small, most trajectories fall into $\mathcal{F}^c$, and the primary learning signal comes from avoiding structural violations. As p_θ increases through training, an increasing proportion of sampled trajectories fall into $\mathcal{F}$. For these trajectories, $R(\tau) = R_{\text{answer}}(\tau)$, and the training signal naturally shifts to optimizing answer correctness. This creates the two-stage behavior: first learn to satisfy structural constraints, then learn to maximize answer quality.

The conditional release mechanism ensures that shortcuts are suppressed: any trajectory that directly outputs an answer without constructing a proper workflow will have $R_{\text{diversity}}(\tau) < 1$ and thus $R(\tau) < 0$, regardless of answer correctness. This negative reward signal systematically discourages such shortcuts.

(iii) Bounded policy updates via clipping and KL regularization. To ensure training stability, our RL objective incorporates two mechanisms that bound the magnitude of policy updates: importance ratio clipping and KL divergence regularization.

For importance sampling, define the probability ratio between the current policy and the behavior policy (used for sampling) as

$$\rho_\theta(\tau) = \frac{\pi_\theta(\tau)}{\pi_{\text{old}}(\tau)} = \prod_{t=1}^{T} \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{old}}(a_t \mid s_t)}, \tag{53}$$

where π_{old} is the policy at the beginning of the current optimization epoch. The clipped objective restricts the effective ratio to the interval $[1 - \epsilon_{\text{clip}}, 1 + \epsilon_{\text{clip}}]$:

$$L^{\text{clip}}(\theta) = \mathbb{E}_{\tau\sim\pi_{\text{old}}}\left[\min\left(\rho_\theta(\tau)\hat{A}(\tau),\ \text{clip}(\rho_\theta, 1-\epsilon, 1+\epsilon)\hat{A}(\tau)\right)\right]. \tag{54}$$

This clipping prevents excessively large policy updates when the importance ratio deviates significantly from 1, which would otherwise destabilize training.

The KL regularization term provides a soft constraint on the policy deviation:

$$D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}}) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=1}^{T} \log \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\mathrm{ref}}(a_t \mid s_t)}\right]. \tag{55}$$

By penalizing this divergence, the optimization ensures that the learned policy remains close to the reference policy, preventing catastrophic forgetting of useful behaviors learned during supervised fine-tuning.

Together, clipping and KL regularization keep each policy update within a trust region around the behavior policy π_{old} and the reference policy π_{ref}, which prevents training oscillations and supports smooth convergence.

(iv) Unbiasedness and variance reduction of token-level mask. An interaction trajectory at the token level consists of two interleaved types of segments: policy-generated tokens comprising reflection and action text, and environment feedback tokens comprising canvas returns. Crucially, the generation distribution of environment feedback does not contain the learnable parameters θ; it is determined by the canvas environment given the action.

Let $w_{1:|\tau|}$ denote the entire trajectory token sequence. We partition it into two disjoint sets: $\mathcal{T}_\pi$ for policy-generated tokens and $\mathcal{T}_{\mathrm{env}}$ for environment feedback tokens. The joint log-likelihood of the trajectory can be decomposed as

$$\log p_\theta(w_{1:|\tau|}) = \sum_{t \in \mathcal{T}_\pi} \log \pi_\theta(w_t \mid w_{<t}) + \sum_{t \in \mathcal{T}_{\mathrm{env}}} \log p_{\mathrm{env}}(w_t \mid w_{<t}), \tag{56}$$

where the first term sums over policy tokens and depends on θ, while the second term sums over environment tokens and does not depend on θ.

Taking the gradient with respect to θ, the second term vanishes since $\nabla_\theta \log p_{\mathrm{env}}(w_t \mid w_{<t}) = 0$:

$$\nabla_\theta \log p_\theta(w_{1:|\tau|}) = \sum_{t \in \mathcal{T}_\pi} \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}). \tag{57}$$

Let $\mathrm{mask}_t \in \{0, 1\}$ be the token-level mask that takes value 1 for policy tokens ($t \in \mathcal{T}_\pi$) and 0 for environment tokens ($t \in \mathcal{T}_{\mathrm{env}}$). Then the gradient can be written equivalently as

$$\nabla_\theta \log p_\theta(w_{1:|\tau|}) = \sum_{t=1}^{|\tau|} \mathrm{mask}_t \cdot \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}). \tag{58}$$

Using the mask to select only policy tokens $\mathcal{T}_\pi$ implements this equality exactly. Therefore, the masked gradient estimator is unbiased:

$$\mathbb{E}\left[\sum_{t=1}^{|\tau|} \mathrm{mask}_t \cdot \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}) \cdot R(\tau)\right] = \nabla_\theta J(\theta). \tag{59}$$

Furthermore, the mask reduces gradient variance. Without the mask, if all tokens were naively included, the gradient estimator would have variance

$$\mathrm{Var}_{\mathrm{no-mask}} = \mathrm{Var}\left[\sum_{t=1}^{|\tau|} \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}) \cdot R(\tau)\right]. \tag{60}$$

With the mask, the variance is

$$\mathrm{Var}_{\mathrm{mask}} = \mathrm{Var}\left[\sum_{t \in \mathcal{T}_\pi} \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}) \cdot R(\tau)\right]. \tag{61}$$

Since $|\mathcal{T}_\pi| < |\tau|$ and the environment tokens contribute noise unrelated to θ, we have $\mathrm{Var}_{\mathrm{mask}} < \mathrm{Var}_{\mathrm{no-mask}}$. This variance reduction leads to more stable gradient updates and faster convergence.

In conclusion, the reward design and optimization framework achieve four complementary goals for stable and effective learning. First, sign separation numerically enforces strong constraint separation between feasible and non-feasible trajectories, ensuring that structural violations are systematically penalized with negative rewards. Second, conditional release creates a natural two-stage optimization where the policy first learns to satisfy structural constraints (increasing $p_\theta = \mathbb{P}(\tau \in \mathcal{F})$) before optimizing answer correctness, thereby suppressing shortcuts and preventing structural collapse. Third, clipping and KL regularization bound the magnitude of policy updates, preventing training oscillations and ensuring smooth convergence while maintaining proximity to the reference policy. Fourth, the token-level mask ensures that gradients act only on policy-controllable tokens, maintaining unbiased gradient estimates while reducing variance from environment feedback, thereby stabilizing the training process. Together, these mechanisms provide stable learning signals, bounded updates, and effective prevention of shortcut behaviors, enabling reliable end-to-end reinforcement learning for workflow orchestration. □

C FlowSteer Algorithm Details

FlowSteer is a multi-turn workflow orchestration framework built on the collaboration between a Flow-Director (small-scale LLM) and a Workflow Canvas (execution environment): the Flow-Director handles planning and action generation, while the Workflow Canvas provides structural feedback and its executor runs the constructed workflow to produce the answer. The process is divided into three connected stages: first, given a task description, the Flow-Director constructs workflow nodes with structural guidance using the operator library, and the Canvas returns execution feedback to form the initial interaction history; then, during multi-turn interaction, the Flow-Director continuously generates reasoning and the next action from the accumulated history, the Canvas validates and executes each action to extend the history until the termination condition is reached, and the accumulated trajectory is utilized to execute the final workflow and generate the answer; finally, end-to-end reinforcement learning is applied to update the policy of the Flow-Director. The reward function jointly evaluates structural diversity compliance and answer correctness, enabling the Flow-Director to gradually learn how to effectively construct complex workflows within a limited budget. This coordination reduces invalid actions and stabilizes interaction dynamics. This design enables progressive and adaptive workflow construction, resulting in improved accuracy and stability on complex reasoning tasks.

Training and Inference Flow. During training, the algorithm proceeds in three stages (A $\rightarrow$ B $\rightarrow$ C): the Flow-Director initializes the system prompt and triggers the Canvas for an initial feedback, then enters multi-turn interaction to generate actions, receive feedback, and terminate with `finish` in a sequential manner, after which the executor runs the final workflow to produce the answer, and finally updates the policy in Stage C using rewards and advantages. During testing, it runs only two stages (A $\rightarrow$ B) in a simplified form: initialization and multi-turn interaction, after which the final workflow is executed and the answer is produced directly without parameter updates.

Complexity Analysis. The computational complexity of FlowSteer mainly comes from initialization, multi-turn interaction, and reinforcement learning optimization. The initialization stage involves one call to the Flow-Director and a call to the Canvas, which is a constant overhead. The multi-turn interaction stage requires up to T rounds in the worst case, where each round includes one planning step by the Flow-Director and one call to the Canvas, yielding time complexity $O(T)$. The memory consumption grows linearly with the history length, which can be controlled through windowing or summarization. The reinforcement learning stage requires sampling N trajectories per update, each trajectory containing up to T action-feedback steps, leading to complexity $O(NT)$. It also requires storing trajectory information for reward and advantage computation. In total, the complexity of FlowSteer is $O(NT + T)$, scaling linearly with the number of rounds and sampled trajectories during training, while inference requires $O(T)$. Since the Flow-Director is responsible for planning and constructing workflows, the Canvas execution is more focused, ensuring stability on complex reasoning tasks.

