CORE-Flow: Counterfactual Observation and Retrieval-Guided Exploration for Efficient Agentic Workflow Evolution

Anonymous authors
Paper under double-blind review

Abstract

Automated workflow design enables adaptive solving of complex tasks through fine-grained composition of capability modules and multi-round iterative optimization. However, efficiently searching the large space of workflow structures, operators, and prompts remains difficult. Existing methods typically rely on an optimizer model to diagnose the deficiencies of candidate workflows, thereby improving search efficiency. However, such diagnosis introduces uncertainty into the update direction due to the stochasticity of model outputs, while overlooking reusable historical evidence. Therefore, we propose CORE-Flow, an experience-driven framework for efficient workflow evolution through evidence reuse. CORE-Flow uses counterfactual observations of workflow edits to construct reusable optimization experience and retrieval-guided exploration to focus subsequent search on historically supported directions. Across six benchmarks spanning mathematical reasoning, code generation, and question answering, CORE-Flow achieves the best average score of 89.41, outperforming JUDGEFLOW and AFLOW by 2.92 and 7.89 points, respectively. The cost analysis tracked on the MATH dataset further demonstrates strong potential to reduce both token consumption and wall-clock execution time.

1 Introduction

Background. Recent advances in large language models (LLMs) have driven growing interest in agentic workflows (Yao et al., 2023b; Hong et al., 2024). By composing multiple model calls, prompt templates, and control logic, workflows can substantially improve complex problem solving, including mathematical reasoning, code generation, and multi-hop question answering (Wei et al., 2022; Wang et al., 2022; Yao et al., 2023b; Madaan et al., 2023). However, conventional workflow construction still relies heavily on expert knowledge, task-specific debugging, and manual trial-and-error, which limits the rapid transfer of LLM systems to new tasks Ma et al. (2026). Recent work has therefore turned to automated workflow generation (Zhang et al., 2025b). A common formulation treats workflow optimization as a search under the space of model-invocation nodes and code-defined edges, and iteratively refines candidate workflows using execution feedback (Zhao et al., 2025). While this paradigm offers a standardized pathway for automated workflow design, it also creates a challenge for efficient exploration and optimization because the search space contains many code structures and node configurations.

Shortcomings. To improve search efficiency, mainstream methods commonly place an LLM-based diagnostic loop before workflow editing: a model interprets execution traces, localizes a suspected defect, and proposes a modification. For example, AFlow generates reusable feedback signals based on iterative results (Zhang et al., 2025b), while JUDGEFLOW abstracts workflows into logic blocks and introduces a judge module for structured process-level feedback (Ma et al., 2026). Although effective, these methods rely on model-based judgments for diagnosis, which may introduce potential diagnostic bias and output uncertainty, as well as additional inference costs. Moreover, when a new workflow state resembles one evaluated in an earlier round, the optimizer often re-analyzes the trace and re-explores candi-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

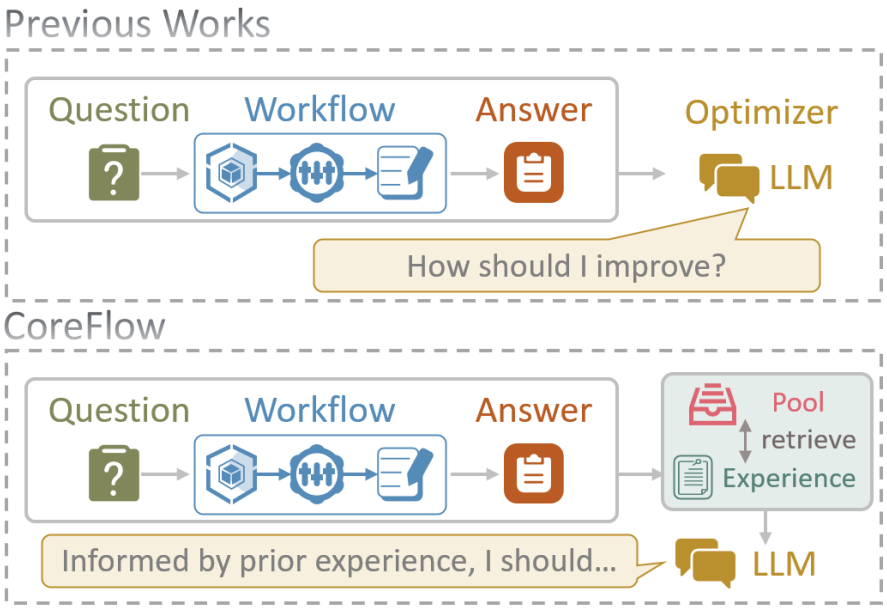


Figure 1: Experience-pool retrieval guides agentic workflow evolution by reusing prior workflow-edit effects for similar workflow states, avoiding repeated diagnosis and unguided exploration from scratch.

date edits instead of reusing the observed action effect. Historical evidence is consequently underused, and repeated diagnosis and exploration consume extra LLM calls, validation evaluations, and execution time.

Challenges and Solution. To address these limitations, we propose CORE-Flow, an experience-driven framework based on Counterfactual Observation and Retrieval-Guided Exploration for Workflow Evolution. CORE-FLOW moves beyond the traditional paradigm of model-based localization and refinement, enabling efficient workflow iteration through experience accumulation and reuse as illustrated in Figure 1. This design raises two key challenges. The first is experience accuracy: an experience should faithfully capture what changed in the workflow, why the change affected execution, and what impact it produced on validation behavior. The second is experience reusability: the system must select relevant experiences for the current workflow state and apply them as useful guidance rather than as rigid templates.

