FUSIONFLOW: Enabling Deep Structural Exploration for Automated Agentic Workflow Generation

Anonymous ACL submission

Abstract

Agentic workflows are commonly used to guide large language models in solving complex reasoning tasks. However, existing automated workflow generation methods primarily rely on stepwise local refinement or tree-based search over a single evolving workflow. Under limited optimization budgets, this paradigm constrains **structural depth**, hindering the discovery of workflows that require deep compositional structure. To address this limitation, we propose **FUSIONFLOW**, a framework centered on workflow fusion. Unlike incremental refinement, fusion enables structural leaps by synthesizing multiple independently evolved workflows, allowing exploration of deeper regions of the workflow space within a finite budget. To make fusion effective, **FUSIONFLOW** integrates local optimization, task-specific differentiation, and a dynamic scheduling mechanism. Experiments on six reasoning benchmarks demonstrate that **FUSIONFLOW** consistently outperforms existing automated workflow generation methods. Further ablation and analysis confirm that fusion is the key driver of deep structural exploration, highlighting fusion-driven exploration as an effective approach for overcoming depth limitations in automated workflow generation.

1 Introduction

Large language models (LLMs) (Touvron et al., 2023; Achiam et al., 2023; Anthropic, 2025b) have demonstrated strong capabilities in reasoning, planning, and decision-making (Wang et al., 2024a; Zaharia et al., 2024; Xi et al., 2025), giving rise to LLM-based agent systems that autonomously tackle complex tasks across diverse domains with minimal human intervention (Hong et al., 2023; Qian et al., 2024a; Ridnik et al., 2024). Early agent systems emphasize free-form reasoning, enabling autonomous planning and dynamic interaction with environments (Chen et al., 2023; Qiao et al., 2024; Zhu et al., 2024). Despite their adaptability and creativity, such systems often suffer from instability and limited controllability. To address these limitations, recent work has introduced *agentic workflows*, which structure reasoning into repeatable, explicitly defined pipelines of LLM calls (Khattab et al., 2024; Zhou et al., 2024; Yuksekgonul et al., 2024). By constraining decision-making within well-defined workflows, these approaches provide more stable and controllable foundations for complex problem solving.

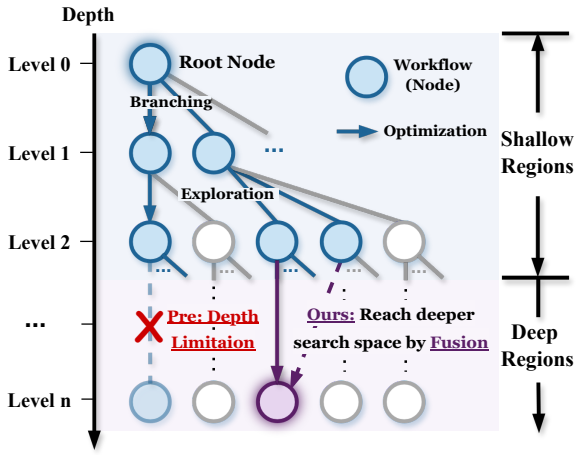

Figure 1: Illustration of **structural depth** limitation under a finite optimization budget. Incremental optimization methods are restricted to a shallow **structural depth** proportional to the iteration count. In contrast, fusion-based exploration enables structural leaps that allow the search to reach deeper regions of the workflow space within the same budget.

Existing automated workflow generation frameworks explore large combinatorial spaces of code- or graph-based workflows and iteratively refine candidate workflows based on execution feedback (Zhuge et al., 2024; Hu et al., 2024; Liu et al., 2025), shifting the role of LLMs from solving individual tasks to constructing structured problem-solving procedures. Representative methods such as ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2024) operationalize this paradigm through heuris-

tic optimization and tree-based exploration. Under practical optimization budgets, however, such incremental local refinement often constrains exploration to limited **structural depth**, i.e., the number of effective structural modifications accumulated along an optimization path, making it difficult to discover workflows with sufficiently complex structures, as illustrated in Figure 1 and further analyzed in Appendix D.

To overcome this structural depth limitation, we introduce the **Fusion Operator**. Unlike incremental local refinement methods that optimize a single workflow through step-wise modifications, fusion synthesizes effective components from multiple candidate workflows, enabling *structural leaps* during search. However, fusion alone is insufficient for effective structural exploration. Its success depends on the availability of diverse and complementary workflow structures to integrate. To this end, we leverage empirical observations about the organization of complex reasoning tasks. An analysis of the MATH benchmark (Figure 2A) reveals that such tasks often exhibit clear sub-domain structure, with distinct reasoning patterns across problem categories. This motivates a search strategy in which workflows are first specialized for different sub-domains and subsequently fused to construct a more robust global solution. Guided by this observation, we introduce the **Differentiation Operator**, which explicitly encourages the emergence of diverse, specialized workflow variants that serve as complementary building blocks for effective fusion.

Based on these insights, we propose **FusionFlow**, a workflow generation framework centered on fusion-driven structural exploration. The core of this framework is the **Fusion Operator**, which serves as the primary mechanism for breaking **structural depth** limitations, while other components are designed to support its effectiveness. Specifically, we incorporate an **Optimization Operator** to perform stable local refinement and a **Differentiation Operator** to generate diverse, complementary workflows for fusion. Coordinating these operators introduces a challenge, as they involve different trade-offs between exploitation and exploration. To address this issue, we introduce a **Dynamic Scheduling Algorithm** as the core control mechanism of **FusionFlow**. Modeled as a non-stationary Markov process, this algorithm adaptively selects operators, balancing stable local improvement with high-variance structural evolution.

Experiments on six reasoning benchmarks demonstrate that **FusionFlow** consistently outperforms existing automated workflow generation methods. Further ablation and analysis confirm that fusion is the key driver of deep structural exploration, highlighting fusion-driven exploration as an effective approach for overcoming depth limitations in automated workflow generation.

Our contributions are summarized as follows:

- We characterize the **structural depth** limitation of incremental workflow optimization methods and introduce the **Fusion Operator** to enable structural exploration beyond single-step refinement.
- We present **FusionFlow**, a fusion-centered workflow generation framework in which optimization, differentiation, and dynamic scheduling are designed to support effective fusion.
- Experiments on multiple benchmarks validate that fusion is the key driver for overcoming depth limitations and achieving superior performance.

2 Related Work

2.1 LLM-based Autonomous Agent Planning

Early agent systems were mainly built for specific domains such as code generation (Hong et al., 2023; Qian et al., 2024a; Ridnik et al., 2024), question answering (Zhou et al., 2022, 2023; Xu et al., 2024), and robotics (Liang et al., 2022; Singh et al., 2022; Song et al., 2023). These systems followed fixed pipelines or rule-based designs for task planning and reasoning (Wang et al., 2024a; Xi et al., 2025). As large language models (LLMs) became more capable (Achiam et al., 2023; Anthropic, 2025b), research shifted toward agents that can plan and make decisions more flexibly (Chen et al., 2023; Zhuge et al., 2023; Huang et al., 2024; Qiao et al., 2024; Zhu et al., 2024; Liu et al., 2025). Such LLM-based agents can decompose and solve problems through natural language reasoning, showing strong flexibility and creativity. However, their decision process is not explicitly structured, so their behavior and results are often unpredictable before execution. This makes it difficult to optimize them for specific tasks or to ensure consistent performance. Current improvements mainly focus on higher-level strategies, such as using multiple agents to cooperate

(Guo et al., 2024; Wang et al., 2024a; Qian et al., 2024b) or adding external feedback loops (Shinn et al., 2023; Wang et al., 2024b,c). While these methods improve stability to some extent, they still lack precise control over the reasoning process.

2.2 Automated Agentic Workflow Optimization

Agentic workflows provide a structured alternative to fully autonomous agents by organizing multiple LLM calls into explicit, executable reasoning pipelines (Zhuge et al., 2024; Hu et al., 2024; Liu et al., 2025). By constraining decision making within predefined workflow structures, these approaches improve controllability and reproducibility compared to free-form agent behaviors.

Early work primarily focused on improving performance within fixed workflows through prompt or parameter optimization, without modifying the underlying structure (Chen et al., 2023; Xu et al., 2023; Yang et al., 2023; Fernando et al., 2023; Khattab et al., 2024; Zhou et al., 2024; Yuksekgonul et al., 2024). More recent methods extend this line of research by directly searching over workflow structures, representing workflows as graphs or executable programs and refining them through iterative optimization (Li et al., 2024; Cheng et al., 2024; Zhuge et al., 2024; Hu et al., 2024; Zhang et al., 2024). Despite differences in representation and search strategy, most existing approaches incrementally refine a single candidate workflow through local modifications, limiting achievable search depth under practical budgets. Our work departs from this paradigm by enabling structural exploration beyond single-path refinement, addressing the depth limitation of prior methods.

3 Method

3.1 Preliminary

Agentic Workflow. Formally, an agentic workflow $\mathcal{W}$ is defined as an executable program modeled by a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ represents the set of functional nodes and $\mathcal{E}$ represents the control flow logic connecting them.

Each node $n_i \in \mathcal{N}$ encapsulates a specific atomic operation performed by a LLM. A node is parameterized as $n_i = (M, P, \tau, \mathcal{F})$, where M, P, τ and $\mathcal{F}$ denotes the backbone model, the natural language prompt instruction, the sampling temperature, and the output format constraints respectively. The edge set $\mathcal{E}$ is implemented via programming language constructs, supporting sequential execution, conditional branching, loops, and recursive calls. We provide representative code examples of the workflow graph structure, operator interfaces, and prompt organization used in **FusionFlow** in Appendix G.

Given an input task x, the workflow executes by traversing the directed graph according to the control-flow edges, invokes a sequence of LLM-based nodes, and finally returns a single output as the final answer. Mathematically, a workflow $\mathcal{W}$ acts as a mapping function $f_{\mathcal{W}} : \mathcal{X} \to \mathcal{Y}$, which takes an input task $x \in \mathcal{X}$ and produces a final answer $y \in \mathcal{Y}$, where $\mathcal{X}$ denotes the input space of task instances and $\mathcal{Y}$ denotes the output space of answers.

Problem Definition. We formulate the automated generation of agentic workflows as a search problem within the infinite workflow space $\mathbb{S}$. Given a task dataset $\mathcal{D} = \{(x_j, y_j)\}_{j=1}^{|\mathcal{D}|}$ and an evaluation metric $G(\hat{y}, y)$, for each task instance (x, y) sampled from the dataset, the workflow produces a prediction $\hat{y} = f_{\mathcal{W}}(x)$, which is then compared against the ground-truth label y via the metric $G(\hat{y}, y)$ to obtain a scalar score. The overarching objective is to identify an optimal workflow $\mathcal{W}^*$ that maximizes the expected performance over the data distribution:

$$\mathcal{W}^* = \arg\max_{\mathcal{W} \in \mathbb{S}} \mathbb{E}_{(x,y)\sim\mathcal{D}}[G(\hat{y}, y)] \qquad (1)$$

Here, $\mathbb{E}_{(x,y)\sim\mathcal{D}}[\cdot]$ denotes the empirical expectation over the dataset.

Structural Depth. We define the **structural depth** $d(\mathcal{W})$ of a workflow as the number of effective modifications it accumulates during the search process. Intuitively, each optimization step adds one modification, differentiation preserves the parent's depth, and fusion aggregates modifications from multiple parent workflows. Formally, let $\mathcal{N}(\mathcal{W})$ denote the set of modification nodes in $\mathcal{W}$, then $d(\mathcal{W}) = |\mathcal{N}(\mathcal{W})|$. Under purely incremental optimization, the maximum achievable depth is bounded by the length of a single optimization chain. In contrast, fusion enables $d(\mathcal{W}_{\text{fuse}}) \geq \max_i d(\mathcal{W}_i)$ by merging modification histories from multiple trajectories, thereby allowing workflows to reach deeper structural regions within a finite iteration budget. The formal recur-

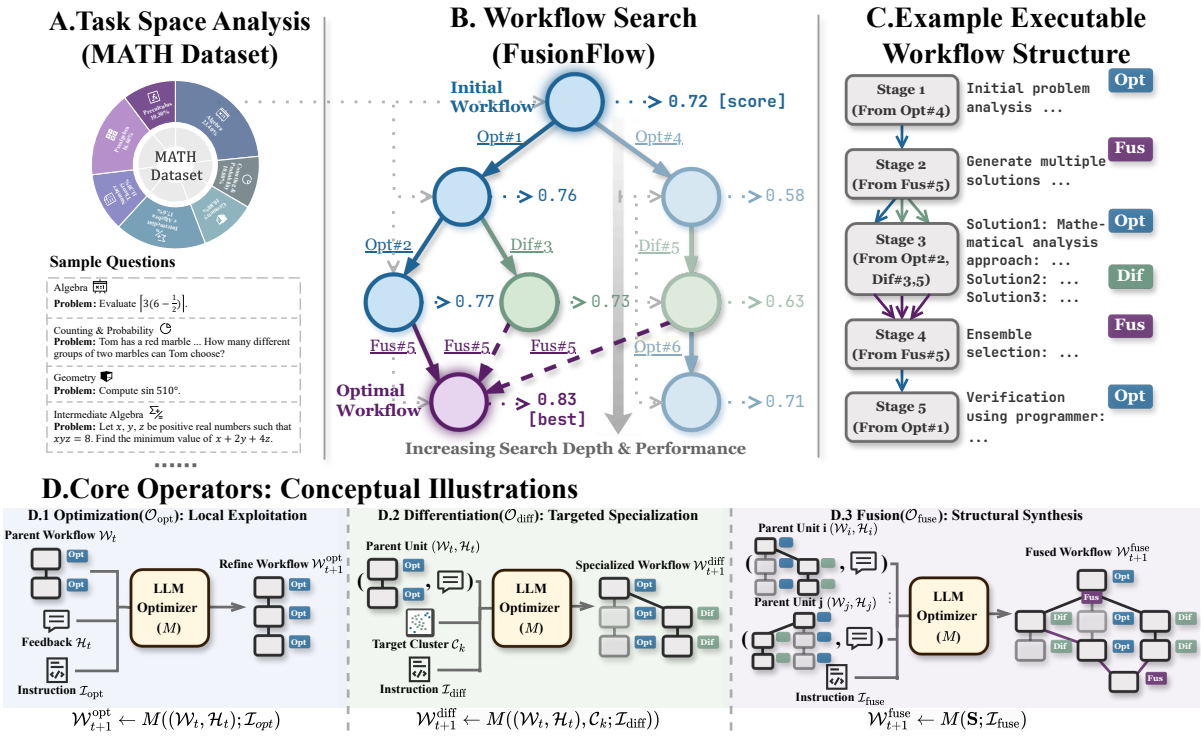

Figure 2: Overall framework of **FusionFlow**. (A) The task space exhibits a clustered structure (e.g., Algebra, Goemetry in MATH Dataset), motivating targeted search strategies. (B) The evolutionary search tree where nodes represent workflows and edges denote operations. The illustration shows how the framework progressively improves by leveraging information accumulated from previously explored workflows through Optimization, Differentiation, and Fusion, evolving from a baseline (0.72) to the optimal workflow (0.83). (C) An example structure of an executable workflow. (D) Conceptual illustrations of the three core operators.

sive definition and empirical analysis are provided in Appendix D.1.

