FlowSteer: Interactive Agentic Workflow Orchestration via End-to-End Reinforcement Learning

Anonymous Authors[1]

Abstract

In recent years, a variety of powerful agentic workflows have been applied to solve a wide range of human problems. However, existing workflow orchestration still faces key challenges, including high manual cost, reliance on specific operators/large language models (LLMs), and sparse reward signals. To address these challenges, we propose FlowSteer, an end-to-end reinforcement learning framework that takes a lightweight policy model as the agent and an executable canvas environment, automating workflow orchestration through multi-turn interaction. In this process, the policy model analyzes execution states and selects editing actions, while the canvas executes operators and returns feedback for iterative refinement. Moreover, FlowSteer provides a plug-and-play framework that supports diverse operator libraries and interchangeable LLM backends. To effectively train this interaction paradigm, we propose Canvas Workflow Relative Policy Optimization (CWRPO), which introduces diversity-constrained rewards with conditional release to stabilize learning and suppress shortcut behaviors. Experimental results on twelve datasets show that FlowSteer significantly outperforms baselines across various tasks. Our code is available at https://anonymous.4open.science/r/FlowSteer-9B2E.

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

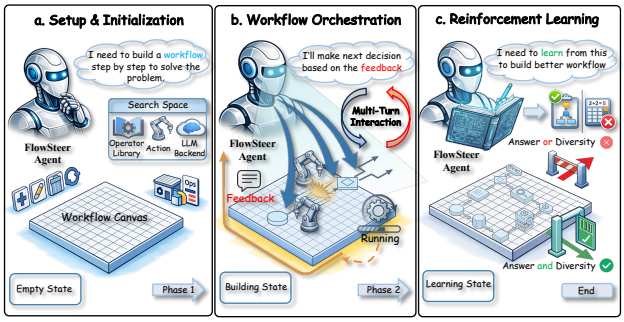


Figure 1. Overview of the FlowSteer framework pipeline. The agent first initializes with the task and explores the search space. Then, through multi-turn interaction with the canvas, it analyzes workflow states, selects editing actions, and receives execution feedback to iteratively build and refine the workflow. Finally, the agent learns from diversity-constrained rewards to continuously improve its workflow orchestration strategies across diverse tasks.

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., 2024a) to generate workflows. Third, automated workflow optimization—such as AFlow (Zhang et al., 2024a), GPTSwarm (Zhuge et al., 2024), and LATS (Zhou et al., 2023)—combines search and execution feedback to iteratively improve workflow structures.

However, these methods still face several challenges: **(i) High manual/heuristic dependence**—rules and templates require continual maintenance and are often tightly coupled to specific scenarios, limiting reuse and generaliza-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

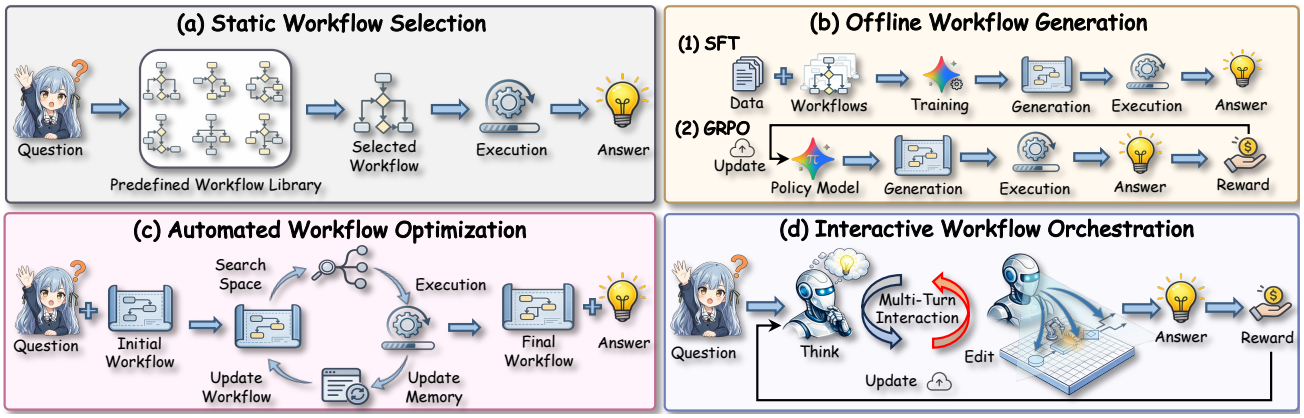


Figure 2. Comparison of different workflow orchestration paradigms: static workflow selection, offline workflow generation, automated workflow optimization, and our interactive workflow orchestration framework FlowSteer.

tion (Schick et al., 2023; Wang et al., 2024); **(ii) Operator/backend lock-in (path lock-in)**—existing approaches tend to rely on fixed operators or a single strong large language model (LLM) backend, making it difficult for a policy model to compose operators in a plug-and-play manner, so performance and robustness drop sharply when the operator library or backend environment changes (Yao et al., 2023; Zhou et al., 2024); **(iii) Sparse and unstable learning signals**—training with only terminal correctness rewards can lead to shortcut behaviors (e.g., premature termination and oversimplified graphs) and reward hacking, and may further suffer from intra-group advantage collapse, making long-horizon credit assignment unstable (Yu et al., 2025; DeepSeek-AI, 2025).

To address these challenges, we propose FlowSteer, an end-to-end reinforcement learning (RL)–enhanced framework for workflow orchestration that supports plug-and-play deployment across different operator libraries and interchangeable LLM backends. In our framework, a lightweight policy model acts as an agent that selects and revises compositions of operators within an executable Workflow Canvas environment. The canvas executes the selected nodes and returns execution traces and feedback, enabling the policy to learn transferable and diverse orchestration strategies from real execution loops. We further design a diversity-constrained reward and a conditionally released answer-based reward to jointly improve workflow structure quality and task correctness.Overall, FlowSteer provides a low-cost, transferable, training-stable, and scalable paradigm for automated orchestration of agentic workflows across diverse tasks.

We evaluate FlowSteer on three task categories: QA, mathematical reasoning, and code generation. Experimental results demonstrate that FlowSteer enhances both generation quality and reasoning accuracy through its RL-driven multi-turn workflow interaction framework, surpassing baselines and existing methods, as shown in Figure 2. Beyond strengthening the reasoning capabilities of large-scale LLM backends, FlowSteer also improves the performance of smaller-scale LLMs, while reducing token consumption and interaction turns through more efficient orchestration strategies. Furthermore, FlowSteer can adapt across diverse task types without task-specific fine-tuning, demonstrating broad adaptability and strong practical potential.

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; Shang et al., 2025; Hong et al., 2024). 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., 2025); orchestration, which uses LLM controllers for tool/model routing (Shang et al., 2025) or constrained application programming interface (API) planning to ground intent (Wang et al., 2024); and multi-agent collaboration, which relies on standard operating procedures (SOPs)/roles (Hong et al., 2024) or cross-team orchestration (Du et al., 2025). For sustained capability, agentic tool reasoning (Wu et al., 2025) and reusable workflow memory further improve efficiency (Wang et al., 2024), supporting scalable automation in practice (Qian et al., 2024). In this paper, we propose FlowSteer, an end-to-end RL framework that learns workflow orchestration from executable canvas feedback.

Reinforcement Learning for Agents. With recent progress in RL for LLM-based agents, agent RL models interaction as a long-horizon Markov decision process (MDP) (Zhou et al., 2024). In hierarchical multi-turn RL, coupling value learning with token-level policy learning eases delayed credit

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`	$\text{op}(v) \leftarrow o'$
Revision	`Revise`	revised solution	Config	`set_prompt`	$\text{prompt}(v) \leftarrow p$
Ensemble	`ScEnsemble`, `Aggregate`	voted, combined	Terminal	`finish`	$y_q = \text{Execute}(\mathcal{G}_T, q)$
Formatting	`Format`	final answer y_q	Control	`parallel`, `cond`, `loop`	branch/merge in E_t

assignment (Zhou et al., 2024). For tool-chain control, step-grained shaping (Yu et al., 2025) and outcome feedback improve tool selection and self-correction (Feng et al., 2025). When retrieval is treated as an action, search rollouts learn think-then-search behaviors (Jin et al., 2025). Moreover, large-scale RL motivates GRPO-style objectives based on verifiable or group-relative signals (DeepSeek-AI, 2025; Shao et al., 2024a).

3. Preliminaries

Definition 1: Workflow Graph. A workflow graph is a directed acyclic graph $\mathcal{G} = (V, E, \text{attr})$, 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 $\text{attr}(v) = (\text{op}(v), \text{param}(v), \text{prompt}(v))$ 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 $\mathcal{G}_\tau$:

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

where a_t^{think} denotes the reasoning reflection at step t, $a_t = (\alpha_t, a_t^{\text{out}})$ is the editing action with type $\alpha_t \in \mathcal{A}_{\text{type}}$ and content a_t^{out}, and o_t^{exec} is the execution feedback from the canvas environment. The complete definitions of operators and actions are summarized in Table 1.

