FlowBank: Query-Adaptive Agentic Workflows Optimization through Precompute-and-Reuse

Lingzhi Yuan[1] **Chenghao Deng**[1] **Fangxu Yu**[1] **Souradip Chakraborty**[1]
Mohammad Rostami[2] **Furong Huang**[1,∀†]

[1]University of Maryland, College Park [2]Amazon [∀]All Purpose AI

Abstract

An agentic system's capabilities depend on both its language model and the harness that organizes its work. Workflow design optimizes a central part of harness: how a task is decomposed, which roles agents assume, and how their outputs are coordinated and checked. However, current agentic workflow optimization paradigms make an unsatisfying trade-off. Task-level methods search offline but deploy only a single workflow, leaving complementary candidates unused, while query-level generators synthesize a new workflow for each query at substantial inference cost. Our motivating analysis shows that workflows discovered offline solve different subsets of queries, and many queries handled by expensive query-level generation can already be solved by cheaper precomputed workflows. These observations motivate precompute-and-reuse: building a compact bank of complementary workflows offline and selecting among them adaptively at inference time. Doing so requires solving three coupled problems: generating complementary rather than redundant candidates, compressing them into a deployable portfolio, and assigning each query to the right workflow under a performance–cost trade-off. To this end, we present `FlowBank`, a precompute-and-reuse framework for portfolio-based agentic workflow optimization. *Diversifying* uses DiverseFlow to steer search toward under-covered queries and produce a high-coverage candidate pool. *Curating* proposes CuraFlow to compress this pool into a compact portfolio with minimal redundancy. *Matching* learns a query–workflow utility predictor that selects one portfolio member under a performance–cost trade-off, including a dynamic generation option when useful. Across five benchmarks, `FlowBank` achieves the highest average score among the evaluated methods while remaining cost-competitive, improving over the strongest automated and handcrafted baselines by 3.00 and 9.53 percentage points respectively. Our code is available at https://agentic-flowbank.github.io/.

1 Introduction

Large Language Models (LLMs) have enabled increasingly capable agentic systems across question answering [41, 17, 22, 33, 43], code generation [2, 1, 12], web navigation [54, 4], and beyond. However, single-agent systems often struggle with intricate real-world requirements, catalyzing a paradigm shift toward multi-agent systems, wherein coordinated ensembles of specialized agents tackle complex tasks [15, 6, 37]. The effectiveness of these systems depends on both the underlying LLMs and the agent harness: the software and protocols that organize model calls, tool use, and agent interactions. Within this harness, workflows specify how responsibilities are allocated, what information agents exchange, and how their outputs are combined [56, 19]. While many human-designed workflows have been proposed [30, 6, 15, 24, 34], the design space of possible agent

†Corresponding author: furongh@umd.edu

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

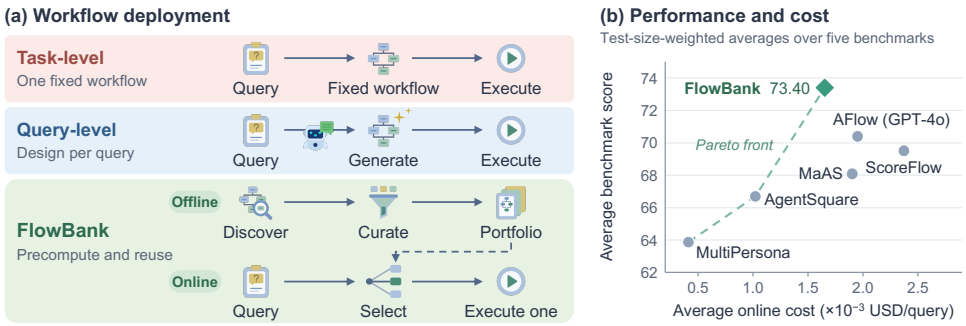

Figure 1: **Query-adaptive workflow deployment through precompute-and-reuse.** (a) Task-level methods deploy one fixed workflow, while query-level methods generate a workflow for each query. `FlowBank` discovers and curates a portfolio offline, then selects one member to execute for each incoming query. (b) Average performance and online cost across five benchmarks for representative methods. `FlowBank` achieves the highest average performance and lies on the Pareto frontier.

topologies, communication patterns, and configurations grows rapidly with system complexity, rendering manual design increasingly untenable and motivating automated workflow optimization [56, 16, 47, 28].

Existing automated workflow optimization methods largely follow two strategies. Task-level methods run multiple rounds of offline search and then deploy a single universal workflow for all queries, enabling stable and low-cost deployment but enforcing a one-size-fits-all policy [56, 47, 16, 28, 23, 51, 35]. Query-level methods instead synthesize a workflow on the fly for each query, offering stronger adaptivity but pushing substantial reasoning and search cost into inference [32, 42, 46, 44, 11, 29]. The resulting trade-off leaves a natural question: can we recover query-level adaptivity without paying the full cost of per-query workflow generation?

Our motivating analysis in Section 2 with detailed experiments in Appendix D reveals two under-exploited forms of workflow complementarity. Within task-level search, workflows discarded after optimization often solve queries that the selected workflow misses, retaining substantial reuse value. Across paradigms, many queries handled by expensive query-level generation can also be solved by cheaper static workflows, revealing opportunities to reduce online cost through query-adaptive selection. These observations motivate `FlowBank`'s perspective: rather than commit to one workflow or regenerate a new one for every query, we should precompute a bank of complementary workflows offline and reuse it adaptively at test time.

Turning this perspective into a practical system is challenging. **(C1)** First, *complementary bank construction* requires more than optimizing individual workflow accuracy: a high-scoring workflow may repeat capabilities already covered by the bank, while a weaker specialist may solve otherwise uncovered queries. Search must therefore discover workflows that complement the current pool. **(C2)** Second, *compact portfolio curation* must reduce selector burden while preserving joint coverage. Retaining every discovered workflow enlarges the decision space, while selecting only individually strong workflows can discard useful complementary members. **(C3)** Third, *query-adaptive workflow selection* requires identifying a suitable portfolio member before observing its execution outcome. The selector must translate the bank's complementary capabilities into effective per-query decisions under a performance–cost trade-off.

To address these challenges, we introduce `FlowBank`, a portfolio-based framework organized around three stages: *Diversifying*, *Curating*, and *Matching*. **(S1) Complementarity-oriented Workflow Diversification**: we use DiverseFlow to steer search toward under-covered queries and produce a candidate pool that is complementary rather than merely individually high-scoring. **(S2) Coverage-aware Combinatorial Curation**: we use CuraFlow to perform coverage-aware combinatorial search and compress this pool into a compact, high-coverage portfolio. **(S3) Bipartite Graph-based Matching**: we cast deployment as edge-value prediction on a query–workflow bipartite graph and assign each incoming query to the portfolio member with the best predicted performance–cost trade-off. As shown in Figure 1, by separating heavy offline workflow discovery from lightweight online selection, `FlowBank` recovers much of the adaptivity of query-level methods while preserving the deployment efficiency of offline methods. In this way, `FlowBank` turns workflows into reusable assets rather than one-shot design outcomes.

Across five benchmarks spanning mathematical reasoning [13, 5], question answering [33], reading comprehension [7], and code generation [1], FlowBank achieves the highest average score among the evaluated methods while remaining competitive in inference cost, improving over the strongest automated and handcrafted baselines by 3.00 and 9.53 percentage points, respectively. These results show that selecting from a curated workflow bank is more effective than committing to a single workflow or a single paradigm.

Our main contributions are summarized as follows:

- We identify two under-exploited forms of workflow complementarity: discarded task-level workflows retain set-level reuse value, and many queries handled by expensive query-level generation can also be solved by cheaper precomputed workflows.
- We propose FlowBank, a three-stage framework that builds and adaptively reuses a complementary workflow bank through pool-aware query reweighting for discovery, coverage-aware combinatorial curation, and utility prediction on a query–workflow bipartite graph.
- We demonstrate that FlowBank achieves the strongest average performance and competitive performance–cost trade-offs across five benchmarks, validating portfolio-based workflow reuse as a stronger alternative to committing to a single workflow or a single paradigm.

2 Background

2.1 Problem Formulation

Agentic Workflows. Given a task dataset D, an agentic workflow $\omega(\cdot)$ is defined as a computation graph of LLM calls that maps a query $q \in D$ to a prediction $\omega(q)$. An ideal workflow aims to produce correct answers with lower token usage, balancing accuracy and computational cost. To quantify this, we evaluate the utility of ω on a query q using two metrics: **(1)** the performance $e(\omega, q) \in [0, 1]$, measured by a task-specific evaluation function, and **(2)** the cost $c(\omega, q) > 0$, defined as the total token consumption incurred by ω while processing q.

Agentic Workflow Optimization. Agentic workflow optimization aims to maximize the performance of a workflow on a training dataset $D_{\text{train}} \subset D$. Existing agentic workflow optimization methods generally fall into two paradigms.

Task-level optimization searches for a static workflow by maximizing its overall performance on D_{train}, i.e., $\omega_{\text{stat}} = \arg\max_{\omega} \sum_{q \in D_{\text{train}}} e(\omega, q)$. This workflow is chosen through multiple candidates discovered through iterative exploration of the workflow space. The per-query cost is affordable in deployment, as tokens are only consumed during workflow execution. But using a single workflow for all queries may limit deployment performance due to the lack of query adaptation.

Query-level optimization uses a meta-generator G_ϕ to design a dynamic workflow $\omega^q_{\text{dyn}} = G_\phi(q)$ for each query $q \in D$. The optimization objective is to maximize the aggregate performance of query-specific workflows on D_{train} by optimizing the parameters of G_ϕ, i.e., $\phi^* = \arg\max_\phi \sum_{q \in D_{\text{train}}} e(G_\phi(q), q)$. The per-query cost of this paradigm includes both workflow generation and workflow execution. This paradigm enjoys query adaptation by workflow customization but the high inference cost can make large-scale deployment expensive.

2.2 Motivating Analysis

We investigate whether workflows discovered offline can be reused to improve performance and reduce the cost of query adaptation. To measure the performance attainable from a workflow set Ω on dataset D, we define its *coverage* as

$$\text{Coverage}(\Omega) = \frac{1}{|D|} \sum_{q \in D} \max_{\omega \in \Omega} e(\omega, q), \tag{1}$$

which measures the query-wise best performance available from the set.

We first examine whether workflows discarded by task-level search retain reusable value. Using candidates discovered by AFlow [47], we compare top-k sets ranked by individual accuracy with size-k subsets selected to maximize coverage. As shown in Figure 5, coverage increases as more

candidates are retained, and coverage-based subsets consistently outperform accuracy-ranked ones. These results show that discarded workflows retain complementary capabilities, and individual accuracy alone does not capture their value to a workflow bank.

We then examine whether query adaptation requires generating a new workflow for every query. We compare ScoreFlow [32] with an oracle that chooses between its query-level generator and AFlow's best static workflow for each query. As shown in figure 6, the oracle achieves a better performance–cost trade-off than always invoking ScoreFlow, as many queries handled by dynamic generation can already be solved by the cheaper static workflow. This demonstrates the potential of reducing generation costs through query-adaptive workflow reuse.

Together, these findings motivate building a compact bank of complementary workflows and selecting among them adaptively. Detailed experimental settings and results are provided in the Appendix D.

3 Methodology

However, realizing this idea requires addressing three challenges: **(C1)** searching for complementary workflows beyond individual accuracy, **(C2)** preserving their joint coverage in a compact portfolio, and **(C3)** selecting a suitable member for each query before observing its execution outcome. To address the above challenges, `FlowBank` adopts a portfolio-based methodology for workflow deployment. Rather than searching for a single universally best workflow, we aim to build, curate, and deploy a compact set of complementary workflows. Concretely, as shown in Figure 2, `FlowBank` consists of three stages. In *Diversifying*, DiverseFlow explores the workflow search space to generate a raw candidate pool with broad query coverage. In *Curating*, CuraFlow distills this pool into a compact portfolio that preserves most attainable coverage while removing redundant workflows. In *Matching*, we learn to predict edge utilities on a query–workflow bipartite graph and select the portfolio member with the best predicted performance–cost trade-off for each incoming query. Workflow discovery, portfolio curation, and selector training are performed offline; online inference consists of query embedding, workflow selection, and execution of the selected member.

3.1 Diversifying via DiverseFlow

Discovering Complementary Workflows. Challenge C1 arises because individual accuracy does not capture a workflow's contribution to the current pool. DiverseFlow first establishes a pool of strong individual workflows, then redirects search toward queries the pool handles poorly. Its inputs are a training set D_{train}, a seed workflow, and number of search rounds; its output is a candidate pool Ω_{raw} together with each candidate's performance and cost for each query in D_{train}.

Workflow Candidate Pool Generation. DiverseFlow builds on the tree-structured search procedure of AFlow [47]. Each search round consists of four steps. **(1)** Among the discovered workflows, one is sampled as the parent workflow using a score-based probability distribution defined below in Equation 2. **(2)** The optimizer LLM proposes a new workflow by modifying the parent's code using recorded optimization experience. **(3)** The new workflow is evaluated on D_{train}, and its per-query performance $e(\omega, q)$ and cost $c(\omega, q)$ are recorded before it is added to the candidate pool. **(4)** The optimization experience of the current round is tagged as Success/Failure based on whether it outperforms the parent and is then attached to the search tree.

Performance-oriented Warm-up. During the first N_0 rounds, DiverseFlow uses aggregate training performance as the parent-selection score, $\eta(\omega) = \sum_{q \in D_{\text{train}}} e(\omega, q)$ to favor strong overall performers. At each round, a parent workflow is sampled from the current pool according to

$$P(\omega_i) = \rho \cdot \frac{1}{n} + (1 - \rho) \cdot \frac{\exp\big(\alpha(\eta(\omega_i) - \eta_{\max})\big)}{\sum_{j=1}^{n} \exp\big(\alpha(\eta(\omega_j) - \eta_{\max})\big)}, \tag{2}$$

where n is the number of workflows currently in the pool and $\eta_{\max} = \max_{1 \leqslant j \leqslant n} \eta(\omega_j)$. The parameter ρ controls the share of uniform exploration, while α controls how strongly sampling favors high-scoring workflows. This warm-up phase initializes the pool with strong individual workflows.