3.2 Overview of FusionFlow

Figure 2 illustrates the overall design of **FusionFlow**. The framework operates over a search tree $\mathcal{T} = (\mathcal{W}, \mathcal{O})$ where each node $\mathcal{W}$ represents a fully executable workflow and edges $\mathcal{O}$ correspond to transformation operators. At each iteration t, the framework selects an operator $\mathcal{O}_t \in \{\mathcal{O}_{\text{opt}}, \mathcal{O}_{\text{diff}}, \mathcal{O}_{\text{fuse}}\}$ and one or more parent workflows from the previously explored nodes, then applies the operator to generate a new workflow node, thereby expanding the search tree.

In our framework, agentic workflows are composed of atomic operators connected by code control logic, which allows a large language model to synthesize a composite workflow by identifying shared structures and reconciling complementary components across multiple workflows. Fusion itself is therefore not difficult to apply (as demonstrated in Appendix B.5). The main challenge, however, lies in determining when fusion should be applied and which workflows should be fused, as indiscriminate fusion can result in redundancy, incompatible coordination, or performance degradation.

To address these challenges, **FusionFlow** incorporates a set of auxiliary components that jointly ensure the effectiveness of fusion:

- **Optimization** refines individual workflows to produce candidates with reliable local behaviors.
- **Differentiation** intentionally introduces structural and functional diversity, creating complementary candidate workflows.
- **Dynamic Scheduling Algorithm** coordinates these components with the fusion operator, adaptively regulating fusion timing and candidate selection to maximize depth expansion under limited iteration budgets.

3.3 Fusion Operator

We formalize workflow fusion as a structure level composition operator $\mathcal{O}_{\text{fuse}}$ that synthesizes multiple agentic workflows into a single complex workflow. Let $\mathbf{S} = \{(\mathcal{W}_i, \mathcal{H}_i)\}_{i=1}^{K}$ denote a selected set of K parent units ($K \geq 2$), where $\mathcal{H}_i$ denotes

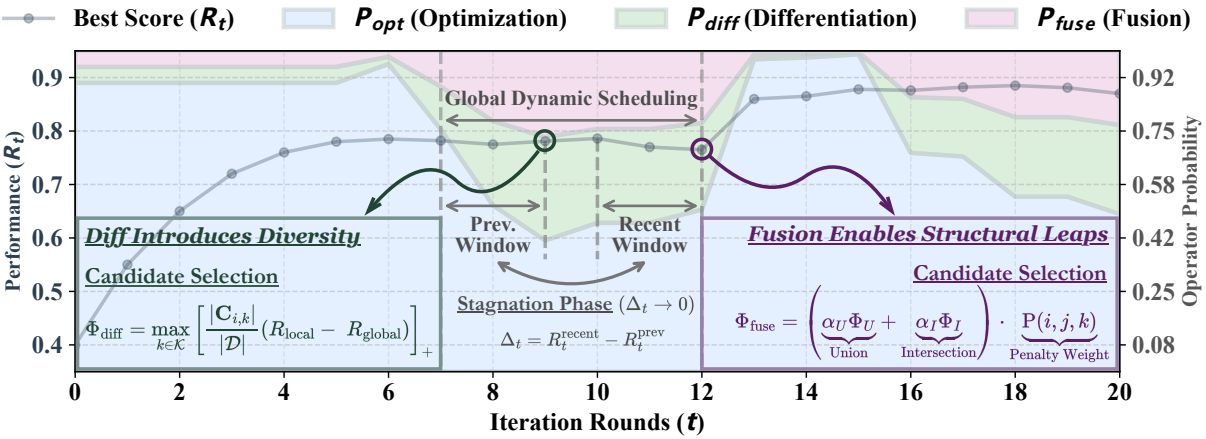

Figure 3: Conceptual illustration of the dynamic scheduling process in **FusionFlow** (real quantitative results are reported in Appendix F). The curve shows the evolution of the best workflow performance R_t over iteration rounds, while shaded regions indicate the probabilistic allocation among optimization (p_{opt}), differentiation (p_{diff}), and fusion (p_{fuse}). The middle panel highlights the two sliding windows used to detect stagnation. The formula in the lower-left corner defines the differentiation potential. The formula in the lower-right corner defines the fusion potential. Circled iterations indicate rounds where differentiation and fusion are applied.

execution feedback of $\mathcal{W}_i$ on the task dataset. Formally, the fusion process is defined as:

$$\mathcal{W}_{t+1}^{\text{fuse}} \leftarrow M(\mathbf{S}; \mathcal{I}_{\text{fuse}}) \tag{2}$$

Unlike optimization, fusion integrates the internal structures of multiple workflows. Concretely, each parent workflow $\mathcal{W}_i$ is represented as a directed graph $\mathcal{G}_i = (\mathcal{N}_i, \mathcal{E}_i)$. The fused workflow $\mathcal{W}_{t+1}^{\text{fuse}}$ integrates selected subgraphs from $\mathcal{G}_i$ through newly constructed control-flow nodes. As a result, $\mathcal{W}_{t+1}^{\text{fuse}}$ preserves the functional behaviors of its parents while introducing additional coordination logic. The core logic of the fusion operator is summarized in Appendix H.2.

3.4 Support Components

3.4.1 Optimization Operator

The optimization operator $\mathcal{O}_{\text{opt}}$ performs local refinement on a single workflow to improve reliability and correctness. Formally, it is defined as:

$$\mathcal{W}_{t+1}^{\text{opt}} \leftarrow M((\mathcal{W}_t, \mathcal{H}_t); \mathcal{I}_{\text{opt}}) \tag{3}$$

where $\mathcal{I}_{\text{opt}}$ specifies optimization instructions that constrain the modification to prompt content and node parameters. Within **FusionFlow**, optimization primarily serves as a preparatory operator that stabilizes local behaviors and ensures that workflows are sufficiently mature before being considered as reliable building blocks for fusion. Appendix H.1 provides the main logic used to produce and validate a single local refinement.

3.4.2 Differentiation Operator

The differentiation operator $\mathcal{O}_{\text{diff}}$ generates specialized variants of a workflow with respect to different task subspaces. Given a workflow $\mathcal{W}_t$, differentiation produces a specialized workflow:

$$\mathcal{W}_{t+1}^{\text{diff}} \leftarrow M((\mathcal{W}_t, \mathcal{H}_t), \mathcal{C}_k; \mathcal{I}_{\text{diff}}) \tag{4}$$

where $\mathcal{C}_k \subset \mathcal{D}$ represents a task sub-cluster and $\mathcal{I}_{\text{diff}}$ guides structural or reasoning-level modifications conditioned on the characteristics of $\mathcal{C}_k$. Differentiation increases functional and structural diversity in the pool, aiming to produce complementary workflows that make subsequent fusion more likely to yield non-redundant gains. We summarize the differentiation main execution logic in Appendix H.3.

3.4.3 Dynamic Scheduling Algorithm

We model the search process as a partially observable Markov decision process which is formally illustrated in Appendix B.7. To coordinate the overall search process, we introduce a dynamic scheduling mechanism (as shown in Figure 3) that regulates the application of all operators in the framework. Since workflow fusion constitutes the core contributor to depth expansion in **FusionFlow**, we focus our presentation here on fusion scheduling, including when fusion is applied and how fusion candidates are selected. The complete scheduling procedure over all operators is provided in the Appendix C and Appendix E.

Probabilistic Fusion Scheduling Within the Markov decision framework, the scheduler must

decide when to transition from exploitation to exploration. We use recent performance dynamics as an observable proxy for the latent advantage of each operator. Let R_t denote the best performance achieved at iteration t. Over a sliding window of size k, we define

$$R_t^{\text{recent}} = \max_{u \in [t-k+1,\, t]} R_u, \qquad R_t^{\text{prev}} = \max_{u \in [t-2k+1,\, t-k]} R_u. \tag{5}$$

Using the window maximum provides a robust progress indicator that tracks upper-bound improvement while filtering evaluation noise. The improvement difference $\Delta_t = R_t^{\text{recent}} - R_t^{\text{prev}}$ serves as an empirical estimator for the marginal return of continued local refinement. Based on this quantity, we define a fusion activation signal

$$\pi_t = \frac{1}{1 + \exp\big[\kappa \cdot \Delta_t\big]} \tag{6}$$

where κ controls sensitivity to performance stagnation. A higher π_t indicates reduced marginal gains from local refinement, under which the scheduler assigns higher probability to selecting fusion (as illustrated in Appendix B.7 and Appendix C).

Fusion Candidate Selection Once fusion is activated, the scheduler must select which workflows to fuse. This corresponds to choosing the action operands within the Markov decision process. Candidate selection is based on structural complementarity and behavioral consensus, which approximate the latent compatibility between workflows that determines fusion success.

Concretely, the scheduler first filters a small subset $\mathcal{P}' \subset \mathcal{P}_t$ of good performance workflows. We then evaluate candidate workflow groups $\mathcal{S} = \{\mathcal{W}_1, \ldots, \mathcal{W}_m\} \subset \mathcal{P}'$ using a fusion potential that jointly captures complementary coverage and behavioral consensus. For each workflow $\mathcal{W}_i$, let $\mathbf{C}_i \subseteq \mathcal{D}$ denote the subset of task instances in the dataset that $\mathcal{W}_i$ successfully solves. Complementarity and consensus are measured by both pairwise set operations within the group and group-level aggregation, and the group with the highest combined score Φ_{fuse} is selected for fusion (as illustrated in Appendix C.4).

4 Experiments

4.1 Experimental Setup

Datasets. Following the convention of previous work on automated workflow generation (Hu et al., 2024; Zhang et al., 2024), we evaluate our method on six public benchmarks, including HotpotQA (Yang et al., 2018), DROP (Dua et al., 2019), HumanEval (Chen, 2021), MBPP (Austin et al., 2021), GSM8K (Cobbe et al., 2021), and MATH (Hendrycks et al., 2021). As is common practice, these methods require a small annotated set for workflow optimization; therefore, we split each dataset into a validation set and a test set at a 1:4 ratio, using the former for optimization. Further details on the datasets and their splits are provided in Appendix A.1.

Baselines. We compare our method against two types of baselines. First, we consider a range of manually designed prompting methods, including standard IO (Direct LLM invocation), Chain-of-Thought (Wei et al., 2022), Self-Consistency with CoT (5 answers) (Wang et al., 2022), Multi-Persona Debate (Wang et al., 2024d), Self-Refine (max 3 iterations) (Madaan et al., 2023), and MedPrompt (Nori et al., 2023). Second, we compare against automated workflow optimization methods, specifically the recent approach such as ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2024).

Inference Details. We use different models for the optimization and execution phases. Specifically, we employ Claude-3-7-Sonnet (Anthropic, 2025a) as the optimizer and GPT-4o-mini (OpenAI, 2024) as the executor. All models are accessed via APIs with a temperature of 0. Further details are provided in Appendix A.2.

4.2 Main Results

Table 1 summarizes the performance of all methods across six benchmarks. API costs are reported separately in Appendix A.3.

Comparison with Manual Prompting. Overall, **FusionFlow** outperforms most manually designed prompting strategies and achieves the highest average accuracy across benchmarks. Notably, it yields substantial improvements on MATH, DROP, and MBPP. For example, on MATH, **FusionFlow** surpasses MedPrompt by +5.1 points (67.5 vs. 62.4), indicating that automatically discovered workflows can outperform static prompt designs under identical model settings.

Comparison with Automated Optimization. We further compare **FusionFlow** with automated workflow optimization baselines. Across all benchmarks, **FusionFlow** consistently achieves

Method	Benchmarks						Avg.
	HotpotQA	DROP	HumanEval	MBPP	GSM8K	MATH	
`Manual Prompting Methods`							
IO (Direct)	69.1	78.3	87.0	72.1	91.5	56.4	75.7
CoT (Wei et al., 2022)	66.6	79.4	91.6	71.6	90.3	59.3	76.5
CoT SC (5-shot) (Wang et al., 2022)	63.8	77.0	87.0	73.9	**92.5**	62.1	76.1
MultiPersona (Wang et al., 2024d)	68.8	78.5	90.8	69.5	91.4	62.6	76.9
Self-Refine (Madaan et al., 2023)	64.3	75.0	**92.4**	73.9	85.5	56.4	74.6
MedPrompt (Nori et al., 2023)	70.6	78.9	90.1	74.2	91.2	62.4	77.9
`Automated Workflow Optimization`							
ADAS (Hu et al., 2024)	74.3	75.6	85.5	70.1	88.4	52.9	74.5
AFlow (Zhang et al., 2024)	68.8	83.3	89.3	81.3	91.2	62.6	79.4
FusionFlow(ours)	**77.0**	**84.9**	90.8	**84.5**	90.8	**67.5**	**82.6**

Table 1: Performance comparison on six public benchmarks. All methods utilize `GPT-4o-mini` as the executor. We report accuracy (%) for all tasks. Best results are highlighted in bold.

Variant	Accuracy(%)
IO	56.4
CoT	59.3
w/ FusionFlow	
Optimization Only	62.6 (↑ 6.2)
+ Differentiation	61.7 (↑ 5.3)
+ Fusion	64.8 (↑ 8.4)
+ Diff. & Fusion	67.5 (↑ 11.1)

Table 2: Ablation study of **FusionFlow** components on the MATH dataset, using GPT-4o-mini as the executor model.

Backbone	Method	Accuracy (%)
GPT-4o-mini	IO	56.4
	AFlow	62.6
	Ours	**67.5**
GPT-4o	IO	63.6
	AFlow	68.1
	Ours	**74.7**
Qwen-2.5-14b	IO	60.5
	AFlow	64.4
	Ours	**68.1**
Llama-3-8b	IO	10.3
	AFlow	28.8
	Ours	**29.8**

Table 3: **Performance Comparison across Different Models.** We compare our method with IO and AFlow using four different foundation models.

stronger performance, outperforming AFlow by an average margin of +3.2 points. While AFlow remains competitive on several benchmarks, its overall performance is limited under the same optimization budget. In contrast, **FusionFlow** demonstrates more robust gains, suggesting the effectiveness of its workflow generation strategy beyond incremental refinement.

Summary of Observations. Taken together, the results in Table 1 show that **FusionFlow** delivers consistent improvements over both manually designed prompting methods and existing automated optimization approaches. These gains motivate a deeper analysis of how different optimization strategies explore the workflow space, which we examine in detail through optimization dynamics (Section 4.5) and structural depth analyses (Appendix D) in subsequent sections.

4.3 Ablation Study

To investigate the contribution of each component in **FusionFlow**, we conduct an ablation study on the MATH dataset. We define four variants: (1) **Optimization Only**: which degenerates to a standard iterative search similar to AFlow; (2) **+ Differentiation**: which enables Differentiation Operator; (3) **+ Fusion**: which enables Fusion Operator and (4) **+ Diff. & Fusion**: which activates both operators and represents the full version of **FusionFlow**.

As presented in Table 2, optimization alone quickly saturates, confirming the limited depth reachable by purely local search. Differentiation yields modest gains by increasing exploration diversity, but improvements remain constrained.