CORE-Flow addresses these challenges by integrating counterfactual attribution into experience generation and establishing mechanisms for semantic retrieval and breadth-first search. Specifically, CORE-Flow leverages counterfactual attribution to construct parent-child sample pairs and compares state transitions and transfer effects between parent and child workflows under controlled conditions, thereby generating more faithful and reusable experiences. Before each iterative refinement step, CORE-Flow retrieves relevant experiences and uses them to guide problem diagnosis and subsequent improvement. With these mechanisms, workflow search shifts from blind exploration over a large structural and prompt space to evidence-based optimization.

Contributions. Our main contributions are:

- We introduce an experience-retrieval paradigm for automated workflow evolution that extends localization–repair with counterfactual parent-child observation, structured experience accumulation, semantic retrieval, and retrieval-guided breadth search.
- Across six benchmarks, CORE-Flow attains the best average score of 89.41, surpassing JUDGEFLOW and AFLOW by 2.92 and 7.89 points, respectively. Furthermore, in the cost analysis tracked on the MATH dataset, it also incurs lower token cost and wall-clock execution time than both efficiency baselines.

2 Related Work

2.1 Agentic Workflow

Agentic workflows have become a common abstraction for building LLM-based systems that solve complex tasks through multiple coordinated model invocations. Early reasoning-oriented methods such as chain-of-thought prompting, self-consistency, Tree-of-Thought, debate, and self-refinement can be viewed as representative workflow patterns that improve reasoning by decomposing or repeatedly revising the solution process (Wei et al., 2022; Wang et al., 2022; Yao et al., 2023a; Du et al., 2023; Madaan et al., 2023). More recent agentic systems further combine language models with planning, memory, external tools, and environment interaction, extending workflows to code generation, web navigation, embodied control, software engineering, question answering, and decision-making (Yao et al., 2023b; Wang et al., 2024a; Hong et al., 2024; Jimenez et al., 2024).Traditional workflow construction depends substantially on manual expertise, motivating the development of automated workflow design methods.

2.2 Automated Agentic Optimization

Automated agentic optimization aims to automatically improve the structure, prompts, parameters, or execution policies of LLM-based agentic systems. Mainstream approaches achieve improvements through multiple rounds of iteration. However, due to the vast space of improvable parameters, this process still poses challenges in accurately identifying errors in the current workflow and effectively correcting them. Relevant scholars have conducted a series of studies on this. ADAS represents an agentic system as executable code and searches over generated agent programs (Hu et al., 2024). AFLOW formulates code-represented workflows as compositions of LLM-invoking nodes and executable edges, and combines reusable operators with an MCTS-style search loop driven by execution feedback (Zhang et al., 2025b). A^2Flow extends workflow generation with self-adaptive abstraction operators (Zhao et al., 2025), while MaAS-LLM searches over multi-agent architectures through an agentic supernet (Zhang et al., 2025a). JUDGEFLOW instead decomposes a workflow into logic blocks and uses an LLM-based judge to assign responsibility scores before applying targeted edits (Ma et al., 2026).Despite this progress, most of these methods still rely on LLM-based explanations, judgments, or the generation of suggestions for attribution and repair. Our work establishes a new workflow update paradigm that enhances iterative efficiency through experience-driven rather than model-subjective refinement.

2.3 Counterfactual Attribution

Counterfactual reasoning provides a principled way to attribute an observed outcome to a specific intervention by asking how the outcome would change if one factor were altered while other conditions remained fixed (Pearl, 2009). In machine learning explanation, counterfactual explanations have been used to identify minimal changes that would alter model decisions without requiring full access to the internal decision process (Wachter et al., 2017). Recent work on failure attribution for LLM multi-agent systems further highlights the importance of identifying which agent or component is responsible for task failure, while also showing that attribution can be sensitive to judge-model behavior and evaluation context (Zhang et al., 2025c). These ideas are well aligned with workflow evolution: a workflow update can be treated as an intervention on a workflow component, and the resulting score change provides empirical evidence about the effect of that component-level edit. Therefore, CORE-Flow introduces counterfactual attribution as a design constraint for experience construction: restrict modifications to parent-child workflows within a single fixed component, and observe the effects induced by variations in this component to derive empirical grounded judgments.

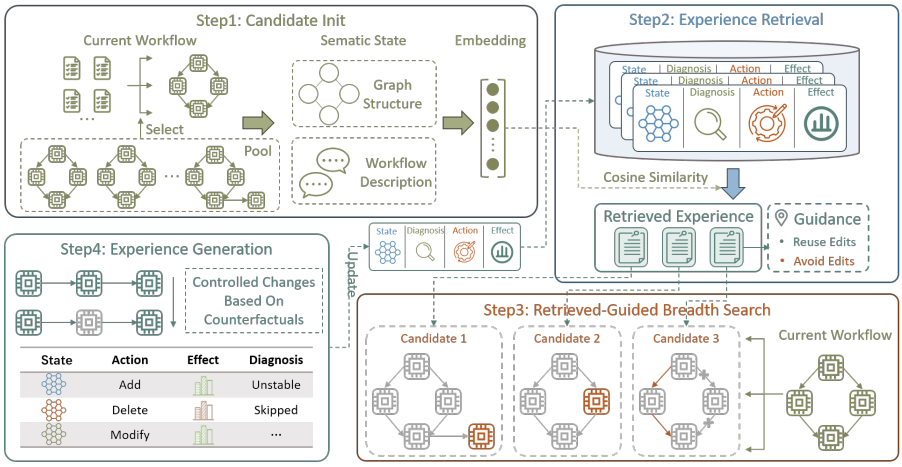


Figure 2: Overview of CORE-Flow. The optimizer represents parent-child workflow transitions as state-diagnosis-action-effect experiences, retrieves similar historical experiences to guide edit actions, evaluates candidate workflows, and updates both the workflow candidate pool and the experience pool.