Complementarity-oriented Expansion. After the warm-up phase, DiverseFlow shifts to complementarity-oriented expansion to improve coverage of queries the current pool handles poorly. The search procedure and parent-sampling rule remain unchanged. The parent-selection score now

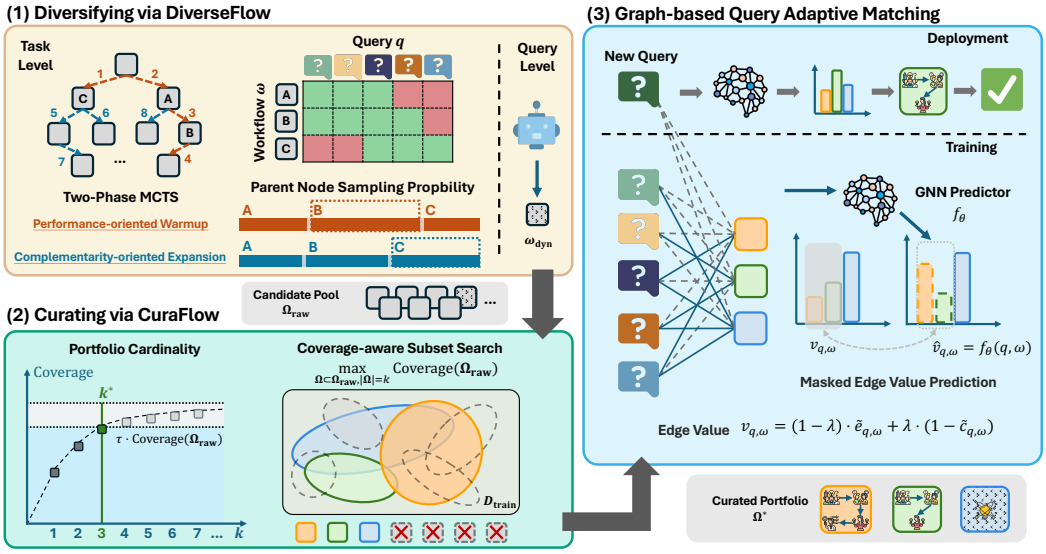

Figure 2: Overview of `FlowBank`. DiverseFlow first builds a diverse raw pool Ω_{raw} through performance-oriented warm-up followed by coverage-oriented expansion. CuraFlow then selects a compact portfolio Ω^* that retains most attainable coverage while pruning redundant workflows. Given a new query, a bipartite query–workflow matcher predicts each portfolio member's utility under the performance–cost trade-off and executes the highest-scoring workflow.

becomes $\eta(\omega) = \sum_{q \in D_{\text{train}}} \bar{\mu}(q) e(\omega, q)$, where the query weights are defined as

$$\bar{\mu}(q) = \frac{\mu(q)}{\sum_{q \in D_{\text{train}}} \mu(q)}, \quad \mu(q) = \frac{1}{1 + \sum_{i=1}^{n} e(\omega_i, q)}. \tag{3}$$

Here $\mu(q)$ reflects the difficulty of q to the pool. It reaches the maximum of 1 when no workflow solves q. The query weights are updated as the pool grows. As the toy example provided in Figure 2, workflows that perform well on under-covered queries now receive higher parent-selection scores and are therefore more likely to be sampled as parents, steering subsequent expansion toward complementary capabilities.

Completing the Pool with a Dynamic Candidate. DiverseFlow produces reusable workflow programs, each with a fixed coordination structure. Query-specific generation can provide additional solving strategies for queries these programs handle poorly, as motivated by the cross-paradigm analysis in Section 2.2. After search, we optionally augment the pool with a query-level method as one additional candidate. This candidate represents the complete generate-and-execute procedure ω_{dyn}, whose generated workflow varies with the query. We evaluate ω_{dyn} on the same training queries as the static candidates, recording its performance $e(\omega_{\text{dyn}}, q)$ and total generation-plus-execution cost $c(\omega_{\text{dyn}}, q)$. These per-query records use the same format as those of static workflows, allowing CuraFlow in the next stage to consider all candidates jointly under the same coverage objective. In our implementation, we use ScoreFlow [32] as the query-level candidate. We denote the resulting candidate pool by Ω_{raw}.

3.2 Curating via CuraFlow

From a High-Coverage Pool to a Compact Portfolio. Challenge C2 is to remove redundancy without discarding the complementary capabilities exposed during discovery. Many workflows are partially redundant, and retaining all candidates increases selector burden and deployment complexity. We therefore introduce CuraFlow, a coverage-aware combinatorial curation procedure. Given the candidate pool Ω_{raw} and its per-query performance records, CuraFlow searches for the smallest subset that preserves most of the pool's attainable coverage. We refer to this selected subset as the *curated portfolio*, which provides the workflow choices for the subsequent matching stage.

Coverage-aware Portfolio Search. We initialize the portfolio cardinality k at 1 and expand it incrementally. For each cardinality k, we find the size-k subset that maximizes coverage:

$$\Omega_k^* = \underset{\Omega \subseteq \Omega_{\text{raw}},\, |\Omega|=k}{\arg\max} \text{Coverage}(\Omega). \tag{4}$$

We enumerate over all combinatorial subsets which remains feasible when k is modest and the candidate pool is of moderate size. Coverage is computed from the recorded per-query performance, so subset comparisons require no additional workflow executions. When coverage ties, we prefer the subset with lower mean pairwise correlation between its members' per-query performance vectors. This encourages behavioral heterogeneity within the curated portfolio.

Choosing Portfolio Size by Coverage Retention. We choose the portfolio size by requiring its coverage to reach a saturation ratio $\tau \in (0, 1]$ of the full candidate pool's coverage:

$$k^* = \min\{k : \max_{\Omega \subset \Omega_{\text{raw}}, |\Omega|=k} \text{Coverage}(\Omega) \geqslant \tau \cdot \text{Coverage}(\Omega_{\text{raw}})\} \tag{5}$$

This criterion sets portfolio size according to the fraction of the pool's empirical coverage that must be retained. Because coverage is a submodular set function, its marginal gain diminishes as more workflows are added. As a result, a subset of size k can retain most of the candidate pool's coverage while removing redundant workflows, even when k is much smaller than $|\Omega_{\text{raw}}|$. When the threshold lies on a plateau of the coverage curve, we pass both Ω_k^* and $\Omega_{k^-1}^*$ to the matching stage as alternative curated portfolios and treat the portfolio size as a tunable hyperparameter.

Why DiverseFlow and CuraFlow Are Complementary. DiverseFlow discovers workflows through sequential exploration, while CuraFlow selects a compact portfolio from the completed candidate pool. Later discoveries can change which earlier workflows are most useful in combination, so post-search curation can identify high-coverage portfolios that would be missed by simply stopping exploration early. A controlled comparison is provided in Appendix F.

3.3 Matching via a Query-Workflow Graph

From Oracle Potential to an Executable Decision. Challenge C3 is the gap between available coverage and an executable selection rule: at deployment, the system knows the query and the portfolio Ω^*, but not how any member will perform on that query. Executing all members to find the winner would defeat the purpose of selective reuse. We instead train a predictor of the utility of each query-workflow pair and execute only the member with the highest predicted value. The selector must learn both when additional coordination is useful and when a cheaper workflow is sufficient.

Learning Query-Workflow Compatibility. Building on graph-based routing [9], we represent the training records as a bipartite graph $G = (D \cup \Omega^*, E)$, where D represents query nodes and Ω^* represents workflow nodes. Query nodes carry embeddings of their text; workflow nodes carry embeddings of natural-language descriptions of their behavior. This representation lets the predictor use similarities among queries and among solving procedures to estimate their compatibility. Descriptions and implementation details appear in Appendix C.4 and E.

For every pair $(q, \omega) \in D_{\text{train}} \times \Omega^*$, we create an edge $(q, \omega) \in E$ whose supervision value combines performance and cost:

$$v_{q,\omega} = (1 - \lambda) \cdot \tilde{e}_{q,\omega} + \lambda \cdot (1 - \tilde{c}_{q,\omega}). \tag{6}$$

where $\tilde{e}_{q,\omega}$ and $\tilde{c}_{q,\omega}$ are the normalized performance and cost of workflow ω on query q over D_{train}, and $\lambda \in [0, 1]$ controls the deployment trade-off between performance and token cost. Unlike a single-winner classification target, these pairwise targets preserve supervision for every candidate and allow several workflows to be equally suitable.

Train on Hidden Outcomes; Predict for Unseen Queries. We train a heterogeneous 2-layer GNN encoder followed by an MLP decoder to predict $\hat{v}_{q,\omega} = f_\theta(q, \omega)$. During training, we randomly sample minibatches of edges and mask their supervision values. Then we pass the node representations and unmasked graph information to the model, and minimize binary cross-entropy with soft targets on the masked edges. Masking requires the model to infer each held-out pairing's value from its representations and the remaining relational evidence.

Table 1: Performance and online cost across five benchmarks. All results are averaged over three runs. Cost is reported in $\times 10^{-3}$ USD per query. The Average columns are weighted by test-set size. Bold and underlined values indicate the best and second-best results, respectively. Performance is ranked across all methods, while cost is ranked among automated optimization methods and `FlowBank`.

Method	MATH		AMC		MBPP		DROP		MMLU Pro		Average	
	Perf.	Cost	Perf.	Cost	Perf.	Cost	Perf.	Cost	Perf.	Cost	Perf.	Cost
IO	56.58	0.51	53.59	0.53	72.92	0.08	75.49	0.08	50.89	0.03	60.44	0.22
Manually Designed Agentic Workflows												
CoT	55.76	0.53	52.47	0.54	74.29	0.08	77.96	0.11	51.35	0.04	60.98	0.23
ComplexCoT	54.39	0.62	49.52	0.64	71.68	0.10	77.74	0.33	64.03	0.35	63.85	0.42
Self-Consistency	57.82	3.38	53.89	3.44	73.98	0.52	78.33	0.74	50.93	0.34	61.49	1.51
Multi-agent Debate	58.92	7.66	53.82	7.96	74.68	0.86	78.65	1.64	57.56	0.88	63.86	3.45
Self-Refine	55.14	1.08	52.88	1.06	71.07	0.27	76.71	0.34	43.75	0.46	57.96	0.62
MedPrompt	56.69	5.64	55.19	5.49	71.85	0.61	81.77	1.75	52.79	0.63	62.77	2.58
MultiPersona	57.41	0.56	53.74	0.58	70.38	0.28	76.13	0.29	61.70	0.36	63.87	0.41
Automated Agentic Workflow Optimization Methods												
GPTSwarm	54.94	<u>1.69</u>	55.42	9.04	71.26	**0.25**	78.37	**0.51**	58.37	1.14	63.43	2.54
ADAS	56.17	2.95	47.33	3.22	72.34	0.77	81.17	0.98	59.71	1.05	63.22	1.71
AgentSquare	62.76	**1.35**	58.78	**1.79**	72.73	<u>0.52</u>	76.84	<u>0.76</u>	63.75	**0.75**	66.70	**1.02**
AFlow (Qwen3-8B)	63.24	3.23	62.75	2.51	79.37	1.02	77.08	1.05	64.62	1.47	68.56	1.79
AFlow (GPT-4o)	63.99	2.77	<u>63.77</u>	4.86	<u>83.77</u>	0.57	80.26	1.02	<u>65.61</u>	<u>0.89</u>	<u>70.40</u>	1.95
MaAS	<u>65.02</u>	2.43	63.36	2.70	79.08	2.04	76.64	1.64	62.31	1.29	68.09	1.90
ScoreFlow	62.00	2.36	60.15	3.51	83.63	1.70	<u>82.10</u>	1.42	64.58	2.62	69.51	2.37
`FlowBank`	**69.34**	1.78	**67.94**	<u>2.49</u>	**84.26**	1.62	**83.49**	1.06	**67.40**	1.52	**73.40**	<u>1.65</u>

At deployment, for a new query q, we embed its text, attach it as a new query node connected to the portfolio's workflow nodes in Ω^*, and predict all edge values $f_\theta(q, \omega)$ in one forward pass. The selector is then defined as

$$\pi_\theta(q) = \arg\max_{\omega \in \Omega^*} f_\theta(q, \omega), \tag{7}$$

and then execute workflow $\pi_\theta(q)$ on q to produce the final answer. Predicted ties favor the workflow with lower mean training cost, followed by a fixed deterministic order if costs also tie. The inference cost of matching a new query is one GNN forward pass over a small graph, negligible compared to the cost of executing any workflow in the portfolio.

4 Experiments

4.1 Experimental Setup

Tasks and Benchmarks. We evaluate `FlowBank` on five public benchmarks covering four domains: **(1) math reasoning**, MATH [13], AMC [5]; **(2) code generation**, MBPP [1]; **(3) question answering**, MMLU Pro [33] and **(4) reading comprehension**, DROP [7]. Following prior work, each benchmark is split into training and test sets with a 1:4 ratio for workflow optimization and final evaluation. More details can be found in Appendix C.1.

Baselines. We compare `FlowBank` with two series of baselines: **(1)manually designed workflows**: Chain-of-Thought [36], ComplexCoT [10], Self-Consistency [30], Multi-agent Debate [6], Self-Refine [21], MedPrompt [24], MultiPersona [34]; **(2) automated workflow optimization methods**: GPTSwarm [56], ADAS [16], AgentSquare [28], AFlow [47], MaAS [46], ScoreFlow [32].

LLM Backbones. For ADAS and AgentSquare, we use GPT-4o [26] as the optimizer LLM. For AFlow, we evaluate two variants using GPT-4o and Qwen3-8B [40] as the optimizer LLM respectively,

treating each as a separate baseline. For ScoreFlow, we use Qwen3-8B as the generator to design a workflow based on the given query. For FlowBank, we use Qwen3-8B as the optimizer at Stage 1. To ensure a fair comparison, we use GPT-4o mini [25] with a fixed temperature of 0 as the executor LLM across all agentic workflows. More details can be found in Appendix C.2.

Implementation Details. We run AFlow and ADAS for 30 search iteration rounds, and AgentSquare for 20. For ScoreFlow, we follow the official setting in their codebase and train its generator via Score-DPO for 3 epochs on each benchmark's training split. For FlowBank, Stage 1 runs the optimizer for 30 rounds with 7 warm-up rounds to construct a diverse candidate workflow pool. Stage 2 fixes the saturation ratio at $\tau = 0.96$ and curates two compact portfolios of sizes k^* and $k^* - 1$. Stage 3 then trains the query-adaptive selector over a grid of cost weights λ on each dataset. We provide more implementation details in Appendix C.3 and C.4.