In contrast, enabling fusion leads to a substantial performance gain (+8.4 points), demonstrating that fusion is the primary driver for escaping local optima. Combining differentiation and fusion achieves the best performance, as diverse workflow variants provide complementary structures for effective fusion. These results directly support our

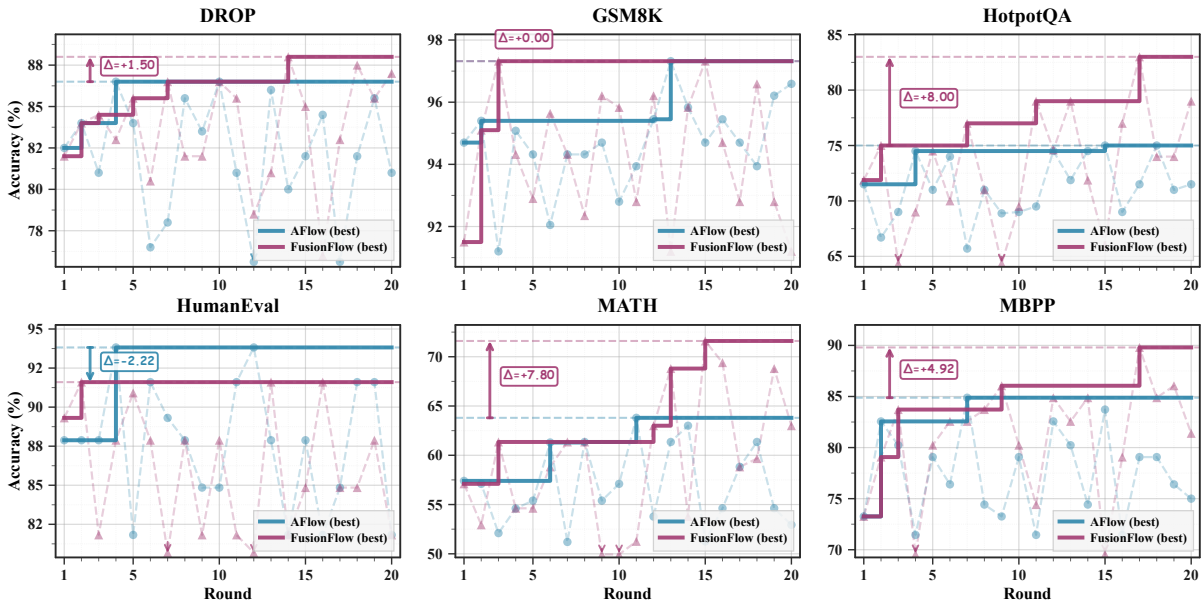

Figure 4: Validation accuracy during optimization across 20 rounds. Scatter points show the validation accuracy of individual candidate workflows at each round, while solid step curves report the best-so-far validation accuracy, which serves as the workflow selection criterion. Horizontal dashed lines indicate the final best validation accuracy achieved by each method, with Δ denoting the absolute improvement (percentage points) of **FusionFlow** over AFlow.

claim that structural exploration through fusion is an effective way to overcome depth limitations.

4.4 Robustness Across Execution Models

To evaluate robustness, we test **FusionFlow** with four different executor models on the MATH dataset: GPT-4o-mini, GPT-4o, Qwen-2.5-14b, and Llama-3-8b. As shown in Table 3, **FusionFlow** consistently outperforms Direct IO and AFlow across all executors. While absolute accuracy varies with model capability, the relative improvement over AFlow remains stable, indicating that the benefits of fusion-based structural exploration are largely model-agnostic.

4.5 Optimization Dynamics

We analyze optimization trajectories under identical iteration budgets of 20 rounds across six benchmarks. The primary metric is the **best-so-far validation accuracy** achieved across all rounds (step curves in Figure 4), since the final workflow is selected based on validation performance and evaluated only once on the test set.

As shown in Figure 4, two distinct regimes emerge. On relatively simple benchmarks (GSM8K and HumanEval), both methods converge rapidly, with best-so-far curves plateauing within a few iterations, indicating that shallow workflows are sufficient and leaving limited room for structural exploration. In contrast, on more complex benchmarks (HotpotQA, MBPP, and MATH), AFlow stagnates early, whereas **FusionFlow** continues to discover improved workflows, achieving absolute accuracy gains of +8.0%, +4.9%, and +7.8%, respectively.

Notably, **FusionFlow** exhibits higher per-round variance, which is a direct consequence of the exploratory operators (differentiation and fusion) that go beyond single-step refinement. This exploration mechanism enables **FusionFlow** to break through the structural depth limitations, escaping local optima and ultimately attaining higher best-so-far performance.

5 Conclusion

Existing automated workflow generation methods based on incremental refinement are constrained by structural depth limitations, often leading to early stagnation under finite optimization budgets. To address this issue, we propose **FusionFlow**, a framework that enables structural exploration through workflow fusion, supported by optimization, differentiation, and dynamic scheduling. Experiments on six benchmarks demonstrate consistent improvements over existing methods, showing that fusion-driven exploration is an effective approach for overcoming depth limitations in automated workflow generation.

Limitations

This work shares several limitations common to automated agentic workflow generation frameworks. First, introducing workflow optimization inevitably incurs additional computational overhead compared to directly applying a fixed model at inference time. Although our experiments adopt relatively low-cost execution models, which may lead to comparable or even lower deployment costs than using stronger models directly, the optimization stage still introduces extra expense that may be undesirable in latency- or budget-critical settings.

Second, workflow optimization can reduce flexibility and generalization across diverse task distributions. Effective workflows often require optimization on a representative subset of tasks before deployment, which limits their ability to adapt instantly to unseen or highly heterogeneous problem domains. As a result, automated workflow generation is better suited to scenarios where task distributions are relatively stable, domain specialization is desired, and low-cost models are preferred, while some degree of offline optimization or preprocessing is acceptable.

Addressing these limitations, particularly improving generalization efficiency and reducing optimization overhead, remains an important direction for future research.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2025a. Claude 3.7 sonnet and claude code.

Anthropic. 2025b. Introducing claude 4.

Jacob Austin, Augustus Odena, Maxwell I Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J Cai, Michael Terry, Quoc V Le, and 1 others. 2021. Program synthesis with large language models. corr abs/2108.07732 (2021). *arXiv preprint arXiv:2108.07732*.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. 2024. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *Advances in Neural Information Processing Systems*, 37:71596–71642.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, and 1 others. 2024. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and

Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2022. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*.

Bang Liu, Xinfeng Li, Jiayi Zhang, Jinlin Wang, Tanjin He, Sirui Hong, Hongzhang Liu, Shaokun Zhang, Kaitao Song, Kunlun Zhu, and 1 others. 2025. Advances and challenges in foundation agents: From brain-inspired intelligence to evolutionary, collaborative, and safe systems. *arXiv preprint arXiv:2504.01990*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, and 1 others. 2023. Can generalist foundation models outcompete special-purpose tuning? case study in medicine. *arXiv preprint arXiv:2311.16452*.

OpenAI. 2024. Gpt-4o mini: advancing cost-efficient intelligence.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2024a. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186.

Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, and 1 others. 2024b. Scaling large language model-based multi-agent collaboration. *arXiv preprint arXiv:2406.07155*.

Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Eleanor Jiang, Chengfei Lv, and Huajun Chen. 2024. Autoact: Automatic agent learning from scratch for qa via self-planning. *arXiv preprint arXiv:2401.05268*.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2022. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2998–3009.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, and 1 others. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yaoke Wang, Yun Zhu, Xintong Bao, Wenqiao Zhang, Suyang Dai, Kehan Chen, Wenqiang Li, Gang Huang, Siliang Tang, and Yueting Zhuang. 2024c. Meta-reflection: A feedback-free reflection learning framework. *arXiv preprint arXiv:2412.13781*.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2024d. 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 (Volume 1: Long Papers)*, pages 257–279.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. 2023. Expertprompting: Instructing large language models to be distinguished experts. *arXiv preprint arXiv:2305.14688*.

Shicheng Xu, Liang Pang, Huawei Shen, Xueqi Cheng, and Tat-Seng Chua. 2024. Search-in-the-chain: Interactively enhancing large language models with search for knowledge-intensive tasks. In *Proceedings of the ACM Web Conference 2024*, pages 1362–1373.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

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

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2024. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

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

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V Le, Ed Chi, Denny Zhou, Swaroop Mishra, and Huaixiu Steven Zheng. 2024. Self-discover: Large language models self-compose reasoning structures. *Advances in Neural Information Processing Systems*, 37:126032–126058.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, Huajun Chen, and Ningyu Zhang. 2024. Knowagent: Knowledge-augmented planning for llm-based agents. *arXiv preprint arXiv:2403.03101*.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, and 1 others. 2023. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*.

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

A Appendix: Datasets and Experimental Setup

In this appendix, we provide a detailed account of the datasets, data partitioning, and inference details in our experiments.

A.1 Dataset Descriptions

We conducted our experiments on six widely-used public benchmarks spanning a diverse range of reasoning tasks, including question answering, code generation, and mathematical problem-solving. For each dataset, we used a subset of instances, which were partitioned into a validation set for workflow optimization and a test set for final performance evaluation.

HotpotQA (Yang et al., 2018) A multi-hop question answering dataset that requires finding and reasoning over multiple supporting documents to answer a question.

- **Task Type**: Multi-hop QA
- **Data Source**: 1,000 instances randomly sampled from the original test set.
- **Data Split**: 200 instances for the validation set and 800 for the test set.

DROP (Dua et al., 2019) A challenging reading comprehension benchmark requiring discrete reasoning over paragraphs, such as addition, counting, or sorting.

- **Task Type**: Reading Comprehension
- **Data Source**: 1,000 instances randomly sampled from the original development (dev) set.
- **Data Split**: 200 instances for the validation set and 800 for the test set.

HumanEval (Chen, 2021) A code generation dataset comprising 164 handwritten programming problems. Each problem includes a function signature, docstring, body, and several unit tests.

- **Task Type**: Code Generation
- **Data Source**: The full original test set.
- **Data Split**: 33 instances for the validation set and the remaining 131 for the test set, following prior work.

MBPP (Mostly Basic Python Problems) (Austin et al., 2021) An entry-level Python programming dataset consisting of around 500 crowd-sourced tasks, each with a short description, a code solution, and three test cases.

- **Task Type**: Code Generation
- **Data Source**: The full original test set.
- **Data Split**: 86 instances for the validation set and the remaining 341 for the test set.

GSM8K (Cobbe et al., 2021) A dataset of high-quality, grade-school math word problems that require a sequence of elementary arithmetic operations to solve.

- **Task Type**: Math Word Problems
- **Data Source**: The full original test set.
- **Data Split**: 264 instances for the validation set and 1055 for the test set.

MATH (Hendrycks et al., 2021) A challenging dataset of mathematics competition problems. Our experiments focus on a curated subset of 605 problems from four specific domains-Combinatorics & Probability, Number Theory, Pre-algebra, and Pre-calculus-all at difficulty level 5.

- **Task Type**: Advanced Mathematics
- **Data Source**: A curated subset of 605 problems at difficulty level 5.
- **Data Split**: 119 instances for the validation set and 486 for the test set.

A.2 Inference Details

We use different models for the optimization and execution phases. Specifically, we employ Claude-3-7-Sonnet (Anthropic, 2025a) as the optimizer and GPT-4o-mini (OpenAI, 2024) as the executor. All models are accessed via APIs with a temperature of 0. For our method and AFlow, we set the maximum number of optimization iterations to 20. This setting follows the standard configuration in prior work and ensures a fair comparison under limited optimization budgets. For the manually designed prompting methods, we use the same executor model (GPT-4o-mini) to ensure a fair comparison. This design choice is intentional: we leverage a more powerful model for the optimizer, which is called infrequently, and a more cost-effective model for the executor, which is invoked for every task step. This strategy allows us to achieve high-quality workflow discovery while controlling the overall inference cost.

A.3 API Cost Analysis

This section provides a breakdown of the total API costs incurred for running the complete **FusionFlow** and **AFlow** optimization process (20 iterations) on each of the six benchmark datasets. The costs, summarized in Table 4, are reported in US Dollars ($) and reflect the expenditure for the entire workflow discovery phase.

Table 4: Total API cost comparison between **FusionFlow** and AFlow.

Dataset	FusionFlow ($)	AFlow ($)
HotpotQA	5.8	6.7
DROP	6.5	4.2
HumanEval	3.0	3.3
MBPP	4.7	4.1
GSM8K	14.8	15.7
MATH	18.4	16.0

It is important to note that these figures represent the one-time cost for the workflow discovery and optimization phase. Once the optimal workflow is generated, the inference cost for solving individual tasks using the more economical executor model (GPT-4o-mini) is substantially lower.

B Theoretical Analysis

This appendix provides theoretical insights that motivate the design of **FusionFlow**. Our analysis

clarifies (i) why incremental optimization alone is insufficient under finite budgets, (ii) why workflow fusion induces different search dynamics, and (iii) why scheduling and candidate selection are necessarily post-hoc and heuristic in nature.

B.1 Problem Structure and Search Space

Let $\mathbb{S}$ denote the space of all executable agentic workflows. Each workflow $\mathcal{W} \in \mathbb{S}$ is represented as a directed control-flow graph composed of atomic operators and explicit coordination logic. We denote by $d(\mathcal{W})$ the **structural depth** of a workflow, defined as the number of effective modifications it contains. Each workflow is associated with a modification-node set $\mathcal{N}(\mathcal{W})$, and its depth is $d(\mathcal{W}) = |\mathcal{N}(\mathcal{W})|$. The formal recursive definition and detailed analysis are provided in Appendix D.

The automated workflow generation problem can be viewed as a search process over $\mathbb{S}$ under a finite iteration budget T. At each iteration, the search applies a transformation operator that maps one or more existing workflows to new candidates. Crucially, the evaluation of a workflow's utility is only observable through execution feedback on a finite dataset, making the search process partially observable and non-stationary.

B.2 Limitations of Incremental Optimization

We first analyze the limitations of purely incremental optimization. Optimization operators modify a workflow through local edits, such as prompt refinement, parameter tuning, or small structural adjustments. Let $\mathcal{O}_{\text{opt}}$ denote such an operator.

Observation 1 (Locality of Optimization). Each application of $\mathcal{O}_{\text{opt}}$ induces only a bounded structural change to the workflow graph. Consequently, under a finite budget T, the structural depth achievable through repeated optimization is at most linear in T.

This implies that even if optimization consistently improves performance, it explores the workflow space along a single trajectory, accumulating depth gradually. In practice, this leads to early saturation: once local refinements exhaust easily accessible improvements, further iterations yield diminishing returns.

B.3 Differentiation as Width Expansion

Differentiation operators address this limitation by explicitly increasing exploration width. Given a parent workflow, differentiation generates specialized variants that target different task subspaces.

From a search perspective, differentiation expands the frontier of exploration by creating multiple distinct trajectories. However, differentiation alone does not increase structural depth: each differentiated workflow remains confined to a similar depth regime as its parent.

Observation 2 (Width without Depth). Differentiation increases diversity across trajectories but does not, by itself, induce super-linear depth growth. Under finite budgets, it primarily redistributes exploration rather than enabling deeper compositions.

Thus, while differentiation is essential for discovering complementary behaviors, it cannot overcome depth limitations on its own.

B.4 Fusion as a Non-local Search Operator

We now turn to fusion, which constitutes the central mechanism of **FusionFlow**. Let $\mathcal{O}_{\text{fuse}}$ denote the fusion operator that maps a group of workflows $\mathcal{S} = \{\mathcal{W}_1, \ldots, \mathcal{W}_m\}$ to a new workflow $\mathcal{W}_{\text{fuse}}$.