3 Method

CORE-Flow innovates an experience-driven paradigm for automated workflow evolution that integrates counterfactual attribution into experience generation and incorporates semantic retrieval and breadth search mechanisms. These designs enhance the attribution fidelity and reusability of optimization experiences, enabling more effective workflow refinement. We first formulate this process as MCTS-inspired experience-augmented search, followed by its specific implementation.

3.1 Problem Formulation

Workflow state. We adopt the code-represented view of workflow optimization (Zhang et al., 2025b) and define an agentic workflow as an executable graph

$$W = (\mathcal{V}, \mathcal{E}, \Theta), \tag{1}$$

where $\mathcal{V} = \{v_i\}_{i=1}^{n}$ denotes workflow nodes, $\mathcal{E}$ specifies executable dependencies, and Θ contains node-level attributes. Each node is associated with an operator type, prompt, output schema, and optional execution parameters:

$$v_i = (o_i, p_i, f_i, \theta_i). \tag{2}$$

The operator set may include generation operators such as Custom, tool-like operators such as Programmer, and fusion operators such as ScEnsemble. Edges are represented by executable code, allowing sequential execution, conditional routing, loops, and aggregation.

Optimization Objective. Given a validation distribution $\mathcal{D}_{val}$ and a task evaluator $G(W, x)$, automated workflow search aims to find a workflow with high expected validation performance:

$$W^* = \arg\max_{W \in \mathcal{S}} \mathbb{E}_{x \sim \mathcal{D}_{val}}[G(W, x)]. \tag{3}$$

3.2 Overview

As shown in 2, we organize iterative workflow improvement as an MCTS-inspired search process following prior work (Zhang et al., 2025b; Ma et al., 2026), augmented with a reusable experience pool. It retains selection and expansion concepts but uses a scored candidate pool rather than standard UCT and tree backpropagation:

$$\mathcal{F}_\Omega = (\mathcal{S}, \mathcal{A}, \Omega, \mu, \pi_{\text{des}}, T, G, \mathcal{E}), \tag{4}$$

where $\mathcal{S}$ is the workflow search space induced by operators, prompts, attributes, and graph structures; $\mathcal{A}$ is the workflow-edit action space; Ω is a dynamic experience pool; μ retrieves relevant experiences; π_{des} proposes experience-conditioned edits; T applies an edit to a workflow; G evaluates a workflow; and $\mathcal{E}$ updates the experience pool. CORE-Flow additionally maintains a candidate pool $\mathcal{P}_t$ of evaluated workflows and their validation scores. At round t, one optimization iteration consists of four steps.

Step 1: candidate workflow initialization. Given $\mathcal{P}_t = \{(W_i, P_i)\}$, CORE-Flow selects the parent workflow for the current round using a softmax distribution over validation scores:

$$\Pr(W_t = W_i) = \frac{\exp(P_i/\tau)}{\sum_{(W_j,P_j)\in\mathcal{P}_t} \exp(P_j/\tau)}, \tag{5}$$

where τ controls the trade-off between exploiting high-scoring workflows and exploring other candidates.

Step 2: experience retrieval. The selected workflow is converted into the state $s_t = (g_t, m_t)$, and the retrieval function μ uses its normalized description m_t to select the top-K historical experiences:

$$\Omega_t^{(K)} = \mu(m_t, \Omega_t). \tag{6}$$

Step 3: experience-guided breadth search. Conditioned on W_t, its diagnostic context d_t, and the retrieved experiences, CORE-Flow jointly proposes B localized edits, applies them to the same parent, evaluates the resulting candidate set $W_{t+1}^{(b)}$ on the shared validation subset, and retains the best child in $\mathcal{P}_{t+1}$. We summarize this stage as

$$(W_{t+1}^{(b)}, \mathcal{P}_{t+1}) = \text{Search}_B\Big(W_t, d_t, \Omega_t^{(K)}, \mathcal{P}_t\Big). \tag{7}$$

Step 4: experience generation. Every evaluated parent-child transition, including a child not retained in $\mathcal{P}_{t+1}$, is converted into a state-diagnosis-action-effect record. The resulting experiences are incorporated into the pool through a single update:

$$\Omega_{t+1} = \mathcal{E}\Big(\Omega_t, \{\langle s_t, d_t^{(b)}, a_t^{(b)}, e_t^{(b)}\rangle\}_{b=1}^{B}\Big). \tag{8}$$

The four-step loop couples workflow selection, historical evidence retrieval, controlled lateral exploration, and experience accumulation within a single MCTS-inspired search process.

3.3 Experience Structure

CORE-Flow represents each evaluated parent-child transition as a state-diagnosis-action-effect experience, which contains four components:

$$\omega = \langle s_\omega, d_\omega, a_\omega, e_\omega\rangle. \tag{9}$$

State. The state is a two-tuple

$$s_\omega = (g_\omega, m_\omega). \tag{10}$$

Here, g_ω records the parent workflow graph before the edit, and m_ω is a one-sentence description of its information flow generated by the LLM-based optimizer, such as "first send the task input to three Custom nodes and then feed their outputs into one ScEnsemble node.". The state-description prompt is provided in Appendix B.

Diagnosis. The diagnosis is represented as

$$d_\omega = (c_\omega, r_\omega). \tag{11}$$

Here, c_ω identifies the weak component or local region in the parent workflow, and r_ω summarizes why the recorded action and effect indicate this diagnosis. It is generated from the parent-child workflow difference described in Section 3.5 and provides the rationale needed to judge whether a historical edit is applicable to a new workflow.