4.2 Performance & Online Cost Analysis

Main Results. Table 1 shows that FlowBank achieves the best average performance (73.40) among all methods and consistently ranks first across all five benchmarks. Compared with the strongest automated baseline, AFlow (GPT-4o) at 70.40, FlowBank improves by 3.00 absolute points (4.26% relative), despite using Qwen3-8B rather than GPT-4o as the optimizer. The gain over the strongest manually designed workflow, MultiPersona at 63.87, is even larger at 9.53 points (14.92% relative). At the benchmark level, FlowBank attains the best results on MATH (69.34), AMC (67.94), MMLU Pro (67.40), DROP (83.49), and MBPP (84.26), surpassing the next best method by clear margins on several reasoning-intensive tasks. Overall, these results show that the performance gains come from FlowBank's 3-stage framework itself, rather than simply from using a stronger optimizer model. See Appendix E for more detailed case study.

Performance–Online Cost Trade-off. Beyond raw accuracy, FlowBank remains efficient. Its average inference cost is 1.65, below AFlow (GPT-4o) at 1.95 and ScoreFlow at 2.37, while its average performance is higher than both. AgentSquare is cheaper at 1.02, but it trails FlowBank by 6.70 points in average performance. As Figure 3 shows, no compared method offers both higher average performance and lower average cost than FlowBank. This result shows that FlowBank improves both task effectiveness and the overall performance–cost balance.

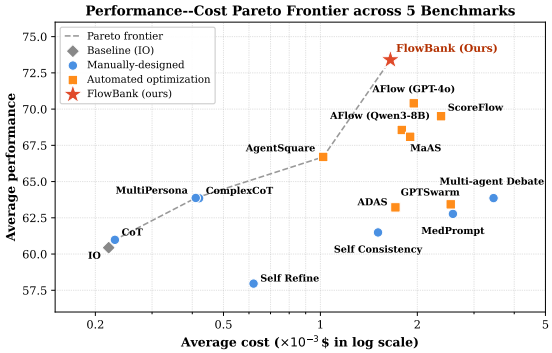

Figure 3: Performance–cost trade-off across all methods. FlowBank is on the Pareto frontier.

4.3 Framework Analysis

Ablation Study. Table 2 compares the full system against four variants to isolate the contributions of each component. Compared with Row 2, the full system replaces AFlow with DiverseFlow in Stage 1, yielding a portfolio with higher oracle performance and, in turn, better final performance, which confirms the value of constructing a more diverse candidate pool. Compared with Row 3, CuraFlow improves over the top-k accuracy set by using combinatorial search to directly maximize coverage, leading to a stronger oracle portfolio

Table 2: Ablation study of FlowBank on MATH and DROP. "Oracle" denotes the performance of an oracle selector on test data.

Method			MATH		DROP	
Stage 1	Stage 2	Stage 3	Perf.	Oracle	Perf.	Oracle
✓	✓	✓	**69.34**	83.74	**83.49**	90.47
×(AFlow)	✓	✓	67.69	79.01	82.98	89.95
✓	×(Top-k Accuracy set)	✓	67.90	80.66	82.37	86.73
✓	×(Full candidate pool)	✓	68.11	**92.04**	82.45	**96.59**
✓	✓	×(MLP Classifier)	68.72	83.74	82.83	90.47

Table 3: Effect of removing the ScoreFlow candidate from FlowBank. Cost denotes average online cost in $\times 10^{-3}$ USD per query. The Average columns are weighted by test-set size.

Method	MATH		AMC		MBPP		DROP		MMLU Pro		Average	
	Perf.	Cost	Perf.	Cost	Perf.	Cost	Perf.	Cost	Perf.	Cost	Perf.	Cost
FlowBank	69.34	1.78	67.94	2.49	84.26	1.62	83.49	1.06	67.40	1.52	73.40	1.65
FlowBank w/o ScoreFlow	69.34†	1.78	67.48	1.83	82.70	1.01	82.14	0.30	67.40†	1.52	72.82	1.27

ScoreFlow is removed from the candidate pool before portfolio curation, and curation and selector training are rerun on the affected benchmarks. †The original portfolios on MATH and MMLU Pro already exclude ScoreFlow, so their results are unchanged and reused.

and higher downstream performance as well. Compared with Row 4, training the selector on the full candidate pool without curation causes noticeable degradation, suggesting that redundancy makes it harder for the selector to exploit the distinct strengths of individual workflows. Finally, compared with Row 5, the bipartite-graph based GNN selector consistently outperforms an independent flat MLP classifier, validating the benefit of explicitly modeling query–workflow interactions over a graph. Overall, the best results are achieved only when all three stages are activated together.

Effect of the Dynamic Candidate. We evaluate the contribution of the dynamic candidate by removing ScoreFlow from the candidate pool before CuraFlow and rerunning portfolio curation and selector training on the affected benchmarks. As shown in Table 3, average performance decreases by only 0.58 percentage points, from 73.40 to 72.82, while average online cost falls from 1.65 to 1.27, a reduction of approximately 23%. The largest benchmark-level performance drop is 1.56 points, and the static configuration still surpasses the strongest baseline's average performance of 70.40. The small performance difference reflects ScoreFlow's limited exclusive coverage: removing it leaves the pool's coverage unchanged on DROP and MBPP and reduces it by only 1.6 points on AMC. CuraFlow can therefore select effective alternative portfolios from the complementary static workflows. These results show that a fully static candidate pool retains the main benefits of FlowBank, while dynamic generation offers an additional performance gain at higher online cost.

Hyperparameter Effects. Figure 4 studies two key hyperparameters in FlowBank and shows that AMC performance increases from 61.07 at $K{=}1$ to 67.94 at $K{=}5$ before dipping slightly at $K{=}6$. This pattern supports choosing a compact, diverse portfolio for effective query-adaptive matching. On MMLU Pro, larger λ steadily lowers inference cost while performance stays stable for $\lambda \in [0.1, 0.4]$ and drops at $\lambda{=}0.5$, showing the balance between effect and token efficiency. Overall, FlowBank is robust to moderate changes in λ and maintains a strong performance–cost trade-off without heavy tuning.

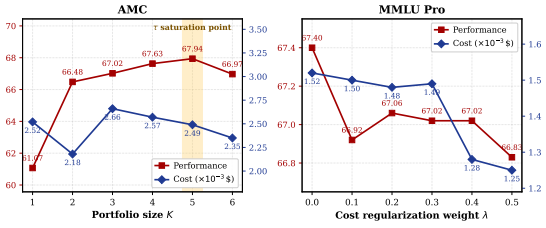

Figure 4: **Left:** Impact of portfolio size k on AMC; **Right:** Impact of cost regularization weight λ on MMLU Pro.

4.4 Offline Cost Audit

Accounting Scope. We audit offline costs on MATH using uniform token and GPU prices across methods. The accounting covers workflow execution during search and evaluation, optimizer LLM calls, model training, workflow description generation, embeddings, portfolio curation, and selector training. Detailed unit prices and accounting protocols are provided in the appendix.

Offline Cost Breakdown. Table 4 compares offline costs across methods and presents the component-wise breakdown of FlowBank. The full configuration costs $\$36.97$, of which $\$27.94$ (approximately 76%) comes from the optional ScoreFlow pipeline. Building the candidate pool without ScoreFlow reduces the offline cost to $\$9.03$, comparable to AFlow (Qwen3-8B) at $\$8.96$ and MaAS at $\$8.39$. On MATH, the deployed portfolio already excludes ScoreFlow, so this reduction leaves test performance unchanged at 69.34. Outside the ScoreFlow pipeline, the main expense is

Table 4: Offline cost audit on MATH. (a) Total offline cost comparison. (b) Component-wise breakdown of FlowBank. All costs are reported in USD under uniform pricing.

(a) Offline Cost Comparison

Method	Cost ($)
ADAS	11.56
AFlow (Qwen3-8B)	8.96
AFlow (GPT-4o)	13.41
MaAS	8.39
ScoreFlow	27.94
FlowBank	**36.97**
FlowBank w/o ScoreFlow	**9.03**

(b) FlowBank Cost Breakdown

Component	Cost ($)
DiverseFlow: workflow execution	8.95
DiverseFlow: optimizer calls	0.03
ScoreFlow: offline pipeline	27.94
Workflow descriptions	0.03
Embeddings computation	0.001
CuraFlow: selection (CPU)	≈ 0
GNN selector training	0.02

The w/o ScoreFlow configuration excludes the optional ScoreFlow offline pipeline. Totals are rounded to two decimal places.

DiverseFlow's workflow execution ($8.95), while GNN selector training costs only $0.02. Together with the static-portfolio results in Table 3, this audit demonstrates that FlowBank can retain strong performance with substantially lower offline investment when using an entirely static candidate pool.

5 Related Work

LLM-based Agentic Workflow. Recent work has shown that LLMs can benefit substantially from explicit workflow design rather than a single prompt-response pass. Representative manually designed strategies include Chain-of-Thought, Self-Consistency, debate-based collaboration, and iterative refinement, as well as more specialized designs such as MedPrompt, MultiPersona and MetaGPT [36, 30, 6, 21, 24, 34, 15]. These methods demonstrate the value of workflow structure and prompting, but they usually rely on manually designed workflows.

Automated Agentic Workflow Optimization. Another line of work studies how to automatically discover and optimize workflows, including task-level offline workflow search methods such as GPTSwarm, ADAS, AgentSquare, AFlow and Weak-for-strong, as well as more adaptive query-level online generation systems such as MAS-GPT, ScoreFlow, MaAS, MasRouter and DyFlow [56, 16, 28, 47, 42, 32, 46, 44, 23, 31]. These methods reduce manual effort and often improve average task performance, but most either optimize for a single final workflow or shift substantial workflow construction to inference time. In contrast, FlowBank constructs a compact portfolio of complementary workflows offline and learns to select the best one for each query at inference time, making our method closer to a precompute-and-reuse framework than to either single-workflow optimization or fully online workflow generation.

6 Conclusion

We revisit agentic workflow optimization from a different perspective. Rather than choosing between a single workflow for all queries and synthesizing a new workflow for every query, we ask how to recover query-level adaptivity through offline computation and reusable workflow assets. FlowBank operationalizes this idea with three stages—*Diversifying*, *Curating*, and *Matching*—that discover complementary workflows, distill them into a compact portfolio, and assign each query to the right workflow at inference time. Across five benchmarks, this precompute-and-reuse strategy delivers the strongest average performance among the evaluated methods while maintaining competitive cost. More detailed discussion about limitations and future directions can be found at Appendix G.

References

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

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

[3] Yitong Cui, Liu Liu, Baosheng Yu, Jiayan Qiu, Xikai Zhang, Likang Xiao, Yixing Liu, and Quan Chen. Spogw: a score-based preference optimization method via group-wise comparison for workflows. *arXiv preprint arXiv:2510.04089*, 2025.

[4] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

[5] Mucong Ding, Chenghao Deng, Jocelyn Choo, Zichu Wu, Aakriti Agrawal, Avi Schwarzschild, Tianyi Zhou, Tom Goldstein, John Langford, Anima Anandkumar, et al. Easy2hard-bench: Standardized difficulty labels for profiling llm performance and generalization. *Advances in Neural Information Processing Systems*, 37:44323–44365, 2024.

[6] Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first international conference on machine learning*, 2024.

[7] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378, 2019.

[8] Batu El, Mert Yuksekgonul, and James Zou. Inefficiencies of meta agents for agent design. In *ICML 2025 Workshop on Programmatic Representations for Agent Learning*, 2025.

[9] Tao Feng, Yanzhen Shen, and Jiaxuan You. Graphrouter: A graph-based router for llm selections. *arXiv preprint arXiv:2410.03834*, 2024.

[10] Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. Complexity-based prompting for multi-step reasoning. *arXiv preprint arXiv:2210.00720*, 2022.

[11] Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

[12] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

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

[14] Matthew Ho, Chen Si, Zhaoxiang Feng, Fangxu Yu, Yichi Yang, Zhijian Liu, Zhiting Hu, and Lianhui Qin. Arcmemo: Abstract reasoning composition with lifelong llm memory. *arXiv preprint arXiv:2509.04439*, 2025.

[15] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pages 23247–23275, 2024.

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

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

[18] Xiaoxiao Li. When single-agent with skills replace multi-agent systems and when they fail. *arXiv preprint arXiv:2601.04748*, 2026.

[19] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*, 2023.

[20] Yinyi Luo, Yiqiao Jin, Weichen Yu, Mengqi Zhang, Srijan Kumar, Xiaoxiao Li, Weijie Xu, Xin Chen, and Jindong Wang. Agentark: Distilling multi-agent intelligence into a single llm agent. *arXiv preprint arXiv:2602.03955*, 2026.

[21] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

[22] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[23] Fan Nie, Lan Feng, Haotian Ye, Weixin Liang, Pan Lu, Huaxiu Yao, Alexandre Alahi, and James Zou. Weak-for-strong: Training weak meta-agent to harness strong executors. *arXiv preprint arXiv:2504.04785*, 2025.

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

[25] OpenAI. GPT-4o mini: Advancing cost-efficient intelligence, 2024. Accessed: 2026-04-30.

[26] OpenAI. Hello GPT-4o, 2024. Accessed: 2026-04-30.

[27] Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026.

[28] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. *arXiv preprint arXiv:2410.06153*, 2024.

[29] Kun Wang, Guibin Zhang, ManKit Ye, Xinyu Deng, Dongxia Wang, Xiaobin Hu, Jinyang Guo, Yang Liu, and Yufei Guo. MAS^2: Self-generative, self-configuring, self-rectifying multi-agent systems. *arXiv preprint arXiv:2509.24323*, 2025.

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

[31] Yanbo Wang, Zixiang Xu, Yue Huang, Xiangqi Wang, Zirui Song, Lang Gao, Chenxi Wang, Xiangru Tang, Yue Zhao, Arman Cohan, et al. Dyflow: Dynamic workflow framework for agentic reasoning. *arXiv preprint arXiv:2509.26062*, 2025.

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

[33] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.

[34] Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 257–279, 2024.

[35] Zixu Wang, Bingbing Xu, Yige Yuan, Huawei Shen, and Xueqi Cheng. Do we always need query-level workflows? rethinking agentic workflow generation for multi-agent systems. *arXiv preprint arXiv:2601.11147*, 2026.

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

[37] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

[38] Jiawei Xu, Arief Koesdwiady, Sisong Bei, Yan Han, Baixiang Huang, Dakuo Wang, Yutong Chen, Zheshen Wang, Peihao Wang, Pan Li, et al. Rethinking the value of multi-agent workflow: A strong single agent baseline. *arXiv preprint arXiv:2601.12307*, 2026.

[39] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

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

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

[42] Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. Mas-gpt: Training llms to build llm-based multi-agent systems. *arXiv preprint arXiv:2503.03686*, 2025.