C.1 Reward Components

The diversity-constrained reward $R_{\text{diversity}}(\tau)$ aggregates four structural checks over the produced workflow, with the total score capped at 1.0:

Algorithm 1 FlowSteer: End-to-End Workflow Orchestration via Multi-Turn Reinforcement Learning

Require: Task q, Flow-Director π_θ, Workflow Canvas $\mathcal{C}$, operator library $\mathcal{O}$, reward function $R(\cdot)$, maximum interaction turns T
Ensure: Final answer y

// Stage A: Workflow Initialization
Initialize: workflow graph $\mathcal{G}_0 = \emptyset$, interaction history $H_0 = []$
Construct system prompt: $p_{\text{sys}} \leftarrow \textsc{BuildPrompt}(\mathcal{O}, q)$
First think and action: $(a_1^{\text{think}}, a_1) \leftarrow \pi_\theta(p_{\text{sys}}, q)$
First canvas feedback: $o_1 \leftarrow \mathcal{C}.\textsc{Step}(a_1)$
First update history: $H_1 \leftarrow \{(a_1^{\text{think}}, a_1, o_1)\}$
// Stage B: Multi-turn Collaborative Workflow Building
for $t = 2$ **to** T **do**
  Plan think: $a_t^{\text{think}} \leftarrow \pi_\theta(p_{\text{sys}}, q, H_{t-1})$
  Generate action: $a_t \leftarrow \pi_\theta(p_{\text{sys}}, q, H_{t-1})$
  Canvas feedback: $o_t \leftarrow \mathcal{C}.\textsc{Step}(a_t)$
  Update history: $H_t \leftarrow H_{t-1} \cup \{(a_t^{\text{think}}, a_t, o_t)\}$
  if $a_t =$ `finish` **and** $\mathcal{C}.\textsc{CheckConstraints}()$ **then**
    break
  end if
end for
Execute workflow: $y \leftarrow \mathcal{C}.\textsc{Execute}(\mathcal{G}_T, q)$
// Stage C: End-to-End Reinforcement Learning Optimization
Sample trajectories: $\{\tau_i\}_{i=1}^N \sim \pi_\theta$
for each τ_i **do**
  Compute reward: $R(\tau_i) = -1 + R_{\text{diversity}}(\tau_i) + \mathbb{I}\{R_{\text{diversity}}(\tau_i) = 1\} \cdot R_{\text{answer}}(\tau_i)$
  Compute advantage: $\hat{A}(\tau_i) = \frac{R(\tau_i) - \bar{R}}{\sqrt{\frac{1}{M}\sum_{j=1}^{M}(R(\tau_j) - \bar{R})^2 + \epsilon}}$
end for
Update policy with the canvas-masked GRPO objective:
$\mathcal{J} \propto \sum_{i=1}^{N} \sum_{t=1}^{|\tau_i|} \textsc{Mask}_t^{(i)} \cdot \min\left(\rho_\theta(w_t^{(i)}), \textsc{clip}(\rho_\theta(w_t^{(i)}), 1 \pm \epsilon)\right) \hat{A}(\tau_i)$
where $\rho_\theta(w_t^{(i)}) = \frac{\pi_\theta(w_t^{(i)} | \tau_{<t}^{(i)})}{\pi_{\theta_{\text{old}}}(w_t^{(i)} | \tau_{<t}^{(i)})}$
Parameter update: $\theta \leftarrow \theta - \eta \nabla_\theta(-\mathcal{J})$
return y

- R_{checker}**:** Encourages the inclusion of verification operators (`Test`, `Review`, `Verify`); fires if at least one verification operator is present.
- R_{format}**:** Encourages proper answer formatting; fires if the `Format` operator is included as the final step before termination.
- R_{operator}**:** Requires a minimum operator count for structural diversity; fires if the workflow contains at least 3 distinct operators.
- R_{control}**:** Encourages control flow structures; fires if the workflow includes at least one control structure (`parallel`, `conditional`, or `loop`).

The four checks together form $R_{\text{diversity}}(\tau) \in [0, 1]$. This design ensures that workflows must exhibit structural diversity (achieving $R_{\text{diversity}} = 1.0$) before the answer reward R_{answer} is released, effectively preventing shortcut behaviors where the policy learns to generate overly simple or degenerate workflows.

D Dataset Details

We selected 12 public datasets (including mathematical reasoning, question answering, and code generation) for training and testing. Six of these datasets were used for training and testing. Six datasets were used for out-of-distribution testing to verify the generalization of the proposed FlowSteer. These datasets are as follows:

• **GSM8K** [Cobbe et al., 2021]: A collection of grade-school math word problems with concise statements, emphasizing step-by-step calculation and accurate numeric results. Problems involve basic operations with real-world contexts such as shopping, time calculations, and quantity comparisons.

• **MATH** [Hendrycks et al., 2021b]: A dataset of competition-level mathematics problems from AMC, AIME, and other mathematical olympiads. Problems span seven categories including algebra, geometry, number theory, combinatorics, probability, precalculus, and intermediate algebra, requiring sophisticated mathematical reasoning and symbolic manipulation.

• **HotPotQA** [Yang et al., 2018]: A large-scale multi-hop question answering dataset requiring reasoning across multiple Wikipedia paragraphs. Questions are designed to require finding and combining information from different sources, with supporting facts annotated for interpretability.

• **SQuAD v2** [Rajpurkar et al., 2018]: A Wikipedia-based QA dataset combining answerable and unanswerable questions, constructed to evaluate comprehension under mixed conditions. This tests both reading comprehension and the ability to recognize insufficient information.

• **MBPP** [Austin et al., 2021]: Mostly Basic Python Problems, a crowd-sourced collection of Python programming problems with natural language descriptions and test cases. Problems range from simple string manipulation to basic algorithms, designed to test fundamental programming skills.

• **HumanEval** [Chen et al., 2021]: A hand-written collection of Python programming problems with function signatures, docstrings, and unit tests. Problems are designed to test functional correctness through execution, covering tasks like string processing, mathematical operations, and data structure manipulation.

• **TriviaQA** [Joshi et al., 2017]: A knowledge-intensive dataset with questions from trivia websites and Wikipedia, containing a wide range of facts and lesser-known topics. The dataset covers diverse domains including history, science, geography, and entertainment.

• **NaturalQuestions** [Kwiatkowski et al., 2019]: A dataset of real anonymized queries from Google Search with answers from Wikipedia articles. Questions reflect genuine information needs of users, making them more diverse and challenging than synthetic questions.

• **MathQA** [Amini et al., 2019]: A math word problem dataset curated from multi-domain exam problems, covering arithmetic, algebra, geometry, probability, and other sub-disciplines. Each problem includes annotated rationales explaining the solution steps.

• **AIME 2025**: Problems from the 2025 American Invitational Mathematics Examination, representing challenging competition-level mathematics. AIME problems require creative problem-solving and deep mathematical insight, with answers being integers from 0 to 999.

• **APPS** [Hendrycks et al., 2021a]: Automated Programming Progress Standard, a collection of competitive programming problems from Codeforces, Kattis, and other platforms. Problems range from introductory to competition-level difficulty, requiring algorithmic thinking and efficient implementation.

• **DS-1000** [Lai et al., 2023]: A data science code generation benchmark covering NumPy, Pandas, TensorFlow, PyTorch, SciPy, Scikit-learn, and Matplotlib. Problems are derived from real StackOverflow questions, testing practical data science skills.

To ensure consistency and fairness for training and testing, we construct the training set by mixing samples from all IID datasets with the following sampling strategy: 2,560 samples from GSM8K, 2,560 from MATH, 2,560 from HotPotQA, 2,560 from SQuAD v2, 374 from MBPP, and 164 from HumanEval, resulting in a total of 10,778 training instances. For evaluation, all methods use the same fixed set of 1,438 test instances (768 IID and 670 OOD): we randomly sample 128 instances once from the test set of each of the other eleven datasets, use all 30 AIME 2025 problems, and keep this set fixed for every method.

E Baseline Details

To accurately evaluate the performance of FlowSteer, we conducted comparative experiments against multiple baselines. These baselines can be broadly divided into four categories: direct LLM inference,

supervised fine-tuning methods, search-based workflow methods, and agent with reinforcement learning methods.

E.1 Direct LLM Baselines

• **Qwen3-8B** [Yang et al., 2025]: An 8-billion parameter language model from Alibaba Cloud, serving as the backbone for our Flow-Director. As a baseline, it is tested with zero-shot chain-of-thought prompting, measuring the model's inherent reasoning capacity without workflow orchestration. The model provides strong efficiency while maintaining competitive performance on reasoning benchmarks, making it an ideal foundation for lightweight policy learning.

• **GPT-4o-mini** [Hurst et al., 2024]: A lightweight variant of GPT-4o optimized for cost and latency, while retaining strong language and reasoning abilities. As a baseline, it is tested with standard instruction prompting without workflow orchestration, measuring the model's inherent generation capacity under constrained resources. It serves as the default backend for our Workflow Canvas, executing the actual reasoning operations specified by the workflow.

E.2 Fine-Tuning Baselines

• **Supervised Fine-Tuning** [Ouyang et al., 2022]: An SFT baseline built on Qwen3-8B, trained with workflow annotation data to improve instruction following and DSL generation accuracy. Unlike FlowSteer's multi-turn interaction paradigm, SFT generates the complete workflow in a single turn without execution feedback. We use the same LoRA configuration as FlowSteer (rank 64, $\alpha = 64$, dropout 0.05) for fair comparison, evaluating how standard supervised adaptation enhances the raw backbone's workflow construction capabilities.