Action. The action a_ω records the localized modification that transforms the parent workflow into its child:

$$a_\omega \in \{\text{AddNode}, \text{RemoveNode}, \text{ModifyNode}\}. \tag{12}$$

Inspired by Zihan Ma et al.(Ma et al., 2026), we define three action families — AddNode, RemoveNode, and ModifyNode — corresponding to node insertion, deletion, and attribute modification.

Effect. The effect is the observed parent-child performance change on the shared validation subset:

$$e_\omega = G(W_{\text{child}}, \mathcal{B}_{cand}) - G(W_{\text{parent}}, \mathcal{B}_{cand}). \tag{13}$$

The direction and magnitude of the effect indicate whether the modification is beneficial and provide an empirical signal for diagnosis generation.

3.4 Experience Selection and Application

Experience selection identifies historical transitions whose workflow context is most relevant to the current state. CORE-Flow compares the current information-flow description m_t with each historical description m_ω using embedding cosine similarity:

$$\text{Sim}_\phi(m_t, m_\omega) = \cos\left(E_\phi(m_t), E_\phi(m_\omega)\right) \in [0, 1], \tag{14}$$

where $E_\phi(\cdot)$ denotes a text embedding model. Instead of applying a manually chosen similarity cutoff, CORE-Flow always retrieves the top-K most similar historical experiences.

Experience application uses the retrieved experiences as optimization signals for updating a selected workflow. Given the current workflow W_t, its current diagnosis d_t, and the retrieved experience set $\Omega_t^{(K)}$, the optimizer produces B experience-guided edit actions through the prompt Π_{exp} in Appendix D, and applies each action to generate new workflows:

$$\begin{aligned} \{a_t^{(b)}\}_{b=1}^{B} &\leftarrow \text{Optimizer}\Big(W_t, d_t, \Omega_t^{(K)}, \Pi_{\text{exp}}, B\Big), \\ W_{t+1}^{(b)} &= T(W_t, a_t^{(b)}), \qquad b = 1, \ldots, B. \end{aligned} \tag{15}$$

CORE-Flow then selects the best candidate workflow by evaluating these candidate workflows on the same validation subset:

$$\begin{aligned} W_{t+1}^\star &= \arg\max G(W_{t+1}^{(b)}, \mathcal{B}_{cand}), \\ P_{t+1}^\star &= G(W_{t+1}^\star, \mathcal{B}_{cand}). \end{aligned} \tag{16}$$

After evaluation, the selected child is inserted into the workflow candidate pool:

$$\mathcal{P}_{t+1} = \text{TopM}\big(\mathcal{P}_t \cup \{(W_{t+1}^\star, P_{t+1}^\star)\}\big), \tag{17}$$

where TopM keeps the best M workflows according to validation performance. At the next iteration, the initial workflow is sampled from the updated pool according to the equation (5). Candidate workflows that are not selected for expansion are still retained as evaluated transitions for experience construction.

3.5 Experience Pool Evolution

Experience pool evolution converts evaluated parent-child workflow updates into reusable state-diagnosis-action-effect records. The experience is generated by the LLM-based optimizer that compares the parent workflow with each candidate child, observes the applied action and its measured impact, and summarizes what changed, why the change mattered, and what effect it produced. To improve experience accuracy, CORE-Flow adopts the principle of counterfactual attribution (Pearl, 2009; Wachter et al., 2017): each candidate update is constrained to modify only one workflow node. This constraint is implemented through the single-component editing prompt in Appendix C and checked by a deterministic validator. Let $\Delta W_t^{(b)}$ denote the difference between parent workflow W_t and candidate child $W_{t+1}^{(b)}$. The validator is defined as

$$\begin{aligned} \text{Verify}\Big(\Delta W_t^{(b)}\Big) = \mathbb{I}\big[&\mathcal{V}_t = \mathcal{V}_{t+1}^{(b)} \wedge \mathcal{E}_t = \mathcal{E}_{t+1}^{(b)} \\ &\wedge \Big|\{v \in \mathcal{V}_t : \Theta_{t,v} \neq \Theta_{t+1,v}^{(b)}\}\Big| = 1\big]. \end{aligned} \tag{18}$$

The edit is accepted only when $\text{Verify}(\Delta W_t^{(b)}) = 1$, otherwise, the candidate is regenerated. For each candidate, CORE-Flow computes its observed effect and asks the LLM-based optimizer to generate a diagnosis conditioned on the concrete action and measured effect:

$$\begin{aligned} e_t^{(b)} &= G(W_{t+1}^{(b)}, \mathcal{B}_{cand}) - G(W_t, \mathcal{B}_{cand}), \\ d_t^{(b)} &= \text{LLM}_{\text{exp}}\left(\Delta W_t^{(b)}, a_t^{(b)}, e_t^{(b)}\right). \end{aligned} \tag{19}$$

The diagnosis is therefore generated from the action and its observed impact rather than from an unconstrained subjective judgment. Coupled with information such as the workflow state, a new experience is then generated.

$$\omega_t^{(b)} = \langle s_t, d_t^{(b)}, a_t^{(b)}, e_t^{(b)} \rangle. \tag{20}$$

All evaluated candidates will generate an experience and be added to the experience pool:

$$\Omega_{t+1} = \mathcal{E}\left(\Omega_t, \{\omega_t^{(b)}\}_{b=1}^{B}\right), \tag{21}$$

where $\mathcal{E}$ adds the newly generated records to the pool and retains both positive and non-positive effects. As the search proceeds, the pool accumulates evidence about which localized edits helped or harmed similar workflow states.