[43] Fangxu Yu, Xingang Guo, Lingzhi Yuan, Haoqiang Kang, Hongyu Zhao, Lianhui Qin, Furong Huang, Bin Hu, and Tianyi Zhou. Tsrbench: A comprehensive multi-task multi-modal time series reasoning benchmark for generalist models. *arXiv preprint arXiv:2601.18744*, 2026.

[44] Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. Masrouter: Learning to route llms for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15549–15572, 2025.

[45] Guibin Zhang, Kaijie Chen, Guancheng Wan, Heng Chang, Hong Cheng, Kun Wang, Shuyue Hu, and Lei Bai. Evoflow: Evolving diverse agentic workflows on the fly. *arXiv preprint arXiv:2502.07373*, 2025.

[46] Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, LEI BAI, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Forty-second International Conference on Machine Learning*, 2025.

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

[48] Mingda Zhang, Haoran Luo, Tiesunlong Shen, Qika Lin, Xiaoying Tang, Rui Mao, and Erik Cambria. Flowsteer: Interactive agentic workflow orchestration via end-to-end reinforcement learning. *arXiv preprint arXiv:2602.01664*, 2026.

[49] Yuanshuo Zhang, Yuchen Hou, Bohan Tang, Shuo Chen, Muhan Zhang, Xiaowen Dong, and Siheng Chen. FLORA: GNNs as predictors of agentic workflow performances. In *The Fourth Learning on Graphs Conference*, 2025.

[50] Yuxuan Zhao, Sijia Chen, and Ningxin Su. On the uncertainty of large language model-based multi-agent systems. *arXiv preprint arXiv:2602.04234*, 2026.

[51] Chengqi Zheng, Jianda Chen, Yueming Lyu, Wen Zheng Terence Ng, Haopeng Zhang, Yew-Soon Ong, Ivor Tsang, and Haiyan Yin. Mermaidflow: Redefining agentic workflow generation via safety-constrained evolutionary programming. *arXiv preprint arXiv:2505.22967*, 2025.

[52] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731, 2024.

[53] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*, 2025.

[54] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

[55] Runchuan Zhu, Bowen Jiang, Lingrui Mei, Fangkai Yang, Lu Wang, Haoxiang Gao, Fengshuo Bai, Pu Zhao, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AdaptFlow: Adaptive workflow optimization via meta-learning. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 3287–3302, Suzhou, China, November 2025. Association for Computational Linguistics.

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

A Notations

This appendix summarizes the main symbols used in Section 2 and Section 3.

Table 5: Notation summary for the formulation and the three stages of `FlowBank`.

Symbol	Meaning
D	Task dataset.
D_{train}	Training split.
q	Input query.
ω	Agentic workflow.
$\omega(q)$	Workflow prediction on q.
$e(\omega, q)$	Workflow performance on q.
$c(\omega, q)$	Workflow token cost on q.
ω_{stat}	Static workflow shared by all queries.
G_ϕ	Meta-generator for dynamic workflows.
$\omega^q_{\text{dyn}} = G_\phi(q)$	Dynamic workflow generated for q.
Ω	A set consists of workflows.
Ω_{raw}	Raw candidate workflow pool.
$\mu(q)$	Complementarity-aware weight of query q.
$\text{Coverage}(\Omega)$	Set-level utility of subset Ω.
k^*	Chosen portfolio size.
τ	Coverage saturation ratio.
Ω^*	Final curated portfolio.
$G = (D \cup \Omega^*, E)$	Query-workflow bipartite graph.
$v_{q,\omega}$	Edge supervision value for (q, ω).
$\tilde{e}_{q,\omega}$	Normalized effect of ω on q.
$\tilde{c}_{q,\omega}$	Normalized cost of ω on q.
λ	Effect-cost trade-off weight.
$f_\theta(q, \omega)$	Predicted value of matching q with ω.
$\pi_\theta(q)$	Selected workflow for query q.

B Algorithm Table

Algorithm 1 summarizes Stage 1, where DiverseFlow maintains a growing workflow pool and shifts from performance-oriented search to coverage-oriented search. Algorithm 2 summarizes CuraFlow, the combinatorial curation procedure in Stage 2. Algorithm 3 describes Stage 3, where a lightweight graph-based selector predicts the best workflow for each query.

Algorithm 1 DiverseFlow for Building a High-Coverage Candidate Pool

Require: Training split D_{train}, search budget N, warm-up rounds N_0
Ensure: Raw workflow pool Ω_{raw}
  Initialize workflow pool Ω_0 with seed workflows
  for $t = 1$ to N **do**
    if $t \leqslant N_0$ **then**
      Score each workflow in Ω_{t-1} by its aggregate performance on D_{train}
    else
      Compute query weights $\mu(q)$ from the current pool coverage on D_{train}
      Re-score each workflow in Ω_{t-1} by weighted performance under $\mu(q)$
    end if
    Sample a parent workflow ω^{parent} from Ω_{t-1}
    Ask the optimizer LLM to propose a new workflow ω^{new} conditioned on ω^{parent} and search experience
    Execute ω^{new} on D_{train} to obtain per-query effect and token cost
    Compare ω^{new} against ω^{parent} under the current scoring rule and record success/failure feedback
    Update the workflow pool: $\Omega_t \leftarrow \Omega_{t-1} \cup \{\omega^{\text{new}}\}$
  end for
  return $\Omega_{\text{raw}} \leftarrow \Omega_N$

Algorithm 2 CuraFlow for Combinatorial Portfolio Curation

Require: Raw candidate pool Ω_{raw}, training split D_{train}, saturation ratio τ
Ensure: Candidate curated portfolios for the matching stage
  for $k = 1$ to $|\Omega_{\text{raw}}|$ **do**
    Enumerate size-k subsets $\Omega \subseteq \Omega_{\text{raw}}$
    Compute $\text{Coverage}(\Omega)$ on D_{train}
    Select $\Omega_k^* = \arg\max_{|\Omega|=k} \text{Coverage}(\Omega)$
    Break ties by preferring lower mean pairwise correlation between per-query effect vectors
  end for
  Find the smallest k^* such that
    $\text{Coverage}(\Omega_{k*}^*) \geqslant \tau \cdot \text{Coverage}(\Omega_{\text{raw}})$
  Form candidate curated portfolios from Ω_{k*}^* and, when applicable, Ω_{k*-1}^*
  return candidate portfolios Ω^* for Stage 3

Algorithm 3 Query-Adaptive Matching over the Curated Portfolio

Require: Curated portfolio Ω^*, training split D_{train}, trade-off coefficient λ, new query q
Ensure: Selected workflow $\pi_\theta(q)$
  Build bipartite graph $G = (D_{\text{train}} \cup \Omega^*, E)$ with complete query-workflow edges
  for all $(q', \omega) \in D_{\text{train}} \times \Omega^*$ **do**
    Compute normalized effect $\tilde{e}_{q',\omega}$ and normalized cost $\tilde{c}_{q',\omega}$
    Assign edge supervision value $v_{q',\omega} = (1-\lambda)\tilde{e}_{q',\omega} + \lambda(1-\tilde{c}_{q',\omega})$
  end for
  Train a heterogeneous 2-layer GNN with an MLP decoder to predict $f_\theta(q', \omega)$ from node embeddings and observed edges
  Add the new query q as a query node and connect it to every workflow in Ω^*
  Perform one forward pass to obtain $f_\theta(q, \omega)$ for all $\omega \in \Omega^*$
  Select the best workflow:
    $\pi_\theta(q) \leftarrow \arg\max_{\omega \in \Omega*} f_\theta(q, \omega)$
  Break ties by preferring the lower-cost workflow
  return $\pi_\theta(q)$

Table 6: Statistics of the training and test splits for the five benchmarks.

Dataset	Training	Testing
MATH	119	486
AMC	165	655
MBPP	86	341
DROP	200	800
MMLU Pro	260	1040

C Detailed Experimental Setup

C.1 Datasets and Splits

We evaluate all methods on five public benchmarks spanning four domains: MATH and AMC for mathematical reasoning, MBPP for code generation, MMLU Pro for question answering, and DROP for reading comprehension. Following prior workflow-optimization work [16, 47, 32, 23], we split each benchmark into training and test sets with a $1{:}4$ ratio, where the training split is used for workflow search, curation, or selector training, and the test split is reserved for final evaluation. Table 6 summarizes the resulting dataset statistics.

For MATH, we follow previous works [16, 47] in selecting 605 problems from four typical problem types (Combinatorics & Probability, Number Theory, Pre-algebra, Pre-calculus) at difficulty level 5. For AMC, we use the Easy2Hard benchmark [5] and retain problems whose IRT difficulty ratings fall in $[0.25, 0.40]$, resulting in 820 problems in total. For MMLU Pro, we exclude the duplicated math category and then uniformly sample 100 problems from each of the remaining 13 categories, yielding 1,300 problems.

C.2 Model Configurations

We use GPT-4o [26] and Qwen3-8B [40] as optimizer-side large language models in different methods, depending on each baseline's design. Specifically, ADAS and AgentSquare use GPT-4o as the optimizer LLM. For AFlow, we report two variants, one with GPT-4o and the other with Qwen3-8B as the optimizer, and treat them as separate baselines. ScoreFlow uses Qwen3-8B as the generator for workflow design, and `FlowBank` also uses Qwen3-8B as the optimizer in Stage 1.

To keep execution-time comparison fair across all agentic workflows, we use GPT-4o mini [25] with temperature fixed to 0 as the executor LLM throughout the experiments. For GPT-4o, we use the default configuration. For Qwen3-8B, we use temperature 0.7 in non-reasoning mode.

C.3 Baselines and Setup

We compare `FlowBank` against two categories of baselines. The first category consists of manually designed workflows, including Chain-of-Thought [36], ComplexCoT [10], Self-Consistency [30], Multi-agent Debate [6], Self Refine [21], MedPrompt [24], and MultiPersona [34]. The second category consists of automated workflow optimization methods, including four task-level methods: GPTSwarm [56], ADAS [16], AgentSquare [28], AFlow [47] and two query-level methods: MaAS [46] and ScoreFlow [32].

For manually designed workflows, we implement each method according to its standard formulation. For CoT, we prepend the instruction "let's think step by step" before answer generation. For ComplexCoT, we use a more structured prompt that instructs the model to identify variables, formulate a plan, execute the solution, and verify the result. For Self-Consistency, we sample $n = 5$ CoT responses and aggregate them using an LLM-based ensemble vote. For Self-Refine, we first generate an initial CoT solution and then perform up to three rounds of review and revision, with early stopping when the reviewer judges the current answer to be correct. For Multi-Agent Debate, three agents independently produce CoT solutions, exchange their answers for two rounds of debate, and then produce a final prediction through ensemble voting. For MultiPersona, a single LLM call identifies task-relevant personas and simulates a multi-turn collaboration among them within one generation. For MedPrompt, we retrieve the top-3 most similar few-shot exemplars using embedding similarity, sample five conditioned solutions, and ensemble the resulting predictions.

For search-based baselines, we follow the standard optimization budgets and implementation settings whenever applicable. Specifically, we run AFlow and ADAS for 30 search iterations, and AgentSquare for 20 iterations. For GPTSwarm, we adopt the edge-optimization framework with REINFORCE training, following the official implementation settings, with a learning rate of 0.1, an initial connection probability of 0.5, and a per-problem moving-average baseline. For MaAS, we follow the default configuration reported in the original paper [46], using a four-layer query-aware controller trained for four sampling repetitions and GPT-4o mini as the TextGrad-based prompt-refinement model. For ScoreFlow, we follow the official setting in its released codebase and train the generator with Score-DPO for three epochs on the training split of each benchmark. All methods are evaluated under the same executor model and test protocol described above.

C.4 Implementation Details of `FlowBank`

Our implementation follows the three-stage framework introduced in Section 3. In Stage 1, we run DiverseFlow with Qwen3-8B as the optimizer for 30 rounds on the training split, using 7 warmup rounds before the main diversified search. After that, we also incorporate ScoreFlow into the final candidate pool Ω_{raw}. Specifically, for each training query, ScoreFlow first generates a workflow and then executes it on that query. We record its task-specific performance and total token cost on each training query, including the cost of both workflow generation and execution. These per-query performance–cost records use the same format as those of static workflows, allowing CuraFlow and the selector to treat ScoreFlow through the same candidate interface.

In Stage 2, we fix the saturation ratio to $\tau = 0.96$ across all benchmarks and determine k^* from the training-split coverage curve. We retain the coverage-maximizing portfolios of sizes k^* and $k^* - 1$ as two candidates for the matching stage. In practice, the raw candidate pool remains moderate, with $|\Omega_{\text{raw}}| = 30 + 1 = 31$, where the additional candidate comes from ScoreFlow. Moreover, as shown in Figure 8, training-split coverage typically saturates with at most six workflows. Since

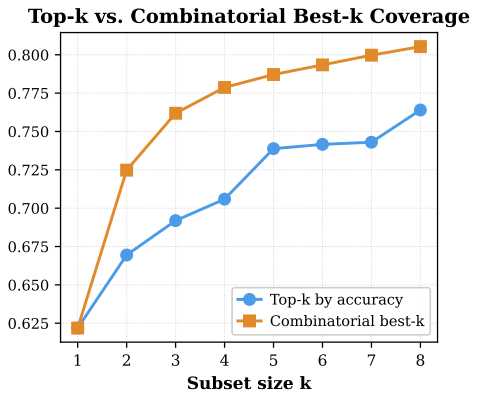

Figure 5: Coverage on MATH for workflow sets built from AFlow candidates.

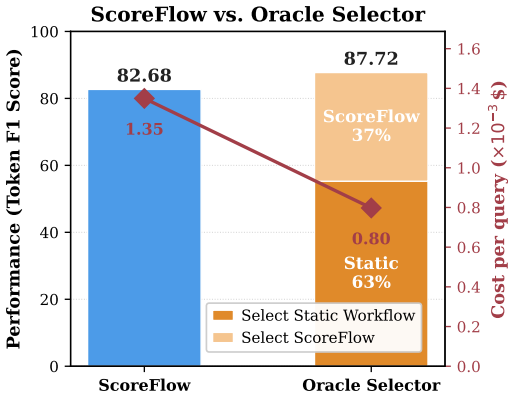

Figure 6: Performance–cost comparison on DROP between ScoreFlow and an oracle selector.

both the candidate-pool size and the effective search range are small, exhaustive subset search is computationally affordable in our setting. This design gives a near-saturated portfolio together with a slightly smaller alternative, which supports the subsequent performance–cost trade-off analysis. The final portfolios include ScoreFlow on AMC, MBPP, and DROP, but not on MATH or MMLU Pro.