Observation 3 (Non-local Transition). Fusion induces a non-local transition in the workflow space by composing multiple independently evolved substructures into a single executable graph.

Specifically, if each $\mathcal{W}_i$ contains a mature subgraph discovered along a distinct optimization trajectory, fusion materializes these subgraphs simultaneously. Under our depth definition in Appendix D.1, each workflow $\mathcal{W}$ is associated with a modification-node set $\mathcal{N}(\mathcal{W})$ and $d(\mathcal{W}) = |\mathcal{N}(\mathcal{W})|$. For a fusion executed at round t, the recursive construction gives $\mathcal{N}(\mathcal{W}_{\text{fuse}}) = \bigcup_i \mathcal{N}(\mathcal{W}_i) \cup \{t\}$. Therefore,

$$d(\mathcal{W}_{\text{fuse}}) = |\mathcal{N}(\mathcal{W}_{\text{fuse}})| = \left| \bigcup_i \mathcal{N}(\mathcal{W}_i) \cup \{t\} \right| \geq \max_i |\mathcal{N}(\mathcal{W}_i)| = \max_i d(\mathcal{W}_i). \tag{7}$$

Moreover, the fusion step introduces new coordination logic (captured by the fresh fusion node $\{t\}$ in Appendix D.1), and the union can be strictly larger than any single parent when the parents contribute complementary modifications, in which case $d(\mathcal{W}_{\text{fuse}})$ strictly exceeds all parent depths.

Implication. Under a fixed iteration budget, fusion enables discrete depth expansion that cannot

be achieved through sequential local edits alone. It effectively collapses multiple exploration trajectories into a single step, bypassing intermediate states that would otherwise require many rounds of optimization.

This property distinguishes fusion from both optimization and differentiation.

B.5 Why Fusion Is Feasible but Not Trivial

Although fusion induces powerful search transitions, it is not difficult to imply from an implementation perspective. Agentic workflows are composed of atomic operators connected by explicit, code-level control logic. This structure allows a large language model to synthesize a composite workflow by identifying shared components and reconciling complementary substructures.

However, effective fusion is not guaranteed. Arbitrarily fusing workflows may result in redundancy, incompatible coordination, or degraded performance. Thus, the challenge of fusion lies not in feasibility, but in selectivity.

B.6 Post-hoc Nature of Fusion Scheduling

A central difficulty in fusion-based search is deciding *when* to apply fusion and *which* workflows to fuse.

The utility of fusing a candidate group depends on latent properties such as structural compatibility and complementary task coverage. These properties are only observable through execution feedback and cannot be inferred reliably a priori.

B.7 Theoretical Motivation and Practical Approximation

This section justifies why **FusionFlow** adopts a feedback-driven scheduling policy. Our key claim is that workflow scheduling is inherently a sequential decision problem under partial observability: each operator choice not only changes the current best score, but also reshapes the future search frontier (what workflows exist, what can be fused, and what information becomes available).

Scheduling as a POMDP. Let the search process maintain a workflow pool $\mathcal{P}_t$ at iteration t. The scheduler chooses an operator (and corresponding targets) to apply, producing $\mathcal{P}_{t+1}$ and new evaluation feedback. This can be abstracted as a POMDP:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, \Omega, r, \gamma), \tag{8}$$

where the latent state $s_t \in \mathcal{S}$ summarizes the unobserved properties of the current search frontier (e.g., latent structural compatibility between workflows, potential synergy from fusion, and the "distance" to high-performing regions). The action $a_t \in \mathcal{A}$ corresponds to selecting an operator (Optimization/Differentiation/Fusion) and its operands (selected workflow(s) or group). After executing a_t, the process transitions according to an unknown dynamics $\mathcal{T}$, and produces an observation $o_t \in \mathcal{O}$, which includes measurable statistics such as best score R_t, per-workflow coverage sets $\mathbf{C}_i$, and execution logs. The scheduler only observes $o_t \sim \Omega(\cdot \mid s_t, a_{t-1})$ and thus cannot directly compute an optimal policy over s_t. Importantly, the consequence of an operator is intrinsically multi-step: a differentiation operator primarily increases future fusion upside by creating complementary specialists, and a fusion operator becomes beneficial after sufficient maturation of its parents. This induces a long-horizon credit assignment problem, where the value of an action depends on how it alters future workflow pools and future available compositions, which is exactly the dependency that a POMDP formulation makes explicit.

Two practical properties make the process additionally challenging: (i) the transition is non-stationary because the pool distribution changes as new workflows are generated; (ii) the reward is expensive and delayed since any operator's effect is only known after execution evaluation. Under these conditions, an effective scheduler must depend on history feedback. This motivates a feedback-driven policy class that uses lightweight sufficient statistics extracted from o_t to approximate the decision-making signal required by the underlying POMDP.

Exploration–Exploitation View: Operators as Arms in a Contextual Bandit. At a coarse level, the scheduler repeatedly chooses among operator types, which can be viewed as a contextual bandit with three arms:

$$a_t \in \{\mathcal{O}_{\text{opt}}, \mathcal{O}_{\text{diff}}, \mathcal{O}_{\text{fuse}}\}, \tag{9}$$

where the context is the observation o_t (recent score trajectory, pool diversity, coverage patterns). Optimization corresponds to exploitation (local improvements), while differentiation and fusion correspond to exploration in structure space. Under finite budgets, an effective policy should allocate more probability mass to exploration when the marginal returns of exploitation diminish.

Stagnation as a Proxy for Negative Advantage. In a rigorous RL setting, action selection is of-

ten driven by an advantage function $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$, which indicates whether a specific action improves upon the baseline value. However, since the latent state s_t is unobservable and computing Q^π requires prohibitive sampling costs, we seek a computationally tractable estimator. We observe that when the marginal return of local optimization diminishes, the advantage of exploitation decreases relative to exploration. Therefore, we employ the measurable improvement rate Δ_t (defined below) as an empirical estimator for the advantage of repeated local refinement. Let

$$R_t^{\text{recent}} = \max_{u \in [t-k+1,\, t]} R_u, \tag{10}$$

$$R_t^{\text{prev}} = \max_{u \in [t-2k+1,\, t-k]} R_u, \tag{11}$$

$$\Delta_t = R_t^{\text{recent}} - R_t^{\text{prev}}. \tag{12}$$

When Δ_t is small or negative, the empirical advantage of repeated local refinement becomes weak, suggesting that the expected gain of $\mathcal{O}_{\text{opt}}$ is decreasing. We therefore define a monotone transformation into a stagnation signal:

$$\pi_t = \sigma(-\kappa \Delta_t) = \frac{1}{1 + \exp(\kappa \Delta_t)}, \tag{13}$$

which can be interpreted as a *soft gating mechanism*, approximating a stochastic policy that shifts probability mass toward high-variance structural exploration (Fusion/Differentiation) when the estimated advantage of exploitation vanishes.

Why Fusion Should be Scheduled More Aggressively than Generic Exploration. Differentiation and fusion both expand exploration, but fusion uniquely induces non-local transitions by composing mature substructures. In the POMDP view, fusion is a high-variance, potentially high-reward action whose payoff depends on latent compatibility between workflows. It is therefore sensible to schedule fusion under two conditions: (i) *exploitation plateau* (high π_t), when local search returns diminish; (ii) *candidate maturity*, when the pool contains sufficiently competent and diverse workflows that make positive fusion payoff more likely. This matches standard RL intuitions: reserve high-variance actions for states where the baseline action underperforms, and when the context indicates high upside.

Connection to Tree Search and Mixed-Probability Selection. Prior automated workflow search frameworks have also adopted RL-inspired selection mechanisms to balance exploration and exploitation, e.g., Monte Carlo Tree Search variants and mixed-probability selection rules. Such designs empirically validate the necessity of stochastic, feedback-driven scheduling in large, expensive, and partially observable search spaces. (Zhang et al., 2024) uses MCTS-style selection and execution feedback to guide workflow expansion, which is conceptually aligned with our view that operator scheduling should be adaptive and history-dependent rather than fixed.

Takeaway. In summary, **FUSIONFLOW**'s scheduling is theoretically motivated as a tractable instantiation of decision-making under partial observability and expensive feedback: (1) model the process as a POMDP / contextual bandit; (2) use measurable score momentum Δ_t as a proxy for exploitation advantage; and (3) increase the probability of high-upside structural actions (especially fusion) when exploitation stagnates and candidate maturity is satisfied. This provides a principled justification for our lightweight, feedback-driven scheduling mechanism.

Observation 4 (A Posteriori Decision Problem). Fusion scheduling constitutes an a posteriori decision problem under partial observability. No fixed or optimal fusion policy can be defined in advance without executing the fused workflow itself.

As a result, scheduling must rely on indirect signals, such as performance stagnation, as proxies for diminishing returns from local refinement. This motivates the use of lightweight, feedback-driven scheduling mechanisms rather than explicit policy optimization.

B.8 Rationale for Heuristic Candidate Selection

Similarly, selecting fusion or differentiation candidates involves balancing complementary objectives-coverage, consensus, and computational tractability-under incomplete information.

Attempting exhaustive or optimal selection is infeasible due to combinatorial complexity and the cost of evaluation. Instead, **FUSIONFLOW** adopts heuristic scoring functions that are grounded in observable execution statistics, such as task-level coverage sets.

Observation 5 (Principled Heuristics). Although heuristic, the candidate selection mechanisms in **FUSIONFLOW** are principled in the sense that they directly operationalize the structural objectives required for effective fusion and differentiation.

B.9 Summary

Taken together, this analysis explains why **FUSIONFLOW** combines optimization, differentiation, and fusion under a dynamic, feedback-driven scheduling framework. Incremental optimization alone is insufficient for deep exploration under finite budgets. Differentiation expands exploration width but not depth. Fusion enables non-local depth expansion but requires careful, post-hoc control.

The design of **FUSIONFLOW** is therefore not ad hoc, but a direct response to the structural constraints of the workflow search problem.

C Detailed Algorithmic Description

This appendix provides a complete and formal specification of the algorithmic components of **FUSIONFLOW**, including execution feedback representation, dynamic scheduling, workflow selection for differentiation, fusion candidate selection, and fusion execution. The notation and formulation here are fully consistent with the main text.

C.1 Workflow Pool and Execution Feedback

At each iteration t, **FUSIONFLOW** maintains a workflow pool

$$\mathcal{P}_t = \{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_{|\mathcal{P}_t|}\}, \tag{14}$$

where each $\mathcal{W}_i$ is a fully executable agentic workflow.

Each workflow is evaluated on the task dataset

$$\mathcal{D} = \{(x_j, y_j)\}_{j=1}^{|\mathcal{D}|}, \tag{15}$$

using a task-specific evaluation metric $G(\cdot)$. The execution feedback of workflow $\mathcal{W}_i$ is summarized by:

Scalar Performance Score

$$R_i = \mathbb{E}_{(x,y)\sim\mathcal{D}}[G(f_{\mathcal{W}_i}(x), y)], \tag{16}$$

which reflects the overall task performance of the workflow.

Task-level Coverage Set

$$C_i \subset \mathcal{D}, \tag{17}$$

where C_i denotes the subset of task instances in $\mathcal{D}$ that $\mathcal{W}_i$ successfully solves. The set C_i serves as a discrete approximation of the functional coverage of the workflow and forms the basis for fusion candidate selection.

C.2 Global Scope of Dynamic Scheduling

The dynamic scheduling mechanism operates over all transformation operators in **FUSIONFLOW**, including optimization, differentiation, and fusion. At each iteration, the scheduler determines which operator to apply based on recent performance dynamics and operator usage history.

Since workflow fusion constitutes the core mechanism for inducing non-local structural transitions, the main text focuses on fusion-related scheduling. This appendix presents the complete scheduling formulation.

C.3 Performance Stagnation Signal

Let

$$R_t = \max_{\mathcal{W}_i \in \mathcal{P}_t} R_i \tag{18}$$

denote the best performance achieved at iteration t.

To detect diminishing returns from local refinement, we compute two sliding-window statistics with window size k:

$$\begin{aligned} R_t^{\text{recent}} &= \max_{u\in[t-k+1,\,t]} R_u, \\ R_t^{\text{prev}} &= \max_{u\in[t-2k+1,\,t-k]} R_u. \end{aligned} \tag{19}$$

The improvement difference is defined as

$$\Delta_t = R_t^{\text{recent}} - R_t^{\text{prev}}. \tag{20}$$

We map this quantity to a continuous stagnation signal:

$$\pi_t = \frac{1}{1 + \exp(\kappa \cdot \Delta_t)}, \tag{21}$$

where $\kappa > 0$ controls sensitivity to performance stagnation. Larger values of π_t indicate reduced marginal gains from local refinement.

The signal π_t is not treated as a decision rule, but as a soft indicator that biases the scheduler toward exploration-oriented operators.

C.4 Workflow Selection Mechanism

Both differentiation and fusion workflows are essentially selection processes driven by distinct potential algorithms. We unify their formulations based on the set of correctly solved tasks. Let $\mathbf{C}_i \subseteq \mathcal{D}$ denote the set of task instances correctly processed by workflow $\mathcal{W}_i$.

C.4.1 Differentiation Candidate Selection

Differentiation aims to select a parent workflow $\mathcal{W}_i$ that exhibits high potential for specializing in a specific task cluster k.

Task Partition. The dataset $\mathcal{D}$ is partitioned into disjoint clusters $\mathcal{K} = \{k_1, \ldots, k_{|\mathcal{K}|}\}$. For any cluster $k \in \mathcal{K}$, the local correct set of workflow $\mathcal{W}_i$ is defined as:

$$\mathbf{C}_{i,k} = \mathbf{C}_i \cap \mathcal{D}_k. \tag{22}$$

Performance Metrics. We define the global accuracy rate R_{global} and the local cluster recall rate R_{local} as:

$$\begin{aligned} R_{\text{global}}(\mathcal{W}_i) &= \frac{|\mathbf{C}_i|}{|\mathcal{D}|}, \\ R_{\text{local}}(\mathcal{W}_i, k) &= \frac{|\mathbf{C}_{i,k}|}{|\mathcal{D}_k|}. \end{aligned} \tag{23}$$

A workflow is a valid candidate for cluster k only if $R_{\text{local}}(\mathcal{W}_i, k) > R_{\text{global}}(\mathcal{W}_i)$.

Differentiation Potential. The potential focuses on the absolute volume of specialized tasks. We define the differentiation potential Φ_{diff} for workflow $\mathcal{W}_i$ as the maximum specialization gain across all clusters:

$$\begin{aligned} \Phi_{\text{diff}}(\mathcal{W}_i) = \max_{k \in \mathcal{K}} \Bigg[\frac{|\mathbf{C}_{i,k}|}{|\mathcal{D}|} \cdot \big(R_{\text{local}}(\mathcal{W}_i, k) \\ - R_{\text{global}}(\mathcal{W}_i)\big) \Bigg]_+ . \end{aligned} \tag{24}$$

where $[\cdot]_+ = \max(0, \cdot)$.

C.4.2 Fusion Candidate Selection

Fusion aims to select a group of workflows $\mathcal{S} = \{\mathcal{W}_1, \ldots, \mathcal{W}_m\}$ that maximizes the joint potential defined by complementarity and consensus.