4 Experiments

4.1 Experimental Setup

Benchmarks. Following the evaluation protocol of automated workflow generation (Wang et al., 2026), we consider six public benchmarks spanning three domains. GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) evaluate mathematical reasoning; MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) evaluate code generation; and DROP (Dua et al., 2019) and HotpotQA (Yang et al., 2018) evaluate reading comprehension and multi-hop question answering. We follow the AFLOW data protocol (Zhang et al., 2025b), using the validation split only for workflow evolution and the held-out test split for final evaluation. We report solve rate for GSM8K and MATH, pass@1 for MBPP and HumanEval, and F1 for DROP and HotpotQA; all values are percentages.

Baselines. We compare CORE-Flow with three groups of methods represented in our evaluation. Manual prompting methods include direct input-output prompting (IO), Chain-of-Thought (CoT) (Wei et al., 2022), and Self-Consistency (CoT-SC) (Wang et al., 2022). Hand-crafted multi-agent workflows include Self-Refine (Madaan et al., 2023), Solo Performance Prompting (SPP) (Wang et al., 2024b), and MedPrompt (Nori et al., 2023). Autonomous workflow optimization methods include ADAS (Hu et al., 2024), AFLOW (Zhang et al., 2025b), A^2Flow (Zhao et al., 2025), MaAS (Zhang et al., 2025a), and JUDGEFLOW (Ma et al., 2026). AFLOW and JUDGEFLOW serve as the primary efficiency baselines because they respectively represent score-guided search and LLM-based fine-grained localization.

Implementation Details. We use GPT-4.1-mini (OpenAI, 2025a) as the workflow optimizer, GPT-5-mini (OpenAI, 2025b) as the default executor, and OpenAI text-embedding-3-small (OpenAI, 2024) to encode normalized workflow descriptions. Each method uses the same executor, dataset split, and validation budget. Workflow optimization runs for 10 rounds; for CORE-Flow, the default retrieval depth and search breadth are $K = 2$ and $B = 2$, respectively. Main task results are averaged over three independent runs. The cost study instead tracks one complete 10-round MATH run so that token charges and end-to-end execution time can be attributed to a concrete optimization trajectory.

4.2 Performance Analyses

Table 1 shows that CORE-Flow obtains the best overall score of 89.41, exceeding the strongest baseline, JUDGEFLOW, by 2.92 points and AFLOW by 7.89 points. The improvement is broad rather than concentrated in one domain: CORE-Flow ranks first on

Table 1: Main performance averaged over three independent runs. Bold and underlined values denote the best and second-best results in each column, respectively.

Method	GSM8K ↑	MATH ↑	MBPP ↑	HumanEval ↑	DROP ↑	HotpotQA ↑	Avg. ↑
Manual Prompting Methods							
IO	<u>94.31</u>	71.60	75.37	93.13	88.12	<u>77.88</u>	83.40
CoT	93.74	75.10	77.13	90.84	91.25	74.00	83.68
CoT-SC	92.04	82.72	75.07	93.89	82.38	76.88	83.83
Hand-Crafted Multi-Agent Systems							
Self-Refine	83.13	72.63	74.78	87.02	78.25	76.88	78.78
SPP	93.84	74.49	83.87	92.37	72.25	65.75	80.43
MedPrompt	90.71	75.10	79.77	88.55	84.88	71.88	81.81
Autonomous Multi-Agent Systems							
ADAS	80.66	75.51	73.31	91.60	82.38	63.75	77.87
AFLOW	78.96	75.93	77.13	97.71	84.75	74.62	81.52
A^2Flow	89.76	73.87	77.42	88.55	87.38	75.38	82.06
MaAS-LLM	85.69	<u>87.89</u>	70.09	90.08	80.38	75.50	81.60
JUDGEFLOW	94.50	85.84	74.19	93.89	<u>92.88</u>	77.62	<u>86.49</u>
CORE-Flow	93.93	88.92	<u>81.52</u>	<u>96.18</u>	94.12	81.75	89.41

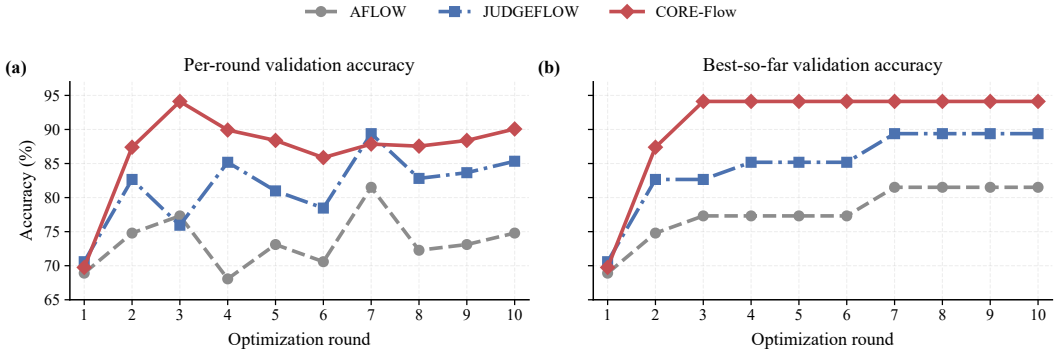


Figure 3: Optimization dynamics on MATH. (a) Best-so-far accuracy measures convergence speed. (b) Per-round accuracy measures the stability of explored candidates. CORE-Flow converges by round 3 while maintaining a consistently higher accuracy band.

MATH, DROP, and HotpotQA and second on both MBPP and HumanEval. Relative to JUDGEFLOW, it improves five of the six benchmarks, with gains of 7.33 points on MBPP, 4.13 on HotpotQA, 3.08 on MATH, 2.29 on HumanEval, and 1.25 on DROP. In particular, its MBPP score rises to 81.52, second only to the specialized SPP workflow. These results indicate that reusing empirically successful edits improves reasoning and evidence aggregation while also providing strong code-generation performance, yielding the most balanced aggregate result.