For larger candidate pools or target portfolio sizes, CuraFlow admits a scalable greedy variant in place of exhaustive subset enumeration. Starting from the empty set, this variant grows the portfolio one workflow at a time, and at each step adds the candidate with the largest marginal increase in Coverage. Under our definition of Coverage as the average query-wise best performance over a workflow set, the resulting objective is monotone submodular as long as per-query performance is nonnegative. Therefore, for any fixed budget k, the greedy portfolio achieves at least a $(1 - 1/e)$ fraction of the optimal size-k Coverage. In practice, one can first build the nested greedy sequence and then select the smallest k whose Coverage reaches the saturation threshold, preserving the core behavior of CuraFlow while scaling to substantially larger search spaces.

In Stage 3, following [9], we first use GPT-4o to generate a natural-language description for each workflow in the curated portfolio (specific examples provided in Appendix E). We then encode both user queries and workflow descriptions using OpenAI's `text-embedding-3-small` model, construct the bipartite query–workflow graph, and train the GNN-based selector over a grid of cost regularization weights $\lambda \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ on each benchmark. At inference time, the selector predicts the most suitable workflow for each query under the learned performance–cost trade-off. All GNN-based selectors are trained on a single NVIDIA RTX A5000 GPU with 24 GB of memory. Each run uses approximately 300–400 MB of GPU memory, allowing us to evaluate multiple hyperparameter configurations concurrently on a single device. A 500-step training run takes approximately 7–8 minutes, and a complete hyperparameter sweep can be finished within one hour of wall-clock time on a single GPU.

D Detailed Motivating Analysis

D.1 Potential of Reusing Workflows through Task-level Searching.

Task-level workflow optimization explores many candidate workflows, but deployment ultimately uses only one. This raises a natural question: do the remaining workflows still provide reusable value as a set? We instantiate this analysis with AFlow [47], a representative task-level workflow optimizer that performs offline Monte Carlo tree search and ultimately deploys a fixed workflow. Specifically, we study the 20 candidate workflows produced by 20 rounds of AFlow optimization on MATH [13]. For each set size $k \in \{1, \ldots, 20\}$, we first select the top-k workflows ranked by individual accuracy. As shown by the blue curve in Fig. 5, the coverage of this set increases steadily with k, indicating that unselected workflows for final deployment still retain reusable value when combined as a set. We then compare it with the combinatorially best size-k set, obtained by exhaustive search over all $\binom{20}{k}$ subsets, shown by the orange curve. The consistent gap between the two curves shows that

accuracy-based construction is suboptimal for set coverage. This suggests a workflow pool optimized for single-workflow accuracy may not be the best one for set-level reuse.

D.2 Potential of Reducing Redundant Workflow Generation for Query Adaptation.

To examine whether the extra inference cost for query adaptation is always necessary, we use ScoreFlow [32] as a representative query-level method, which uses DPO to train a meta-generator and designs a customized workflow for each query at inference time. We combine it with the best static workflow by AFlow. For each query, an oracle chooses the workflow with higher performance, breaking ties in favor of the one with lower cost. As shown in Fig. 6, a substantial fraction of the queries solved by ScoreFlow can be handled by the cheaper static workflow. This suggests that token cost in query-level method can be reduced by diverting queries smartly. The oracle's superior performance–cost trade-off further highlights the potential of integrating static and query-level workflows within a unified framework rather than treating them as competing paradigms.

E Case Study

E.1 AMC

- **AMC W1**

Description

Three-way diverse-prompt ensemble (no code verification). Custom operator generates 3 independent solutions under different instruction styles — SOLVE_PROMPT (general), ANALYTICAL_SOLVE_PROMPT (deductive analysis), and VERBOSE_SOLVE_PROMPT (verbose explanation) — then ScEnsemble selects the most consistent answer via majority voting. Four LLM calls in total. Lighter than W2/W3 because it does not use Programmer-based verification. It is suited to AMC problems where reframing the question through different prompt styles surfaces the correct reasoning path more reliably than a single fixed prompt.

Python Code

```
SOLVE_PROMPT = """
You are a math problem solver. Please solve the following problem step by step, showing all your
      work and providing a final answer in a box.

Problem:
{problem}

Solution:
"""

ANALYTICAL_SOLVE_PROMPT = """
You are a math problem solver. Please solve the following problem step by step, showing all your
      work and providing a final answer in a box. Focus on breaking down the problem into smaller
      parts and analyze each part carefully.

Problem:
{problem}

Solution:
"""

VERBOSE_SOLVE_PROMPT = """
You are a math problem solver. Please solve the following problem step by step, showing all your
      work and providing a final answer in a box. Be as detailed as possible in your explanation,
      and make sure to explain the reasoning behind each step.

Problem:
{problem}

Solution:
"""
```

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

    async def __call__(self, problem: str):
        # Three diverse-style candidates, then ensemble-vote
        prompts = [SOLVE_PROMPT, ANALYTICAL_SOLVE_PROMPT, VERBOSE_SOLVE_PROMPT]
        solutions = [await self.custom(input=problem, instruction=p) for p in prompts]
        best_solution = await self.sc_ensemble(solutions=solutions, problem=problem)
        return best_solution, self.llm.usage_tracker.get_summary()["total_cost"]
```

Assigned Query Example

Suppose a and b are single-digit positive integers chosen independently and at random. What is the probability that the point (a, b) lies above the parabola $y = ax^2 - bx$?

- **AMC W2**

Description

Three-way analytical Custom ensemble with code verification. Generates 3 independent analytical solutions using a Custom operator with a focused analytical prompt, ensembles them via ScEnsemble majority voting, then verifies the best solution with a Programmer operator that executes Python code. Effective for AMC problems that benefit from diverse analytical reasoning plus computational validation.

Python Code

```python
ANALYTICAL_SOLVE_PROMPT = """
You are a math problem solver. Please solve the following problem step by step, showing all your
      work and providing a final answer in a box. Focus on breaking down the problem into smaller
      parts and analyze each part carefully. Make sure to explain your reasoning and avoid
      assumptions.

Problem:
{problem}

Solution:
"""


class Workflow:
    def __init__(self, name, llm_config, dataset):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)
        self.programmer = operator.Programmer(self.llm)

    async def __call__(self, problem: str):
        # Three samples from the same analytical prompt, then ensemble-vote
        solutions = [
            await self.custom(input=problem, instruction=ANALYTICAL_SOLVE_PROMPT)
            for _ in range(3)
        ]
        best_solution = await self.sc_ensemble(solutions=solutions, problem=problem)

        # Verify the ensemble pick with executed Python code
        final_solution = await self.programmer(problem=problem, analysis=best_solution)
        return final_solution, self.llm.usage_tracker.get_summary()["total_cost"]
```

Assigned Query Example

Two distinct numbers a and b are chosen randomly from the set $\{2, 2^2, 2^3, ..., 2^{25}\}$. What is the probability that $\log_a b$ is an integer?

- **AMC W3**

Description

Three-way diverse-prompting ensemble with code verification. Generates 3 solutions using different prompt strategies (logical reasoning, thorough explanation, concise structure), ensembles via ScEnsemble, then verifies the chosen solution with a Programmer operator. Effective for AMC problems where varied reasoning approaches increase coverage of the solution space and code execution catches arithmetic slips.

Python Code

```
LOGICAL_PROMPT = "You are a math problem solver. Please solve the following problem step by step,
      showing all your work and providing a final answer in a box. Focus on logical reasoning and
      clarity."

THOROUGH_PROMPT = "You are a math problem solver. Please solve the following problem step by step,
      showing all your work and providing a final answer in a box. Be thorough and explain your
      thought process."

CONCISE_PROMPT = "You are a math problem solver. Please solve the following problem step by step,
      showing all your work and providing a final answer in a box. Make sure your solution is
      concise and well-structured."


class Workflow:
    def __init__(self, name, llm_config, dataset):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)
        self.programmer = operator.Programmer(self.llm)

    async def __call__(self, problem: str):
        # Three diverse-style candidates, then ensemble-vote
        prompts = [LOGICAL_PROMPT, THOROUGH_PROMPT, CONCISE_PROMPT]
        solutions = [await self.custom(input=problem, instruction=p) for p in prompts]
        best_solution = await self.sc_ensemble(solutions=solutions, problem=problem)

        # Verify with executed Python code
        final_solution = await self.programmer(problem=problem, analysis=best_solution)
        return final_solution, self.llm.usage_tracker.get_summary()["total_cost"]
```

Assigned Query Example

Kiana has two older twin brothers. The product of their three ages is 128. What is the sum of their three ages?

- **AMC W4**

Description

Classify-then-solve conditional routing workflow. First classifies the problem into a category (geometry, combinatorics, number theory, or algebra) via a Custom operator with a classifier prompt, then routes the query to a domain-specialist Custom call whose prompt is tailored for that category. Two LLM calls total — the cheapest WF in the

AMC pool. Effective when domain-specific reasoning strategies significantly outperform generic prompting; weak when the classifier mis-routes.

Python Code

```
CLASSIFY_PROMPT = """Analyze this math problem and classify it into exactly one category. Output
      ONLY the category name, nothing else.

Categories:
- ALGEBRA: equations, inequalities, functions, polynomials, sequences
- GEOMETRY: shapes, angles, areas, volumes, coordinate geometry
- COMBINATORICS: counting, probability, permutations, combinations
- NUMBER_THEORY: divisibility, primes, modular arithmetic, digits
"""

ALGEBRA_PROMPT = """You are an algebra specialist. Solve this problem using algebraic techniques:
- Set up equations from the problem conditions
- Solve systematically (substitution, elimination, factoring)
- Check your answer satisfies all original conditions

Put your final answer in \\boxed{{}}.
"""

GEOMETRY_PROMPT = """You are a geometry specialist. Solve this problem using geometric reasoning:
- Draw a mental picture and label key points/lengths/angles
- Apply relevant theorems (Pythagorean, similar triangles, circle properties)
- Use coordinate geometry if the problem involves positions

Put your final answer in \\boxed{{}}.
"""

COMBINATORICS_PROMPT = """You are a combinatorics specialist. Solve this problem using counting
      techniques:
- Identify what is being counted or the probability space
- Choose the right technique: multiplication principle, combinations, inclusion-exclusion,
      complementary counting
- Double-check by considering small cases if possible

Put your final answer in \\boxed{{}}.
"""

NUMBER_THEORY_PROMPT = """You are a number theory specialist. Solve this problem using number-
      theoretic techniques:
- Consider divisibility rules, prime factorizations, modular arithmetic
- Look for patterns in small cases
- Apply relevant theorems (CRT, Fermat's little theorem, etc.)

Put your final answer in \\boxed{{}}.
"""


class Workflow:
    """Classify problem type, then route to a domain-specialist prompt."""
    def __init__(self, name, llm_config, dataset):
        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):
        # Classify into one of four math domains
        category = (await self.custom(input=problem, instruction=CLASSIFY_PROMPT)).strip().upper()

        # Route to a specialist prompt (algebra is the default)
        if "GEOMETRY" in category:
            specialist = GEOMETRY_PROMPT
        elif "COMBINATORICS" in category:
            specialist = COMBINATORICS_PROMPT
        elif "NUMBER" in category:
            specialist = NUMBER_THEORY_PROMPT
        else:
            specialist = ALGEBRA_PROMPT

        solution = await self.custom(input=problem, instruction=specialist)
        return solution, self.llm.usage_tracker.get_summary()["total_cost"]
```

Assigned Query Example

What is the volume of a cube whose surface area is twice that of a cube with volume 1?

- **AMC W5**

Description

ScoreFlow query-level workflow generator. Unlike fixed-graph workflows that apply one static computation graph to every query, ScoreFlow synthesizes a custom workflow on-the-fly for each individual query at inference time. An optimizer LLM inspects the query and produces a tailored graph before execution. The exposed cost includes both the generated workflow's runtime cost and ScoreFlow's own generation overhead, making it more expensive than fixed-graph candidates. Strong overall accuracy, especially on queries where fixed routing heuristics fail; provides genuine per-query adaptability that cannot be replicated by any single fixed graph.

Python Code

```
class Workflow:
    def __init__(
        self,
        config,
        problem
    ) -> None:
        self.problem = problem
        self.config = create(config)
        self.gid = None
        self.custom = operator.Custom(self.config, self.gid, self.problem)
        self.sc_ensemble = operator.ScEnsemble(self.config, self.gid, self.problem)
        self.programmer = operator.Programmer(self.config, self.gid, self.problem)
        self.review = operator.Review(self.config, self.gid, self.problem)

    async def run_workflow(self):
        """
        This is a workflow graph.
        """
        step1 = await self.custom(instruction="Can you break down the problem into smaller steps and
              explain the reasoning behind each step?")
        step2 = await self.custom(instruction="Can you solve the problem by providing a detailed
              solution?")
        solution = await self.sc_ensemble(solutions=[step1, step2])
        final_solution = await self.programmer(analysis=solution)
        reviewed_solution = await self.review(pre_solution=final_solution)
        return reviewed_solution
```

Assigned Query Example

A singles tournament had six players. Each player played every other player only once, with no ties. If Helen won 4 games, Ines won 3 games, Janet won 2 games, Kendra won 2 games and Lara won 2 games, how many games did Monica win?

E.2 MBPP

- **MBPP W1**

Description

ScoreFlow query-level workflow generator. Unlike fixed-graph workflows that apply one static computation graph to every query, ScoreFlow synthesizes a custom workflow on-the-fly for each individual query at inference time. An optimizer LLM inspects the query and produces a tailored graph before execution. The exposed cost includes both the

generated workflow's runtime cost and ScoreFlow's own generation overhead, making it more expensive than fixed-graph candidates. Strong overall accuracy, especially on queries where fixed routing heuristics fail; provides genuine per-query adaptability that cannot be replicated by any single fixed graph.

Python Code

```
class Workflow:
    def __init__(
        self,
        config,
        problem
    ) -> None:
        self.problem = problem
        self.config = create(config)
        self.code_generate = operator.CustomCodeGenerate(self.config, self.problem)
        self.sc_ensemble = operator.ScEnsemble(self.config, self.problem)
        self.test = operator.Test(self.config, self.problem)

    async def run_workflow(self):
        """
        This is a workflow graph.
        """
        solution_1 = await self.code_generate(instruction="Can you analyze this problem step by step
             and generate a function to find the nth decagonal number?")
        solution_2 = await self.code_generate(instruction="Can you think of another way to generate a
              function to find the nth decagonal number?")
        solution_3 = await self.code_generate(instruction="Can you provide a different approach to
             generate a function to find the nth decagonal number?")

        ensembled_solution = await self.sc_ensemble(solutions=[solution_1, solution_2, solution_3])

        tested_solution = await self.test(solution=ensembled_solution)

        return tested_solution
```

Assigned Query Example

Write a function to find the nth decagonal number.
def is_num_decagonal(n):

- **MBPP W2**

Description

Clarify-Solve-Iterate pipeline without ensembling. First a Custom operator rewrites the problem through a clarify prompt to pin down any ambiguous requirements. Then a single CustomCodeGenerate call under a step-by-step reasoning prompt produces one candidate function implementation. The Test operator runs the problem's unit checks; if they pass, return immediately. On failure the workflow enters a bounded refine loop: up to three Custom refine passes each consume the last failed solution plus the error trace and emit a corrected version, with Test run after each iteration. No majority voting, no parallel candidates — commits to one reasoning path and iterates on errors via focused refinement. Up to 5 LLM calls in the worst case, typically 2 on problems the first-pass solution already solves. Architecturally a direct contrast to W1's per-query-generated workflow: this is a fixed graph that's been tuned to one particular problem-solving strategy.

Python Code

```
CLARIFY_PROMPT = "Refine the problem understanding and clarify the requirements."

