CB-Orchestrator: Adaptive Workflow Optimization for LLM Agents via Contextual Bandits

Jiahang Sun [1] **Zhiwei Shang** [1] **Zhipiao Liu** [2] **Hongwei Yang** [2] **Guoqing Xie** [2] **Shuang Qiu** [3] **Zhongxiang Dai** [1]

Abstract

Large Language Model (LLM) agents have demonstrated remarkable capabilities in tackling complex tasks through agentic workflows. However, manually designing these workflows is labor-intensive and lacks flexibility to adapt to diverse queries. While automated workflow optimization is a promising direction, existing methods often incur expensive API costs during training-phase feedback collection. Furthermore, they fail to adaptively reuse high-quality workflows, which creates a bottleneck for both performance and cost-effectiveness. To address these limitations, we propose **CB-Orchestrator**, an adaptive framework that **decouples workflow generation from selection**. In the first stage, we construct a diverse pool of candidate workflows via evolutionary search. In the second stage, we formulate workflow selection as a **Contextual Bandit** problem, which enables sample-efficient learning by balancing exploration and exploitation, thereby significantly reducing the feedback required for training. During inference, the model adaptively selects the optimal workflow tailored to each query. Evaluations across five diverse benchmarks demonstrate that CB-Orchestrator consistently outperforms all baselines. Notably, compared to strong recent baselines, CB-Orchestrator achieves better performance while reducing training-phase API token overhead by no less than 40.55% and total end-to-end training (including workflow pool generation) token overhead by no less than 29.73%. Code is available at https://github.com/Jiahang-Sun/CB-Orchestrator.

[1]School of Data Science, The Chinese University of HongKong, Shenzhen, China [2]Huawei Technologies Co., Ltd., China [3]Department of Systems Engineering, City University of Hong Kong, China. Correspondence to: Zhongxiang Dai <daizhongxiang@cuhk.edu.cn>.

Accepted to the 2nd Workshop on Compositional Learning at ICML 2026, Seoul, South Korea. Copyright 2026 by the author(s).

1. Introduction

Large Language Model (LLM) agents have demonstrated remarkable capabilities in handling complex tasks through *agentic workflows* (Wu et al., 2023; Hong et al., 2023). An *agentic workflow* is a structured program that orchestrates a sequence of LLM calls and tool invocations to solve a query (Hu et al., 2025; Zhang et al., 2025e). However, manually designing such workflows is labor-intensive and often fails to adapt to diverse queries. This has motivated growing interest in automated workflow optimization.

As illustrated in Figure 1, current automatic workflow optimization largely falls into two paradigms: (i) *task-level optimization*, which implements a universal fixed workflow for a task type (Li et al., 2024; Hu et al., 2025; Zhang et al., 2025e); and (ii) *query-level optimization*, which generates a bespoke workflow for each incoming query (Zhang et al., 2025a; Zhu et al., 2025; Zhang et al., 2025h). However, these paradigms present a fundamental dilemma. Task-level approaches suffer from rigidity and fail to tailor solutions to diverse queries. In contrast, query-level approaches treat every query as a new generation problem. This significantly harder learning objective creates a bottleneck, inflating both the sample complexity needed for training-phase feedback collection and the computational overhead required for inference. Furthermore, neither paradigm effectively reuses high-quality workflows: learned solutions are either too generic or discarded after one-off use, preventing the system from benefiting from accumulated knowledge.

Beyond efficiency bottlenecks, a fundamental issue is that *workflow effectiveness is inherently query-dependent*. A single fixed workflow optimized at the task level can be unnecessarily heavy for simple instances yet insufficient for complex ones (e.g., arithmetic vs. multi-step verification in math reasoning; one-shot utility functions vs. iterative testing in code generation). While query-level methods address this heterogeneity, they often conflate *adaptation* with *generation*: generating a new workflow from scratch is a difficult program-synthesis problem with a vast search space, whereas selecting a suitable strategy from a reusable set is a much simpler decision problem. This motivates our key rationale: *choosing is much easier than creating*. We therefore aim to *reuse* a pool of strong workflows and spend