Problem Statement. Given a task $q \in \mathcal{D}_Q$, an operator library $\mathcal{O}$, and a pluggable backend LLM $\mathcal{M}_{\text{backend}}$, 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:

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

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, this section introduces the FlowSteer framework, including Workflow Canvas and agent initialization (Section 4.1), workflow orchestration via multi-turn interaction (Section 4.2), and outcome-directed end-to-end reinforcement learning (Section 4.3).

4.1. Workflow Canvas and Agent Initialization

FlowSteer follows a ReAct-based agent paradigm (Yao et al., 2023), where a lightweight policy model (Flow-Director) interacts with an executable canvas environment (Workflow Canvas) to construct the workflow graph $\mathcal{G}$ (Definition 1). We define the canvas environment, operator library, action space, state space, and orchestration target.

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:

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

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, $\mathcal{M}_{\text{backend}}$ 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 $\mathcal{A}$. As summarized in Table 1, we employ 12 functional operators organized into six categories (planning, solving, verification, revision, ensemble, and formatting), along with control structures (`parallel`, `conditional`, `loop`). At each step t, the agent generates a reflection a_t^{think} that analyzes the current state, followed by an editing action $a_t = (\alpha_t, a_t^{\text{out}})$, where the action type $\alpha_t \in \mathcal{A}_{\text{type}}$ covers node insertion, removal, modification, prompt configuration, and termination. Orchestration terminates when $\alpha_t =$ `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:

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

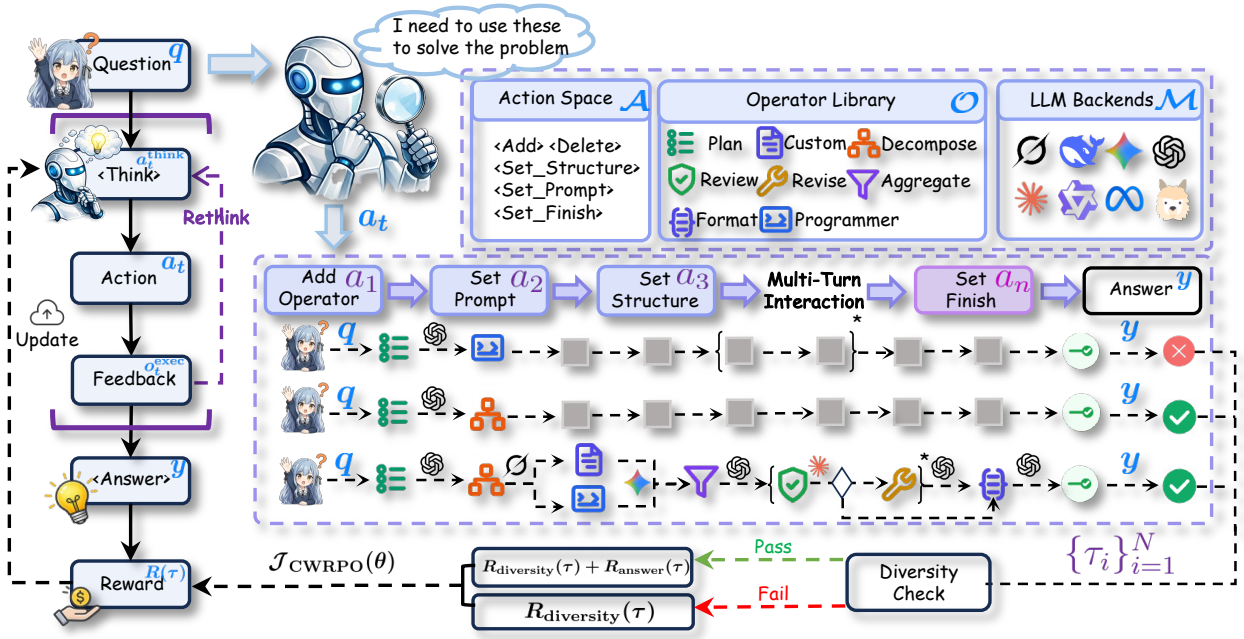


Figure 3. An overview of the FlowSteer framework. The policy model (Flow-Director) interacts with Workflow Canvas through multi-turn interactions and learns from diversity-constrained rewards via CWRPO.

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. We provide experimental results in Section 5.2 and Section 5.3, and theoretical proofs in Appendix B.1. □

4.2. Workflow Orchestration via Multi-Turn Interaction

Building on the canvas $\mathcal{C}$ and state H_0 defined above, we model workflow orchestration as a multi-turn interaction between policy π_θ (Flow-Director) and canvas $\mathcal{C}$ (Workflow Canvas) to iteratively construct the workflow. The interaction follows the prompt template shown in Table 2.

Modeling the Step-wise Orchestration Policy. Each turn submits one atomic editing action, decomposing long-horizon planning into checkable and repairable local decisions. At each step t, Flow-Director outputs reflection a_t^{think} and editing action a_t, modeled as a hierarchical policy conditioned on history H_{t-1}:

$$\pi_\theta(a_t^{\text{think}}, a_t \mid H_{t-1}) = \pi_\theta(a_t^{\text{think}} \mid H_{t-1}) \cdot \pi_\theta(\alpha_t \mid a_t^{\text{think}}, H_{t-1}) \cdot \pi_\theta(a_t^{\text{out}} \mid \alpha_t, a_t^{\text{think}}, H_{t-1}). \tag{6}$$

Canvas-Feedback-Driven Workflow Interaction. Given action a_t from Flow-Director, the Workflow Canvas executes the action and returns feedback o_t^{exec}, forming an iterative closed-loop of "diagnose-edit-verify". This interaction comprises three stages: *(i) Action Execution and Feedback Generation.* The canvas performs syntax parsing and constraint checking, then generates feedback o_t^{exec} including action success status, failure reasons, and repair suggestions:

$$o_t^{\text{exec}} \sim \mathcal{C}_{\text{exec}}(\cdot \mid \mathcal{G}_{t-1}, a_t), \tag{7}$$

where $\mathcal{C}_{\text{exec}}$ denotes the canvas feedback distribution. *(ii) State Update and History Accumulation.* The canvas updates the workflow graph (see Table 1 for update operations) and the state evolves via Eq. 5:

$$\mathcal{G}_t = \text{Update}(\mathcal{G}_{t-1}, a_t, o_t^{\text{exec}}). \tag{8}$$

(iii) Iterative Correction until Termination. Flow-Director continues decision-making based on H_t until $\alpha_t =$ `finish` or $t = T_{\max}$, enabling error discovery and repair without manual rules. **Trajectory Distribution and Optimization.** Following Definition 2, a complete trajectory τ records the interaction from initialization to termination. The trajectory distribution is jointly determined by the policy and environment:

$$P_\theta(\tau) = \prod_{t=1}^{T} \Big[\pi_\theta(a_t^{\text{think}}, a_t \mid H_{t-1}) \cdot \mathcal{C}_{\text{exec}}(o_t^{\text{exec}} \mid \mathcal{G}_{t-1}, a_t)\Big], \tag{9}$$

where only π_θ is optimized. The training objective follows Eq. 3, with $R(\tau)$ detailed in Section 4.3.

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 action). The goal is to build a reliable workflow, not a single-shot answer. Keep your thinking brief and focus on choosing the next action, not solving the whole problem yourself. Let the operators do the computation. Proceed step by step with the following rules: **<think>** (brief reasoning for the current state and next action) **</think>** **<action>** (exactly one editing action with operator/target) **</action>** After the first step, in each interaction with the canvas, write: **<think>** (your reasoning based on the **<feedback>**...**</feedback>** from canvas) **</think>** **<action>** (new action to extend or refine the workflow) **</action>** Each **<action>** must build on what came before. Let the content of the workflow evolve naturally (for example: plan → programmer → verify → format). Continue producing think within **<think>**...**</think>** and action within **<action>**...**</action>** until the workflow is complete. Once the workflow is ready, write: **<think>** (final check that Format operator is added) **</think>** **<action>finish</action>** Task: **{task}**.

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

Proposition 2. *The canvas-feedback-driven multi-turn interaction replaces manual configuration, improving orchestration efficiency and reliability.*

Proof. We provide experimental results in Section 5.5 and theoretical proofs in Appendix B.2. □

4.3. Outcome-Directed End-to-End Reinforcement Learning

To optimize policy π_θ toward generating well-structured workflows, we propose Canvas Workflow Relative Policy Optimization (CWRPO), which optimizes through multi-turn agent-canvas interactions.