• **Group Relative Policy Optimization** [Shao et al., 2024]: GRPO is a reinforcement learning algorithm that normalizes rewards within sampled groups of trajectories. This reduces variance in policy updates, stabilizes training, and improves convergence efficiency compared to standard Proximal Policy Optimization (PPO). Unlike FlowSteer's multi-turn interaction with execution feedback, GRPO generates the entire workflow in one shot, limiting its ability to adapt based on intermediate results.

E.3 Search-Based Workflow Methods

• **AFlow** [Zhang et al., 2024]: A workflow optimization framework that uses Monte Carlo Tree Search (MCTS) to explore the workflow space. The method systematically searches over predefined operator combinations through tree expansion and backpropagation, evaluating candidate workflows by execution outcomes. While effective at finding good operator sequences, AFlow lacks end-to-end learning capability and cannot learn from accumulated experience across different problems. Its search process can be computationally expensive as the workflow complexity increases.

E.4 Agent with Reinforcement Learning Methods

• **AgentFlow**: An agent framework combined with GRPO-based reinforcement learning for workflow construction. The agent dynamically selects tools from a predefined set at each step based on the current state, enabling adaptive decision-making through the interaction process. However, AgentFlow does not support custom prompt specification for operators, limiting its flexibility in fine-tuning the behavior of individual workflow components. This constraint reduces its ability to optimize task-specific reasoning strategies.

• **Router-R1**: A router-style architecture where a small policy model learns to route queries to different processing paths using PPO for policy optimization. The router makes a single decision per query without iterative refinement, selecting from a predefined set of workflow templates. While this approach is computationally efficient, the single-shot routing mechanism cannot adapt to intermediate execution results or refine workflows on partial feedback, limiting its performance on complex multi-step reasoning.

• **Orchestrator**: An orchestrator-style architecture that sequentially selects operators from a library using GRPO. The orchestrator receives partial execution feedback to inform subsequent decisions, enabling some degree of adaptive behavior. However, it lacks the two-step interaction mechanism (add + set_prompt) that FlowSteer employs for fine-grained control over operator configuration.

Table 8: Architectural comparison with baselines.

Method	Dynamic Orchestration	Multi-turn	Exec. Feedback	Custom Prompts	End-to-End RL	Pluggable Backend
Direct LLM	✗	✗	✗	✗	✗	✗
SFT/GRPO	✗	✗	✗	✗	Partial	✗
AFlow	✓	✓	✓	✓	✗	✓
AgentFlow	✗	✓	✓	✓	✓	✓
Router-R1	✗	✓	✓	✗	✓	✓
Orchestrator	✗	✓	Partial	✗	✓	✗
FlowSteer (Ours)	✓	✓	✓	✓	✓	✓

Table 9: Baseline implementation details. All trainable baselines share the same LoRA configuration (rank 64, $\alpha = 64$, dropout 0.05), generation budget (max_tokens=2048, temperature 0.6, top-p 0.95), and training budget (300 steps, batch 36) as FlowSteer wherever the algorithm permits.

Method	Base Model	Training	Key Hyperparameters
Qwen3-8B	Qwen3-8B-Instruct	None	temperature=0.6, top_p=0.95, max_tokens=2048
GPT-4o-mini	GPT-4o-mini	None	temperature=0.6, top_p=0.95, max_tokens=2048
SFT	Qwen3-8B	LoRA	r=64, α=64, dropout=0.05, LR=1×10^{-4}, epochs=3, batch=36
GRPO	Qwen3-8B	GRPO + LoRA	r=64, α=64, dropout=0.05, LR=1×10^{-5}, G=36, steps=300
AFlow	GPT-4o-mini	MCTS	temperature=0.6, search_rounds=21
AgentFlow	Qwen3-8B	GRPO + LoRA	r=64, α=64, dropout=0.05, LR=1×10^{-5}, rollout_n=8, steps=300
Router-R1	Qwen3-8B	PPO + LoRA	r=64, α=64, dropout=0.05, LR=1×10^{-5}, clip=0.2, steps=300
Orchestrator	Qwen3-8B	GRPO + LoRA	r=64, α=64, dropout=0.05, LR=1×10^{-5}, batch=36, max_turns=20

This limitation prevents precise customization of individual operator behaviors, reducing the overall workflow quality.

Table 8 summarizes the key architectural differences between FlowSteer and the baselines, and Table 9 provides the detailed implementation configurations for each method. To ensure fairness, all trainable baselines are aligned with FlowSteer on shared knobs — LoRA configuration ($r = 64$, $\alpha = 64$, dropout 0.05), generation budget (max_tokens=2048, temperature 0.6, top-p 0.95), training budget (300 steps, batch 36, learning rate 1×10^{-5}) — so that performance differences reflect algorithmic design rather than tuning advantage. Algorithm-specific knobs (e.g., rollout count, PPO clip range, MCTS search rounds) follow each baseline's original recommendation.

For a fair comparison, we run FlowSteer and all baselines under identical evaluation settings and report the averaged results. The protocol uses single generation per problem and task-specific metrics (EM/F1 for QA, Accuracy for Math, Pass@1 for Code).

E.5 Other LLM Backends

For transferability experiments (RQ3), we additionally evaluate on six LLM backends to assess the generalization of Flow-Director across different backend models:

• **DeepSeek-V3.2**: DeepSeek-V3.2 adopts advanced context understanding algorithms, enabling it to achieve excellent performance in long-context reasoning and multi-step inference tasks. Its powerful semantic understanding capabilities make it an ideal backend for testing how Flow-Director handles complex reasoning chains.

• **Grok-4.1-Fast**: Grok-4.1-Fast incorporates an efficient inference optimization mechanism, achieving a balance between generation speed and quality, making it suitable for latency-sensitive application scenarios. This backend helps us evaluate Flow-Director's performance under speed-optimized conditions.

• **GPT-5.2**: GPT-5.2 achieves breakthroughs in natural language understanding and generation through enhanced data processing and training strategies, becoming one of the most versatile large language models. It serves as a strong upper bound for backend capability assessment.

• **Claude-Opus-4.5**: Claude-Opus-4.5 is Anthropic's frontier model with strong long-horizon reasoning, code understanding, and tool-use capabilities. Its closed-source frontier design provides a representative test case for high-capacity proprietary backends.

• **Gemini-3-Flash**: Gemini-3-Flash combines a fast inference engine with multimodal understanding capabilities, optimized for real-time interactive applications. Its unique architecture provides a diverse test case for backend generalization.

• **Qwen-Plus-Latest**: Qwen-Plus-Latest incorporates an adaptive learning mechanism based on the Qwen architecture, excelling in few-shot scenarios and cross-language migration tasks. As a model from the same family as our Flow-Director backbone, it provides insights into intra-family transferability.

F Evaluation Metrics

We use task-specific evaluation metrics following standard practices in each domain.

F.1 Exact Match (EM) for Question Answering

Exact Match measures whether the predicted answer exactly matches the ground truth after normalization:

$$\text{EM} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\text{normalize}(y_i) = \text{normalize}(y_i^*)\right), \tag{62}$$

where normalize$(\cdot)$ applies the following transformations: (1) convert to lowercase, (2) remove punctuation (except hyphens in compound words), (3) remove articles ("a", "an", "the"), (4) collapse multiple whitespaces to single space, and (5) strip leading/trailing whitespace.

Applicable datasets: HotPotQA, SQuAD v2, TriviaQA, NaturalQuestions

F.2 F1 Score for Question Answering

F1 Score measures token-level overlap between prediction and ground truth:

$$\text{Precision} = \frac{|y \cap y^*|}{|y|}, \tag{63}$$

$$\text{Recall} = \frac{|y \cap y^*|}{|y^*|}, \tag{64}$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{65}$$

where y and y^* are the sets of tokens in the predicted and ground truth answers respectively, after applying the same normalization as EM.

Applicable datasets: HotPotQA, SQuAD v2, TriviaQA, NaturalQuestions

F.3 Accuracy for Mathematical Reasoning

Accuracy measures whether the predicted numerical answer matches the ground truth within a tolerance:

$$\text{Acc} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(|y_i - y_i^*| < \epsilon\right), \tag{66}$$

where $\epsilon = 10^{-6}$ is the numerical tolerance for floating-point comparisons.

For symbolic answers (e.g., fractions, algebraic expressions), we apply symbolic equivalence checking using SymPy:

$$\text{Acc}_{\text{symbolic}} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\text{simplify}(y_i - y_i^*) = 0\right). \tag{67}$$

Applicable datasets: GSM8K, MATH, MathQA, AIME 2025

F.4 Pass@k for Code Generation

Pass@k measures the probability that at least one of k generated solutions passes all test cases:

$$\text{Pass@}k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \tag{68}$$

where n is the total number of generated samples and c is the number of correct samples (passing all tests).

For our evaluation, we use Pass@1 with a single generation per problem:

$$\text{Pass@1} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\left(\text{execute}(y_i, \text{tests}_i) = \text{pass}\right), \tag{69}$$

where $\text{execute}(y_i, \text{tests}_i)$ runs the generated code y_i against the test suite tests_i.