Set-Based Metrics. We evaluate the group $\mathcal{S}$ using two fundamental set operations on their correct sets:

- **Complementarity (Union):** Represents the collective coverage.

$$U(\mathcal{S}) = \left| \bigcup_{\mathcal{W}_i \in \mathcal{S}} \mathbf{C}_i \right|. \tag{25}$$

- **Consensus (Intersection):** Represents the behavioral consistency.

$$I(\mathcal{S}) = \left| \bigcap_{\mathcal{W}_i \in \mathcal{S}} \mathbf{C}_i \right|. \tag{26}$$

Ideally, a fusion group should maximize the union (solving more problems) while maintaining a sufficient intersection (agreement on ground truths).

Fusion Potential. The fusion potential Φ_{fuse} combines the set-based metrics. To capture both pair-wise interactions and group-wise properties, we formulate:

$$\begin{aligned} \Phi_{\text{fuse}}(\mathcal{S}) = \big(\lambda_1 \bar{U}_{\text{pair}} + \lambda_2 U(\mathcal{S})\big) + \\ \big(\mu_1 \bar{I}_{\text{pair}} + \mu_2 I(\mathcal{S})\big) \cdot \eta(\mathcal{S}). \end{aligned} \tag{27}$$

where $\bar{U}_{\text{pair}}$ and $\bar{I}_{\text{pair}}$ are the average pairwise union and intersection sizes within $\mathcal{S}$, λ, μ are weighting coefficients, and $\eta(\mathcal{S})$ is a penalty term for group complexity.

The group maximizing the potential is selected:

$$\mathcal{S}^* = \arg\max_{\mathcal{S} \subset \mathcal{P}'} \Phi_{\text{fuse}}(\mathcal{S}). \tag{28}$$

C.5 Fusion Execution

Given a selected group $\mathcal{S}$, fusion is performed by prompting a large language model to synthesize a composite workflow that preserves complementary substructures, reconciles shared components, and introduces coordination logic.

The resulting workflow $\mathcal{W}_{\text{fuse}}$ is validated and added to the pool:

$$\mathcal{P}_{t+1} \leftarrow \mathcal{P}_t \cup \{\mathcal{W}_{\text{fuse}}\}. \tag{29}$$

C.6 Computational Considerations

Although both differentiation and fusion involve combinatorial elements, practical complexity is controlled through coverage-based filtering, bounded group size, and probabilistic selection. In practice, these mechanisms keep the overhead of structure-level operations negligible relative to workflow evaluation.

D Empirical Analysis of structural depth

This section provides an empirical analysis of the **structural depth** achieved by different workflow generation methods. Intuitively, depth captures how many *effective structural changes* have been accumulated into a workflow. The key empirical

claim is that **fusion** can aggregate effective changes produced in different branches, thereby overcoming the depth limitations of incremental optimization approaches.

D.1 Depth Definition and Computation

We define the **structural depth** of a workflow $\mathcal{W}$ as the number of effective modifications it contains. Each workflow can be represented as a set of modification nodes, where each node corresponds to an optimization or fusion operation applied during the search process.

Definition 1 (Structural Depth)**.** *Let $\mathcal{N}(\mathcal{W})$ denote the set of modification nodes contained in workflow $\mathcal{W}$, and $d(\mathcal{W}) = |\mathcal{N}(\mathcal{W})|$ be its depth. The node set is computed recursively:*

- ***Root workflow**: $\mathcal{N}(\mathcal{W}_{root}) = \{r\}$, where r is the root node.*
- ***Optimization**: $\mathcal{N}(\mathcal{W}_{opt}) = \mathcal{N}(\mathcal{W}_{parent}) \cup \{t\}$, where t is the current round.*
- ***Differentiation**: $\mathcal{N}(\mathcal{W}_{diff}) = \mathcal{N}(\mathcal{W}_{parent})$ (no new modification).*
- ***Fusion**: $\mathcal{N}(\mathcal{W}_{fuse}) = \bigcup_i \mathcal{N}(\mathcal{W}_i) \cup \{t\}$, where $\{\mathcal{W}_i\}$ are parent workflows.*

This definition captures the intuition that:

- Each workflow represents a tree of modifications in the search space, where each node is a chain from the root.
- Optimization adds one new modification node to the parent's tree.
- Differentiation creates specialized variants without introducing new modifications (breadth expansion).
- Fusion merges multiple independently evolved trees into a single tree (taking the union), plus the fusion operation itself. This enables aggregation of modifications from different evolutionary branches.

D.2 Depth as Structural Capacity under Finite Budgets

It is important to clarify the interpretation and limitations of the structural depth metric used in this analysis. The depth defined here does not directly measure the semantic quality or functional optimality of a workflow. Instead, it characterizes the maximum structural capacity that a workflow can embody under a finite optimization budget.

Under a fixed number of optimization rounds, exploratory operations such as differentiation are unavoidable in complex search spaces, as they are necessary to discover diverse and potentially complementary solution strategies. However, such exploration incurs an opportunity cost: once the budget is consumed by branching into multiple trajectories, purely incremental methods that refine a single workflow cannot subsequently integrate the effective modifications discovered along different branches into a single executable structure. As a result, the maximum structural complexity that any individual workflow can attain is inherently bounded by the depth of a single optimization chain.

Fusion addresses this limitation by enabling ex post aggregation of independently evolved workflows. By integrating modification histories from multiple search trajectories, fusion allows the optimization process to recover and consolidate structural information that would otherwise remain fragmented across branches. In this sense, fusion does not guarantee that the resulting workflow is intrinsically more effective or semantically superior, but it expands the space of structurally realizable workflows that can be reached within a fixed iteration budget.

In **FUSIONFLOW**, this increased structural capacity is further guided by optimization, differentiation, and dynamic scheduling, which are informed by empirical observations of task structure (e.g., sub-domain specialization in the dataset). While deeper workflows are not inherently better, this coordinated process increases the likelihood that workflows with higher structural capacity can represent and support more complex reasoning behaviors when such structure is required by the task distribution.

D.3 Depth Accumulation and Structure-Aligned Exploration

While increased structural depth alone does not guarantee improved workflow quality, the manner in which depth is accumulated plays a critical role. In purely incremental optimization, additional modifications are applied along a single trajectory, often reflecting local adjustments whose composition may be incidental rather than structurally meaningful. As a result, deeper workflows obtained through such processes do not necessarily correspond to

Table 5: **structural depth** comparison between AFlow and **FusionFlow** on MBPP dataset. Depth is defined as the number of effective modifications. Fusion operations (marked with †) aggregate modifications from multiple branches.

	AFlow	FusionFlow		
Round	**Depth**	**Depth**	**Op.**	**Modification Set**
1	1	1	Root	$\{1\}$
2	2	2	Opt	$\{1, 2\}$
3	3	3	Opt	$\{1, 2, 3\}$
4	2	2	Diff	$\{1, 2\}$
5	3	2	Diff	$\{1, 2\}$
6	2	4	Fuse†	$\{1, 2, 3, 6\}$
7	4	3	Opt	$\{1, 2, 7\}$
8	2	3	Opt	$\{1, 2, 8\}$
9	2	4	Opt	$\{1, 2, 8, 9\}$
10	3	2	Diff	$\{1, 2\}$
11	4	5	Opt	$\{1, 2, 8, 9, 11\}$
12	3	5	Opt	$\{1, 2, 8, 9, 12\}$
13	3	6	Fuse†	$\{1, 2, 3, 8, 9, 13\}$
14	3	5	Opt	$\{1, 2, 8, 9, 14\}$
15	3	7	Fuse†	$\{1, 2, 3, 8, 9, 14, 15\}$
16	3	5	Opt	$\{1, 2, 8, 9, 16\}$
17	3	2	Diff	$\{1, 2\}$
18	2	7	Fuse†	$\{1, 2, 3, 8, 9, 14, 18\}$
19	4	6	Opt	$\{1, 2, 8, 9, 12, 19\}$
20	4	5	Opt	$\{1, 2, 8, 9, 20\}$
Max	4	7	–	–

more effective reasoning structures.

In contrast, **FusionFlow** accumulates depth through an explicit differentiation–fusion pattern that mirrors a natural problem-solving strategy: exploring diverse solution variants first, and subsequently integrating complementary components into a unified workflow. Differentiation encourages specialization across branches, allowing distinct substructures to emerge under different inductive biases or task sub-distributions. Fusion then selectively aggregates these independently evolved structures, resulting in deeper workflows whose modifications originate from semantically distinct yet complementary search trajectories.

Under this paradigm, depth growth reflects not merely the accumulation of more changes, but the consolidation of diverse structural insights discovered across the search space. Although such depth remains an upper-bound measure of structural capacity rather than a direct indicator of quality, aligning depth accumulation with the compositional structure of natural tasks increases the likelihood that deeper workflows correspond to more effective reasoning pipelines when such structure is required.

D.4 Depth Comparison: AFlow vs. FusionFlow

We compare the **structural depth** trajectories of AFlow (optimization-only) and **FusionFlow** (with fusion) on the MBPP dataset over 20 optimization rounds. The results are presented in Table 5 and visualized in Figure 5.

Key Observations.

1. **Modification Aggregation**: AFlow's depth is bounded by the maximum chain length in the search tree, reaching only depth 4. **FusionFlow** achieves a maximum depth of 7 by aggregating modifications from multiple evolutionary branches.

2. **Fusion as Branch Merger**: Fusion takes the union of modification sets from parent workflows, effectively combining insights from different search trajectories. For example:

   - Round 6: Fuses workflows from rounds 2, 3, 5, merging their modification histories into $\{1, 2, 3, 6\}$.
   - Round 15: Fuses rounds 3, 9, 14, combining $\{1, 2, 3\} \cup \{1, 2, 8, 9\} \cup \{1, 2, 8, 9, 14\} \cup \{15\} = \{1, 2, 3, 8, 9, 14, 15\}$.

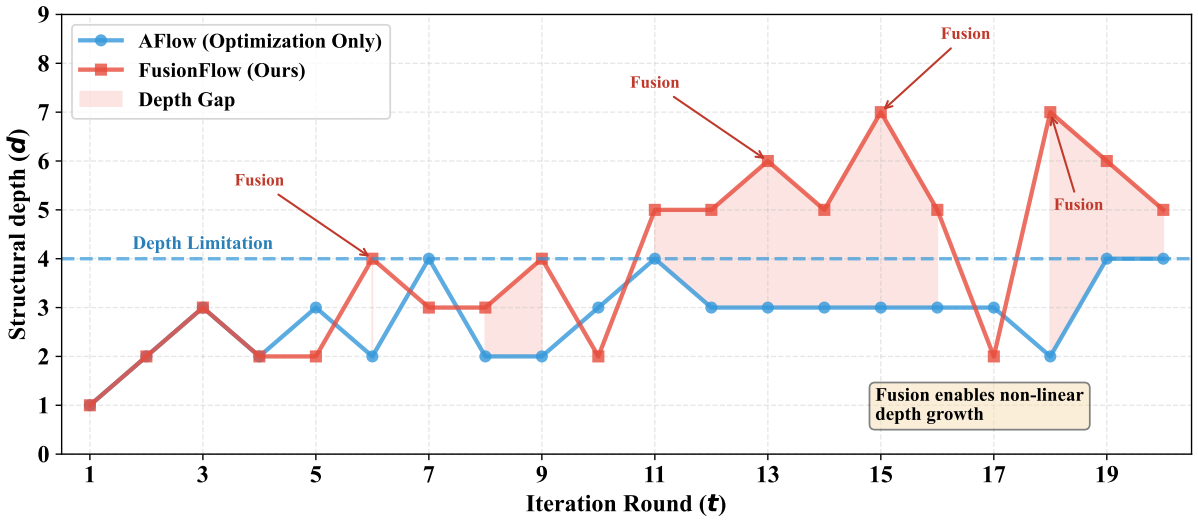

Figure 5: *Structural depth* comparison between AFlow (blue) and **FusionFlow** (red) over 20 optimization rounds on the MBPP dataset. Depth is defined as the number of effective modifications contained in each workflow. Fusion operations (annotated) aggregate modifications from multiple evolutionary branches, enabling **FusionFlow** to reach deeper regions of the modification space.

- Round 18: Fuses rounds 9, 14, 3, yielding $\{1, 2, 3, 8, 9, 14, 18\}$.

3. **Depth Advantage**: The maximum depth ratio is $7/4 = 1.75\times$, demonstrating that fusion enables access to deeper regions of the modification space by combining complementary evolutionary branches.

4. **Differentiation Preserves Diversity**: Differentiation operations (rounds 4, 5, 10, 17) create low-depth specialized variants that serve as complementary building blocks for subsequent fusion, without introducing new modifications.

E Pseudocode

E.1 Global Dynamic Scheduling Algorithm

Table 6: Global Dynamic Scheduling Algorithm

Algorithm 1: Global Dynamic Scheduling Algorithm

Input: Initial workflow pool $\mathcal{P}_0$, Iteration budget T, Dataset $\mathcal{D}$
Parameters: Window size k, Sensitivity κ
Output: Optimized workflow pool $\mathcal{P}_{T+1}$

Initialize $\mathcal{P}_1 \leftarrow \mathcal{P}_0$
for $t = 1$ **to** T **do**
  Evaluate all $\mathcal{W}_i \in \mathcal{P}_t$ on $\mathcal{D}$, obtain R_i and coverage sets $\mathbf{C}_i$
  $R_t \leftarrow \max_{\mathcal{W}_i \in \mathcal{P}_t} R_i$ ▷ Compute best score
  if $t \geq 2k$ **then** ▷ Check sufficient history
    $R_t^{\text{recent}} \leftarrow \max_{u \in [t-k+1,t]} R_u$
    $R_t^{\text{prev}} \leftarrow \max_{u \in [t-2k+1,t-k]} R_u$
    $\Delta_t \leftarrow R_t^{\text{recent}} - R_t^{\text{prev}}$ ▷ Compute performance gap
    $\pi_t \leftarrow \frac{1}{1+\exp(\kappa \cdot \Delta_t)}$ ▷ Sigmoid stagnation signal
  else
    $\pi_t \leftarrow 0$
  end if
  Sample operator $\mathcal{O}_t \in \{\mathcal{O}_{\text{opt}}, \mathcal{O}_{\text{diff}}, \mathcal{O}_{\text{fuse}}\}$ ▷ Sample operator
  if $\mathcal{O}_t = \mathcal{O}_{\text{fuse}}$ **then** ▷ Apply fusion branch
    $\mathcal{S}^\star \leftarrow \text{FusionWorkflowSelection}(\mathcal{P}_t)$ ▷ Algorithm 2
    $\mathcal{W}_{\text{fuse}} \leftarrow \mathcal{O}_{\text{fuse}}(\mathcal{S}^\star)$
    $\mathcal{P}_{t+1} \leftarrow \mathcal{P}_t \cup \{\mathcal{W}_{\text{fuse}}\}$
  else if $\mathcal{O}_t = \mathcal{O}_{\text{diff}}$ **then** ▷ Apply differentiation branch
    $(\mathcal{W}^\star, k^\star) \leftarrow \text{DifferentiationWorkflowSelection}(\mathcal{P}_t, \{\mathcal{D}_k\})$ ▷ Algorithm 3
    $\mathcal{W}_{\text{diff}} \leftarrow \mathcal{O}_{\text{diff}}(\mathcal{W}^\star, k^\star)$
    $\mathcal{P}_{t+1} \leftarrow \mathcal{P}_t \cup \{\mathcal{W}_{\text{diff}}\}$
  else ▷ Apply optimization branch
    Select $\mathcal{W} \in \mathcal{P}_t$ for local optimization
    $\mathcal{W}_{\text{opt}} \leftarrow \mathcal{O}_{\text{opt}}(\mathcal{W})$
    $\mathcal{P}_{t+1} \leftarrow \mathcal{P}_t \cup \{\mathcal{W}_{\text{opt}}\}$
  end if
end for