CWRPO Objective $\mathcal{J}_{\text{CWRPO}}(\theta)$**.** Given task $q \in \mathcal{D}_Q$, the agent interacts with canvas $\mathcal{C}$ to generate a group of N trajectories $\{\tau_i\}_{i=1}^N$, where each $\tau_i = \{(a_t^{\text{think},(i)}, a_t^{(i)}, o_t^{\text{exec},(i)})\}_{t=1}^T$. The CWRPO objective is:

$$\mathcal{J}_{\text{CWRPO}}(\theta) = \mathbb{E}\Bigg[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{|\tau_i|_{\text{mask}}}\sum_{t=1}^{|\tau_i|}\text{mask}_t^{(i)} \cdot \min\Big(\rho_\theta^{(i,t)}\hat{A}_i, \text{clip}(\rho_\theta^{(i,t)}, 1\pm\epsilon)\hat{A}_i\Big) - \beta D_{\text{KL}}(\pi_\theta\|\pi_{\text{ref}})\Bigg], \tag{10}$$

where the importance ratio is defined as:

$$\rho_\theta^{(i,t)} = \frac{\pi_\theta(a_t^{\text{think},(i)}, a_t^{(i)} \mid H_{t-1}^{(i)})}{\pi_{\theta_{\text{old}}}(a_t^{\text{think},(i)}, a_t^{(i)} \mid H_{t-1}^{(i)})}, \tag{11}$$

which measures the ratio between current policy π_θ and behavior policy $\pi_{\theta_{\text{old}}}$ used for trajectory sampling, and the normalized advantage $\hat{A}(\tau_i) = (R(\tau_i) - \mu_{\text{src}})/(\sigma_{\text{src}} + \epsilon)$ uses within-group statistics (Shao et al., 2024a) μ_{src}, σ_{src} partitioned by data source. The mask $\text{mask}_t^{(i)} \in \{0, 1\}$ is set to 1 for policy-generated tokens and 0 for environment feedback, ensuring gradients only affect policy decisions; $|\tau_i|_{\text{mask}}$ is the masked token count. The $\text{clip}(\cdot)$ operator stabilizes updates (Schulman et al., 2017), and the KL term regularizes toward reference π_{ref} with strength β.

Outcome-directed Reward Function $R(\tau)$**.** We decompose the reward into diversity constraint $R_{\text{diversity}}(\tau)$ and answer reward $R_{\text{answer}}(\tau)$, encouraging both structural reliability and answer correctness. *(i) Diversity Constraint Reward.* This ensures workflows possess necessary structural "skeleton":

$$R_{\text{diversity}}(\tau) = \min\big(1.0, R_{\text{checker}} + R_{\text{format}} + R_{\text{operator}} + R_{\text{control}}\big), \tag{12}$$

where R_{checker} encourages verification operators, R_{format} encourages formatting operators, R_{operator} requires minimum operator count, and R_{control} encourages control structures. These components jointly ensure structural completeness for reliable workflow execution. Component weights are detailed in Appendix C.1. *(ii) Answer Reward.* This measures semantic correctness between execution output y_q (obtained via Eq. 2) and ground-truth y_q^*:

$$R_{\text{answer}}(\tau) = \text{Evaluate}(y_q, y_q^*, \text{type}(q)), \tag{13}$$

where $\text{type}(q)$ indicates the task type for metric selection, such as exact match for QA, pass rate for code generation, and accuracy for math reasoning. *(iii) Overall Outcome Reward.* The total reward employs conditional release, using diversity as prerequisite for answer reward:

$$R(\tau) = -1.0 + R_{\text{diversity}}(\tau) + \mathbb{I}\{R_{\text{diversity}}(\tau) = 1.0\} \cdot R_{\text{answer}}(\tau), \tag{14}$$

where $\mathbb{I}\{\cdot\}$ is the indicator function that implements a curriculum-like gating mechanism. When $R_{\text{diversity}} < 1.0$, the answer reward is not released, prompting the policy to first construct qualified skeletons; when $R_{\text{diversity}} = 1.0$, the answer reward is released, shifting focus to correctness and suppressing shortcut behaviors.

Proposition 3. *The diversity constraint and conditional release mechanism stabilize learning signals and suppress shortcut behaviors.*

Proof. We provide experimental results in Section 5.5 and Section 5.6, and theoretical proofs in Appendix B.3. □

		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 (Δ ↑)**
GSM8K	**Acc.**	$91.41_{\pm 0.4}$	$92.97_{\pm 0.6}$	$92.19_{\pm 0.3}$	$92.97_{\pm 0.8}$	$94.53_{\pm 0.5}$	$93.75_{\pm 0.7}$	$94.01_{\pm 0.4}$	$93.94_{\pm 0.9}$	**96.09 (+3.12)**
MATH	**Acc.**	$66.41_{\pm 0.7}$	$60.94_{\pm 0.5}$	$61.72_{\pm 0.9}$	$68.75_{\pm 0.4}$	$70.31_{\pm 0.8}$	$71.87_{\pm 0.6}$	$76.56_{\pm 0.3}$	$72.26_{\pm 0.5}$	**81.25 (+20.31)**
HotPotQA	**EM**	$67.19_{\pm 0.8}$	$63.28_{\pm 0.4}$	$70.31_{\pm 0.6}$	$59.38_{\pm 0.7}$	$68.75_{\pm 0.3}$	$67.19_{\pm 0.9}$	$72.00_{\pm 0.5}$	$67.97_{\pm 0.4}$	**78.12 (+14.84)**
	F1	$74.05_{\pm 0.3}$	$73.03_{\pm 0.8}$	$75.25_{\pm 0.5}$	$64.95_{\pm 0.6}$	$77.90_{\pm 0.7}$	$77.88_{\pm 0.4}$	$79.84_{\pm 0.6}$	$75.61_{\pm 0.8}$	**84.98 (+11.95)**
SQuAD v2	**EM**	$54.69_{\pm 0.6}$	$47.66_{\pm 0.7}$	$73.44_{\pm 0.4}$	$66.41_{\pm 0.5}$	$73.44_{\pm 0.9}$	$64.06_{\pm 0.3}$	$59.84_{\pm 0.8}$	$70.34_{\pm 0.7}$	**78.12 (+30.46)**
	F1	$61.54_{\pm 0.5}$	$59.42_{\pm 0.6}$	$77.31_{\pm 0.8}$	$72.00_{\pm 0.4}$	$82.41_{\pm 0.3}$	$72.45_{\pm 0.9}$	$65.29_{\pm 0.5}$	$75.24_{\pm 0.6}$	**83.67 (+24.25)**
MBPP	**Pass@1**	$63.28_{\pm 0.4}$	$64.84_{\pm 0.9}$	$57.03_{\pm 0.6}$	$77.34_{\pm 0.8}$	$83.20_{\pm 0.4}$	$79.69_{\pm 0.5}$	$73.43_{\pm 0.7}$	$74.22_{\pm 0.3}$	**84.38 (+19.54)**
HumanEval	**Pass@1**	$81.25_{\pm 0.8}$	$82.81_{\pm 0.3}$	$61.72_{\pm 0.7}$	$86.72_{\pm 0.6}$	$90.62_{\pm 0.5}$	$87.50_{\pm 0.4}$	$85.15_{\pm 0.9}$	$89.06_{\pm 0.5}$	**92.96 (+10.15)**
Avg.	**EM**	$60.94_{\pm 0.5}$	$55.47_{\pm 0.6}$	$71.88_{\pm 0.4}$	$62.90_{\pm 0.7}$	$71.10_{\pm 0.8}$	$65.63_{\pm 0.5}$	$65.92_{\pm 0.4}$	$69.16_{\pm 0.6}$	**78.12 (+22.65)**
	F1	$67.80_{\pm 0.4}$	$66.23_{\pm 0.5}$	$76.28_{\pm 0.7}$	$68.48_{\pm 0.8}$	$80.16_{\pm 0.6}$	$75.17_{\pm 0.4}$	$72.57_{\pm 0.5}$	$75.43_{\pm 0.7}$	**84.33 (+18.10)**
	Acc./Pass	$75.59_{\pm 0.6}$	$75.39_{\pm 0.4}$	$68.17_{\pm 0.8}$	$81.45_{\pm 0.5}$	$84.67_{\pm 0.7}$	$83.20_{\pm 0.6}$	$82.29_{\pm 0.3}$	$82.37_{\pm 0.4}$	**88.67 (+13.28)**

Table 3. Main results on reasoning and code generation benchmarks. "Baseline" uses the original model without fine-tuning. "SFT" and "GRPO" apply supervised fine-tuning and group relative policy optimization on Qwen3-8B respectively. "AFlow" uses search-based workflow generation. "Agent+RL" denotes reinforcement learning-based agent methods. All workflow methods use GPT-4o-mini as backend.

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 CWRPO compare against other RL algorithms (GRPO, DAPO)?

5.1. Experimental Setup

Datasets. We evaluate FlowSteer on six in-distribution (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 further test on six out-of-distribution (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). More details of the datasets are provided in Appendix D.

Baselines. We compare FlowSteer against the following baselines: *(i)* direct LLM baselines (Qwen3-8B (Qwen Team, 2025), GPT-4o-mini (OpenAI, 2024)), *(ii)* standard fine-tuning methods (SFT, GRPO (Shao et al., 2024a)), *(iii)* workflow-based methods (AFlow (Zhang et al., 2024a)), and *(iv)* agent with RL methods (AgentFlow (Li et al., 2025b), Router-R1 (Zhang et al., 2025), Orchestrator (Dang et al., 2025)). 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.