Applicable datasets: MBPP, HumanEval, APPS, DS-1000

G Implementation Details

To ensure reproducibility and fair comparison, we summarize the complete hyperparameter configurations for FlowSteer in Table 10. The table covers all aspects of our training pipeline, including model configuration, training hyperparameters, RL objective settings, generation parameters, interaction constraints, reward function design, canvas backend configuration, and hardware setup.

Model Configuration. We use Qwen3-8B as the policy model (Flow-Director) and apply LoRA-based fine-tuning with rank 64 and alpha 64, targeting the query, key, value, and output projection layers (q_proj, k_proj, v_proj, o_proj) with a dropout rate of 0.05. The model is trained in bfloat16 precision with gradient checkpointing enabled for memory efficiency.

Training Configuration. We train the Flow-Director agent using the AdamW optimizer with a learning rate of 1×10^{-5} and weight decay of 0.01. The batch size is 36, computed as 6 samples per data source multiplied by 6 data sources (GSM8K, MATH, HotPotQA, SQuAD v2, MBPP, HumanEval). We train for 300 steps with a cosine learning rate schedule.

RL Objective Configuration. For the RL objective described in Section 4.3 (standard GRPO with a canvas-aware token mask), we use the GRPO advantage estimator with a clip range of 0.20, KL coefficient of 0.005, and entropy coefficient of 0.005. Each training step samples 36 trajectories for group-relative advantage estimation.

Generation Configuration. During trajectory generation, we use temperature 0.6, top-p 0.95, and top-k 20 following Qwen3's recommended parameters. The maximum new tokens per turn is set to 2,048 to allow sufficient space for thinking and action generation. We enable Qwen3's thinking mode for enhanced reasoning capabilities and use vLLM with maximum concurrency of 32 for efficient parallel inference.

Interaction Configuration. The maximum interaction rounds $T_{\max}$ is set to 20, and the maximum context length is 16,384 tokens. To ensure workflow quality, we enforce a minimum operator count before allowing the `finish` action, require at least one checker operator (Verify/Test/Review), and require at least one control structure (parallel/conditional/loop) on tasks that benefit from such structures.

Reward Function. The diversity-constrained reward follows the formulation in Section 4.3, with detailed components in Appendix C.1. The structural reward is capped at 1.0, and the answer reward is only released when the structural reward reaches 1.0. This conditional release mechanism prevents shortcut behaviors where the policy might generate trivial workflows to maximize answer rewards.

Canvas Backend and Hardware. The Workflow Canvas uses GPT-OSS-120B as the executor model with temperature 0 for deterministic execution and a timeout of 600 seconds. All experiments are conducted on two NVIDIA A100 80GB GPUs with CUDA 12.5 and vLLM LoRA support enabled for dynamic weight synchronization during training. We use mixed precision training with bfloat16 to reduce memory footprint while maintaining numerical stability.

H Additional Experimental Results

This appendix reports the head-to-head comparison with Workflow-R1, the study of the control-structure prior, the details of cross-executor transfer, and the orchestration-efficiency measurements referenced in Section 5.

Category	Hyperparameter	Value
Model Configuration	Base Model	Qwen3-8B
	LoRA Rank / Alpha	64 / 64
	LoRA Target Modules	q_proj, k_proj, v_proj, o_proj
	LoRA Dropout	0.05
	Data Type	bfloat16
	Gradient Checkpointing	Enabled
Training	Batch Size	36 (6 samples $\times$ 6 sources)
	Learning Rate	1×10^{-5}
	Optimizer	AdamW (weight decay 0.01)
	LR Schedule	Cosine
	Max Training Steps	300
RL Objective	Advantage Estimator	GRPO
	Clip Range (ϵ)	0.20
	KL Coefficient (β)	0.005
	Samples per Group (N)	36
Generation	Temperature	0.6
	Top-p / Top-k	0.95 / 20
	Max New Tokens	2,048
	Enable Thinking Mode	True
	vLLM Max Concurrency	32
Interaction	Max Interaction Rounds ($T_{\max}$)	20
	Max Context Length	16,384
	Min Operators for Finish	enforced
	Require Checker/Structure	True / True
Reward	Base Reward	-1.0
	Structural Reward Cap	1.0
	Correctness Activation	Gate (structural reward $= 1.0$)
Canvas Backend	Executor Model	GPT-OSS-120B (temp=0)
	Execution Timeout	600s
Hardware	GPU Type	NVIDIA A100 80GB $\times$ 2
	CUDA / vLLM LoRA	12.5 / Enabled

Table 10: Complete hyperparameter settings for FlowSteer training.

H.1 Comparison with Workflow-R1

We reproduce Workflow-R1 and compare the best configurations of both methods end to end under the evaluation protocol of Table 3. The two methods differ in what the trainable policy is responsible for and in how the workflow is represented. In Workflow-R1, the policy may call another operator or directly output the final answer, and the selected operators and their results are appended to a think–action–observation history. In FlowSteer, the Flow-Director is restricted to canvas edits and `finish`; the canvas maintains the workflow as operator nodes, node prompts, directed dependency edges, and control structures; and the executor runs the complete graph and alone produces the answer.

Table 11: End-to-end comparison with Workflow-R1 on the twelve datasets under the same evaluation protocol (EM for QA, Acc. for math, Pass@1 for code).

Method	**GSM8K**	**MATH**	**HotPotQA**	**SQuAD v2**	**MBPP**	**HumanEval**	**IID Avg.**
Workflow-R1	**96.09**	78.91	75.00	73.44	76.56	82.03	80.34
FlowSteer	**96.09**	**81.25**	**78.12**	**78.12**	**84.38**	**92.96**	**85.15**
Method	**TriviaQA**	**NQ**	**MathQA**	**AIME 2025**	**APPS**	**DS-1000**	**OOD Avg.**
Workflow-R1	75.78	51.56	83.59	16.67	30.47	37.50	49.26
FlowSteer	**79.69**	**54.69**	**88.67**	**26.67**	**49.21**	**58.59**	**59.59**

As shown in Table 11, FlowSteer outperforms Workflow-R1 on eleven datasets and ties it on GSM8K, and its average advantage grows from 4.82 points on IID tasks to 10.33 points on OOD tasks. Two design choices explain the widening gap. First, restricting the policy to orchestration avoids the

competition between orchestrating and answering directly; in our reproduction, the joint policy tended to answer prematurely on complex tasks. Second, the explicit graph lets the Flow-Director add a verification node, revise a weak node prompt, or rewire a faulty dependency while preserving the validated remainder, and this local repair matters most as task complexity and distribution shift grow.

H.2 Control-Structure Prior versus Template Selection

The control check R_{control} (Appendix C.1) is a training-time structural signal: it fires when the workflow contains at least one `parallel`, `conditional`, or `loop` structure, and it leaves the type and position of that structure, the node composition, the branch count, the dependencies, and the prompts to the Flow-Director. The reward is also absent at inference, where the Flow-Director determines the graph from the task and the canvas feedback. To test whether this prior acts as a hidden template, we compare four settings with the GPT-4o-mini executor on the same evaluation set: a fixed single graph (identical to w/o Agent in Table 4), selection among predefined workflow templates, FlowSteer trained with only R_{control} removed under otherwise matched conditions, and the full method. The fixed graph and the selected template cannot be revised from intermediate feedback, whereas both FlowSteer variants can.

Table 12: Control-structure prior versus fixed and template-based workflows (EM for QA, Acc. for math, Pass@1 for code). w/o R_{control} and Full are FlowSteer variants; All averages the twelve datasets.

Method	**GSM8K**	**MATH**	**HotPotQA**	**SQuAD v2**	**MBPP**	**HumanEval**	**IID Avg.**	
Fixed Single Graph	94.53	70.31	72.66	66.41	60.94	83.59	74.74	
Template Selection	94.01	76.56	72.00	59.84	73.43	85.15	76.83	
w/o R_{control}	95.31	77.34	68.75	58.59	78.91	86.72	77.60	
Full	**96.09**	**81.25**	**78.12**	**78.12**	**84.38**	**92.96**	**85.15**	
Method	**TriviaQA**	**NQ**	**MathQA**	**AIME 2025**	**APPS**	**DS-1000**	**OOD Avg.**	**All**
Fixed Single Graph	69.53	53.12	80.47	23.33	41.40	40.63	51.41	63.08
Template Selection	75.78	49.22	80.47	10.00	42.95	42.97	50.23	63.53
w/o R_{control}	76.56	53.91	82.81	16.67	38.28	36.72	50.83	64.21
Full	**79.69**	**54.69**	**88.67**	**26.67**	**49.21**	**58.59**	**59.59**	**72.37**

As shown in Table 12, FlowSteer without R_{control} still averages 64.21 over the twelve datasets, above the fixed graph (63.08) and template selection (63.53). Feedback-driven graph construction is therefore neither created by this reward nor equivalent to choosing a template. The full method reaches 72.37 and is best on all twelve datasets, so R_{control} provides useful exploration guidance during training without specifying the resulting graph. Consistently, at inference the Flow-Director answers the simple QA task in Appendix I.4 with three sequential operators and no control structure.