E.2 Fusion Workflow Selection Algorithm

Table 7: Fusion Workflow Selection Algorithm

Algorithm 2: Fusion Workflow Selection

Input: Current workflow pool $\mathcal{P}_t$
Parameters: Selection count M, fusion group size m, weights $\lambda_1, \lambda_2, \mu_1, \mu_2$
Output: Selected workflow group $\mathcal{S}^\star$

$\mathcal{P}' \leftarrow \textsc{TopMByCoverage}(\mathcal{P}_t, M)$ ▷ Filter top-M by $|\mathbf{C}_i|$
$\Phi_{\text{best}} \leftarrow -\infty$
$\mathcal{S}^\star \leftarrow \emptyset$
for each candidate group $\mathcal{S} = \{\mathcal{W}_1, \ldots, \mathcal{W}_m\} \subset \mathcal{P}'$ **do**
    Retrieve coverage sets $\mathbf{C}_1, \ldots, \mathbf{C}_m$ ▷ $\mathbf{C}_i \subseteq \mathcal{D}$: solved instances by $\mathcal{W}_i$
    $\bar{U}_{\text{pair}} \leftarrow \frac{1}{m(m-1)} \sum_{i \neq j} |\mathbf{C}_i \cup \mathbf{C}_j|$ ▷ Pairwise complementarity (union)
    $U(\mathcal{S}) \leftarrow |\bigcup_{i=1}^{m} \mathbf{C}_i|$ ▷ Group-level complementarity
    $\bar{I}_{\text{pair}} \leftarrow \frac{1}{m(m-1)} \sum_{i \neq j} |\mathbf{C}_i \cap \mathbf{C}_j|$ ▷ Pairwise consensus (intersection)
    $I(\mathcal{S}) \leftarrow |\bigcap_{i=1}^{m} \mathbf{C}_i|$ ▷ Group-level consensus
    $\Phi_{\text{fuse}}(\mathcal{S}) \leftarrow \big(\lambda_1 \bar{U}_{\text{pair}} + \lambda_2 U(\mathcal{S})\big) + \big(\mu_1 \bar{I}_{\text{pair}} + \mu_2 I(\mathcal{S})\big) \cdot \eta(\mathcal{S})$ ▷ Fusion potential
    if $\Phi_{\text{fuse}}(\mathcal{S}) > \Phi_{\text{best}}$ **then**
        $\Phi_{\text{best}} \leftarrow \Phi_{\text{fuse}}(\mathcal{S})$
        $\mathcal{S}^\star \leftarrow \mathcal{S}$
    end if
end for
return $\mathcal{S}^\star$

E.3 Differentiation Workflow Selection Algorithm

Table 8: Differentiation Workflow Selection Algorithm

Algorithm 3: Differentiation Workflow Selection

Input: Current workflow pool $\mathcal{P}_t$, dataset partition $\{\mathcal{D}_k\}_{k\in\mathcal{K}}$
Parameters: Softmax temperature τ
Output: Selected workflow $\mathcal{W}^\star$ and target cluster $k^\star$

$N \leftarrow |\mathcal{D}|$ ▷ $N = \sum_{k\in\mathcal{K}} |\mathcal{D}_k|$
Initialize score list $\mathcal{Q} \leftarrow [\,]$
for each workflow $\mathcal{W}_i \in \mathcal{P}_t$ **do**
  Retrieve $\mathbf{C}_i \subseteq \mathcal{D}$ ▷ Solved instances of $\mathcal{W}_i$
  $c_i \leftarrow |\mathbf{C}_i|$
  if $c_i = 0$ **then**
    Append $(\mathcal{W}_i, 0)$ to $\mathcal{Q}$
    continue
  end if
  $\mathrm{Acc}_{\mathrm{global}}(\mathcal{W}_i) \leftarrow \frac{|\mathbf{C}_i|}{N}$ ▷ Global accuracy
  $S_i^{\max} \leftarrow -\infty$
  for each cluster $k \in \mathcal{K}$ **do**
    $n_k \leftarrow |\mathcal{D}_k|$
    if $n_k = 0$ **then**
      continue
    end if
    $\mathbf{C}_{i,k} \leftarrow \mathbf{C}_i \cap \mathcal{D}_k$
    $c_{i,k} \leftarrow |\mathbf{C}_{i,k}|$
    $\mathrm{Recall}_{i,k} \leftarrow \frac{c_{i,k}}{n_k}$ ▷ Cluster-level recall
    if $\mathrm{Recall}_{i,k} > \mathrm{Acc}_{\mathrm{global}}(\mathcal{W}_i)$ **then**
      $\mathrm{Cov}_{i,k} \leftarrow \frac{c_{i,k}}{N}$ ▷ Absolute coverage
      $S_{i,k} \leftarrow \mathrm{Cov}_{i,k} \cdot \big(\mathrm{Recall}_{i,k} - \mathrm{Acc}_{\mathrm{global}}(\mathcal{W}_i)\big)$ ▷ Split potential on cluster k
      $S_i^{\max} \leftarrow \max(S_i^{\max}, S_{i,k})$
    end if
  end for
  $S_i \leftarrow \max(0,\ S_i^{\max})$ ▷ Workflow differentiation potential
  Append $(\mathcal{W}_i, S_i)$ to $\mathcal{Q}$
end for
Sample $(\mathcal{W}^\star, S^\star)$ from $\mathcal{Q}$ with probability $\propto \exp(S_i/\tau)$ ▷ Softmax selection
$k^\star \leftarrow \arg\max_{k\in\mathcal{K}} \left[\frac{|\mathbf{C}_{\star,k}|}{N} \cdot \left(\frac{|\mathbf{C}_{\star,k}|}{|\mathcal{D}_k|} - \frac{|\mathbf{C}_\star|}{N}\right)\right]$ ▷ Choose specialization target
return $(\mathcal{W}^\star, k^\star)$

F Example of Adaptive Operator Scheduling

This section demonstrates the dynamic scheduling mechanism in action through a complete 20-round optimization trace on the DROP dataset, showing how stagnation signals and operator probabilities evolve throughout the process. Note that this trace is from a different experimental run than the case study in Appendix I.

F.1 Execution Trace: 20 Rounds

Table 9 shows the actual values from a complete optimization run.

Table 9: Evolution of Stagnation Signal and Operator Selection Probabilities over 20 Optimization Rounds

Round	π_t	P_{opt}	P_{diff}	P_{fuse}	Selected
1	0.0000	1.0000	0.0000	0.0000	Optimization
2	0.0000	1.0000	0.0000	0.0000	Optimization
3	0.4137	0.5450	0.2482	0.2068	Differentiation
4	1.0000	0.0000	0.5206	0.4794	Differentiation
5	0.7401	0.2664	0.3636	0.3701	Fusion
6	0.5863	0.4467	0.2880	0.2653	Optimization
7	0.3323	0.6864	0.1633	0.1504	Optimization
8	0.4137	0.6096	0.2032	0.1871	Optimization
9	0.2599	0.7547	0.1277	0.1176	Differentiation
10	0.3323	0.7019	0.1477	0.1504	Optimization
11	0.8512	0.2365	0.3784	0.3851	Optimization
12	0.5863	0.4741	0.2606	0.2653	Fusion
13	0.1986	0.8305	0.0883	0.0813	Optimization
14	0.5000	0.5731	0.2222	0.2047	Fusion
15	0.5000	0.5925	0.2222	0.1852	Optimization
16	0.8512	0.3063	0.3784	0.3153	Differentiation
17	0.8512	0.3423	0.3424	0.3153	Fusion
18	0.1488	0.8903	0.0598	0.0499	Optimization
19	0.1098	0.9191	0.0442	0.0368	Optimization

F.2 Observable Patterns

The trace reveals several characteristic behaviors:

- **Initial exploration (Rounds 1–2):** $\pi_t = 0 \Rightarrow P_{\text{opt}} = 1.0$ (pure exploitation)
- **First plateau (Rounds 3–5):** π_t increases from $0.41 \rightarrow 1.0 \rightarrow 0.74$, triggering Differentiation and Fusion
- **Recovery phase (Rounds 6–10):** π_t drops to 0.26–0.59, returning to Optimization dominance
- **Second plateau (Rounds 11–12):** $\pi_t = 0.85$ triggers Fusion
- **Balanced adaptation (Rounds 13–19):** π_t varies between 0.11 and 0.85, showing continuous adjustment

Fusion operations occur at Rounds 5, 12, 14, and 17, precisely when $\pi_t \in [0.50, 0.85]$, demonstrating adaptive triggering of structural exploration.

F.3 Visual Illustration

Figure 6 shows how π_t evolves over time, with peaks at Rounds 4–5, 11–12, and 16–17 corresponding to performance plateaus.

Figure 7 visualizes the operator probability dynamics, showing the complementary relationship between Optimization (exploitation) and Differentiation/Fusion (exploration).

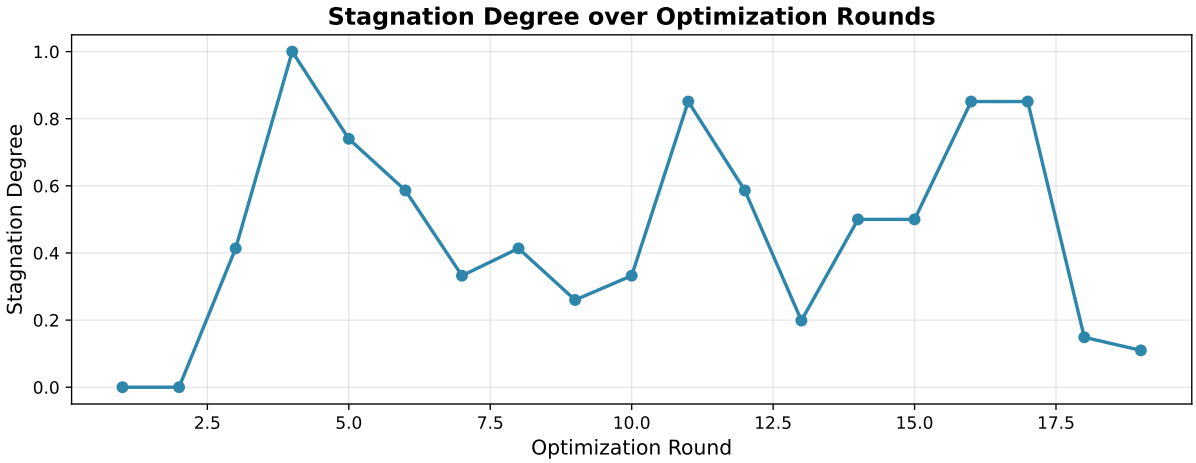

Figure 6: Stagnation signal π_t over 20 rounds. Peaks indicate performance plateaus.

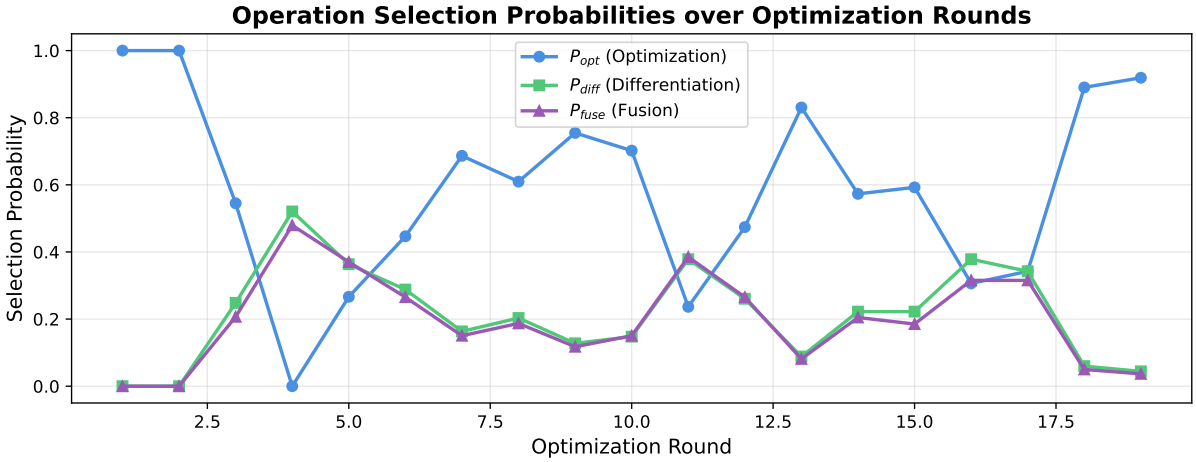

Figure 7: Operator selection probabilities. Blue: Optimization, Green: Differentiation, Purple: Fusion.

G Framework Architecture Details

This section provides detailed examples of the core components in our FusionFlow framework, including workflow structures, prompts, operators, and optimization metadata.

G.1 Workflow Structure Example

A workflow in FusionFlow consists of a directed graph that orchestrates various operators to solve specific problems. Below is a complete example of a workflow for solving GSM8K mathematical reasoning problems:

```python
from typing import Literal
import workspace.GSM8K.workflows.template.operator as operator
import workspace.GSM8K.workflows.round_1.prompt as prompt_custom
from scripts.async_llm import create_llm_instance
from scripts.evaluator import DatasetType

class Workflow:
    def __init__(
        self,
        name: str,
        llm_config,
        dataset: DatasetType,
    ) -> None:
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)

    async def __call__(self, problem: str):
        """
        Implementation of the workflow
        """
        solution = await self.custom(
            input=problem,
            instruction=""
        )
        return solution['response'], \
               self.llm.get_usage_summary()["total_cost"]
```

Listing 1: Example Workflow Graph for GSM8K Dataset

This simple workflow demonstrates the basic structure where:

- The workflow is initialized with an LLM configuration
- A Custom operator is instantiated to handle problem-solving
- The `__call__` method defines the execution flow
- The workflow returns both the solution and the cost incurred

G.2 Node Implementation Example

Operators are the fundamental building blocks in FusionFlow. Here we show the implementation of the `Custom` operator and the `Programmer` operator:

```python
from scripts.operators import Operator

class Custom(Operator):
    def __init__(self, llm: AsyncLLM, name: str = "Custom"):
        super().__init__(llm, name)

    async def __call__(self, input, instruction):
        prompt = instruction + input
```

```
        response = await self._fill_node(
            GenerateOp,
            prompt,
            mode="single_fill"
        )
        return response
```

Listing 2: Custom Operator Implementation

The `Custom` operator provides flexible problem-solving capability by combining custom instructions with input problems. It serves as the most versatile operator in the framework.

```
class Programmer(Operator):
    def __init__(self, llm: AsyncLLM, name: str = "Programmer"):
        super().__init__(llm, name)

    async def exec_code(self, code, timeout=30):
        """
        Asynchronously execute code and return error if timeout.
        """
        loop = asyncio.get_running_loop()
        with concurrent.futures.ProcessPoolExecutor() as executor:
            try:
                future = loop.run_in_executor(
                    executor, run_code, code
                )
                result = await asyncio.wait_for(
                    future, timeout=timeout
                )
                return result
            except asyncio.TimeoutError:
                executor.shutdown(wait=False, cancel_futures=True)
                return "Error", "Code execution timed out"
            except Exception as e:
                return "Error", f"Unknown error: {str(e)}"
```

Listing 3: Programmer Operator with Code Execution

The `Programmer` operator is designed for problems requiring code execution, such as mathematical calculations or algorithmic solutions.