Implementation Details. We use Qwen3-8B (Qwen Team, 2025) as the policy model (Flow-Director), with GPT-4o-mini (OpenAI, 2024) as the default workflow execution backend. For transferability experiments, we also test GPT-OSS-120B as an alternative backend. More implementation details are illustrated in Appendix G.

5.2. Main Results (RQ1)

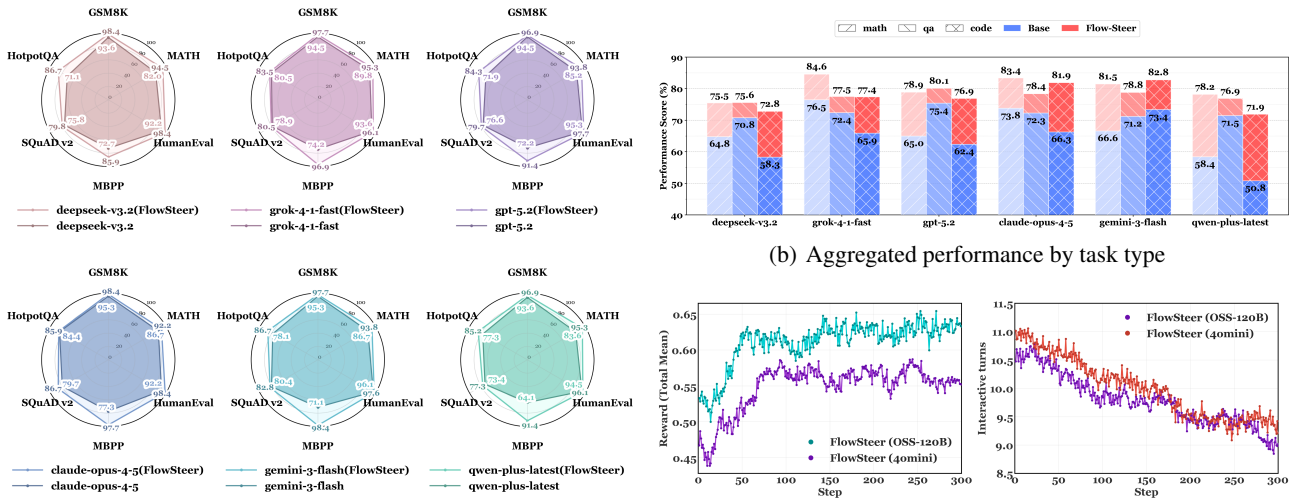
As illustrated in Table 3, FlowSteer achieves consistent improvements over the backend model (GPT-4o-mini) across six IID benchmarks and outperforms multiple baseline categories. The largest improvements are observed on mathematical reasoning datasets, while more stable performance is achieved on QA tasks compared to other learning or search-based methods. Strong Pass@1 results are also maintained on code generation benchmarks. These results confirm the broad applicability of FlowSteer and its ability to amplify backend model capabilities in complex reasoning scenarios.

5.3. Generalization Results (RQ2)

As shown in Table 4, FlowSteer maintains overall superiority over baseline methods across six OOD benchmarks, with more stable improvement trends in question answering and mathematical reasoning tasks. In comparison, directly using large-scale LLM backends typically exhibits stronger zero-shot capabilities but lacks transferable orchestration strategies, while search-based workflow methods are prone to efficiency limitations in OOD scenarios. These results demonstrate that FlowSteer achieves robust cross-task and cross-distribution improvements without task-specific fine-

		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 (Δ ↑)**
TriviaQA	**EM**	$60.16_{\pm 0.5}$	$71.09_{\pm 0.8}$	$60.94_{\pm 0.4}$	$59.38_{\pm 0.6}$	$73.44_{\pm 0.7}$	$75.00_{\pm 0.3}$	$75.78_{\pm 0.9}$	$77.36_{\pm 0.5}$	**79.69 (+8.60)**
	F1	$69.17_{\pm 0.9}$	$81.40_{\pm 0.4}$	$69.88_{\pm 0.6}$	$69.23_{\pm 0.5}$	$82.50_{\pm 0.8}$	$81.47_{\pm 0.7}$	$80.43_{\pm 0.3}$	$83.23_{\pm 0.6}$	**84.11 (+2.71)**
NaturalQuestions	**EM**	$39.84_{\pm 0.6}$	$39.84_{\pm 0.5}$	$46.09_{\pm 0.8}$	$43.75_{\pm 0.4}$	$42.97_{\pm 0.9}$	$45.70_{\pm 0.7}$	$49.22_{\pm 0.6}$	$50.00_{\pm 0.3}$	**54.69 (+14.85)**
	F1	$50.75_{\pm 0.4}$	$51.42_{\pm 0.9}$	$53.40_{\pm 0.5}$	$53.24_{\pm 0.8}$	$49.92_{\pm 0.6}$	$55.98_{\pm 0.4}$	$52.79_{\pm 0.5}$	$55.41_{\pm 0.7}$	**62.56 (+11.14)**
MathQA	**Acc.**	$75.00_{\pm 0.8}$	$79.69_{\pm 0.4}$	$61.71_{\pm 0.7}$	$60.15_{\pm 0.6}$	$83.59_{\pm 0.3}$	$82.81_{\pm 0.9}$	$80.47_{\pm 0.5}$	$82.03_{\pm 0.4}$	**88.67 (+8.98)**
AIME 2025	**Acc.**	$16.66_{\pm 0.7}$	$10.00_{\pm 0.6}$	$0.00_{\pm 0.0}$	$8.33_{\pm 0.5}$	$13.33_{\pm 0.9}$	$10.00_{\pm 0.4}$	$10.00_{\pm 0.8}$	$20.00_{\pm 0.7}$	**26.67 (+16.67)**
APPS	**Pass@1**	$39.84_{\pm 0.5}$	$40.62_{\pm 0.8}$	$26.56_{\pm 0.6}$	$34.38_{\pm 0.7}$	$42.97_{\pm 0.4}$	$41.41_{\pm 0.9}$	$42.95_{\pm 0.5}$	$44.53_{\pm 0.6}$	**49.21 (+8.59)**
DS-1000	**Pass@1**	$34.38_{\pm 0.6}$	$45.31_{\pm 0.5}$	$25.78_{\pm 0.8}$	$38.28_{\pm 0.4}$	$53.91_{\pm 0.7}$	$46.88_{\pm 0.5}$	$42.97_{\pm 0.9}$	$51.56_{\pm 0.6}$	**58.59 (+13.28)**
Avg.	**EM**	$50.00_{\pm 0.4}$	$55.47_{\pm 0.6}$	$53.52_{\pm 0.5}$	$51.57_{\pm 0.7}$	$58.21_{\pm 0.8}$	$60.35_{\pm 0.4}$	$62.50_{\pm 0.6}$	$63.68_{\pm 0.5}$	**67.19 (+11.72)**
	F1	$59.96_{\pm 0.5}$	$66.41_{\pm 0.4}$	$61.64_{\pm 0.8}$	$61.24_{\pm 0.6}$	$66.21_{\pm 0.5}$	$68.73_{\pm 0.7}$	$66.61_{\pm 0.4}$	$69.32_{\pm 0.6}$	**73.34 (+6.93)**
	Acc./Pass	$41.47_{\pm 0.7}$	$43.91_{\pm 0.5}$	$28.51_{\pm 0.6}$	$35.29_{\pm 0.9}$	$48.45_{\pm 0.4}$	$45.28_{\pm 0.8}$	$44.10_{\pm 0.5}$	$49.53_{\pm 0.7}$	**55.79 (+11.88)**

Table 4. OOD generalization results on QA, math, and code benchmarks. "Baseline" uses the original model without fine-tuning. "SFT" and "GRPO" apply supervised fine-tuning and group relative policy optimization on Qwen3-8B respectively. "AFlow" uses search-based workflow generation. "Agent+RL" denotes reinforcement learning-based agent methods. All workflow methods use GPT-4o-mini as backend.



(a) Radar charts on different backends

(b) Aggregated performance by task type

(c) Training dynamics

Figure 4. **Transferability analysis of Flow-Director across LLM backends (RQ3).** (a) Radar charts comparing six LLM backends (DeepSeek-V3.2, Grok-4.1-Fast, GPT-5.2, Claude-Opus-4.5, Gemini-3-Flash, Qwen-Plus) across six IID benchmarks, showing performance with and without Flow-Director trained on different backends. (b) Aggregated performance comparison across backends, grouped by task type (math, QA, code), comparing base LLMs vs. Flow-Director trained with 4o-mini vs. oss-120b. (c) Training dynamics showing F1 score, interaction turns, and operator counts over training steps for different backend configurations.

tuning of the backend LLM.