H.3 Cross-Executor Transfer Details

All three Flow-Directors in Table 5 are initialized from the same Qwen3-8B model and share the training data, training steps, reward function, LoRA configuration, and generation budget; the only variable is the executor used by the canvas during training. GPT-OSS-120B, DeepSeek-V3.2, and Claude-Opus-4.5 form a progression from a locally deployable executor to strong proprietary ones, with increasing Base performance. During transfer, the Flow-Director parameters stay frozen: we only replace the target executor and perform no additional RL training, fine-tuning, or reward tuning. Each entry of Table 5 averages the primary metric over the twelve datasets.

As shown in Table 13, FlowSteer improves all six target executors on both IID and OOD tasks, by 8.10 and 10.97 points on average, and every executor gains more on OOD than on IID tasks. OOD tasks contain problem distributions unseen during training and leave more room for decomposition, verification, and revision, and the frozen Flow-Director adapts the workflow to the actual feedback of each target executor. In Table 5, a stronger training executor gives more reliable intermediate results, verification judgments, and revision feedback, which reduces reward noise from erroneous execution and lets the Flow-Director learn more reliably when to decompose, verify, or revise. What transfers to weaker executors is this orchestration policy rather than the answer-generation ability of the training executor, and for weaker executors structured decomposition and verification further compensate for limitations in single-pass problem solving. When the training and target executors coincide, the

Table 13: IID and OOD averages behind Figure 4 for the frozen Flow-Director trained with GPT-OSS-120B. Base runs the target executor without FlowSteer.

Target Executor	IID Base	IID FlowSteer	OOD Base	OOD FlowSteer
DeepSeek-V3.2	81.23	90.62	64.63	74.63
Grok-4.1-Fast	85.25	91.67	71.60	79.83
GPT-5.2	82.62	90.63	67.60	78.63
Claude-Opus-4.5	85.93	93.22	70.80	81.23
Gemini-3-Flash	84.62	92.83	70.40	81.03
Qwen-Plus-Latest	81.08	90.37	60.23	75.67
Average	83.46	91.56	67.54	78.51

distributions of execution results, error types, and verification feedback are better aligned, which yields the additional gain on the diagonal.

H.4 Orchestration Efficiency

Progressive editing commits one atomic edit per turn, which increases the number of Flow-Director–canvas interactions. At each turn, however, the Flow-Director only outputs the next edit and its parameters from the current graph state and the latest feedback; it neither regenerates the complete workflow nor solves the task. In Figure 5(c), w/o Agent corresponds to a static workflow that executes the same predefined graph for every instance, and w/o Multi-turn corresponds to single-shot workflow generation that produces a complete workflow per instance before executing it. FlowSteer uses fewer tokens than both in all three domains (e.g., 7,096 versus 9,997 and 10,686 on math), because it reuses the graph state maintained by the canvas, emits only local edits, and can issue `finish` as soon as the workflow suffices.

Table 14: Per-dataset orchestration cost of FlowSteer: average interaction turns, Flow-Director orchestration time, workflow execution time, and API calls per instance.

Dataset	Avg. Turns	Orchestration Time (s)	Execution Time (s)	API Calls
GSM8K	8.31	4.55	9.29	11.95
MATH	9.51	5.19	11.19	13.59
MathQA	8.16	5.24	10.93	11.38
AIME 2025	12.37	15.85	41.15	16.70
SQuAD v2	7.02	8.01	15.40	9.95
HotPotQA	9.80	4.60	8.97	13.56
TriviaQA	6.82	4.11	7.60	9.40
NQ	7.05	3.33	5.83	9.66
MBPP	7.24	3.98	10.05	9.61
HumanEval	7.10	3.05	7.51	9.80
APPS	15.58	9.20	28.18	28.33
DS-1000	11.68	6.12	17.97	14.70

Table 14 breaks down the end-to-end cost. Workflow execution takes most of the time on all twelve datasets, so multi-turn canvas interaction is not the main source of latency. Harder tasks such as AIME 2025 and APPS take more turns, and most of their extra time is spent executing the added decomposition, verification, or revision nodes rather than on Flow-Director orchestration. This matches how FlowSteer adapts computation to instance difficulty: simple instances invoke `finish` early, whereas complex instances keep expanding or repairing the graph.

I Case Study

We present four detailed case studies illustrating how FlowSteer orchestrates workflow construction through multi-turn interaction between Flow-Director and Workflow Canvas, demonstrating sequential, parallel, conditional, and simple workflow structures. These case studies provide concrete examples of how our end-to-end reinforcement learning framework addresses the key challenges in workflow orchestration: reducing manual effort, enabling plug-and-play operator composition, and learning from execution feedback. Throughout these examples, we highlight the round-by-round

interaction process, showing how the policy model analyzes execution states, selects editing actions, and iteratively refines the workflow until obtaining the correct answer.

I.1 Case Study 1: Sequential Workflow Structure

We present a case study from AIME 2025 demonstrating how FlowSteer constructs **sequential workflow structures** for mathematical reasoning tasks that require planning, computation, verification, and formatting.

AIME 2025 Problem

From an unlimited supply of 1-cent, 10-cent, and 25-cent coins, Silas wants to find a collection of coins that has a total value of N cents. He uses the **greedy algorithm**, successively choosing the coin of greatest value that does not cause the value to exceed N. For example, to get 42 cents, Silas will choose a 25-cent coin, then a 10-cent coin, then 7 1-cent coins. However, this collection of 9 coins uses more coins than necessary; indeed, choosing 4 10-cent coins and 2 1-cent coins achieves the same total value with only 6 coins.
In general, the greedy algorithm succeeds for a given N if no other collection gives a total value of N cents using strictly fewer coins. Find the number of values of N between 1 and 1000 inclusive for which the greedy algorithm succeeds. **Ground Truth: 610**

Final Workflow: Plan → Programmer → Verify → Format (9 rounds, 4 operators)

I.1.1 Round-by-Round Interaction Log

Round 1-2: Plan Operator

Round 1 - Think:
This is a combinatorial optimization problem involving coin denominations {1, 10, 25}. The task requires counting how many values $N \in [1, 1000]$ satisfy the condition that greedy selection equals optimal selection. This is NOT a simple arithmetic problem—it requires systematic comparison across 1000 cases. Before jumping into implementation, I need a clear strategy. **Decision: Add Plan operator** to decompose the problem into:

- Greedy algorithm formulation
- Optimal solution computation
- Systematic comparison methodology

Round 1 - Action: <action>add</action><operator>Plan</operator>

Round 1 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: (empty)

Round 2 - Input Prompt:
Analyze the coin change problem with denominations {1, 10, 25} cents. The greedy algorithm always picks the largest coin ≤ remaining value. For example, N=42: greedy gives 25+10+7×1=9 coins, but optimal is 4×10+2×1=6 coins.
Create a strategic plan to count how many $N \in [1, 1000]$ have greedy count = optimal count. Include:

- *APPROACH for systematic comparison*
- *KEY_INSIGHTS about when greedy fails*
- *PLAN with concrete algorithmic steps*

Operator Execution Output:

Approach:
Implement both greedy and optimal (DP-based) coin counting algorithms, then systematically compare their outputs for all N from 1 to 1000.

Key Insights:

1. Greedy always picks largest coin $\leq N$, which is locally optimal but not always globally optimal.
2. Greedy fails when using fewer large coins plus more medium coins yields fewer total coins (e.g., N=30: greedy=25+5×1=6 coins, optimal=3×10=3 coins).
3. Dynamic programming guarantees finding the true minimum coin count.
4. The coin system {1,10,25} is NOT canonical (unlike {1,5,10,25} US coins), so greedy can fail.

Plan:

1. Implement `greedy_coins(n)` that iteratively subtracts largest possible coin.
2. Implement `min_coins(n)` using DP with recurrence $dp[i] = \min_{c \in \{1,10,25\}}(dp[i-c]+1)$.
3. Initialize $dp[0] = 0$, $dp[i] = \infty$ for $i > 0$.
4. For each $N \in [1, 1000]$, compute both counts.
5. Count cases where `greedy_coins(N) == min_coins(N)`.
6. Return the final count.

Feedback: [Output]: Strategy generated with 4 key insights and 6-step plan | [Current DSL]: Plan

Workflow State: (empty) → Plan (executed)

Round 3-4: Programmer Operator

Round 3 - Think:
The **Plan** operator has provided a clear 6-step algorithmic strategy:

greedy function → DP function → initialization → iteration → comparison → counting

This is a computational task requiring exact numerical results across 1000 test cases—natural language reasoning would be error-prone and inefficient.
Decision: Add Programmer operator to implement the greedy and DP algorithms in Python, ensuring:

- Precise computation without rounding errors
- Automatic iteration over all $N \in [1, 1000]$
- Verifiable code that can be re-executed

Round 3 - Action: <action>add</action><operator>Programmer</operator>
Round 3 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: Plan

---

Round 4 - Input Prompt:
Based on the plan, implement Python code for the coin problem with denominations [25, 10, 1]. Requirements:

1. *`greedy_coins(n)`: subtract largest coin $\leq n$ repeatedly, count coins used*
2. *`min_coins(n)`: use DP with $dp[i] = \min(dp[i], dp[i-c]+1)$ for each coin c*
3. *`solve()`: iterate $N \in [1, 1000]$, count cases where greedy equals optimal*

Return the final count as an integer.