G.3 Prompt Template Example

Prompts guide the LLM's behavior within operators. In **FusionFlow**, prompts are stored separately and can be dynamically modified during optimization:

```
# File: workspace/GSM8K/workflows/round_1/prompt.py

SOLVE_PROMPT = """
Please solve the following mathematical problem step by step.
Show your reasoning clearly and provide the final numerical answer.

Problem: {problem}

Solution:
"""

VERIFICATION_PROMPT = """
Verify the following solution and check if the answer is correct.
If there are errors, provide the corrected solution.

Original Problem: {problem}
Solution to Verify: {solution}

Verification:
"""
```

Listing 4: Prompt Template File Structure

Custom prompts can be referenced in the workflow using `prompt_custom.PROMPT_NAME`.

G.4 Experience Metadata

During optimization, **FusionFlow** maintains a structured experience database that tracks the success and failure of different modifications. Below is an example from the DROP dataset:


```
{
    "3": {
        "score": 0.8450,
        "success": {
            "14": {
                "modification": "3-way Workflow Fusion: Combined
                  3 high-performing workflows from rounds [3, 5, 11]
                  (scores: ['0.8450', '0.8450', '0.8300']). Integrated
                  complementary strengths to maximize problem coverage.",
                "score": 0.8800
            }
        },
        "failure": {
            "15": {
                "modification": "Add mathematical verification and
                  completeness check step using Custom operator before
                  ensemble selection",
                "score": 0.8200
            }
        }
    },
    "2": {
        "score": 0.8400,
        "success": {
            "3": {
                "modification": "Add ScEnsemble operator to generate
                  multiple reasoning paths and select the most consistent
                  answer, improving accuracy for mathematical and logical
                  reasoning problems",
                "score": 0.8450
            }
        },
        "failure": {}
    },
    "11": {
        "score": 0.8300,
        "success": {},
        "failure": {
            "17": {
                "modification": "Workflow Differentiation: Specialized
                  for Mathematical Derivation but with too narrow focus",
                "score": 0.8100
            }
        }
    }
}
```


Listing 5: Processed Experience Metadata from DROP Dataset

The experience metadata captures:

- **Parent Round**: The round from which modifications originated
- **Success/Failure Tracking**: Categorization of descendant rounds
- **Modification Description**: Natural language description of the change

- **Performance Score**: Validation set accuracy

This structured experience allows the optimization LLM to learn from past attempts and avoid repeating failed modifications.

G.5 Fusion Metadata Example

When performing workflow fusion, **FusionFlow** combines multiple high-performing workflows. The fusion process generates metadata describing how workflows were combined:


```
{
    "fusion_round": 14,
    "parent_workflows": [
        {
            "round": 3,
            "score": 0.8450,
            "solved_problems": 169,
            "key_features": [
                "ScEnsemble for multiple reasoning paths",
                "Majority voting mechanism"
            ]
        },
        {
            "round": 5,
            "score": 0.8450,
            "solved_problems": 169,
            "key_features": [
                "Alternative ensemble configuration",
                "Enhanced problem comprehension"
            ]
        },
        {
            "round": 11,
            "score": 0.8300,
            "solved_problems": 166,
            "key_features": [
                "Specialized for mathematical derivation",
                "Step-by-step calculation verification"
            ]
        }
    ],
    "fusion_strategy": "Adaptive routing to specialized paths
                        based on problem type analysis",
    "combined_coverage": 176,
    "fusion_score": 0.8800,
    "improvement_over_best_parent": 0.0350,
    "unique_problems_solved": 7
}
```


Listing 6: Fusion Metadata Structure from DROP Dataset

G.6 Differentiation Metadata Example

Differentiation creates specialized variants of workflows by analyzing error logs:


```
{
    "parent_round": 9,
    "parent_score": 0.8200,
    "error_analysis": {
        "total_errors": 36,
        "error_categories": {
            "multi_step_calculation_error": 15,
            "passage_comprehension_error": 12,
            "numerical_extraction_error": 9
        },
        "example_errors": [
```

```
            {
                "problem_id": "drop-validation-127",
                "question": "How many more points did team A score
                            in the first half than team B?",
                "passage": "Team A scored 24 in first half...
                           Team B scored 17 in first half...",
                "expected": "7",
                "got": "24",
                "error_type": "multi_step_calculation_error",
                "root_cause": "Extracted first value instead of
                              computing difference"
            }
        ]
    },
    "differentiation_type": "mathematical_specialization",
    "modification": "Enhance workflow with dedicated multi-step
                    calculation handling and intermediate value
                    tracking for mathematical derivation problems",
    "child_round": 11,
    "child_score": 0.8300,
    "improvement": 0.0100,
    "specialization_focus": "Complex multi-step numerical reasoning"
}
```


Listing 7: Differentiation Metadata from DROP Dataset

G.7 Log Entry Example

Execution logs capture detailed information about each problem solved:


```
{
    "question": "How many more field goals did the team make
      in the first half compared to the second half?",
    "passage": "In the first half, the team made 3 field goals.
      ... In the second half, they made 1 field goal.",
    "right_answer": "2",
    "model_output": "To find how many more field goals the team
      made in the first half:\n1. First half field goals: 3\n
      2. Second half field goals: 1\n3. Difference: 3 - 1 = 2\n
      Therefore, the team made 2 more field goals in the first
      half compared to the second half.",
    "score": 1.0,
    "judge_explanation": "Model correctly identified the values
      and computed the difference, arriving at answer '2'.",
    "problem_id": "drop-validation-042",
    "category": "Numerical Reasoning"
}
```


Listing 8: Example Log Entry from DROP Workflow Execution

These logs are used by the differentiation operator to identify systematic errors and guide targeted improvements.

H Core Operator Implementations Logic

This section details the prompts and implementation logic for **FusionFlow**'s three core optimization operators: Optimization, Fusion, and Differentiation.

H.1 Optimization Operator

The optimization operator analyzes existing workflows and proposes incremental improvements.

H.1.1 Optimization Prompt

Prompt for the Optimization Operator

[Objective]
You are building a Graph and corresponding Prompt to jointly solve {type} problems. Referring to the given graph and prompt, which forms a basic example of a {type} solution approach, please reconstruct and optimize them. You can add, modify, or delete nodes, parameters, or prompts. Include your single modification in XML tags in your reply. Ensure they are complete and correct to avoid runtime failures.

[Optimization Guidelines]
When optimizing, you can incorporate critical thinking methods like:

- **Review**: Add verification steps to check intermediate results
- **Revise**: Implement error correction loops
- **Ensemble**: Generate multiple answers through different/similar prompts, then vote/integrate/check the majority to obtain a final answer
- **Self-Ask**: Break down complex problems into sub-questions

Consider Python's loops (for, while, list comprehensions), conditional statements (if-elif-else, ternary operators), or machine learning techniques. The graph complexity should not exceed 10 nodes. Use logical and control flow (IF-ELSE, loops) for enhanced graphical representation.

[Input Format]
You will receive:

```
<sample>
    <experience>{experience}</experience>
    <modification>(e.g., add/delete/modify operator X)</modification>
    <score>{score}</score>
    <graph>{current_graph_code}</graph>
    <prompt>{current_prompts}</prompt>
    <operator_description>{available_operators}</operator_description>
</sample>

Error logs from current workflow:
{error_examples}
```

[Constraints]

- **Single Modification**: Only one detail point can be modified at a time
- **Code Limit**: No more than 5 lines of code may be changed per modification
- **No Placeholders**: Generated prompts must not contain any placeholders

- **Format Output**: Use custom methods to restrict output format, not code
- **No None Output**: Graph must never output None for any field

[Custom Operator Usage]
Example of using the Custom operator in your graph:

```
# Write prompt in prompt_custom and use in Custom method
response = await self.custom(
    input=problem,
    instruction=prompt_custom.XXX_PROMPT
)

# Can concatenate previous results for more context
solution = await self.generate(
    problem=f"question:{problem}, context:{response['response']}"
)
```

Note: In custom, input and instruction are directly concatenated (instruction+input), and placeholders are not supported.

[Output Format]
Provide your optimization in XML format:

```
<modification>
Describe the single modification made
</modification>
<graph>
Complete Python code for optimized workflow
</graph>
<prompt>
All necessary prompts in prompt_custom (if needed)
</prompt>
```

H.1.2 Optimization Implementation Logic

The optimization operator follows this process:

1. **Select Parent**: Choose high-performing workflows
2. **Load Context**: Retrieve the parent workflow's code, prompts, and error logs
3. **Format Experience**: Convert historical experience into structured format showing successful and failed modifications
4. **Generate Prompt**: Create optimization prompt with all context
5. **Call LLM**: Use optimization LLM to generate modified workflow
6. **Validate**: Check that modification is novel (not repeated from history)
7. **Evaluate**: Test the modified workflow on validation set
8. **Record**: Update experience database with success/failure

H.2 Fusion Operator

The fusion operator combines multiple high-performing workflows into a single comprehensive workflow.

H.2.1 Fusion Prompt

Prompt for the Fusion Operator

[Objective]
You are an expert workflow designer tasked with fusing multiple high-performing workflows into a single, more comprehensive workflow that can solve {type} problems. Referring to the given graphs and prompts from the parent workflows, you must intelligently combine the strengths of each input workflow while creating robust routing logic to handle different problem types.

[Core Principle]
The most crucial point: Your task is to perform multi-way workflow fusion. This means you must identify the key components within each of the parent workflows and select only those parts that have fusion value to combine them into a new workflow. **You should not create the core components from scratch**, including graph structure and prompts.

[Input Format]
You will be given multiple high-performing workflows ($K \geq 2$):

```
<workflow_1>
    <round>{round_number}</round>
    <score>{performance_score}</score>
    <solved_problems>{count} problems</solved_problems>
    <prompt>{workflow_prompts}</prompt>
    <graph>{workflow_code}</graph>
</workflow_1>
... (workflow_2, workflow_3, ... similarly)

<operator_description>
{available_operators}
</operator_description>
```

[Fusion Strategy]
Analyze each workflow's strengths and design a fusion strategy that:

- **Maximizes Coverage**: The fused workflow should solve problems that any of the parent workflows can solve
- **Leverages Scores**: Workflows with higher scores can serve as the foundation
- **Intelligent Routing**: Create decision logic to route problems to appropriate solution paths
- **Combines Strengths**: Integrate complementary capabilities from each parent

[Design Patterns]
Common fusion patterns include:

1. **Problem Classification + Routing**:

```
problem_type = await self.classify(problem)
if problem_type == "calculation":
    return await workflow_1_logic(problem)
elif problem_type == "word_problem":
    return await workflow_2_logic(problem)
```

2. **Sequential Enhancement**:

```
# Use workflow_1 for initial solution
initial = await workflow_1_solve(problem)
# Use workflow_2 for verification
verified = await workflow_2_verify(initial)
# Use workflow_3 for refinement if needed
if not verified:
    final = await workflow_3_refine(initial)
```

3. **Parallel Ensemble**:

```
results = await asyncio.gather(
    workflow_1_solve(problem),
    workflow_2_solve(problem),
    workflow_3_solve(problem)
)
final = self.vote_or_combine(results)
```

[Constraints]

- The fused workflow must be different from any single input workflow
- Generated prompts must not contain placeholders
- Use custom methods to restrict output format, not code

[Output Format]
Provide your fusion result in XML format:

```
<modification>
Description of the fusion strategy and how the parent workflows
are combined
</modification>
<graph>
Complete Python code for the fused workflow
</graph>
<prompt>
All necessary prompts in prompt_custom (if needed)
</prompt>
```

H.2.2 Fusion Implementation Logic

The fusion operator executes the following workflow:

1. **Select Candidate Workflows**: Choose high-performing workflows based on the fusion potential score
2. **Analyze Diversity**: Compute solved problem overlap to ensure complementarity
3. **Load Components**: Extract code, prompts, and metadata from selected workflows

4. **Format Input**: Structure the parent workflows into fusion prompt format
5. **Generate Fusion**: Call LLM to create combined workflow
6. **Validate Integration**: Verify that parent workflows contributed to the fusion
7. **Evaluate Performance**: Test on validation set
8. **Update Metadata**: Record fusion genealogy and parent information

H.3 Differentiation Operator

The differentiation operator creates specialized variants of workflows targeting particular task subspaces (e.g., algebraic problems, geometric reasoning, combinatorial counting). Unlike optimization which refines general performance, differentiation tailors workflows to excel on specific problem categories, typically through prompt specialization and targeted structural adjustments.

H.3.1 Differentiation Prompt

Prompt for the Differentiation Operator

[Objective]
You are an expert workflow designer tasked with differentiating an existing workflow to create a specialized variant for {type} problems.

[Core Principle - What is Differentiation]
Differentiation means creating a specialized variant that excels on a particular problem category:

- Focus on making the workflow highly effective for the target task subspace
- Specialize prompts with domain-specific instructions and reasoning patterns
- Structural adjustments are allowed when beneficial for specialization

[What You ARE Doing]

- Converting "analyze problem" → "analyze [specific problem type] characteristics"
- Converting "solve" → "apply [specific technique] to solve"
- Converting "verify" → "verify using [specific domain] validation rules"
- Making prompts more specific and detailed for the target domain

[Input Format]
You will receive:

```
<differentiation_data>
    <dataset>{dataset}</dataset>
    <target_round>{target_round}</target_round>
    <specialization_direction>{direction}</specialization_direction>
    <specialization_focus>{specialization_focus}</specialization_focus>
    <parent_workflow>
        <score>{validation_score}</score>
        <graph>{workflow_code}</graph>
        <prompt>{workflow_prompts}</prompt>
    </parent_workflow>
    <operator_description>{available_operators}</operator_description>
</differentiation_data>
```

For targeted differentiation, a <target_category> section with example problems may also be provided.

[How to Convert Abstract → Concrete]

```
Parent: "analyze the problem"
Child:  "analyze {specialization_focus} characteristics:
         identify key variables, constraints, and target"

Parent: "solve the problem"
Child:  "apply {specialization_focus} techniques:
         [specific method 1], [specific method 2]"

Parent: "verify the solution"
Child:  "verify using {specialization_focus} rules:
         check [specific constraint 1], [specific constraint 2]"
```

[Example Differentiation]
If parent workflow is:

```
analysis = await self.custom(input=problem,
    instruction=prompt_custom.ANALYSIS_PROMPT)
solution = await self.custom(input=analysis['response'],
    instruction=prompt_custom.SOLUTION_PROMPT)
```

Differentiated workflow should be:

```
analysis = await self.custom(input=problem,
    instruction=prompt_custom.ALGEBRAIC_ANALYSIS_PROMPT)
solution = await self.custom(input=analysis['response'],
    instruction=prompt_custom.ALGEBRAIC_SOLUTION_PROMPT)
```

The prompts should contain {specialization_focus}-specific instructions (e.g., "identify variables and equations", "apply substitution or elimination methods"), but the structure remains identical.