5.4. Transferability across LLM Backends (RQ3)

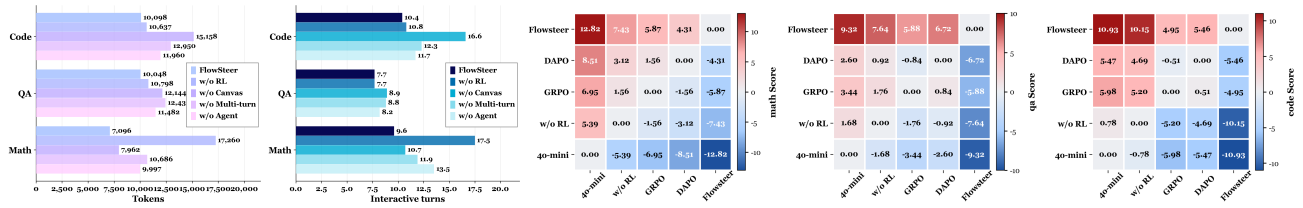
As shown in Figure 4, we evaluate the transferability of Flow-Director across six LLM backends (DeepSeek-V3.2, Grok-4.1-Fast, GPT-5.2, Claude-Opus-4.5, Gemini-3-Flash, and Qwen-Plus-Latest). The radar charts in Figure 4(a) show that FlowSteer yields consistent improvements across all backends on IID benchmarks, with weaker baselines benefiting more significantly. Figure 4(b) shows aggregated OOD performance grouped by task type, where FlowSteer maintains stable gains across math, QA, and code categories regardless of the backend model. The training dynamics in Figure 4(c) compare two training configurations: GPT-4o-mini and locally deployed GPT-OSS-120B. While OSS-120B achieves higher reward with more stable optimization, both configurations show similar convergence trends in interaction turns, indicating that zero-cost local models can effectively replace API backends for training.

5.5. Ablation Study (RQ4)

As shown in Table 5, FlowSteer (Full) performs optimally across all datasets, highlighting the synergy of multi-turn interaction, executable feedback from Workflow Canvas, and end-to-end RL optimization. Removing any component significantly reduces performance, with RL most affected by complex reasoning tasks such as math and code generation. The canvas feedback has greater impact on QA tasks that

Method	IID								OOD							
	GSM8K	MATH	HotPotQA		SQuAD v2		MBPP	HumanEval	TriviaQA		NQ		MathQA	AIME	APPS	DS-1000
	Acc.	Acc.	EM	F1	EM	F1	Pass@1	Pass@1	EM	F1	EM	F1	Acc.	Acc.	Pass@1	Pass@1
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
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**

Table 5. Ablation study of FlowSteer (GPT-4o-mini as backend) on twelve datasets, including FlowSteer (Full), without Flow-Director agent (w/o Agent), without multi-turn interaction (w/o Multi-turn), without Workflow Canvas (w/o Canvas), and without reinforcement learning (w/o RL).



(a) Token consumption (b) Interactive turns (c) Math score comparison (d) QA score comparison (e) Code score comparison

Figure 5. **Ablation and RL algorithm analysis.** (a) Token consumption comparison across task types, showing FlowSteer achieves lower token usage. (b) Average interaction turns comparison, demonstrating FlowSteer requires fewer turns to complete tasks. (c-e) Pairwise performance comparison matrices for math, QA, and code tasks respectively, where positive values (red) indicate row method outperforms column method.

require iterative error correction, showing task-dependent dependencies. In addition, FlowSteer (Full) exhibits more stable performance on the six OOD benchmarks, reflecting strong cross-task generalization ability. These results show that the effectiveness of FlowSteer stems from the close coupling of agent decision-making, canvas interaction, and RL optimization.

As shown in Figure 5(a-b), we further analyze orchestration efficiency in terms of average interaction turns and token consumption. The results show that FlowSteer achieves lower turns and fewer tokens across all task types compared to ablation variants. Removing any component causes the policy to produce redundant interactions and fail to determine proper termination timing, leading to higher time and computational costs. Through the joint effect of all components, the policy learns efficient orchestration strategies and when to stop, reducing both interaction turns and token consumption while maintaining task accuracy.

5.6. Comparison of RL Algorithms (RQ5)

As shown in Table 6, we compare three RL algorithms (DAPO, GRPO, and CWRPO) on six IID benchmarks under identical training settings. CWRPO achieves optimal results across all task types with more stable training dynamics. The pairwise comparison matrices in Figure 5(c–e) show consistent superiority of CWRPO across math, QA, and code tasks. Unlike DAPO and GRPO that primarily focus on answer correctness, CWRPO jointly optimizes structural diversity and task performance through diversity-constrained rewards with conditional release. This design enables the policy to learn not only how to solve problems but also how to orchestrate workflows effectively, achieving better trade-off between workflow quality and answer accuracy.

Method	GSM8K	MATH	HotPotQA		SQuAD v2		MBPP	HumanEval
	Acc.	Acc.	EM	F1	EM	F1	Pass@1	Pass@1
4o-mini	92.97	60.94	63.28	73.03	47.66	59.42	64.84	82.81
w/o RL	91.41	71.81	72.66	82.19	59.38	67.66	64.06	89.06
GRPO	92.97	73.43	72.66	81.80	61.72	68.91	78.91	89.84
DAPO	93.75	74.22	73.44	82.42	61.72	70.08	81.25	89.06
CWRPO	**96.09**	**81.25**	**78.12**	**84.98**	**78.12**	**83.67**	**84.38**	**92.96**

Table 6. Comparison of RL algorithms on six IID benchmarks using GPT-4o-mini as backend. Metrics follow each dataset's standard (EM/F1 for QA, Acc. for math, Pass@1 for code).

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. We design diversity-constrained rewards with conditional release to jointly optimize workflow structure and task correctness. Experiments on twelve benchmarks demonstrate that FlowSteer significantly outperforms baselines across mathematical reasoning, QA, and code generation tasks.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Amini, A., Gabriel, S., Lin, S., Koncel-Kedziorski, R., Choi, Y., and Hajishirzi, H. 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*, 2019.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Dang, Y., Qian, C., Luo, X., Fan, J., Xie, Z., Shi, R., Chen, W., Yang, C., Che, X., Tian, Y., Xiong, X., Han, L., Liu, Z., and Sun, M. Multi-agent collaboration via evolving orchestration. In *Advances in Neural Information Processing Systems*, 2025.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Du, Z., Qian, C., Liu, W., Xie, Z., Wang, Y., Qiu, R., Dang, Y., Chen, W., Yang, C., Tian, Y., Xiong, X., and Han, L. Multi-agent collaboration via cross-team orchestration. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10386–10406. Association for Computational Linguistics, 2025.

Erdogan, L. E., Furuta, H., Kim, S., Lee, N., Moon, S., Anumanchipalli, G., Keutzer, K., and Gholami, A. Plan-and-act: Improving planning of agents for long-horizon tasks. In *International Conference on Machine Learning*, 2025.

Feng, J., Huang, S., Qu, X., Zhang, G., Qin, Y., Zhong, B., Jiang, C., Chi, J., and Zhong, W. ReTool: Reinforcement learning for strategic tool use in LLMs. *arXiv preprint arXiv:2504.11536*, 2025.

Google DeepMind. Gemini 2.5: A family of highly capable multimodal models. *arXiv preprint arXiv:2503.00000*, 2025.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with APPS. *Advances in Neural Information Processing Systems*, 34, 2021a.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset. *Advances in Neural Information Processing Systems*, 34, 2021b.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Jin, B., Zeng, H., Yue, Z., Yoon, J., Arik, S., Wang, D., Zamani, H., and Han, J. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017.

Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., Toutanova, K., Jones, L., Kelcey, M., Chang, M.-W., Dai, A. M., Uszkoreit, J., Le, Q., and Petrov, S. Natural questions: A benchmark for question

answering research. *Transactions of the Association for Computational Linguistics*, 7, 2019.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., tau Yih, S. W., Fried, D., Wang, S., and Yu, T. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, 2023.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Li, Z., Hu, Y., and Wang, W. Encouraging good processes without the need for good answers: Reinforcement learning for LLM agent planning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1654–1666. Association for Computational Linguistics, 2025a.

Li, Z., Zhang, H., Han, S., Liu, S., Xie, J., Zhang, Y., Choi, Y., Zou, J., and Lu, P. In-the-flow agentic system optimization for effective planning and tool use. *arXiv preprint arXiv:2510.05592*, 2025b.

Meta AI. The LLaMA 4 herd of models. *arXiv preprint arXiv:2504.00000*, 2025.

OpenAI. GPT-4o system card. *OpenAI Technical Report*, 2024.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, 2022.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., and Sun, M. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 2018.

Ruan, J., Chen, Y., Zhang, B., Xu, Z., Bao, T., Du, G., Shi, S., Mao, H., Li, Z., Zeng, X., and Zhao, R. TPTU: Large language model-based AI agents for task planning and tool usage. *arXiv preprint arXiv:2308.03427*, 2023.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Shang, Y., Li, Y., Zhao, K., Ma, L., Liu, J., Xu, F., and Li, Y. AgentSquare: Automatic LLM agent search in modular design space. In *International Conference on Learning Representations*, 2025.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., and Guo, D. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024a.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y., Wu, Y., and Guo, D. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024b.

Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. *Advances in Neural Information Processing Systems*, 36, 2023.

Wang, Z. Z., Mao, J., Fried, D., and Neubig, G. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Wu, J., Zhu, J., Liu, Y., Xu, M., and Jin, Y. 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*, pp. 28489–28503. Association for Computational Linguistics, 2025.

Yang, K., Liu, Y., Chaudhary, S., Fakoor, R., Chaudhari, P. A., Karypis, G., and Rangwala, H. AgentOccam: A simple yet strong baseline for LLM-based web agents. In *International Conference on Learning Representations*, 2025.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhutdinov, R., and Manning, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Yu, Y., Wang, Z., Ma, W., Wang, S., Wu, C., Guo, Z., and Zhang, M. 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*, 2025.

Zeng, Z., Watson, W., Cho, N., Rahimi, S., Reynolds, S., Balch, T., and Veloso, M. FlowMind: Automatic workflow generation with LLMs. In *Proceedings of the 4th ACM International Conference on AI in Finance*, 2023.

Zhang, H., Feng, T., and You, J. Router-R1: Teaching LLMs multi-round routing and aggregation via reinforcement learning. *arXiv preprint arXiv:2506.09033*, 2025.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. AFlow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024a.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. AFlow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024b.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning acting and planning in language models. *Advances in Neural Information Processing Systems*, 36, 2023.

Zhou, Y., Zanette, A., Pan, J., Levine, S., and Kumar, A. ArCHer: Training language model agents via hierarchical multi-turn RL. *arXiv preprint arXiv:2402.19446*, 2024.

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., and Schmidhuber, J. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 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 7. 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 Flow-Steer

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 7, the operator library $\mathcal{O}$ consists of 12 functional operators organized into 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.

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

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.

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 8. 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 8. Action types in Flow-Steer.

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

$$\tau = \{(a_t^{\text{think}}, a_t, o_t^{\text{exec}})\}_{t=1}^{T}, \tag{15}$$

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:

$$\begin{aligned}\mathcal{O} = \{&\texttt{Plan}, \texttt{Decompose}, \texttt{Programmer}, \texttt{Custom},\\ &\texttt{AnswerGenerate}, \texttt{Test}, \texttt{Review}, \texttt{Verify},\\ &\texttt{Revise}, \texttt{ScEnsemble}, \texttt{Aggregate}, \texttt{Format}\},\end{aligned} \tag{16}$$

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

$$\begin{aligned}\mathcal{A}_{\text{type}} = \{&\texttt{add}, \texttt{delete}, \texttt{modify}, \texttt{set\_prompt},\\ &\texttt{finish}, \texttt{parallel}, \texttt{conditional}, \texttt{loop}\},\end{aligned} \tag{17}$$

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 4 (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

$$\mathcal{C} = \{\mathbf{P}, \mathbf{D}, \mathbf{E}, \mathbf{M}, \mathbf{R}, \mathbf{I}, \mathbf{O}\}, \tag{18}$$

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; **E** (Execution) performs solving, reasoning, or externalized computation on a subgoal, including symbolic computation, code generation and execution, and other operations; **M** (Monitoring) verifies, reviews, unit tests, and checks constraints on intermediate products, determining whether to continue, backtrack, or redo; **R** (Revision) performs repairs based on monitoring results, including rewriting, supplementing, replacing, and adjusting prompts or parameters; **I** (Integration) fuses, votes, disambiguates, and ensures consistency of results from multiple branches or subproblems; and **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

$$\psi(\texttt{Plan}) = \mathbf{P}. \tag{19}$$

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

$$\psi(\texttt{Decompose}) = \mathbf{D}. \tag{20}$$

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

$$\begin{aligned}\psi(\texttt{Custom}) &= \psi(\texttt{Programmer}) \\ &= \psi(\texttt{AnswerGenerate}) = \mathbf{E}.\end{aligned} \tag{21}$$

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

$$\psi(\texttt{Verify}) = \psi(\texttt{Review}) = \psi(\texttt{Test}) = \mathbf{M}. \tag{22}$$

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

$$\psi(\texttt{Revise}) = \mathbf{R}. \tag{23}$$

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

$$\psi(\texttt{Aggregate}) = \psi(\texttt{ScEnsemble}) = \mathbf{I}. \tag{24}$$

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

$$\psi(\texttt{Format}) = \mathbf{O}. \tag{25}$$

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 back-edges or 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 5 (Monotonic Improvement of Multi-turn Interaction)**.** *Under Assumptions 2–3, multi-turn orchestration based on canvas feedback 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 characterize the value of multi-turn interaction 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.

(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

$$H_t = \{(a_1^{\text{think}}, a_1, o_1^{\text{exec}}), \ldots, (a_t^{\text{think}}, a_t, o_t^{\text{exec}})\}, \quad (26)$$

where H_t encodes all reflection texts, edit actions, and canvas feedback up to turn t. The initial history is $H_0 = \emptyset$. Given history H_t, we define the posterior distribution over Z as

$$\pi_t(z) \triangleq \mathbb{P}(Z = z \mid H_t), \quad (27)$$

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:

$$A(H_t) \triangleq \max_z \pi_t(z), \quad (28)$$

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

$$V(H_t) \triangleq 1 - A(H_t) = 1 - \max_z \pi_t(z), \tag{29}$$

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 $t \geq 1$,

$$\mathbb{E}[V(H_t) \mid H_{t-1}] \leq V(H_{t-1}), \tag{30}$$

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

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

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:

$$\phi(p) \triangleq 1 - \max_z p_z, \quad p \in \Delta_{|Z|-1}, \tag{32}$$

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:

$$\mathbb{E}[V(\pi_t) \mid \mathcal{F}_{t-1}] \leq V(\mathbb{E}[\pi_t \mid \mathcal{F}_{t-1}]) = V(\pi_{t-1}), \tag{33}$$

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

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

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

$$\mathbb{E}[V(H_t) \mid H_{t-1}] < V(H_{t-1}) \tag{35}$$

with positive probability, leading to strict improvement.

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

$$\mathbb{E}[V(H_t)] \leq \mathbb{E}[V(H_{t-1})] \leq \cdots \leq \mathbb{E}[V(H_0)]. \tag{36}$$

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

$$\Delta_t \triangleq \mathbb{E}[V(H_{t-1})] - \mathbb{E}[V(H_t)] \geq 0, \tag{37}$$

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:

$$\mathbb{E}[V(H_t)] = \mathbb{E}[V(H_0)] - \sum_{s=1}^{t} \Delta_s, \tag{38}$$

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

$$\mathbb{E}[A(H_t)] = 1 - \mathbb{E}[V(H_0)] + \sum_{s=1}^{t} \Delta_s, \tag{39}$$

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

$$\mathbb{P}(\hat{z}_t \neq Z \mid H_t) = 1 - \max_z \pi_t(z) = V(H_t). \tag{40}$$

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:

$$\max_{\pi \in \Pi_{\text{multi-turn}}} \mathbb{E}[A(H_T)] \geq \max_{\pi \in \Pi_{\text{single-turn}}} \mathbb{E}[A(H_1)], \tag{41}$$

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. □

B.3. Proof of Proposition 3

Proposition 6 (Structural Constraints, Conditional Release, and Mask Effectiveness)**.** *Let the trajectory-level reward be defined as*

$$\begin{aligned} R(\tau) = -1 &+ R_{\text{diversity}}(\tau) \\ &+ \mathbb{I}\{R_{\text{diversity}}(\tau) = 1\} \cdot R_{\text{answer}}(\tau), \end{aligned} \tag{42}$$

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 CWRPO, 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

$$\mathcal{F} = \{\tau : R_{\text{diversity}}(\tau) = 1\}, \tag{43}$$

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

$$\mathcal{F}^c = \{\tau : R_{\text{diversity}}(\tau) < 1\}, \tag{44}$$

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:

$$\begin{aligned} R(\tau) &= -1 + R_{\text{diversity}}(\tau) + 0 \cdot R_{\text{answer}}(\tau) \\ &= -1 + R_{\text{diversity}}(\tau). \end{aligned} \tag{45}$$

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

$$R(\tau) \in [-1, 0) \quad \text{for all } \tau \in \mathcal{F}^c. \tag{46}$$

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:

$$R(\tau) = -1 + 1 + 1 \cdot R_{\text{answer}}(\tau) = R_{\text{answer}}(\tau). \tag{47}$$

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

$$R(\tau) \geq 0 \quad \text{for all } \tau \in \mathcal{F}. \tag{48}$$

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). The CWRPO objective can be written as

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\right] - \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}), \tag{49}$$

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

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(\tau)\Big], \tag{50}$$

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:

$$\hat{A}(\tau) = \frac{R(\tau) - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \epsilon}, \tag{51}$$

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

$$\mathbb{E}[\hat{A}(\tau) \mid \tau \in \mathcal{F}] > 0 > \mathbb{E}[\hat{A}(\tau) \mid \tau \in \mathcal{F}^c]. \tag{52}$$