Operator Execution Output:

Generated Code:

```
def greedy_coins(n):
    coins = [25, 10, 1]
    count = 0
    for coin in coins:
        while n >= coin:
            n -= coin
            count += 1
    return count

def min_coins(n):
    dp = [float('inf')] * (n + 1)
    dp[0] = 0
    for coin in [25, 10, 1]:
        for i in range(coin, n + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)
    return dp[n]

def solve():
    count = 0
    for n in range(1, 1001):
        if greedy_coins(n) == min_coins(n):
            count += 1
    return count
```

Execution Result: **610**

Feedback: [Output]: **610** | [Current DSL]: Plan → Programmer

Workflow State: Plan → Programmer (executed, output=610)

Round 5-6: Verify Operator

Round 5 - Think:
The **Programmer** returned **610**, but this is a competition-level math problem (AIME 2025) where correctness is critical.
The code logic appears sound, but I should NOT trust a single computation path without independent verification. Potential issues include:

- Off-by-one errors in range [1,1000]
- DP initialization bugs
- Greedy algorithm edge cases

Decision: Add Verify operator to independently re-derive the solution logic and cross-check the numerical result before finalizing.

Round 5 - Action: <action>add</action><operator>Verify</operator>

Round 5 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: Plan → Programmer

---

Round 6 - Input Prompt:
*The Programmer computed **610** as the count of $N \in [1, 1000]$ where greedy equals optimal for coins {1,10,25}.*
Independently verify this result:

1. *Re-examine the problem statement—greedy succeeds iff no collection uses strictly fewer coins*
2. *Validate the greedy logic: for N=42, greedy gives 25+10+7×1=9 coins, optimal is 4×10+2×1=6 coins (greedy fails)*
3. *Check boundary cases N=1, 10, 25, 100, 1000*
4. *Confirm or refute the answer 610 with confidence assessment*

Operator Execution Output:

[Step 1] Problem Re-examination:
The greedy algorithm succeeds for N iff `greedy_coins`(N) = `min_coins`(N). We need to count such N in the range [1, 1000].

[Step 2] Algorithm Validation:

- **Greedy:** Iteratively pick largest coin $\leq$ remaining value
- **DP:** $dp[0] = 0$, $dp[i] = \min_c(dp[i-c] + 1)$
- Both implementations follow standard algorithms and are correct

[Step 3] Test Case Verification:

N	Greedy	Optimal	Result
42	25+10+7×1 = 9	4×10+2×1 = 6	Fails
30	25+5×1 = 6	3×10 = 3	Fails
25	1×25 = 1	1 coin	Succeeds
100	4×25 = 4	10×10 = 10	Succeeds
1000	40×25 = 40	100×10 = 100	Succeeds

[Step 4] Cross-Validation:
Independent recalculation confirms **610** values where greedy succeeds.

Verification Result:
`is_correct`: **True** | `confidence`: **HIGH** | `verified_answer`: **610**

Feedback: [Output]: **is_correct=True**, confidence=HIGH | [Current DSL]: Plan → Programmer → Verify

Workflow State: Plan → Programmer → Verify (verified, is_correct=True)

Round 7-8: Format Operator

Round 7 - Think:
The **Verify** operator has independently confirmed that **610** is correct with HIGH confidence.
The workflow now has a complete reasoning chain:

Plan (strategy) → Programmer (computation) → Verify (validation)

However, the current output contains verbose execution traces and intermediate results. For final submission, I need a clean, concise answer.
Decision: Add Format operator to extract only the final numerical answer "610" without any surrounding explanation or computation details.

Round 7 - Action: <action>add</action><operator>Format</operator>

Round 7 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: Plan → Programmer → Verify

---

Round 8 - Input Prompt:
*The verified answer for the AIME 2025 coin problem is **610** (count of $N \in [1, 1000]$ where greedy = optimal).*
Extract ONLY the final integer answer:

- *Do NOT include any explanation, units, or formatting*
- *Do NOT re-solve the problem*
- *Output must be directly usable as competition submission*
- *Just the number*

Operator Execution Output:

Formatted Answer:

610

Feedback: [Output]: **610** | [Current DSL]: Plan → Programmer → Verify → Format

Workflow State: Plan → Programmer → Verify → Format (formatted)

Round 9: Finish

Round 9 - Think:
The workflow is now complete with four operators forming a robust reasoning pipeline:

Plan	Decomposed the problem into algorithmic steps
Programmer	Implemented greedy and DP algorithms, computed 610
Verify	Independently validated correctness with test cases
Format	Extracted clean answer for submission

All verification checks passed with HIGH confidence. The formatted output "**610**" is ready for submission.
Decision: Execute finish action to terminate the workflow and return the final answer.

Round 9 - Action: <action>finish</action>

Final Status:
[Status]: **SUCCESS** | [Final DSL]: Plan → Programmer → Verify → Format | [Result]: **610**

Final Workflow State:

Plan → Programmer → Verify → Format

Predicted Answer: 610 ✓ **Matches Ground Truth**

Key Observations from Case Study 1: This sequential workflow demonstrates how FlowSteer effectively chains operators to solve complex mathematical problems. The Plan operator provides strategic decomposition, the Programmer operator implements precise algorithms, the Verify operator ensures correctness through independent validation, and the Format operator produces clean output. The workflow's success highlights the importance of verification in competition-level mathematics, where a single computational error can lead to incorrect answers. The 9-round interaction shows efficient operator selection without unnecessary complexity.

I.2 Case Study 2: Parallel Workflow Structure

We present a second case study demonstrating how FlowSteer constructs **parallel workflow structures** for multi-hop QA tasks that benefit from decomposition into independent sub-problems.

Multi-Hop QA Problem

Based on the following passages, answer the question:
[Park Holidays UK] Park Holidays UK operate 28 holiday parks in the South of England...
[Hard Rock Hell] ...the festival was held at Pontin's Holiday Village in Prestatyn, Wales...
[Pontins] Pontins is the trading name of Britannia Jinky Jersey Ltd, a company operating holiday parks in the United Kingdom...

Question: What is the name shared between a city in Prestatyn, Wales and a company operating holiday parks in the UK? **Ground Truth: Pontins**

Final Workflow: Decompose → [Custom, Custom, Custom] → Verify → Aggregate (11 rounds, 5 operators with parallel structure)

I.2.1 Round-by-Round Interaction Log

Round 1-2: Decompose Operator

Round 1 - Think:
This is a multi-hop QA problem requiring information synthesis from multiple passages. The question asks for a name shared between a location (Prestatyn, Wales) and a company (holiday parks operator).
This requires: (1) identifying the city in Prestatyn, (2) identifying UK holiday park companies, and (3) finding the shared name. These are independent sub-problems that can be solved in parallel.
Decision: Add Decompose operator to break down the question into independent sub-problems for parallel processing.

Round 1 - Action: <action>add</action><operator>Decompose</operator>

Round 1 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: (empty)

Round 2 - Input Prompt:
Analyze the multi-hop question about Prestatyn, Wales and UK holiday park companies. Break down into independent sub-problems that can be solved in parallel.
Operator Execution Output:

Sub-problems:
1. Identify the city mentioned in Prestatyn, Wales from the passages.
2. Identify the companies operating holiday parks in the UK mentioned in the passages.
3. Determine if any identified companies share a name with the city in Prestatyn.

Reasoning: The first sub-problem establishes the location reference. The second sub-problem lists relevant companies. The third sub-problem finds the intersection. These can be processed independently then aggregated.

Feedback: [Output]: 3 independent sub-problems identified | [NEXT]: ADD:Custom (3 solvers in parallel, then Aggregate)

Workflow State: (empty) → Decompose (executed)

Rounds 3–10: Parallel Solving, Verification, and Aggregation

Round 3–6: Parallel Custom Operators (3 branches). The agent adds a parallel structure with three Custom solvers, one per sub-problem. Each branch independently produces:

- **Branch 1 (Location):** "Pontin's Holiday Village in Prestatyn, Wales" ⇒ **Pontins**.
- **Branch 2 (Companies):** Park Holidays UK, Park Resorts, **Pontins** (Britannia Jinky Jersey Ltd).
- **Branch 3 (Cross-reference):** "Pontins" appears as both a location in Prestatyn and a company name.

All three branches converge on the answer **Pontins**.

Round 7–8: Verify Operator. The Verify operator independently re-checks the answer against passages: (i) Pontins is associated with Prestatyn, Wales; (ii) Pontins operates UK holiday parks; (iii) no other name satisfies both. Result: `is_correct=True, confidence=HIGH, suggested_answer=Pontins`.

Round 9–10: Aggregate Operator. The Aggregate operator consolidates four consistent sources (3 branches + Verify) into a single output. **Aggregation Summary:** all 4 sources agree on **Pontins**; output: `Pontins`, `is_correct=True`, `confidence=HIGH`.

Workflow State after Aggregation: Decompose → [Custom, Custom, Custom] → Verify → Aggregate (aggregated).

Round 11: Finish

Round 11 - Think:
The workflow is now complete with a parallel structure demonstrating multi-path reasoning:

Decompose	Broke question into 3 independent sub-problems
[Custom×3]	Solved sub-problems in parallel with different perspectives
Verify	Validated answer against original passages
Aggregate	Consolidated parallel outputs into final answer

All verification checks passed. The aggregated output "**Pontins**" is ready for submission.
Decision: Execute finish action to terminate the workflow.