4.3 Cost Analyses

To measure search efficiency, we record the token cost and wall-clock execution time consumed in every round of one complete MATH optimization run and sum both quantities over all 10 rounds. Table 2 reports the resulting totals; the round-wise measurements and detailed analysis are provided in Appendix A.

Table 2: Accuracy and aggregate cost of one tracked 10-round MATH optimization run.

Method	Accuracy ↑ (%)	Token Cost ↓ (USD)	Wall-Clock Time ↓ (s)
AFLOW	75.93	0.1858	2578.3
JUDGEFLOW	85.84	1.6592	1876.1
CORE-Flow	88.92	0.0572	1364.0

Within the tracked MATH run, CORE-Flow exhibits a more favorable accuracy-cost tradeoff. As shown in Table 2, it achieves the highest accuracy while reducing token cost by 69.2%

and end-to-end execution time by 47.1% relative to AFLOW. Against JUDGEFLOW, the reductions reach 96.6% and 27.3%, respectively, together with a 3.08-point accuracy gain. This run-specific Pareto improvement is consistent with the design of CORE-Flow: retrieved parent-child transitions replace repeated LLM-based localization with lightweight precedent lookup, while stored action effects discourage low-yield edits. Retrieval-guided breadth search still evaluates alternatives, but concentrates them on historically supported local changes. Figure 3 further shows that this guidance discovers a strong workflow early rather than relying on prolonged trial and error.

4.4 Ablation Study

Table 3: Ablation on MATH. Δ is measured against optimization only.

Variant	Accuracy (%) ↑	**Δ**
IO	71.60	–
Optimization only	75.93	–
+ Experience retrieval	86.63	+10.70
+ Breadth search	79.84	+3.91
+ Experience & Search	88.92	**+12.99**

As shown in Table 3, optimization alone improves IO by 4.33 points, whereas experience retrieval adds 10.70 points and breadth search adds 3.91 points over this baseline. Combining both reaches 88.92, a 2.29-point gain over retrieval alone, indicating that retrieval supplies promising directions while breadth search compares alternative realizations before commitment.

4.5 Further Analysis

Optimization Dynamics. Figure 3 compares convergence and stability under the same 10-round budget. CORE-Flow reaches 87.39 in round 2 and its maximum of 94.12 in round 3, whereas JUDGEFLOW does not reach 89.39 until round 7 and AFLOW reaches only 81.51 at that round. From round 2 onward, CORE-Flow also remains within a high and narrow accuracy band of 85.87–94.12, compared with 75.94–89.39 for JUDGEFLOW and 68.07–81.51 for AFLOW. These results suggest that experience retrieval provides effective edit priors for rapid convergence, while breadth search limits severe performance degradation, yielding both faster and more stable optimization.

Hyperparameter Sensitivity. We vary the retrieval depth K and breadth B independently while holding the other setting at its default. Figure 4 shows that both parameters attain their best result at 2. With $K = 1$, the limited experience set appears insufficient to provide reliable precedent, whereas the decline from 88.92 to 83.02 at $K = 3$ suggests that less relevant transitions can dilute the retrieval signal. A similar pattern holds for search breadth: $B = 1$ limits lateral comparison, while the lower score at $B = 3$ indicates that additional alternatives can increase selection difficulty without improving the incumbent. The peak at the moderate setting supports the default $K = B = 2$ and suggests that CORE-Flow benefits from selective, rather than indiscriminate, retrieval and exploration.

5 Conclusion

We introduced CORE-Flow, an experience-driven framework for agentic workflow evolution. It reuses controlled parent-child edit effects to guide localized improvement and same-parent breadth search. Across six benchmarks, CORE-Flow achieves the best average score of 89.41, exceeding JUDGEFLOW and AFLOW by 2.92 and 7.89 points, while the tracked MATH run shows strong potential to reduce token cost and execution time. Future work will study more systematic retrieval, effect estimation, and causal experience construction.

AI Use Disclosure

We used generative AI tools to assist with literature discovery, language editing, LaTeX formatting, and figure and table presentation during the preparation of this manuscript. We did not use generative AI tools to generate experimental data, implement the experimental method, formulate the theoretical claims, or interpret the final experimental results. All AI-assisted text and formatting were reviewed by the authors, and the cited literature, mathematical statements, code, figures, tables, and experimental claims were checked against the underlying sources and runs. The authors take responsibility for the final content of this work.

Reproducibility Statement

The main paper specifies the workflow representation, search procedure, single-component validation rule, model roles, datasets, evaluation protocol, and default hyperparameters. The appendix after the references gives the complete round-wise cost trace, detailed cost analysis, and prompts used for state description, single-component editing, and experience-conditioned design. The source code and configuration files can be released through an anonymous supplementary submission.

References

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

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

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 2368–2378. Association for Computational Linguistics, 2019.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset, 2021.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Zhang, Ceyao Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In International Conference on Learning Representations, 2024.

Zihan Ma, Zhikai Zhao, Chuanbo Hua, Federico Berto, and Jinkyoo Park. JUDGEFLOW: Agentic workflow optimization via block judge, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, et al. Can generalist foundation models outcompete special-purpose tuning? case study in medicine. arXiv preprint arXiv:2311.16452, 2023.

OpenAI. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/, 2024. Accessed: 2026-07-28.

OpenAI. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/, 2025a. Accessed: 2026-07-28.

OpenAI. GPT-5 mini model. https://platform.openai.com/docs/models/gpt-5-mini, 2025b. Accessed: 2026-07-28.