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:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta), \tag{53}$$

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 $p_\theta < 1$:

$$\frac{dp_\theta}{d\theta} \cdot \nabla_\theta J(\theta) > 0 \quad \text{when } 0 < p_\theta < 1. \tag{54}$$

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, CWRPO 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

$$\rho_\theta(\tau) = \frac{\pi_\theta(\tau)}{\pi_{\text{old}}(\tau)} = \prod_{t=1}^{T} \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{old}}(a_t \mid s_t)}, \tag{55}$$

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

$$L^{\text{clip}}(\theta) = \mathbb{E}_{\tau\sim\pi_{\text{old}}}\Big[\min\big(\rho_\theta(\tau)\hat{A}(\tau), \text{clip}(\rho_\theta, 1-\epsilon, 1+\epsilon)\hat{A}(\tau)\big)\Big]. \tag{56}$$

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:

$$D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \log \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{ref}}(a_t \mid s_t)}\right]. \tag{57}$$

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 bound the per-step policy change:

$$\|\theta_{k+1} - \theta_k\| \leq \frac{\alpha \cdot \epsilon_{\text{clip}} \cdot \max_\tau |A(\tau)|}{\beta + \lambda_{\min}(\nabla^2_\theta D_{\text{KL}})}, \tag{58}$$

where $\lambda_{\min}$ denotes the minimum eigenvalue of the KL Hessian. This bounded update magnitude prevents training oscillations and ensures 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 $\mathcal{T}_{\text{env}}$ for environment feedback tokens. The joint log-likelihood of the trajectory can be decomposed as

$$\begin{aligned}\log p_\theta(w_{1:|\tau|}) = &\sum_{t \in \mathcal{T}_\pi} \log \pi_\theta(w_t \mid w_{<t}) \\ &+ \sum_{t \in \mathcal{T}_{\text{env}}} \log p_{\text{env}}(w_t \mid w_{<t}),\end{aligned} \tag{59}$$

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 $\nabla_\theta \log p_{\text{env}}(w_t \mid w_{<t}) = 0$:

$$\nabla_\theta \log p_\theta(w_{1:|\tau|}) = \sum_{t \in \mathcal{T}_\pi} \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}). \tag{60}$$

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

$$\nabla_\theta \log p_\theta(w_{1:|\tau|}) = \sum_{t=1}^{|\tau|} \text{mask}_t \cdot \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}). \tag{61}$$

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

$$\mathbb{E}\left[\sum_{t=1}^{|\tau|} \text{mask}_t \cdot \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}) \cdot R(\tau)\right] = \nabla_\theta J(\theta). \tag{62}$$

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

$$\text{Var}_{\text{no-mask}} = \text{Var}\left[\sum_{t=1}^{|\tau|} \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}) \cdot R(\tau)\right]. \tag{63}$$

With the mask, the variance is

$$\text{Var}_{\text{mask}} = \text{Var}\left[\sum_{t \in \mathcal{T}_\pi} \nabla_\theta \log \pi_\theta(w_t \mid w_{<t}) \cdot R(\tau)\right]. \tag{64}$$

Since $|\mathcal{T}_\pi| < |\tau|$ and the environment tokens contribute noise unrelated to θ, we have $\text{Var}_{\text{mask}} < \text{Var}_{\text{no-mask}}$. 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. Flow-Steer Algorithm Details

Flow-Steer 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, action generation, and answer output, while the Workflow Canvas provides structural feedback and executes the constructed workflow. 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

Algorithm 1 Flow-Steer: 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
Final think & answer: $(a_T^{\text{think}}, y) \leftarrow \pi_\theta(p_{\text{sys}}, q, H_T)$
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
CWRPO-based update policy:
$\mathcal{J}_{\text{CWRPO}} \propto \sum_{i=1}^{N} \sum_{t=1}^{|\tau_i|} \textsc{Mask}_t^{(i)} \cdot \min\left(\rho_\theta(w_t^{(i)}), \textsc{clip}(\rho_\theta(w_t^{(i)}), 1 \pm \epsilon)\right) \hat{A}(\tau_i)$
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: $\theta \leftarrow \theta - \eta \nabla_\theta(-\mathcal{J}_{\text{CWRPO}})$
return y

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 an answer in a sequential manner, 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 Flow-Steer 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 Flow-Steer 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 Component Weights

The diversity constraint reward $R_{\text{diversity}}(\tau)$ is composed of four binary checks, each contributing equally to the total score (capped at 1.0):

- R_{checker} **(0.25):** Encourages the inclusion of verification operators (`Test`, `Review`, `Verify`). Set to 0.25 if at least one verification operator is present in the workflow.
- R_{format} **(0.25):** Encourages proper answer formatting. Set to 0.25 if the `Format` operator is included as the final step before termination.
- R_{operator} **(0.25):** Requires a minimum operator count for structural diversity. Set to 0.25 if the workflow contains at least 3 distinct operators.
- R_{control} **(0.25):** Encourages control flow structures. Set to 0.25 if the workflow includes at least one control structure (`parallel`, `conditional`, or `loop`).

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 Flow-Steer. 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 subdisciplines. 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 Program-

ming 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 (full set), and 164 from HumanEval (full set), resulting in a total of 10,778 training instances. To evaluate the generalization performance of Flow-Steer, 128 instances were randomly sampled from each of the six out-of-distribution and six trained datasets for testing, except for AIME 2025 which contains 30 problems.

E. Baseline Details

To accurately evaluate the performance of Flow-Steer, 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** (Qwen Team, 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** (OpenAI, 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

• **SFT (Supervised Fine-Tuning)** (Ouyang et al., 2022): A supervised fine-tuning baseline built on Qwen3-8B, trained with workflow annotation data to improve instruction following and DSL generation accuracy. Unlike Flow-Steer's multi-turn interaction paradigm, SFT generates the complete workflow in a single turn without execution feedback. We use LoRA fine-tuning with rank 8 for parameter efficiency, evaluating how standard supervised adaptation enhances the raw backbone's workflow construction capabilities.

• **GRPO (Group Relative Policy Optimization)** (Shao et al., 2024b): Group Relative Policy Optimization 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 Flow-Steer'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., 2024b): 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 PPO-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 GRPO 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 based on partial feedback,

Table 9. 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	✗	✓	✗
Flow-Steer (Ours)	✓	✓	✓	✓	✓	✓

Table 10. Baseline implementation details.

Method	Base Model	Training	Key Hyperparameters
Qwen3-8B	Qwen3-8B-Instruct	None	temperature=0, max_tokens=512
GPT-4o-mini	GPT-4o-mini	None	temperature=0, top_p=1, max_tokens=512
SFT	Qwen3-8B	LoRA	temperature=0, r=16, α=16, LR=1×10^{-4}, epochs=1, batch=16
GRPO	Qwen3-8B	GRPO + LoRA	temperature=0, r=16, α=32, LR=1×10^{-6}, G=4
AFlow	GPT-4o-mini	MCTS	temperature=0, search_rounds=21
AgentFlow	Qwen2.5-7B-Instruct	GRPO	temperature=0, LR=1×10^{-6}, rollout_n=8, epochs=5
Router-R1	Qwen3-8B	PPO	temperature=0, LR=1×10^{-6}, clip=0.2, epochs=30
Orchestrator	Qwen3-8B	GRPO/PPO	temperature=0, batch=256, max_turns=5

limiting its performance on complex multi-step reasoning tasks.

• **Orchestrator**: An orchestrator-style architecture that sequentially selects operators from a library using PPO. 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 Flow-Steer employs for fine-grained control over operator configuration. This limitation prevents precise customization of individual operator behaviors, reducing the overall workflow quality.

Table 9 summarizes the key architectural differences between Flow-Steer and the baselines, and Table 10 provides the detailed implementation configurations for each method.

For a fair comparison, we conducted three independent runs under identical settings for both Flow-Steer and all baselines and reported the averaged results. For all baselines, we use the same evaluation protocol: 128 test samples per dataset (except AIME 2025 with 30 samples), single generation per sample, 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** (DeepSeek-AI, 2024): DeepSeek-V3 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-Fast**: Grok-4-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.

• **LLaMA-4-Maverick** (Meta AI, 2025): LLaMA-4-Maverick leverages the latest multimodal learning techniques, particularly excelling in tasks that require integrating diverse information sources. Its open-source nature allows for detailed analysis of workflow execution patterns.

• **Qwen-Plus**: Qwen-Plus 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.

• **GPT-5**: GPT-5 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.

• **Gemini-2.5-Flash** (Google DeepMind, 2025): Gemini-2.5-Flash combines a fast inference engine with multimodal understanding capabilities, optimized for real-time interac-

tive applications. Its unique architecture provides a diverse test case for backend generalization.