Round 11 - Action: <action>finish</action>

Final Status:
[Status]: **SUCCESS** | [Final DSL]: Decompose → [Custom×3] → Verify → Aggregate | [Result]: **Pontins**

Final Workflow State:
Decompose → [Custom×3] → Verify → Aggregate

Predicted Answer: Pontins ✓ **Matches Ground Truth**

Key Observations from Case Study 2: This parallel workflow demonstrates FlowSteer's ability to decompose multi-hop QA problems into independent sub-problems that can be solved concurrently. The

Decompose operator identifies three distinct reasoning paths, the parallel Custom operators explore each path independently, and the Aggregate operator consolidates the results. The Verify operator provides an additional layer of confidence by cross-checking against the original passages. This structure is particularly effective for questions requiring information synthesis from multiple sources, as it reduces the risk of missing relevant information by exploring multiple perspectives simultaneously.

I.3 Case Study 3: Conditional Branch Structure

We present a third case study demonstrating how FlowSteer constructs **conditional branch structures** for code generation tasks that require iterative debugging and repair when tests fail.

Codeforces Problem: Snowball Game

In late autumn evening n robots gathered in the cheerful company of friends. Each robot has a unique identifier—an integer from 1 to 10^9.
At some moment, robots decided to play the game "Snowball". The rules are: First, all robots stand in a row. Then the first robot says his identifier. After that the second robot says the identifier of the first robot and then says his own identifier. Then the third robot says the identifiers of the first and second robots, then his own. And so on.
Given n robots with identifiers and a number k, find the k-th identifier pronounced in the game.
Input: n, k, and n robot identifiers **Output:** The k-th pronounced identifier **Ground Truth: Correct Algorithm**

Final Workflow: Plan → Programmer → Test → Test ? Programmer : done → Test ? Programmer : done

(13 rounds, 5 operators with conditional branches for iterative repair)

I.3.1 Round-by-Round Interaction Log

Round 1-2: Plan Operator

Round 1 - Think:
This is a code generation task requiring algorithmic reasoning. The "Snowball" game creates a sequence where robot i pronounces i identifiers. The total count after i robots is $1 + 2 + \cdots + i = \frac{i(i+1)}{2}$.
Before coding, I need to plan the algorithm to find which robot pronounces the k-th identifier and which position within that robot's sequence.
Decision: Add Plan operator to design the algorithmic approach.
Round 1 - Action: <action>add</action><operator>Plan</operator>
Round 1 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: (empty)

Round 2 - Input Prompt:
Design an algorithm for the Snowball game problem. Robot i says i identifiers (all previous robots' IDs plus its own). Given k, find the k-th pronounced identifier.
Operator Execution Output:

Algorithm Design:

1. The cumulative count after robot i is $\sum_{j=1}^{i} j = \frac{i(i+1)}{2}$
2. Find the smallest i such that $\frac{i(i+1)}{2} \geq k$
3. The position within robot i's sequence is $k - \frac{(i-1)i}{2}$
4. Return the identifier at that position (1-indexed from the robot list)

Feedback: [Output]: Algorithm planned | [NEXT]: ADD:Programmer
Workflow State: (empty) → Plan (executed)

Round 3-4: Programmer Operator

Round 3 - Think:
The **Plan** operator provided a clear algorithm. Now I need to implement it in code.
Decision: Add Programmer operator to generate executable Python code.
Round 3 - Action: <action>add</action><operator>Programmer</operator>

Round 4 - Operator Execution Output:

Generated Code (v1):

```
def solve():
    data = input().split()
    n, k = int(data[0]), int(data[1])
    ids = list(map(int, data[2:n+2]))

    current_count = 0
    for i in range(1, n + 1):
        if current_count + i >= k:
            position = k - current_count
            print(ids[position])  # Bug: should be position-1
            return
        current_count += i
```

Feedback: [Output]: CODE_GENERATED | [NEXT]: ADD:Test

Workflow State: Plan → Programmer (code generated)

Round 5-6: Test Operator (First Failure)

Round 5 - Think:
Code has been generated. Before finalizing, I must test it against sample inputs to verify correctness.
Decision: Add Test operator to run unit tests on the generated code.
Round 5 - Action: <action>add</action><operator>Test</operator>

Round 6 - Test Execution Output:

TEST_FAILED: FAILED 4/5 public tests.
[EDGE CASE] 4/5 passed. Check edge cases:

- Off-by-one errors ($<$ vs $\leq$)?
- Array indexing (0-based vs 1-based)?

[First Failure Details]:

```
Input: 4 5
10 4 18 3

Expected: 4
Got: 10
```

Analysis: The algorithm logic has an **off-by-one error** in array indexing.

Feedback: [NEXT]: ADD:conditional(Test, Programmer, done) | Test failed—add conditional branch for repair

Workflow State: Plan → Programmer → Test (FAILED)

Round 7-9: First Conditional Branch (Test ? Programmer : done)

Round 7 - Think:
The test failed with an off-by-one error. Instead of manually fixing, I should add a conditional branch that automatically triggers repair when tests fail.

Decision: Add conditional(Test, Programmer, done) structure:

- If Test fails → execute Programmer (repair)
- If Test passes → done (exit branch)

Round 7 - Action:
<action>add</action><structure>conditional</structure>
<condition>Test</condition><true_branch>Programmer</true_branch>
<false_branch>done</false_branch>

Round 8 - Test (Condition Check):
Re-running test to check condition...

> **Condition Result:** TEST_FAILED → Execute **true_branch** (Programmer)

Round 9 - Programmer (First Repair Attempt):

> **[LAST_ERROR]:** Off-by-one error. Expected 4, got 10.
> **Repair Analysis:** The bug is in `ids[position]`—should be `ids[position-1]` for 0-based indexing.
> **Generated Code (v2):**
>
> ```
> print(ids[position - 1])  # Fixed: 0-based indexing
> ```
>
> **Note:** First repair attempt still has issues...

Feedback: [NEXT]: ADD:conditional(Test, Programmer, done) | Still failing—add another repair loop

Workflow State: Plan → Programmer → Test → Test ? Programmer : done (repair loop 1)

Round 10-12: Second Conditional Branch (Successful Repair)

Round 10 - Think:
First repair attempt still failing. Adding another conditional branch for a second repair iteration.

Round 10 - Action:
<action>add</action><structure>conditional</structure>
<condition>Test</condition><true_branch>Programmer</true_branch>
<false_branch>done</false_branch>

Round 11 - Test (Condition Check):

> **Condition Result:** TEST_FAILED → Execute **true_branch** (Programmer)

Round 12 - Programmer (Second Repair—Success):

[LAST_ERROR]: Still off-by-one. Re-analyzing the algorithm...

Root Cause Analysis:

- Input: $n = 4$, $k = 5$, ids = [10, 4, 18, 3]
- Robot 1 says: 10 (count=1)
- Robot 2 says: 10, 4 (count=3)
- Robot 3 says: 10, 4, 18 (count=6) $\leftarrow k = 5$ falls here
- Position within Robot 3: $5 - 3 = 2 \rightarrow$ ids[1] = **4** ✓

Generated Code (v3 - Final):

```
def solve():
    data = input().split()
    n, k = int(data[0]), int(data[1])
    ids = list(map(int, data[2:n+2]))

    current_count = 0
    for i in range(1, n + 1):
        if current_count + i >= k:
            position = k - current_count - 1  # Fixed!
            print(ids[position])
            return
        current_count += i
```

TEST_PASSED: All 5/5 public tests passed!

Feedback: [BLOCK_FINISH]=0 | [NEXT]: FINISH (Answer correct)

Workflow State: Plan → Programmer → Test → Test?Programmer:done → Test?Programmer:done (PASSED)

Round 13: Finish

Round 13 - Think:
The workflow successfully repaired the code through two conditional branch iterations:

Operator	Role	Status
Plan	Algorithm design	✓
Programmer	Initial code generation	✓
Test	Detect off-by-one bug	✗
Test?Prog:done	First repair attempt	✗
Test?Prog:done	Second repair (success)	✓

Decision: Execute finish action to terminate the workflow.

Round 13 - Action: <action>finish</action>

Final Status:

[Status]: SUCCESS | [Final DSL]: Plan → Prog → Test → Test?Prog:done^2

[Result]: **TEST_PASSED**

Final Workflow Visualization:

Plan → Prog → Test → Test?Prog:done → Test?Prog:done

Predicted Answer: TEST_PASSED ✓ **All Tests Passed**

Key Observations from Case Study 3: This conditional workflow demonstrates FlowSteer's iterative debugging capability through conditional branches. When the initial code fails tests, the system automatically triggers repair loops rather than terminating. The Test operator serves as both a validator and a condition checker, while the Programmer operator performs targeted repairs based on error feedback. This structure mirrors real-world software development practices where code undergoes multiple iterations before passing all tests. The successful repair after two iterations shows the system's ability to learn from failures and progressively refine solutions.

I.4 Case Study 4: Simple QA Workflow

We present a fourth case study demonstrating how FlowSteer constructs a **minimal sequential workflow** for straightforward QA tasks that require only reasoning, review, and formatting.