Judea Pearl. Causality: Models, Reasoning, and Inference. Cambridge University Press, 2 edition, 2009.

Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. Harvard Journal of Law & Technology, 31(2):841–887, 2017.

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

Xiang Wang, Zongtao Yang, Zhuojian Hong, Shuhao Zhang, and Wei Wei. FusionFlow: Enabling deep structural exploration for automated agentic workflow generation. In Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 27718–27760, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.1278.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2022.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 257–279. Association for Computational Linguistics, 2024b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2022.

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023b.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. arXiv preprint arXiv:2502.04180, 2025a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFLOW: Automating agentic workflow generation. In International Conference on Learning Representations, 2025b.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), Proceedings of the 42nd International Conference on Machine Learning, volume 267 of Proceedings of Machine Learning Research, pp. 76583–76599. PMLR, 13–19 Jul 2025c. URL https://proceedings.mlr.press/v267/zhang25cq.html.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A^2Flow: Automating agentic workflow generation via self-adaptive abstraction operators, 2025. Accepted by AAAI 2026.

A Round-Wise Cost Analysis

We trace one complete 10-round MATH optimization run for each optimizer. For every round, the experiment runner records the dollar-denominated token cost returned by the API usage meter and the end-to-end execution time from the round-level timer. Table 4 reports the unaggregated measurements underlying the aggregate cost comparison in the main paper; totals differ only by rounding.

Table 4: Round-wise sample execution time and token cost for the tracked MATH runs.

Round	AFLOW		JUDGEFLOW		CORE-Flow	
	Sample Time (s) ↓	Cost (USD) ↓	Sample Time (s) ↓	Cost (USD) ↓	Sample Time (s) ↓	Cost (USD) ↓
1	960.3	0.02022	1,098.8	0.13341	1,235.2	0.01967
2	1,010.2	0.00476	1,708.5	0.21234	962.8	0.00492
3	2,210.8	0.00775	1,019.6	0.13438	1,113.5	0.00369
4	2,535.2	0.00780	1,241.0	0.15717	1,033.7	0.00329
5	2,149.6	0.02229	1,214.9	0.17099	1,202.1	0.00377
6	2,181.9	0.02135	1,876.7	0.21666	1,312.6	0.00423
7	2,486.8	0.02444	2,091.2	0.18149	1,044.6	0.00405
8	2,237.6	0.02523	2,294.9	0.19585	827.9	0.00434
9	2,333.2	0.02233	1,069.7	0.12737	791.3	0.00478
10	2,521.0	0.02962	1,393.3	0.12953	1,388.4	0.00446
Total	20,626.6	0.18579	15,008.6	1.65920	10,912.1	0.05721

Why AFLOW becomes expensive after round 3. The round-wise increase is explained by deterministic growth in the executed workflow. As summarized in Table 5, round 2 invokes a single Custom operator. Round 3 changes this into two serial calls: the first produces a draft and the second verifies it. This doubles the critical sequence of LLM calls and carries the draft into the verification context; correspondingly, the recorded end-to-end time rises from 1,010.2 to 2,210.8 seconds, while the average per-sample execution time rises from 8.49 to 18.58 seconds. Round 4 retains the same two-stage structure and exhibits a similarly high time.

Table 5: AFLOW structural phases and average successful-sample cost statistics.

Round(s)	Executed Workflow	Time (s)	Cost (USD)
2	Custom	8.49	0.000040
3	Custom → Custom	18.58	0.000065
4	Custom → Custom	21.30	0.000066
5–9	Programmer → Custom	18.06–20.90	0.000179–0.000212
10	Programmer → Custom → Custom	37.99	0.000249

From round 5 onward, AFLOW replaces the first call with Programmer. This operator generates code, launches a subprocess to execute it, and can regenerate failing code up to three times. The following Custom call consumes the program output and, in rounds 7 and 9, the full generated code as additional context. The traces contain 21, 23, and 21 code-generation attempts in rounds 7, 8, and 10, respectively, rather than exactly one attempt per problem. These retries and longer contexts explain why token cost increases from 0.00780 in round 4 to 0.02229 in round 5 and remains elevated thereafter. Round 10 adds a final Custom verification call; its token cost therefore peaks at 0.02962, while its end-to-end time remains near the maximum observed for AFLOW.

Why JUDGEFLOW is slow in rounds 6–8. Rounds 6–8 all use two serial sequence blocks, Verify → Programmer, whereas rounds 5, 9, and 10 use a single Verify block. Each problem therefore requires two ordered model calls in rounds 6–8. The second call also receives the first block's output as its current state together with rolling memory, repeating part of the context. This structural expansion aligns with the increase from 1,214.9 seconds in round 5 to 1,876.7–2,294.9 seconds in rounds 6–8, followed by a decline to 1,069.7 seconds when the workflow returns to one block in round 9. The Programmer label in this JUDGEFLOW reproduction denotes an LLM prompt type only; unlike AFLOW's operator, it does not execute Python code.

Round 8 is a residual latency peak rather than a further structural expansion: rounds 6–8 have the same two-block topology and no recorded API failures. Its token cost is only 7.9% above round 7, and its end-to-end time is 9.7% higher. This mismatch is more consistent with response-length variation, model-service latency, and concurrent scheduling tails than with deterministic algorithmic work. Because the available trace does not provide a per-call token/latency decomposition, we do not attribute this residual peak to any single operator.

B Workflow State Description Prompt

CORE-Flow retrieves experiences by comparing one-sentence semantic descriptions of workflow information flow. The following prompt converts workflow code and graph metadata into m_ω.