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:

$$\text{EM} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\text{normalize}(y_i) = \text{normalize}(y_i^*)\right), \quad (65)$$

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:

$$\text{Precision} = \frac{|y \cap y^*|}{|y|}, \quad (66)$$

$$\text{Recall} = \frac{|y \cap y^*|}{|y^*|}, \quad (67)$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \quad (68)$$

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:

$$\text{Acc} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(|y_i - y_i^*| < \epsilon\right), \quad (69)$$

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:

$$\text{Acc}_{\text{symbolic}} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\text{simplify}(y_i - y_i^*) = 0\right). \quad (70)$$

Applicable datasets: GSM8K, MATH, MathQA, AIME 2024

F.4. Pass@k for Code Generation

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

$$\text{Pass@}k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \quad (71)$$

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:

$$\text{Pass@1} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\text{execute}(y_i, \text{tests}_i) = \text{pass}\right), \quad (72)$$

where execute(y_i, 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 Flow-Steer in Table 11. The table covers all aspects of our training pipeline, including model configuration, training hyperparameters, CWRPO algorithm 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.

CWRPO Configuration. For the CWRPO algorithm, we use GRPO as the advantage estimator with a clip range

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
CWRPO Algorithm	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	5
	Require Checker/Structure	True / True
Reward	Base Reward	-1.0
	Structure Reward Cap	1.0
	Structure Components	R_{chk}=0.2, R_{fmt}=0.2, R_{op}=0.2, R_{ctrl}=0.4
	Correctness Activation	Gate (structure = 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 11. Complete hyperparameter settings for FlowSteer training.

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 several constraints: minimum 5 operators before allowing the `finish` action, requiring at least one checker operator (Verify/Test/Review), and requiring at least one control structure (parallel/conditional/loop).

Reward Function. The diversity-constrained reward follows the formulation in Section 4.3. The base reward is -1.0. The structure reward consists of four components: checker score ($R_{\text{checker}} = 0.2$), format score ($R_{\text{format}} = 0.2$), operator score ($R_{\text{operator}} = 0.2$), and control structure score ($R_{\text{control}} = 0.4$). The structure reward is capped at 1.0, and the answer reward is only released when the structure 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. The total training time for 300 steps is approximately 8 hours. We use mixed precision training with bfloat16 to reduce memory footprint while maintaining numerical stability.

H. 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.

H.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)

H.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.

H.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)

H.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)

Round 3-6: Parallel Custom Operators [Custom, Custom, Custom]

Round 3 - Think:
The **Decompose** operator created 3 independent sub-problems. The feedback recommends adding 3 solvers in parallel to handle each sub-problem concurrently.
Decision: Add parallel structure with **[Custom, Custom, Custom]** to solve each sub-problem independently with different perspectives.
Round 3 - Action:
<action>add</action><structure>parallel</structure><operators>Custom,Custom,Custom</operators>
Round 3 - Feedback: [Status]: Pending - Awaiting Prompts for 3 parallel branches

---

Round 4-6 - Parallel Branch Prompts:

- **Branch 1 (node_2_p0):** *Focus on identifying the city/location in Prestatyn, Wales. Search passages for geographic references.*
- **Branch 2 (node_2_p1):** *Focus on identifying UK holiday park companies. List all company names mentioned in the passages.*
- **Branch 3 (node_2_p2):** *Cross-reference findings to identify the shared name between location and company.*

Parallel Execution Outputs:

Branch 1 Output: The Hard Rock Hell passage mentions "Pontin's Holiday Village in Prestatyn, Wales." The location reference is **Pontins**.

Branch 2 Output: Companies mentioned: Park Holidays UK (28 parks), Park Resorts (48 parks), **Pontins** (Britannia Jinky Jersey Ltd).

Branch 3 Output: Cross-referencing: "Pontins" appears both as a location in Prestatyn and as a company name. Answer: **Pontins**

Feedback: [Output]: All 3 branches converge on "Pontins" | [NEXT]: ADD:Aggregate (parallel structure needs aggregation)

Workflow State: Decompose → [Custom, Custom, Custom] (parallel executed)

Round 7-8: Verify Operator

Round 7 - Think:
All three parallel branches converged on "Pontins" as the answer. Before aggregating, I should verify this answer against the original passages to ensure correctness.
Decision: Add Verify operator to independently validate that "Pontins" satisfies both conditions: (1) associated with Prestatyn, Wales, and (2) is a UK holiday park company.

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

Round 7 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: Decompose → [Custom, Custom, Custom]

Round 8 - Input Prompt:
The parallel Custom operators proposed "Pontins" as the answer. Verify this against the passages:

1. *Confirm Pontins is mentioned in connection with Prestatyn, Wales*
2. *Confirm Pontins operates holiday parks in the UK*
3. *Verify no other name satisfies both conditions*

Operator Execution Output:

[Step 1] Location Verification:
The Hard Rock Hell passage states: "the festival was held at Pontin's Holiday Village in **Prestatyn, Wales**." ✓ Confirmed

[Step 2] Company Verification:
The Pontins passage states: "Pontins is the trading name of Britannia Jinky Jersey Ltd, a company **operating holiday parks in the United Kingdom**." ✓ Confirmed

[Step 3] Uniqueness Check:
Other companies (Park Holidays UK, Park Resorts) are not associated with Prestatyn. ✓ Pontins is unique

Verification Result:
`is_correct`: **True** | `confidence`: **HIGH** | `suggested_answer`: **Pontins**

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

Workflow State: Decompose → [Custom, Custom, Custom] → Verify (verified)

Round 9-10: Aggregate Operator

Round 9 - Think:
The **Verify** operator confirmed "Pontins" with HIGH confidence. The parallel structure produced consistent results across all branches.

Now I need to aggregate the parallel outputs into a single, unified answer. This is essential for parallel workflows to consolidate multiple reasoning paths.
Decision: Add Aggregate operator to combine the verified parallel outputs into the final answer.
Round 9 - Action: <action>add</action><operator>Aggregate</operator>
Round 9 - Feedback: [Status]: Pending - Awaiting Prompt | [Current DSL]: Decompose → [Custom, Custom, Custom] → Verify

---

Round 10 - Input Prompt:
Aggregate the outputs from the parallel Custom operators and the Verify result. All branches produced "Pontins" with high confidence. Consolidate into a single final answer.
Operator Execution Output:

Aggregation Summary:

Source	Answer
Branch 1 (Location)	Pontins
Branch 2 (Companies)	Pontins
Branch 3 (Cross-ref)	Pontins
Verify	Pontins (confirmed)

Consensus: All 4 sources agree on **Pontins**.
Aggregated Output:
`output`: **Pontins** | `is_correct`: **True** | `confidence`: **HIGH**

Feedback: [Output]: **Pontins** | [NEXT]: FINISH (Answer ready)
Workflow State: 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.

H.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)

H.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.

H.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)

H.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.

I. Limitations

Despite its strong empirical performance across various reasoning tasks, Flow-Steer 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 (with our 16,384 token limit affecting approximately 8% of complex tasks), it can lead to substantial information loss or suboptimal workflow decisions, thereby limiting the framework's practical flexibility and effectiveness.

J. Future Work

To further enhance the overall performance and robustness of Flow-Steer, future work should focus on comprehensively improving both scalability and efficiency. Specifically, optimizing the multi-turn workflow interaction process through techniques such as context compression or selective summarization of completed operator outputs could significantly reduce computational overhead while improving inference speed. Additionally, refining the reinforcement learning component, particularly the design of the progressive reward mechanism including process reward models for step-level supervision, can further boost learning efficiency and dynamic adaptability. To expand applicability across diverse domains, incorporating domain adaptation and transfer learning strategies could strengthen Flow-Steer's ability to handle heterogeneous cross-domain tasks and generalize effectively to unseen scenarios with minimal fine-tuning. Addressing long-range dependencies in multi-turn reasoning may be achieved by exploring advanced memory mechanisms and hierarchical planning structures, ensuring contextual coherence over extended interaction histories. Lastly, automatic operator discovery through synthesizing new operators from task requirements and composing existing operators into higher-level abstractions would enhance the framework's adaptability to novel task categories.

K. Applicability Analysis

Flow-Steer, with its advanced multi-turn workflow orchestration and dynamic canvas updating capabilities, demonstrates significant potential for application in highly knowledge-intensive domains that require rigorous logical deduction. Particularly in critical fields such as law, healthcare, and finance, Flow-Steer can leverage powerful large language models through its pluggable backend architecture to handle increasingly complex reasoning tasks efficiently while ensuring data privacy through local deployment options in resource-constrained environments. Moreover, by integrating reinforcement learning techniques via CWRPO, Flow-Steer is not only capable of handling traditional supervised tasks but also adapts seamlessly to complex dynamic environments by continuously optimizing its workflow orchestration strategies via intrinsic verification and refinement operators, thus greatly enhancing the system's adaptive intelligence and long-term planning proficiency. Overall, Flow-Steer provides strong support for trustworthy, transparent, and intelligent decision-making in knowledge-intensive fields, offering promising applications in a wide range of challenging real-world domains with its robust reasoning capabilities, interpretable workflow structures, and cross-backend adaptability.