[Output Format]

```
<modification>
Specialized for {specialization_focus}: {description}
</modification>
<graph>
Differentiated workflow code (same structure, specific prompts)
</graph>
<prompt>
Category-specific prompts
</prompt>
```

H.3.2 Differentiation Modes

The differentiation operator supports two modes:

Exploratory Differentiation: When no specific target category is provided, the system randomly selects a specialization focus from the predefined categories based on the problem type (math/code/qa). This enables broad exploration of the specialization space.

Targeted Differentiation: When a specific `target_category` is provided (along with optional description and example problems), the differentiation becomes directed toward that particular task subspace.

The prompt includes additional context:

```
<target_category>
    <name>{category_name}</name>
    <description>{category_description}</description>
    <examples>
        <example_1>{problem_text_1}</example_1>
        <example_2>{problem_text_2}</example_2>
        ...
    </examples>
</target_category>
```

H.3.3 Differentiation Implementation Logic

The `DifferentiationPromptGenerator` class implements the following process:

1. **Load Parent Workflow**: Retrieve a high-performing general workflow (graph and prompts)
2. **Determine Specialization Focus**:
   - If `target_category` is provided: use it directly for targeted differentiation
   - Otherwise: randomly select from predefined specializations based on `question_type`
3. **Gather Category Examples** (for targeted mode): Collect up to 3 representative problems, truncating if longer than 500 characters
4. **Generate Specialization Prompt**: Combine the differentiation system prompt, input data, and custom operator usage instructions
5. **Call LLM**: Request the model to convert abstract steps into category-specific steps while preserving workflow structure
6. **Parse Output**: Extract the modified graph and category-specific prompts
7. **Evaluate**: Test the differentiated workflow on the validation set

I Case Study: Complete Optimization Cycle

This section presents a detailed walkthrough of a complete optimization cycle on the DROP dataset, demonstrating how **FusionFlow** progressively improves workflow performance through its three core operators. DROP (Discrete Reasoning Over Paragraphs) requires complex numerical reasoning over text passages, making it an ideal showcase for our framework's capabilities.

Important Note: All scores reported in this case study are **validation set scores** (200 problems), which guide the optimization process. In our experimental protocol, we select the workflow with the best validation performance and evaluate it on the held-out test set (800 problems) for final reporting. The validation scores shown here may differ from the final test set scores reported in the main paper's experiments.

I.1 Initial Setup

Dataset: DROP (Discrete Reasoning Over Paragraphs)
Problem Type: Numerical reasoning and question answering
Initial Workflow: Simple Custom operator with no instructions
Validation Set Size: 200 problems
Test Set Size: 800 problems
Challenge: Requires multi-step mathematical reasoning combined with passage comprehension

I.1.1 Round 1: Baseline Workflow

The initial workflow starts with a minimal structure:

```
class Workflow:
    def __init__(self, name: str, llm_config, dataset: DatasetType):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)

    async def __call__(self, problem: str):
        solution = await self.custom(input=problem, instruction="")
        return solution['response'], \
               self.llm.get_usage_summary()["total_cost"]
```

Listing 9: Round 1: Initial Baseline Workflow

Performance:

- Validation Score: 0.8200 (82.00%)
- Solved Problems: 164/200

Error Analysis: Identified issues with multi-step reasoning, mathematical calculations, and extracting the correct numerical values from passages.

I.2 Optimization Operator Applied

I.2.1 Round 2: Adding Structured Reasoning

The optimization operator analyzes Round 1 and proposes adding structured step-by-step reasoning:

Modification Description: "Add AnswerGenerate operator to provide structured step-by-step reasoning before generating the final solution, which should help with mathematical calculations and logical comparisons"

```
class Workflow:
    def __init__(self, name: str, llm_config, dataset: DatasetType):
        self.name = name
        self.dataset = dataset
```

```
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.answer_generate = operator.AnswerGenerate(self.llm)

    async def __call__(self, problem: str):
        # Use AnswerGenerate for structured reasoning
        solution = await self.answer_generate(
            problem=problem,
            instruction=prompt_custom.REASONING_PROMPT
        )
        return solution['response'], \
               self.llm.get_usage_summary()["total_cost"]
```

Listing 10: Round 2: Workflow with Structured Reasoning

Associated prompt in `prompt_custom`:

```
REASONING_PROMPT = """
Carefully read the passage and question.
Break down the problem into steps:
1. Identify relevant information from the passage
2. Determine what calculation is needed
3. Perform the calculation step by step
4. Provide the final numerical answer
"""
```

Performance:

- Validation Score: 0.8400 (84.00%)
- Improvement: +0.0200 (+2.00%)
- Solved Problems: 168/200

Impact: Modest improvement through structured reasoning, helping with some multi-step problems but still struggling with complex cases.

I.2.2 Round 3: Adding Ensemble for Robustness

Building on Round 2's success, another optimization adds ensemble voting:

Modification: "Add ScEnsemble operator to generate multiple reasoning paths and select the most consistent answer, improving accuracy for mathematical and logical reasoning problems"

```
class Workflow:
    def __init__(self, name: str, llm_config, dataset: DatasetType):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.answer_generate = operator.AnswerGenerate(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)

    async def __call__(self, problem: str):
        # Generate multiple reasoning paths
        solutions = await self.sc_ensemble(
            problem=problem,
            instruction=prompt_custom.REASONING_PROMPT,
            num_samples=3
        )
        return solutions['response'], \
               self.llm.get_usage_summary()["total_cost"]
```

Listing 11: Round 3: Workflow with Ensemble Voting

Performance:

- Validation Score: 0.8450 (84.50%)
- Improvement over Round 2: +0.0050 (+0.50%)
- Solved Problems: 169/200

Impact: Ensemble voting provides marginal improvement, reducing some calculation errors but introducing additional cost without dramatic performance gains.

I.2.3 Round 11: Differentiation-Based Specialization

The differentiation operator creates a specialized variant by converting abstract workflow steps into concrete, domain-specific steps for a particular task subspace:

Modification: "Workflow Differentiation: Specialized for Mathematical Derivation problems. The parent workflow's general prompts are converted into concrete, derivation-specific prompts while preserving the exact same workflow structure."

Differentiation Process: The operator takes a high-performing parent workflow and specializes it for the "mathematical derivation" category by:

- Keeping the same operator structure (single Custom operator)
- Converting abstract instructions (e.g., "solve the problem") into concrete, domain-specific instructions (e.g., "apply step-by-step mathematical derivation with explicit variable tracking")
- Adding category-specific reasoning patterns for numerical calculations

Performance:

- Validation Score: 0.8300 (83.00%)
- Solved Problems: 166/200 (different subset than Round 3)
- Specialization: Strong on problems requiring complex multi-step calculations

I.3 Fusion Operator Applied: The Breakthrough

I.3.1 Round 14: Three-Way Workflow Fusion

After multiple rounds of optimization and differentiation, the fusion operator combines three complementary workflows to achieve a significant performance leap:

Parent Workflows:

- **Round 3** (Score: 0.8450): Strong ensemble-based reasoning
- **Round 5** (Score: 0.8450): Alternative ensemble configuration
- **Round 11** (Score: 0.8300): Specialized for mathematical derivation

Key Insight: While Rounds 3 and 5 have similar overall scores, they solve different subsets of problems. Round 11's specialization complements both by handling complex calculation problems that the ensemble approaches miss.

Fusion Strategy: Create an adaptive workflow that routes problems based on their characteristics and combines the strengths of all three parents.

```
class Workflow:
    def __init__(self, name: str, llm_config, dataset: DatasetType):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
```

```
        self.answer_generate = operator.AnswerGenerate(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)

    async def __call__(self, problem: str):
        # Step 1: Analyze problem complexity
        analysis = await self.custom(
            input=problem,
            instruction=prompt_custom.ANALYZE_PROBLEM_PROMPT
        )

        problem_type = analysis['response'].lower()

        # Step 2: Intelligent routing based on problem type
        if "multi-step calculation" in problem_type or \
           "complex derivation" in problem_type:
            # Use Round 11's specialized approach
            solution = await self._solve_with_derivation(problem)
        elif "comparison" in problem_type or \
             "multiple values" in problem_type:
            # Use Round 5's ensemble configuration
            solution = await self._solve_with_enhanced_ensemble(
                problem
            )
        else:
            # Use Round 3's standard ensemble approach
            solution = await self._solve_with_standard_ensemble(
                problem
            )

        # Step 3: Final verification and formatting
        final_answer = await self.custom(
            input=f"Question: {problem}\nAnswer: {solution}",
            instruction=prompt_custom.VERIFY_FORMAT_PROMPT
        )

        return final_answer['response'], \
               self.llm.get_usage_summary()["total_cost"]

    async def _solve_with_derivation(self, problem: str):
        """Round 11's specialized mathematical derivation logic"""
        solution = await self.answer_generate(
            problem=problem,
            instruction=prompt_custom.DERIVATION_PROMPT
        )
        return solution['response']

    async def _solve_with_enhanced_ensemble(self, problem: str):
        """Round 5's ensemble configuration"""
        solutions = await self.sc_ensemble(
            problem=problem,
            instruction=prompt_custom.ENHANCED_REASONING_PROMPT,
            num_samples=5
        )
        return solutions['response']

    async def _solve_with_standard_ensemble(self, problem: str):
        """Round 3's standard ensemble approach"""
        solutions = await self.sc_ensemble(
            problem=problem,
            instruction=prompt_custom.REASONING_PROMPT,
            num_samples=3
        )
        return solutions['response']
```

Listing 12: Round 14: Fused Workflow with Intelligent Routing

Performance - The Breakthrough:

- **Validation Score: 0.8800 (88.00%)**

- **Improvement over best parent: +0.0350 (+3.50%)**
- **Solved Problems: 176/200**
- **New Problems Solved: 7 problems that none of the three parents could solve**
- **Coverage Enhancement: Union of parent coverage (169 + 169 + 166) → 176 unique problems through intelligent routing**

Analysis of the Breakthrough:
The fusion's success demonstrates two key principles:

1. **Complementary Coverage**: Each parent excels on different problem types:
   - Round 3: Strong on standard reasoning problems (optimization result)
   - Round 5: Effective on comparison and multi-value problems (optimization result)
   - Round 11: Superior on complex mathematical derivations (differentiation result—specialized for the "mathematical derivation" task subspace)
2. **Intelligent Integration**: The fused workflow doesn't simply combine approaches randomly, but routes each problem to the most suitable strategy, achieving **3.5% improvement** - significantly larger than the marginal gains from pure optimization (+0.5% from Round 2 to 3). This demonstrates the power of **deep trajectory crossing**.
3. **Differentiation Enables Fusion**: Round 11's task subspace specialization is crucial—without differentiation creating a workflow specialized for mathematical derivations, the fusion would lack the diversity needed to achieve complementary coverage.

I.4 Performance Evolution Summary

Table 10 summarizes the performance evolution across the optimization cycle, highlighting the breakthrough achieved by fusion:

Table 10: Performance Evolution on DROP Dataset Across Optimization Rounds (Validation Set). * Round 11's lower overall score reflects task subspace specialization - it excels on mathematical derivation problems while trading off performance on other problem types

Round	Operator	Modification	Val Score	Δ
1	Baseline	Initial workflow	0.8200	-
2	Optimization	Add structured reasoning	0.8400	+0.0200
3	Optimization	Add ensemble voting	0.8450	+0.0050
5	Optimization	Alternative ensemble	0.8450	+0.0000
11	Differentiation	Math specialization	0.8300	-0.0150*
14	**Fusion**	**3-way fusion [3+5+11]**	**0.8800**	**+0.0350**
Total Improvement from Baseline				**+0.0600**
Fusion Contribution (beyond best parent)				**+0.0350**
Pure Optimization Gains (R1→R3)				**+0.0250**

Key Observations:

- **Diminishing Returns from Optimization**: Pure optimization shows diminishing marginal returns (2.0% → 0.5% → 0.0%)
- **Differentiation Creates Diversity**: Round 11's differentiation produces a specialized workflow with lower overall score (0.8300) but superior performance on mathematical derivation problems, creating valuable diversity for fusion

- **Fusion Breakthrough**: The fusion operator achieves a **3.5% improvement - 7x larger** than the gain from Round 2 to Round 3 (0.5%)
- **Deep Trajectory Crossing**: Fusion solves 7 additional problems that *none* of the three parents could solve individually, demonstrating the power of combining diverse optimization trajectories
- **Validation vs Test**: These scores are on the 200-problem validation set used to guide optimization. The final experiment selects the best validation workflow (Round 14) and evaluates it on the 800-problem test set for unbiased performance reporting.

I.5 Failed Attempts and Lessons Learned

Not all modifications succeed. Here are notable failures from the DROP case study:

I.5.1 Failed Attempt 1: Over-Verification (Round 15)

Modification: "Add a mathematical verification and completeness check step using Custom operator before the ensemble selection to validate calculations and ensure all qualifying answers are included"

Result: Score dropped to 0.8200 (regression of -0.0250 from parent Round 3)

Root Cause: The verification step introduced excessive complexity and second-guessing, causing the model to reject correct answers or overcomplicate simple problems

Lesson: Verification should be selective, not universal. Pre-ensemble verification can interfere with the diversity needed for effective voting.

I.5.2 Failed Attempt 2: Excessive Answer Refinement (Round 16)

Modification: "Add a dedicated answer refinement step using Custom operator after ensemble selection to standardize answer formats and verify calculations, ensuring consistency with expected answer patterns"

Result: Score: 0.7600 (severe regression of -0.0850)

Root Cause: The post-processing step incorrectly reformatted many correct numerical answers, changing "175" to "one hundred seventy-five" when the expected format was numerical

Lesson: Answer formatting should preserve the content's semantic meaning. Format standardization must be carefully designed to avoid introducing new errors.

I.5.3 Failed Attempt 3: Misguided Ensemble Expansion (Round 18)

Modification: "Modify the existing Custom operator call to include mathematical verification and multi-answer handling within the formatting step, and increase the number of solution generations from 3 to 5 for better ensemble coverage"

Result: Score: 0.8450 (no improvement over parent, just increased cost)

Root Cause: Simply increasing ensemble size without improving the underlying reasoning doesn't help. The additional samples added cost without providing new insights.

Lesson: Ensemble size should be tuned based on problem diversity. More samples help when solutions disagree, but don't improve performance when the model consistently makes the same type of error.

I.5.4 Failed Attempt 4: Suboptimal Fusion (Round 19 - Alternative)

Modification: "3-way Workflow Fusion: Combined 3 high-performing workflows from rounds [14, 18, 11]"

Result: Score: 0.8300 (below parent Round 14's 0.8800)

Root Cause: Round 18 (no improvement over its parent) didn't contribute meaningful diversity. Fusing workflows with overlapping strengths doesn't create complementary coverage.

Lesson: Successful fusion requires **diverse** parents with **complementary** strengths. The parent selection strategy should maximize coverage diversity, not just select the highest-scoring workflows.