SOLVE_PROMPT = "Solve the problem step by step and provide a clear solution."
```

```
REFINE_PROMPT = "Refine the solution to fix any errors or improve clarity."


class Workflow:
    def __init__(self, name, llm_config, dataset):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.custom_code_generate = operator.CustomCodeGenerate(self.llm)
        self.test = operator.Test(self.llm, dataset=self.dataset)

    async def __call__(self, problem: str, entry_point: str):
        # Clarify the problem, then generate an initial solution
        refined_problem = await self.custom(input=problem, instruction=CLARIFY_PROMPT)
        solution = await self.custom_code_generate(
            problem=refined_problem, entry_point=entry_point, instruction=SOLVE_PROMPT
        )
        test_result = await self.test(problem=problem, solution=solution, entry_point=entry_point)

        # Up to 3 refinement passes; stop early as soon as tests pass
        for _ in range(3):
            if test_result['result']:
                break
            refined_solution = await self.custom(
                input=test_result['solution'], instruction=REFINE_PROMPT
            )
            test_result = await self.test(
                problem=problem, solution=refined_solution, entry_point=entry_point
            )

        return test_result['solution'], self.llm.usage_tracker.get_summary()["total_cost"]
```

Assigned Query Example

Write a function to extract all the adjacent coordinates of the given coordinate tuple.
def get_coordinates(test_tup):

E.3 DROP

- **DROP W1**

Description

Conditional routing workflow based on question-type classification. First classifies the question into NUMERIC, COMPARE, DATE, SPAN, or other. For NUMERIC/COMPARE questions, routes to QANumerical — a DROP-tailored Programmer variant that generates Python code to extract numbers and compute the answer in a sandboxed subprocess. For DATE questions, uses a Custom operator with a date-extraction prompt. For SPAN questions, uses a Custom operator with a span-extraction prompt. All other questions fall back to AnswerGenerate. This selective routing applies QANumerical's strength (code-based computation) only where it fits, avoiding it on text-span or date questions where it would overfit to numeric patterns. Low cost (2 LLM calls: classify + solve).

Python Code

```
CLASSIFY_PROMPT = """Classify the question into ONE of these categories. Output ONLY the category
      name (a single word).

Categories:
- NUMERIC --- asks for a count, sum, difference, percentage, year, or any calculation.
- SPAN --- asks "who", "what", "which" to extract a text span (name, place, entity) from the passage.

- DATE --- asks for a specific date.
```

```
- COMPARE --- asks to compare two things (e.g., "which is greater", "longer", "earlier").

Output format: just one word, e.g. NUMERIC
"""

SPAN_PROMPT = """Read the passage and find the exact text span that answers the question.
Output ONLY the text span, no explanation, no formatting.
"""

DATE_PROMPT = """Read the passage and find the exact date that answers the question.
Output ONLY the date in the format used by the passage, no explanation.
"""


class Workflow:
    """Classify question type, then route to the best-suited operator."""
    def __init__(self, name, llm_config, dataset):
        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.qa_numerical = operator.QANumerical(self.llm)

    async def __call__(self, problem: str):
        cls = (await self.custom(input=problem, instruction=CLASSIFY_PROMPT)).strip().upper()

        if "NUMERIC" in cls or "COMPARE" in cls:
            solution = await self.qa_numerical(problem=problem, analysis="None")
        elif "DATE" in cls:
            solution = await self.custom(input=problem, instruction=DATE_PROMPT)
        elif "SPAN" in cls:
            solution = await self.custom(input=problem, instruction=SPAN_PROMPT)
        else:
            solution = await self.answer_generate(input=problem)

        return solution, self.llm.usage_tracker.get_summary()["total_cost"]
```

Assigned Query Example

Passage: The median age in the city was 22.1 years. 10.1% of residents were under the age of 18; 56.2% were between the ages of 18 and 24; 16.1% were from 25 to 44; 10.5% were from 45 to 64; and 7% were 65 years of age or older. The gender makeup of the city was 64.3% male and 35.7% female. Question: How many in percent weren't between the ages of 18 and 24? Answer:

- **DROP W2**

Description

ScoreFlow query-level workflow generator. Unlike fixed-graph workflows that apply one static computation graph to every query, ScoreFlow synthesizes a custom workflow on-the-fly for each individual query at inference time. An optimizer LLM inspects the query and produces a tailored graph before execution. The exposed cost includes both the generated workflow's runtime cost and ScoreFlow's own generation overhead, making it more expensive than fixed-graph candidates. Strong overall accuracy, especially on queries where fixed routing heuristics fail; provides genuine per-query adaptability that cannot be replicated by any single fixed graph.

Python Code

```
class Workflow:
    def __init__(
        self,
        config,
        problem
```

```
    ) -> None:
        self.problem = problem
        self.agent = create(config)
        self.custom = operator.Custom(self.agent, self.problem)
        self.answer_generate = operator.AnswerGenerate(self.agent, self.problem)
        self.sc_ensemble = operator.ScEnsemble(self.agent, self.problem)
        self.review = operator.Review(self.agent, self.problem)

    async def run_workflow(self):
        """
        This is a workflow graph.
        """
        initial_solution = await self.answer_generate()
        reviewed_solution = await self.review(pre_solution=initial_solution)
        ensembled_solution = await self.sc_ensemble(solutions=[initial_solution, reviewed_solution])
        final_solution = await self.review(pre_solution=ensembled_solution)