Listing 1: Prompt template for canonical workflow-state description.

```
You are describing an agentic workflow for experience retrieval.
Convert the workflow into exactly one concise English sentence.
Only describe the information flow between workflow operators.
Do not evaluate whether the workflow is good or bad.
Do not mention validation scores or weak-node diagnoses.

Input:
1. Workflow code or graph specification.
2. Operator list and operator types.
3. Edge list and data-flow dependencies.

```

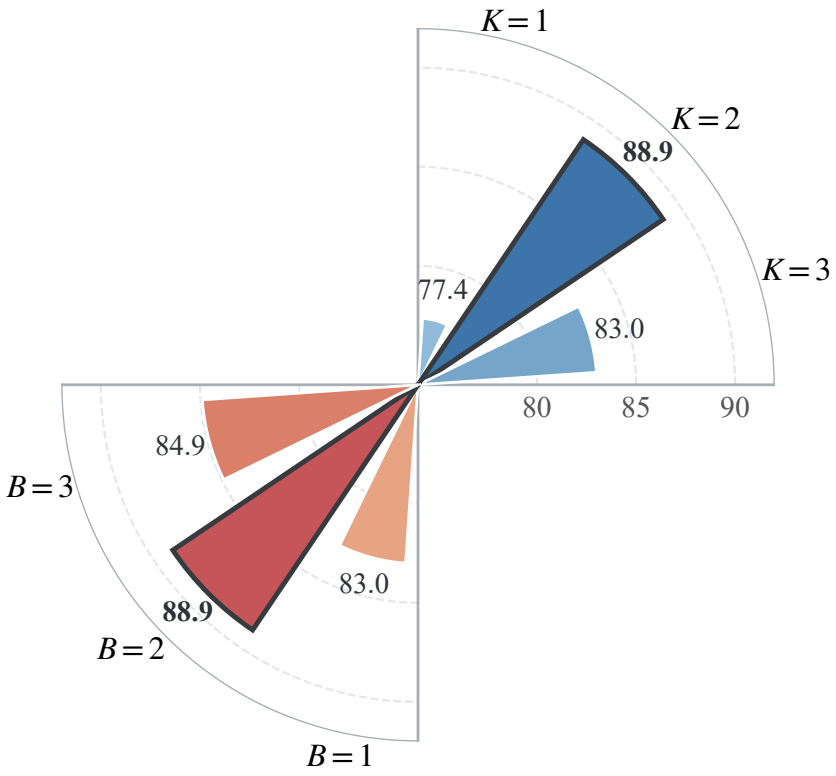


Figure 4: Polar sensitivity of MATH accuracy to retrieval depth K (upper-right sector) and search breadth B (lower-left sector).

```
12  Output format:
13  One sentence only.
14
15  Example:
16  First send the task input to three Custom nodes, then feed their
        candidate answers into one ScEnsemble node to select the final
        answer.
```

C Single-Component Counterfactual Edit Prompt

CORE-Flow treats parent-child workflow transitions as observed counterfactual evidence only when the edit is localized enough to support attribution. The following prompt constrains the designer to modify one target component per candidate.

Listing 2: Prompt template for single-component workflow edits.

```
1   You are proposing a candidate workflow edit for counterfactual
        comparison.
2   The purpose is to estimate the effect of one component-level change.
3
4   Hard constraint:
5   Modify exactly one target component and keep all other components
        unchanged.
6   A target component can be:
7       - one node prompt or node attribute,
8       - one operator instance,
9       - one local edge/data-flow pattern,
10      - one local aggregation/selection rule.
11
12  Do not change multiple nodes, multiple prompts, and global graph
        structure
```

```
in the same candidate. If a broader redesign seems useful, decompose it into
separate single-component candidates.

Output:
1. Target component.
2. Action type: AddNode, RemoveNode, or ModifyNode.
3. Exact edit.
4. Elements intentionally kept unchanged.
5. Expected mechanism of improvement.
```

D Experience-Conditioned Design Prompt

CORE-Flow implements experience selection and application through a design prompt rather than a hard-coded editing rule. The prompt gives the designer the top-K most similar experiences according to embedding cosine similarity, and then asks it to interpret their effects when proposing edits. A simplified version is shown below.

Listing 3: Prompt template for experience selection and application.

```
You are a workflow designer. Your task is to improve the current workflow by using historical workflow-edit experiences when they are applicable.

Input:
1. Current workflow graph state:
   - node/operator list
   - edge and information-flow structure
   - canonical semantic state description

2. Current diagnosis:
   - weak component(s)
   - diagnostic context and failure patterns
   - operator reliability statistics when available

3. Retrieved historical experiences:
   Each experience has:
   - state: parent workflow graph structure
   - diagnosis: weak component and diagnostic context
   - action: AddNode, RemoveNode, or ModifyNode
   - effect: validation score change after the action

Decision policy:
Step 1: Use the retrieved Top-K experiences as historical reference evidence.
If the experience list is empty, treat this as a cold-start case and rely on the current diagnosis.

Step 2: Analyze effects among the retrieved experiences.
If similar positive-effect experiences exist, prefer actions that follow the same design principle, while adapting the target node, prompt, and information flow to the current diagnosis.
If retrieved experiences are neutral or negative, avoid repeating their harmful action patterns and use the current diagnosis to propose a conservative alternative.

Step 3: Produce one candidate workflow edit.
The edit must specify:
   - target component
   - action type
   - concrete graph or prompt modification
   - reason based on current diagnosis
   - which retrieved experiences support or warn against the edit
   - which workflow elements are intentionally kept unchanged
```

```
37
38  Do not blindly copy historical workflows. Reuse the action principle
        only when the current semantic state is similar and the historical
         effect supports it. Modify only one target component so that the
        resulting parent-child transition can be used as counterfactual
        evidence.
```