Simple QA Problem

Based on the following passages, answer the question:
[Chicago State University] Chicago State University (CSU) is a state university of the U.S. state of Illinois, located in Chicago. The University is a member-school of Thurgood Marshall College Fund.
[Emil and Patricia Jones Convocation Center] The Jones Convocation Center (JCC) is a 7,000-seat multi-purpose arena in Chicago, Illinois. It is home to the Chicago State University Cougars basketball teams...

Question: Which school fund is the Emil and Patricia Jones Convocation Center a member of? **Ground Truth: Thurgood Marshall College Fund**

Final Workflow: Custom → Review → Format (7 rounds, 3 operators)

I.4.1 Round-by-Round Interaction Log

Round 1-2: Custom Operator

Round 1 - Think:
This is a straightforward QA task requiring information extraction and reasoning across passages. The question asks about a "school fund" associated with the Jones Convocation Center.
The reasoning chain is simple:

1. JCC is home to Chicago State University teams
2. CSU is a member of Thurgood Marshall College Fund
3. Therefore, JCC is associated with that fund

No code execution or complex decomposition needed. **Decision: Add Custom operator** for natural language reasoning.

Round 1 - Action: <action>add</action><operator>Custom</operator>

Round 2 - Operator Execution Output:

Reasoning:
The Emil and Patricia Jones Convocation Center is home to the Chicago State University Cougars basketball teams. According to the Chicago State University passage, CSU is a member-school of the **Thurgood Marshall College Fund**.
Answer: Thurgood Marshall College Fund

Feedback: [Output]: Thurgood Marshall College Fund | [NEXT]: ADD:Review

Workflow State: (empty) → Custom (executed)

Round 3-4: Review Operator

Round 3 - Think:
The **Custom** operator produced an answer. Before finalizing, I should review the reasoning quality to ensure the answer is well-supported by the passages.
Decision: Add Review operator to evaluate answer quality.

Round 3 - Action: <action>add</action><operator>Review</operator>

Round 4 - Operator Execution Output:

Review Assessment:

- **Reasoning validity:** ✓ Correct chain from JCC → CSU → Fund
- **Evidence support:** ✓ Directly stated in CSU passage
- **Answer completeness:** ✓ Full fund name provided

Verdict: Answer is correct and well-supported.

Feedback: [Output]: Thurgood Marshall College Fund | [NEXT]: ADD:Format

Workflow State: Custom → Review (reviewed)

Round 5-6: Format Operator

Round 5 - Think:
The Review operator confirmed the answer is correct. Now I need to format the output for final submission.
Decision: Add Format operator to extract the concise answer.
Round 5 - Action: <action>add</action><operator>Format</operator>

Round 6 - Operator Execution Output:

Formatted Answer:

Thurgood Marshall College Fund

Feedback: [BLOCK_FINISH]=0 | [NEXT]: FINISH (Answer correct and concise)

Workflow State: Custom → Review → Format (formatted)

Round 7: Finish

Round 7 - Think:
The workflow is complete with a minimal 3-operator pipeline:

Custom	Natural language reasoning
Review	Quality assessment
Format	Answer extraction

Decision: Execute finish action.
Round 7 - Action: <action>finish</action>
Final Status:

[Status]: **SUCCESS** | [Final DSL]: Custom → Review → Format | [Result]: **Thurgood Marshall College Fund**

Final Workflow State:

Custom → Review → Format

Predicted Answer: Thurgood Marshall College Fund ✓ **Matches Ground Truth**

Key Observations from Case Study 4: This minimal workflow demonstrates FlowSteer's ability to recognize when simple problems require simple solutions. Unlike the previous case studies that employed complex structures, this workflow uses only three operators in a straightforward sequence. The Custom operator handles natural language reasoning, the Review operator validates the reasoning quality, and the Format operator extracts the final answer. This efficiency is crucial for practical deployment, as over-engineering simple tasks wastes computational resources. The 7-round completion shows that FlowSteer adapts its workflow complexity to match task requirements.

Summary of Case Studies: These four case studies collectively demonstrate the versatility of FlowSteer's workflow orchestration. Sequential workflows (Case Study 1) excel at multi-step reasoning with verification. Parallel workflows (Case Study 2) enable efficient exploration of multiple reasoning paths. Conditional workflows (Case Study 3) support iterative refinement through automated repair loops. Minimal workflows (Case Study 4) ensure computational efficiency for straightforward tasks. Together, these structures cover a wide range of reasoning scenarios encountered in real-world applications.

J Limitations

Despite its strong empirical performance across various reasoning tasks, FlowSteer has several limitations. First, its heavy reliance on historical context is a key structural constraint. As multi-turn interactions progress, the quality of initial operator outputs and workflow decisions becomes critical for sustaining accurate reasoning. Even subtle errors introduced at early stages (e.g., incorrect problem decomposition by the Plan operator) may accumulate rapidly via error propagation through subsequent operators, affecting the reliability and accuracy of the final output. Consequently, when the historical context is incomplete or noisy, the Flow-Director's orchestration ability can be compromised. Additionally, the method depends heavily on continuous and dynamic updates to the workflow state through the Canvas. If these updates fail to capture execution feedback promptly or the context window becomes saturated on long-horizon tasks, it can lead to information loss or suboptimal workflow decisions, thereby limiting the framework's practical flexibility and effectiveness. Finally, the structural reward and the action interface are defined over the executable graph representation maintained by the Canvas. This representation covers sequential paths, DAG-based collaboration, conditional branches, loops, and their instance-specific compositions, and extending the reward to orchestration paradigms beyond graph-structured workflows is a direction for future work.

K Future Work

A central direction for future work on FlowSteer is to address the memory bottleneck of the underlying large language model when orchestrating long, multi-turn workflows. Because the Flow-Director conditions every editing decision on the full interaction history maintained in the Canvas, its effective planning horizon is ultimately bounded by the model's context window: as trajectories grow, early operator outputs, intermediate verification feedback, and prior structural decisions are increasingly diluted or truncated, weakening the policy's ability to reason globally about the workflow. We therefore plan to investigate dedicated memory mechanisms for the LLM-based Director, including hierarchical context compression that retains structural summaries of completed sub-workflows while discarding token-level detail, retrieval-augmented memory that stores past canvas states in an external store and selectively reinjects only the segments relevant to the current edit, and learned memory tokens that allow the policy itself to decide which historical artifacts to keep, drop, or rewrite. Together, these directions aim to extend FlowSteer's effective reasoning horizon and contextual coherence on long-horizon tasks without inflating per-step inference cost.

L Applicability Analysis

FlowSteer, with its multi-turn workflow orchestration and dynamic canvas updating capabilities, demonstrates clear potential for application in tasks that require rigorous, multi-step logical deduction. Beyond the mathematical reasoning, multi-hop question answering, and code generation benchmarks evaluated in this paper, the same paradigm naturally extends to other compositional reasoning settings where solutions can be expressed as a sequence of plan, decompose, verify, and refine steps. Through the pluggable backend architecture, FlowSteer can pair a small Director with stronger executor models when reasoning quality is paramount, or with lighter executors when latency and cost dominate, allowing the framework to adapt to a range of deployment regimes without retraining the Director. Moreover, by integrating reinforcement learning into reinforced progressive canvas editing, FlowSteer not only handles fixed supervised tasks but also adapts to changing requirements by continuously optimizing its orchestration strategies via intrinsic verification and refinement operators, thereby enhancing adaptive intelligence and long-horizon planning. Overall, FlowSteer offers transparent and interpretable workflow structures, cross-backend adaptability, and strong reasoning capability, providing a general substrate for trustworthy multi-step decision-making across knowledge-intensive tasks of similar structure to the benchmarks studied here.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state our three contributions (the *Agent Designing Agentic Workflows* paradigm, the *Workflow Canvas* executable graph-state environment, and *Reinforced Progressive Canvas Editing*) and the empirical scope (twelve datasets across math, QA, code over six modern LLM backends), all of which are supported by the experimental results in Sections 5.2–5.6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss limitations in Appendix J, including the heavy reliance on historical context, the risk of error propagation through long multi-turn trajectories, the dependence on continuous and dynamic Canvas updates to capture execution feedback, and the graph-based scope of the reward and action interface.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: Three propositions are stated in Sections 4.1–4.3; full proofs are provided in Appendix B.1–B.3.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Implementation details (hyperparameters, training configuration, hardware) are provided in Appendix G, dataset details in Appendix D, and baseline details in Appendix E.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility.

In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Code is released at `https://github.com/beita6969/FlowSteer`; all benchmarks are public.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: See Section 5 and Appendix G.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [N/A]

   Justification:

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Hardware configuration (GPU type and count, CUDA version, mixed-precision setting, vLLM concurrency, executor model and timeout) is reported in Appendix G.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work uses only public benchmarks and pretrained models; no human subjects or sensitive data are involved.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Applicability and broader impacts are discussed in Appendix L.

    Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We do not release new pretrained models or scraped datasets that pose misuse risks beyond the underlying base LLMs.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All datasets, base models, and baseline frameworks used in this work are cited via their original publications in Appendices D and E; we use them only for non-commercial academic research in accordance with their respective terms of use.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The released code repository contains a README with usage instructions.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: No human subjects involved.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: No human subjects involved.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are central to the methodology and are described throughout (Qwen3-8B as Director, GPT-4o-mini / GPT-OSS-120B / six other backends as workflow execution engines).

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.