        return final_solution
```

Assigned Query Example

Passage: The median age in the city was 22.1 years. 10.1% of residents were under the age of 18; 56.2% were between the ages of 18 and 24; 16.1% were from 25 to 44; 10.5% were from 45 to 64; and 7% were 65 years of age or older. The gender makeup of the city was 64.3% male and 35.7% female. Question: Which age group had the second most people? Answer:

F Supplementary Experiment Results

F.1 Distribution of Workflow Assignment on Each Dataset

As shown in Figure 7, FlowBank successfully assigns different queries to different members in the curated portfolio over five benchmarks, demonstrating FlowBank's query adaptivity.

F.2 Coverage-k Curves in CuraFlow on five benchmarks

Figure 8 summarizes the Coverage-k curves obtained by CuraFlow across all five benchmarks. We observe a consistent pattern: coverage improves rapidly when a small number of workflows are added, and then gradually saturates as k increases. This indicates that much of the attainable set-level utility can be captured by a compact portfolio, which supports our design choice of curating a small workflow bank instead of keeping the entire candidate pool. We also observe noticeable variation across datasets: some benchmarks saturate quickly, while others continue to benefit from adding more workflows, suggesting that the degree of workflow complementarity is task dependent.

F.3 Comparison of Coverage-k Curves for AFlow, DiverseFlow and CuraFlow

To better understand why DiverseFlow and CuraFlow are complementary, we conduct a controlled analysis on MATH that directly echoes the "Why DiverseFlow and CuraFlow are Complementary" discussion in Section 3.2. We run AFlow and DiverseFlow for the same 20 optimization rounds, while constraining the first seven rounds of DiverseFlow to produce exactly the same workflows as AFlow. We then compare their online cumulative set coverage during optimization (the blue and orange curves in Figure 9). After DiverseFlow finishes all 20 rounds, we further apply CuraFlow to the resulting workflow pool generated from DiverseFlow and perform post-hoc max-coverage search; this curve is shown as the red line in Figure 9.

Figure 9 shows two key findings. First, after the shared first seven rounds, DiverseFlow consistently achieves higher online coverage than AFlow, indicating that diversity-aware exploration is indeed more effective at discovering complementary workflow sets. Second, although online exploration increases set size, it cannot anticipate which later workflows will best complement earlier ones, so stopping early still yields suboptimal sets. CuraFlow addresses this limitation by searching for the max-coverage subset over DiverseFlow's fully explored candidate pool, allowing a much smaller set

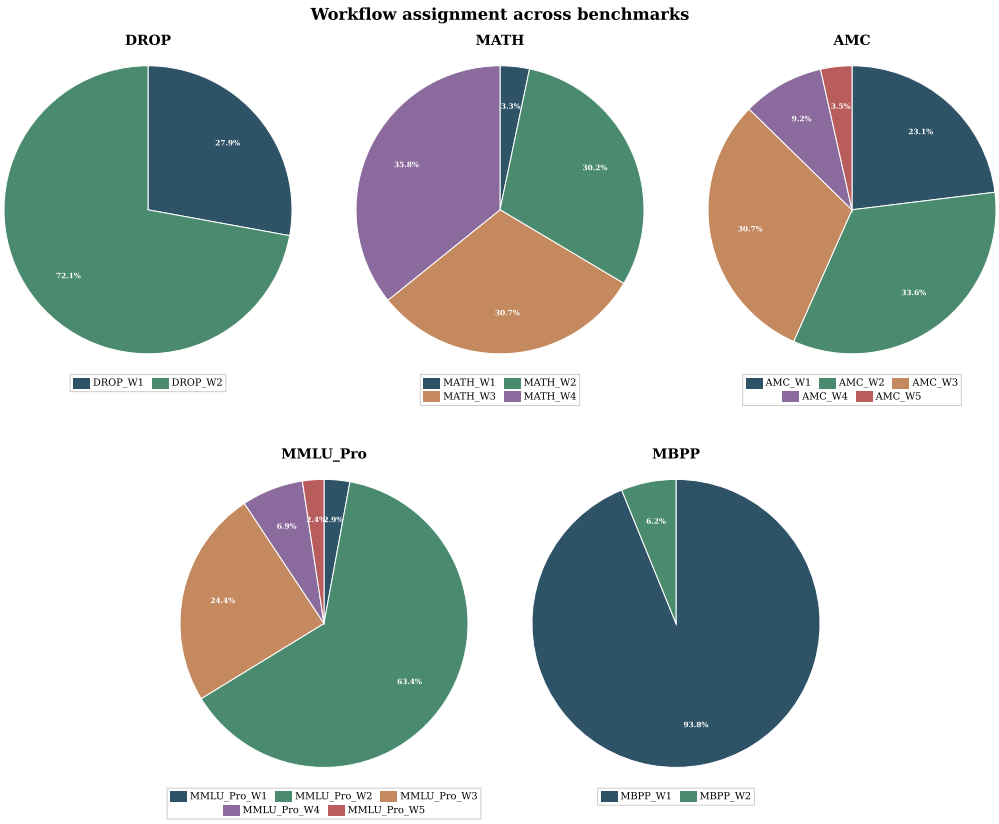

Figure 7: Workflow assignment distribution of `FlowBank` across datasets.

to match or even exceed the coverage of the online cumulative set at the same size k. This explains why the two stages are not contradictory: DiverseFlow discovers complementary candidates, and CuraFlow distills them into a compact high-coverage portfolio.

F.4 Additional Offline Cost Details

We provide the component-level breakdown and accounting protocol underlying the offline cost audit in Table 4. All audited costs in this subsection are measured on MATH using uniform token and GPU prices across methods.

Online and Offline Accounting. Offline costs cover workflow execution during search and training-set evaluation, optimizer calls, model training, workflow description generation, embedding computation, portfolio curation, and selector training, as applicable to each method. Online costs are incurred when processing test queries and are reported separately in Table 1. For static workflows, online cost consists of workflow execution. ScoreFlow additionally incurs generator inference cost for query-specific workflow synthesis. `FlowBank` includes query embedding and execution of the selected portfolio member, including workflow synthesis when ScoreFlow is selected. Its GNN-based selection requires no additional LLM calls and has negligible computational overhead relative to workflow execution.

Baseline Cost Breakdown. Table 7 decomposes the baselines' offline costs into workflow execution, optimizer calls, and GPU consumption. AFlow and ADAS incur token costs for execution and optimization, while ScoreFlow and MaAS additionally require GPU resources for generator fine-tuning and supernet training, respectively. Together with the `FlowBank` breakdown in Table 4, these components account for the reported offline totals.

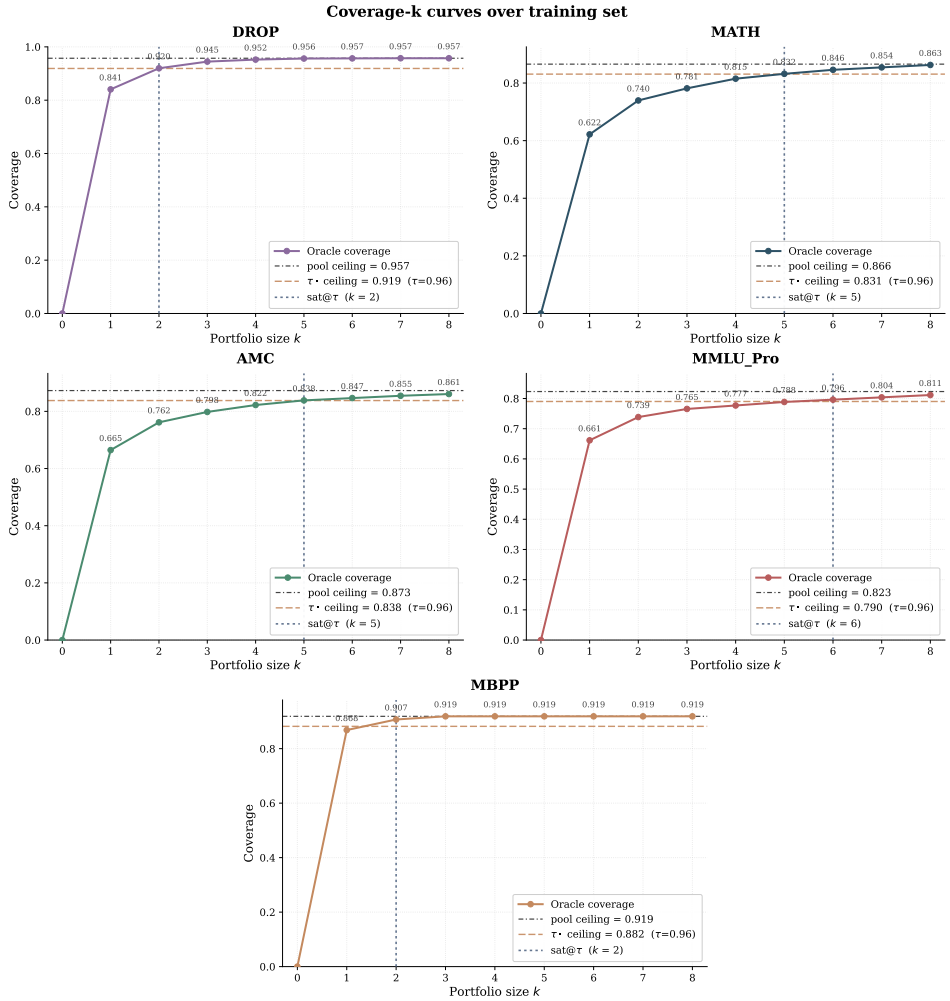

Figure 8: Coverage-k curves of CuraFlow across five benchmarks.

Table 7: Component-level offline costs of the baselines on MATH, in USD. A dash indicates that the corresponding cost component is not incurred. Total offline costs are reported in Table 4.

Method	Workflow Execution	Optimizer Calls	GPU Consumption
AFlow (Qwen3-8B)	8.94	0.02	–
AFlow (GPT-4o)	12.48	0.93	–
ADAS	5.68	5.88	–
ScoreFlow	23.07	0.20	4.67
MaAS	1.14	–	7.25

Unit Prices and GPU Billing. Token costs are computed from input and output token usage using the unit prices in Table 8. GPU costs in the audit are computed at \$0.79 per A6000 GPU-hour, multiplied by the training wall-clock time and the number of allocated GPUs. ScoreFlow's generator inference is charged by token usage both before and after fine-tuning; its GPU charge covers generator training. Thus, the \$4.67 GPU component is only part of ScoreFlow's \$27.94 offline pipeline cost. Likewise, `FlowBank`'s inherited ScoreFlow cost includes the complete offline pipeline, rather than only generator fine-tuning.

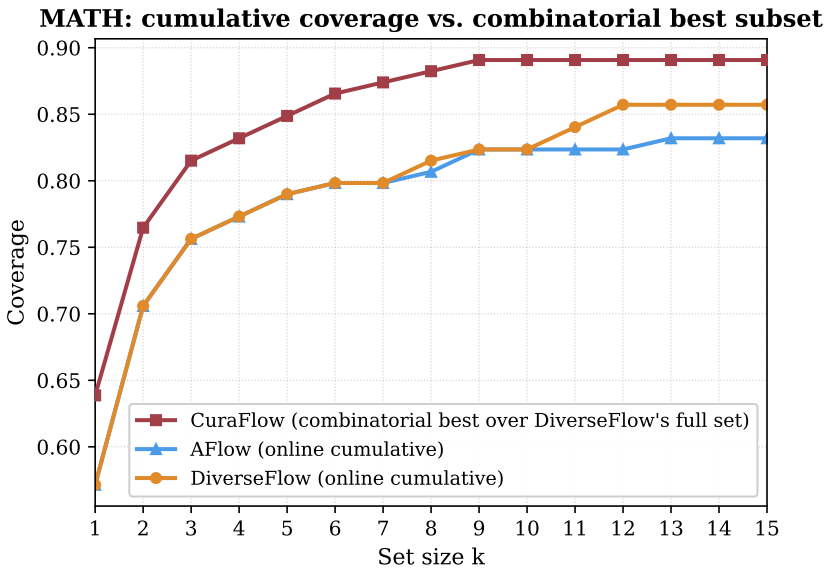

Figure 9: Controlled comparison of Coverage-k curves for AFlow and DiverseFlow (online cumulative), and CuraFlow (combinatorial best over DiverseFlow's full candidate set) on MATH.

Table 8: Token unit prices used in the cost audit, in USD per million tokens.

Model	Input	Output
GPT-4o mini	0.15	0.60
GPT-4o	2.50	10.00
Qwen3-8B	0.05	0.40
`text-embedding-3-small`	0.02	–

F.5 Run-to-Run Variability and Averaging Protocol

Table 9 reports the mean and standard deviation of performance and online cost over three runs for eight methods on MATH, with the executor temperature fixed to zero. Across benchmarks, the Average columns in Tables 1, 3 and 9 weight both performance and online cost by test-set size.

G Additional Discussion

G.1 Limitations

One main limitation of `FlowBank` is that the curated portfolio passed to Stage 3 is fixed once constructed. While this design improves deployment efficiency and simplifies inference-time routing, it also means that workflows discovered after curation are not automatically incorporated into the bank. Extending `FlowBank` into a continually expanding flow bank that grows as new workflows become available is a natural direction we leave to future work. In addition, the saturation ratio τ and the cost trade-off λ are lightweight hyperparameters that may benefit from per-benchmark tuning. Although Section 4 shows that `FlowBank` is robust within a moderate range, fully automating their selection remains an open question.

G.2 Broader Impact

`FlowBank` can have several positive practical impacts. By reusing a compact portfolio of high-utility workflows instead of generating or executing fully dynamic multi-agent systems for every query, it can substantially reduce token consumption and thus lower monetary and environmental costs in large-scale deployments of multi-agent systems. In addition, the framework is naturally expandable:

Table 9: Performance and online-cost variability on MATH. Values denote means and standard deviations over three runs. Cost denotes average online cost in $\times 10^{-3}$ USD per query.

Method	Performance	Online Cost
IO	56.58 ± 0.63	0.51 ± 0.01
CoT	55.76 ± 0.74	0.53 ± 0.02
Multi-agent Debate	58.92 ± 0.76	7.66 ± 0.05
MultiPersona	57.41 ± 0.32	0.56 ± 0.01
AFlow (GPT-4o)	63.99 ± 0.74	2.77 ± 0.03
ScoreFlow	62.00 ± 1.20	2.36 ± 0.04
MaAS	65.02 ± 0.98	2.43 ± 0.05
`FlowBank`	69.34 ± 0.80	1.78 ± 0.03

the workflow bank can continually grow by incorporating workflows discovered from different optimization methods, datasets, domains, or external workflow sources, which may make it a flexible infrastructure layer for future agentic systems. At the same time, `FlowBank` may also introduce negative impacts. If the bank is populated with workflows collected from heterogeneous sources, it may inherit hidden biases, unsafe reasoning patterns, or licensing and attribution issues from those sources, and these problems may be amplified when such workflows are reused at scale.

G.3 Future Work

More broadly, our results suggest that workflows should be treated less as one-off artifacts and more as reusable memory. A natural next step is to move from a fixed portfolio to a continually expanding flow bank that can recognize when existing workflows are insufficient, incorporate newly discovered workflows, and improve routing under distribution shift. We hope this perspective helps shift the field from searching for one universally best workflow toward building reusable workflow libraries that grow in coverage, adaptability, and efficiency over time. One promising direction is to connect FlowBank with recent memory-based agent frameworks such as A-Mem [39], ArcMemo [14], MemoryBank [52], and ReasoningBank [27], so that it stores not only reusable workflows but also structured experience about when they succeed, fail, and should be adapted.

H Additional Related Works

Static Agentic Workflow Optimization. Static agentic workflow optimization focuses on automatically improving agentic workflows and multi-agent architectures to reduce reliance on manually designed prompting and coordination strategies. These approaches typically formulate agentic systems as structured computational graphs and optimize their prompts, topologies, communication patterns, or execution procedures through search, evolutionary algorithms, architecture optimization, or preference-based learning. Early studies such as GPTSwarm [56] formulate language-agent systems as optimizable graphs and improve multi-agent collaboration through graph-based optimization. ADAS [16] further investigates automated agentic system design by performing program-level search over workflow structures and operators. Building upon these ideas, AFlow [47] introduces an MCTS-based framework for automated workflow generation and demonstrates strong reasoning performance across multiple benchmarks. Mass [53] studies the joint optimization of prompts and agent topologies, emphasizing the importance of structural coordination in multi-agent systems. Several subsequent works further improve workflow optimization efficiency and diversity. EvoFlow [45] adopts evolutionary strategies to iteratively evolve diverse workflows during optimization, while AdaptFlow [55] introduces meta-learning techniques to improve workflow adaptation across tasks and environments. Preference-based optimization methods have also attracted growing attention. SPOGW [3] proposes a score-based preference optimization framework using group-wise workflow comparisons to improve workflow quality. In addition, FLORA [49] studies surrogate performance modeling by using graph neural networks to predict workflow effectiveness, enabling more efficient workflow evaluation and search. Despite their effectiveness, most existing methods ultimately optimize toward a single final workflow or architecture for deployment. In contrast, our work focuses on constructing a compact portfolio of complementary workflows and adaptively selecting among

them at inference time, enabling a better trade-off between deployment efficiency and query-level adaptivity.

Query-Adaptive Workflow Generation. Query-adaptive workflow generation focuses on improving adaptability at inference time by dynamically generating or orchestrating workflows for queries that require different reasoning strategies, coordination structures, or agent interactions. Instead of deploying a single static workflow for all inputs, these methods dynamically generate, select, or route workflows at inference time to improve task-specific adaptability and reasoning performance. MAS-GPT [42] trains large language models to directly generate multi-agent systems conditioned on the input query, enabling query-specific workflow construction through meta-generation. MaAS [46] formulates workflow discovery as an architecture search problem and leverages a supernet-based strategy to efficiently explore large workflow design spaces. ScoreFlow [32] further introduces score-based preference optimization for training workflow generators, allowing the system to synthesize customized workflows tailored to individual queries. FlowSteer [48] formulates interactive workflow orchestration as an end-to-end reinforcement learning problem, optimizing adaptive workflow control policies through interaction feedback. While these methods improve query-level flexibility, they often incur substantial online computation cost due to per-query workflow generation or orchestration. In contrast, our work seeks to recover much of the benefit of query adaptivity through a precompute-and-reuse paradigm, where a compact portfolio of complementary workflows is optimized offline and adaptively selected at inference time.

Analysis of Multi-Agent Systems. Analysis of multi-agent systems focuses on examining the foundations, efficiency, and necessity of complex orchestration strategies, asking when multi-agent systems and query-adaptive workflows are truly beneficial. These works investigate issues such as uncertainty propagation, coordination inefficiency, redundant workflow complexity, and the trade-offs between single-agent, static-workflow, and dynamically generated multi-agent systems, providing important insights into the practical limitations of agentic workflows. Zhao et al. [50] study uncertainty propagation and reliability in LLM-based multi-agent systems, showing how interactions among agents can amplify instability and affect downstream reasoning quality. Li [18] investigates the conditions under which a strong single agent augmented with specialized skills can match or outperform multi-agent systems, revealing that additional coordination is not always necessary. Similarly, Xu et al. [38] revisit the empirical value of multi-agent workflows and demonstrate that carefully designed single-agent baselines can remain highly competitive across many tasks. Several works further analyze the efficiency and scalability of complex orchestration strategies. El et al. [8] study the computational inefficiencies introduced by meta-agent-based workflow design and question whether increasingly sophisticated orchestration mechanisms justify their additional overhead. Wang et al. [35] systematically analyze the necessity of query-level workflow generation and show that static workflows can remain competitive in many scenarios despite the growing popularity of dynamic workflow synthesis. Luo et al. [20] further explore whether the collaborative intelligence emerging from multi-agent systems can be distilled into a single LLM agent, studying the relationship between distributed coordination and compact agent representations. Collectively, these works suggest that increasing workflow complexity or online adaptivity does not necessarily translate into better efficiency or scalability. In contrast, our work seeks to balance adaptivity and efficiency through reusable workflow portfolios, enabling query-aware behavior without incurring the substantial overhead of fully dynamic workflow generation.

I Prompts for DiverseFlow

Performance-oriented Warm-up

```
You are building an LLM Graph (Python workflow) and corresponding Prompt to jointly solve {type}
      problems.
{dataset_description}
Referring to the given graph and prompt below, 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.

When optimizing, you can incorporate critical thinking methods like review, revise, ensemble (generating
       multiple answers through different/similar prompts, then voting/integrating/checking the majority
```

```
      to obtain a final answer), selfAsk, etc. Consider Python's loops (for) and conditional statements (
      if-elif-else).
Considering information loss, complex graphs may yield better results, but insufficient information
      transmission can omit the solution. It's crucial to include necessary context during the process.
Introducing multiple operators at appropriate points can enhance performance. If some provided operators
       are not yet used, try incorporating them.

## Operator Initialization and Usage (MUST follow exactly)

All operators MUST be initialized with `self.llm` in `__init__`. Example:
```
        self.custom = operator.Custom(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)
        self.answer_generate = operator.AnswerGenerate(self.llm)
        self.custom_code_generate = operator.CustomCodeGenerate(self.llm)
        self.test = operator.Test(self.llm, dataset=self.dataset)
        self.programmer = operator.Programmer(self.llm)
```

Operator call formats (MUST follow exactly):

1. **Custom**
   Format: `custom(input: str, instruction: str) -> str`
   Example: `solution = await self.custom(input=problem, instruction=prompt_custom.SOLVE_PROMPT)`
   Note: Returns the response string directly. The input and instruction are directly concatenated (
         instruction+input). Placeholders are not supported. You MUST define the prompt variable (e.g., `
         SOLVE_PROMPT = "..."`) in the `<prompt>` section for every `prompt_custom.XXXX` you use in graph.


2. **ScEnsemble**
   Format: `sc_ensemble(solutions: List[str], problem: str) -> str`
   Example: `best = await self.sc_ensemble(solutions=[sol1, sol2, sol3], problem=problem)`
   Note: Returns the best solution string directly. Selects the best from multiple candidates via voting.

3. **AnswerGenerate**
   Format: `answer_generate(input: str) -> str`
   Example: `answer = await self.answer_generate(input=problem)`
   Note: Returns the answer string directly. Generates step-by-step reasoning internally.