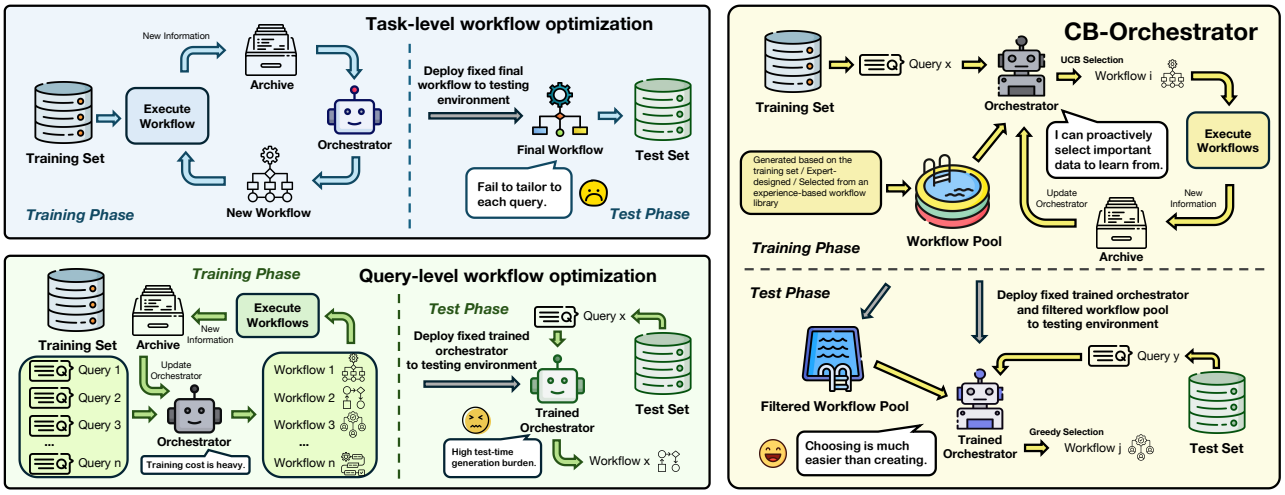

Figure 1. CB-Orchestrator vs. prior workflow-optimization paradigms.

the learning budget on *adaptive selection*.

To this end, we propose **CB-Orchestrator**, which **decouples workflow pool construction from adaptive selection** (Figure 1). CB-Orchestrator proceeds in two stages: Stage 1 constructs a diverse workflow pool automatically from the training set via execution-guided evolutionary search, and Stage 2 learns a query-conditioned selecting policy by formulating workflow selection as a *contextual bandit*. The bandit view enables a principled exploration–exploitation trade-off, allowing the selector to learn sample-efficiently.

Contributions. We make the following key contributions:

(1) **Problem Setting:** We formalize *adaptive workflow reuse* for LLM agents as learning a query-conditioned routing policy over a reusable workflow pool, as an alternative to task-level fixed-workflow deployment and query-level per-query workflow generation (Figure 1).

(2) **CB-Orchestrator:** We propose a two-stage framework that (i) constructs a diverse workflow pool *automatically from the training set* via execution-guided evolutionary search, and (ii) learns a *contextual-bandit* selector to adaptively choose workflows, enabling sample-efficient training by reducing the number of workflow evaluations needed to learn a strong policy.

(3) **Results:** Across five benchmarks, CB-Orchestrator consistently improves over strong baselines while reducing training-phase token overhead by $\geq 40.55\%$ and end-to-end training overhead (including workflow pool generation) by $\geq 29.73\%$.

2. Related Work

2.1. Agentic Workflows and Autonomous Agents

LLM applications can be organized into two paradigms: *agentic workflows* and *autonomous agents* (Zhang et al., 2025e). Agentic workflows decompose a task into a structured sequence of model invocations and tool calls, whereas autonomous agents follow a more flexible, stateful control policy that dynamically decides the next action based on intermediate outcomes and environment feedback. Agentic workflows generalize single-instance reasoning strategies (e.g., Chain-of-Thought (CoT) (Wei et al., 2022) and self-consistency (Wang et al., 2022)) into closed-loop architectures. Representative designs such as Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) incorporate execution feedback or self-evaluation signals to iteratively update intermediate solutions, enabling calibration over multiple turns rather than relying on a one-shot generation.

Autonomous agents maintain an explicit *state* (e.g., memory, plans, and tool results) and use a policy-like controller to select actions online from intermediate outcomes and environment feedback. Such designs are realized in agent frameworks (e.g., CAMEL (Li et al., 2023), AutoGen (Wu et al., 2023), and MetaGPT (Hong et al., 2023)), but are often labor-intensive to specify (prompts, roles, tools, and interaction topologies).