4. **CustomCodeGenerate**
   Format: `custom_code_generate(problem: str, entry_point: str, instruction: str) -> str`
   Example: `code = await self.custom_code_generate(problem=problem, entry_point=entry_point, instruction
         ="Analyze step by step and generate code.")`
   Note: Returns the code string directly. The instruction should encourage step-by-step thinking.

5. **Test**
   Format: `test(problem: str, solution: str, entry_point: str) -> dict with keys 'result' (bool) and '
         solution' (str)`
   Example: `test_result = await self.test(problem=problem, solution=solution, entry_point=entry_point)`
   Note: Modify the input solution solution with public test cases. Always return test_result['solution']
          (the improved solution). Use test_result['result'] (bool) for conditional retry if needed.

6. **Programmer**
   Format: `programmer(problem: str, analysis: str = 'None') -> str`
   Example: `result = await self.programmer(problem=problem, analysis="Step by step analysis")`
   Note: Writes and executes Python code, returns a string with the execution result.

Here is a graph and the corresponding prompt (prompt only related to the custom method) that performed
      excellently in a previous iteration (maximum score is 1). You must make further optimizations and
      improvements based on this graph.

<sample>
    <experience>{experience}</experience>
    <modification>(such as: add / delete / modify / ...)</modification>
    <score>{score}</score>
    <graph>{graph}</graph>
    <prompt>{prompt}</prompt>(only prompt_custom)
</sample>

Below are the logs of some results with the aforementioned Graph that performed well but encountered
      errors, which can be used as references for optimization:
{log}

Now, provide your optimization. The modified graph must differ from the provided example, and the
      specific differences should be noted within the <modification>xxx</modification> section.

**OPERATOR RULES (MUST follow strictly):**
1. Do NOT create new operators. Only use the operators listed in the Operator Usage section above.
```

```
2. All operators MUST be initialized in `__init__` with `self.llm` as the first argument. Example: `self.
      custom = operator.Custom(self.llm)`. Never call `operator.XXX()` without `self.llm`.
3. Follow the exact call format for each operator as specified in the Operator Usage section.
4. Loop iteration MUST <= 5 to avoid timeout.
5. The graph complexity should not exceed 8 nodes.

**`__call__` SIGNATURE AND RETURN FORMAT (MUST follow exactly):**
The `__call__` method signature MUST be:
```python
async def __call__(self, problem: str):
    # ... your workflow logic ...
    return solution, self.llm.usage_tracker.get_summary()["total_cost"]
```
- Input: `problem` (str) -- the math problem text.
- Return: a **2-tuple** of `(solution_string, cost_float)`. Do NOT return only the solution without cost.
       Do NOT return a 3-tuple or any other format.

PROMPT GENERATION RULES (CRITICAL):
- Only generate prompts used by Custom operator via `prompt_custom.YOUR_PROMPT_NAME` in the graph.
- You MUST define every `prompt_custom.XXXX` variable you reference in graph in the `<prompt>` section.
      For example, if you use `prompt_custom.SOLVE_PROMPT` in graph, you must output `SOLVE_PROMPT =
      "..."` in `<prompt>`.
- Other operators (ScEnsemble, Programmer, AnswerGenerate, CustomCodeGenerate, Test) have built-in
      prompts -- do NOT generate prompts for them.
- **The generated prompt MUST be plain text. Do NOT use placeholders like {{problem}}, {{entry_point}},
      or {{input}}. The prompt will be directly concatenated with the input, NOT formatted with .format()
      .**
- Remove any unused prompts from prompt_custom.

MODIFICATION RULES:
- **Only ONE detail point can be modified at a time**, and no more than 5 lines of code (including graph
       and prompt) may be changed per modification -- extensive modifications are strictly prohibited to
      maintain project focus!
- When introducing new functionalities in the graph, please make sure to import the necessary libraries
      or modules yourself, except for operator, prompt_custom, create_llm_instance, and CostManage, which
       have already been automatically imported.
- **Under no circumstances should Graph output None for any field.**
- Use custom methods to restrict your output format, rather than using code (outside of the code, the
      system will extract answers based on certain rules and score them).
- It is very important to format the Graph output answers, you can refer to the standard answer format
      in the log.
```

Complementarity-oriented Expansion

```
You are building an LLM Graph (Python workflow) and corresponding Prompt to jointly solve {type}
      problems.
{dataset_description}
Referring to the given graph and prompt below, 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.

When optimizing, you can incorporate critical thinking methods like review, revise, ensemble (generating
       multiple answers through different/similar prompts, then voting/integrating/checking the majority
      to obtain a final answer), selfAsk, etc. Consider Python's loops (for) and conditional statements (
      if-elif-else).
Considering information loss, complex graphs may yield better results, but insufficient information
      transmission can omit the solution. It's crucial to include necessary context during the process.
Introducing multiple operators at appropriate points can enhance performance. If some provided operators
       are not yet used, try incorporating them.

## Operator Initialization and Usage (MUST follow exactly)

All operators MUST be initialized with `self.llm` in `__init__`. Example:
```
       self.custom = operator.Custom(self.llm)
       self.sc_ensemble = operator.ScEnsemble(self.llm)
       self.answer_generate = operator.AnswerGenerate(self.llm)
       self.custom_code_generate = operator.CustomCodeGenerate(self.llm)
       self.test = operator.Test(self.llm, dataset=self.dataset)
       self.programmer = operator.Programmer(self.llm)
```

Operator call formats (MUST follow exactly):

1. **Custom**
```

```
   Format: `custom(input: str, instruction: str) -> str`
   Example: `solution = await self.custom(input=problem, instruction=prompt_custom.SOLVE_PROMPT)`
   Note: Returns the response string directly. The input and instruction are directly concatenated (
         instruction+input). Placeholders are not supported. You MUST define the prompt variable (e.g., `
         SOLVE_PROMPT = "..."`) in the `<prompt>` section for every `prompt_custom.XXXX` you use in graph.


2. **ScEnsemble**
   Format: `sc_ensemble(solutions: List[str], problem: str) -> str`
   Example: `best = await self.sc_ensemble(solutions=[sol1, sol2, sol3], problem=problem)`
   Note: Returns the best solution string directly. Selects the best from multiple candidates via voting.

3. **AnswerGenerate**
   Format: `answer_generate(input: str) -> str`
   Example: `answer = await self.answer_generate(input=problem)`
   Note: Returns the answer string directly. Generates step-by-step reasoning internally.

4. **CustomCodeGenerate**
   Format: `custom_code_generate(problem: str, entry_point: str, instruction: str) -> str`
   Example: `code = await self.custom_code_generate(problem=problem, entry_point=entry_point, instruction
         ="Analyze step by step and generate code.")`
   Note: Returns the code string directly. The instruction should encourage step-by-step thinking.

5. **Test**
   Format: `test(problem: str, solution: str, entry_point: str) -> dict with keys 'result' (bool) and '
         solution' (str)`
   Example: `test_result = await self.test(problem=problem, solution=solution, entry_point=entry_point)`
   Note: Modify the input solution solution with public test cases. Always return test_result['solution']
          (the improved solution). Use test_result['result'] (bool) for conditional retry if needed.

6. **Programmer**
   Format: `programmer(problem: str, analysis: str = 'None') -> str`
   Example: `result = await self.programmer(problem=problem, analysis="Step by step analysis")`
   Note: Writes and executes Python code, returns a string with the execution result.

Here is a graph and the corresponding prompt (prompt only related to the custom method) that performed
      excellently in a previous iteration (maximum score is 1). You must make further optimizations and
      improvements based on this graph.

<sample>
    <experience>{experience}</experience>
    <modification>(such as: add / delete / modify / ...)</modification>
    <score>{score}</score>
    <graph>{graph}</graph>
    <prompt>{prompt}</prompt>(only prompt_custom)
</sample>

Below are the logs of some results with the aforementioned Graph that performed well but encountered
      errors, which can be used as references for optimization:
{log}

**SCORING**: Your workflow is evaluated by WEIGHTED accuracy --- solving under-covered queries counts
      much more than solving well-covered ones. The score shown above reflects this weighted metric, not
      raw accuracy.

**Query weight distribution**:
- {n_weight_high} queries covered by 0 workflows (weight=1.00, highest impact on your score)
- {n_weight_mid} queries covered by 1-3 workflows (weight=0.25-0.50)
- {n_weight_low} queries covered by 4+ workflows (weight<0.20, low impact)

**High-weight examples** (solving these types has the most impact on your score):
{high_weight_examples}

**Existing strengths** (avoid duplicating):
{existing_strengths}

**The workflow is pure Python --- use the full language freely.** Go beyond simple operator chains.
      Consider these design patterns:
- **Majority Vote**: Loop N calls + `collections.Counter` on extracted answers (more robust than
      ScEnsemble)
- **Dynamic Prompts**: Use `.format()` to inject prior outputs into prompts (define templates with `{
      placeholders}` in `<prompt>`)
- **Multi-Agent Debate**: Proposer -> Critic -> Judge, each seeing prior output
- **Conditional Routing**: Classify problem type -> if/else -> specialist prompt
- **Helper Methods**: `@staticmethod` for regex answer extraction, post-processing
- **Multi-Paradigm**: Combine forward reasoning + backward reasoning + code execution, then ensemble
```

```
You can import ‘re‘, ‘collections‘, ‘json‘, ‘math‘, etc. Think structurally different, not just prompt
      tweaks.

**Relaxed rules for diversity mode** (override the rules below where they conflict):
- Prompts MAY use ‘{placeholders}‘ filled via ‘.format()‘ in graph code (for dynamic prompt composition)
- Up to **10 lines** of code may be changed (structural changes need more room)
- You MAY define ‘@staticmethod‘ or helper methods on the Workflow class

Now, provide your optimization. The modified graph must differ from the provided example, and the
      specific differences should be noted within the <modification>xxx</modification> section.

**OPERATOR RULES (MUST follow strictly):**
1. Do NOT create new operators. Only use the operators listed in the Operator Usage section above.
2. All operators MUST be initialized in ‘__init__‘ with ‘self.llm‘ as the first argument. Example: ‘self.
      custom = operator.Custom(self.llm)‘. Never call ‘operator.XXX()‘ without ‘self.llm‘.
3. Follow the exact call format for each operator as specified in the Operator Usage section.
4. Loop iteration MUST <= 5 to avoid timeout.
5. The graph complexity should not exceed 8 nodes.

**‘__call__‘ SIGNATURE AND RETURN FORMAT (MUST follow exactly):**
The ‘__call__‘ method signature MUST be:
‘‘‘python
async def __call__(self, problem: str):
    # ... your workflow logic ...
    return solution, self.llm.usage_tracker.get_summary()["total_cost"]
‘‘‘
- Input: ‘problem‘ (str) --- the math problem text.
- Return: a **2-tuple** of ‘(solution_string, cost_float)‘. Do NOT return only the solution without cost.
       Do NOT return a 3-tuple or any other format.

**PROMPT GENERATION RULES (CRITICAL):**
- Only generate prompts used by Custom operator via ‘prompt_custom.YOUR_PROMPT_NAME‘ in the graph.
- You MUST define every ‘prompt_custom.XXXX‘ variable you reference in graph in the ‘<prompt>‘ section.
      For example, if you use ‘prompt_custom.SOLVE_PROMPT‘ in graph, you must output ‘SOLVE_PROMPT =
      "..."‘ in ‘<prompt>‘.
- Other operators (ScEnsemble, Programmer, AnswerGenerate, CustomCodeGenerate, Test) have built-in
      prompts --- do NOT generate prompts for them.
- **The generated prompt MUST be plain text. Do NOT use placeholders like {{problem}}, {{entry_point}},
      or {{input}}. The prompt will be directly concatenated with the input, NOT formatted with .format()
      .**
- Remove any unused prompts from prompt_custom.

**MODIFICATION RULES:**
- **Only ONE detail point can be modified at a time**, and no more than 5 lines of code (including graph
       and prompt) may be changed per modification --- extensive modifications are strictly prohibited to
       maintain project focus!
- When introducing new functionalities in the graph, please make sure to import the necessary libraries
      or modules yourself, except for operator, prompt_custom, create_llm_instance, and CostManage, which
       have already been automatically imported.
- **Under no circumstances should Graph output None for any field.**
- Use custom methods to restrict your output format, rather than using code (outside of the code, the
      system will extract answers based on certain rules and score them).
- It is very important to format the Graph output answers, you can refer to the standard answer format
      in the log.
```

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: The abstract and introduction (Section 1) clearly state our three main contributions: (1) We identify two under-exploited forms of workflow complementarity: discarded task-level workflows retain set-level reuse value, and part of the gain from query-level generation can be recovered through adaptive reuse of cheaper precomputed workflows, (2) We propose `FlowBank`, a three-stage framework that builds a reusable workflow bank through diversified search, coverage-aware curation, and graph-based query-adaptive matching, (3) We demonstrate that `FlowBank` achieves the strongest average performance and competitive performance–cost trade-offs across five benchmarks, validating portfolio-based workflow reuse as a stronger alternative to committing to a single workflow or a single paradigm, all of which are supported by experiments in Section 4.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: We provide a discussion of the limitations of this work at Appendix G.1.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

   Answer: [N/A]

   Justification: The paper does not include theoretical results.

   Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: We provide the experiment setup in Section 4.1 and Appendix C.

   Guidelines:

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

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

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

Answer: [Yes]

Justification: We release the source code implementing our framework as part of the supplementary material.

Guidelines:

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

6. **Experimental setting/details**

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

Answer: [Yes]

Justification: We provide the experiment setup in Section 4.1 and Appendix C.

Guidelines:

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

7. **Experiment statistical significance**

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

Answer: [Yes]

Justification: The experiment result shown in Section 4 are reported as the average of 3 random runs.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: We provide the specific type of computation resource, memory and time usage in Appendix C.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: The research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics

   Guidelines:

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

10. **Broader impacts**

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

    Answer: [Yes]

    Justification: We provide a discussion of both the positive and negative societal impacts of this work at Appendix G.2.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

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

    Answer: [N/A]

    Justification: The paper poses no such risks.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: All existing assets used in this work are properly cited in the main text.

    Guidelines:

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

13. **New assets**

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

Answer: [Yes]

Justification: We release the source code implementing our framework as part of the supplementary material.

Guidelines:

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

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

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

Answer: [N/A]

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

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

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

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

Answer: [N/A]

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

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

Answer: [Yes]

Justification: LLMs are a core component of agentic workflow optimization. LLMs work as both optimizer for agentic workflow optimization and executor in the workflow. Section 4.1 and Appendix C show the specific models used, their roles and configurations.

Guidelines:

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