2.2. Automated Workflow Optimization

Recent progress has shifted from *manually designed* workflows to *automatically optimized* agentic systems. At the *task level*, the goal is to learn a *query-invariant* workflow for a given task/benchmark by optimizing a dataset-level objective. Representative examples include AutoFlow (Li et al., 2024), ADAS (Hu et al., 2025), AgentSquare (Shang et al.,

2025), and AFlow (Zhang et al., 2025e), which explore different search paradigms (e.g., heuristic search, modular search, and MCTS-style exploration). Beyond workflow discovery, several works broaden the optimization toolbox for multi-agent collaboration and evolution, including MASS (Zhou et al., 2025), OMAC (Li et al., 2025), SEW (Liu et al., 2025), and safety-aware evolutionary programming in MermaidFlow (Zheng et al., 2025). Swarm-intelligence-based generation (SwarmAgentic) (Zhang et al., 2025g) further pushes toward automated end-to-end system construction. A concurrent anonymous submission (included in the supplementary material) models task-level workflow search as an evolutionary process over a directed acyclic graph with multi-parent fusion. While this approach discovers strong workflows, its performance is limited by the inability to adaptively select workflows conditioned on the query.

In addition to searching over entire workflows, some approaches optimize *inter-agent communication structures* and *topologies* directly, treating MAS as optimizable graphs. GPTSwarm (Zhuge et al., 2024) models language agents as graph-structured systems, while G-Designer (Zhang et al., 2025c) explicitly architects multi-agent communication via graph learning. More broadly, supernet-style architecture optimization has been explored in MaAS (Zhang et al., 2025b), which learns an agentic supernet (a distribution over architecture) instead of a single fixed workflow.

At the *query level*, systems plan or adapt workflows conditioned on each input (Zhang et al., 2025a; Zhu et al., 2025; Zhang et al., 2025h). Researchers train agents (or meta-agents) to construct or orchestrate MAS automatically, such as MAS-GPT (Ye et al., 2025), Weak-for-Strong (Nie et al., 2025) and ScoreFlow (Wang et al., 2025). Complementary efforts study dynamic agent networks (Liu et al., 2024), query-level meta-agent reinforcement (Gao et al., 2025), robust workflow generation (Xu et al., 2025), modularized workflow automation (Niu et al., 2025), and self-optimizing hierarchical workflows (Ho et al., 2025). Researchers also explore efficient refinement signals, including difficulty-aware orchestration (Su et al., 2025) and failure-driven refinement (Zhang et al., 2025d). In contrast to prior work that primarily (i) optimizes a single dataset-level workflow or (ii) directly generates/refines workflows per query, our setting focuses on *adaptive reuse and selection* of workflows.

3. Methodology

We present **CB-Orchestrator**, an adaptive framework that decouples workflow generation (workflow pool construction) from selection. Section 3.1 introduces the workflow representation and formalizes the problem. Section 3.2 describes stage 1, workflow pool construction. Section 3.3 details stage 2, the contextual bandit formulation for adaptive selection. Section 3.4 introduces the inference procedure.

3.1. Workflow Representation and Problem Formulation

We represent workflows as executable Python classes with a standardized async callable interface. Following AFlow (Zhang et al., 2025e), we define a library of reusable *operators* as atomic building blocks that encapsulate common LLM interaction patterns (see Appendix A.9 for details). Each workflow is composed of multiple operator instances that orchestrate their execution to solve problems.

Consider a set of queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$, where each query q_i has an associated ground-truth answer a_i. We partition $\mathcal{Q}$ into two disjoint subsets: $\mathcal{Q}_{\text{train}}$ for workflow pool construction (Stage 1) and bandit policy training (Stage 2), and $\mathcal{Q}_{\text{test}}$ for final evaluation (Stage 3). An *agentic workflow* w is a structured program that orchestrates LLM calls and tool invocations to produce an answer $\hat{a} = w(q)$ for a query q.

We formulate *workflow optimization* as finding, for each query q, a workflow w_q that maximizes the expected reward:

$$w_q^* = \arg\max_{w} \mathbb{E}\left[r(q, w)\right] \tag{1}$$

where $r(q, w) \in [0, 1]$ measures the quality of the answer produced by workflow w for query q. In principle, this requires generating a customized workflow for every query, which is expensive.

CB-Orchestrator addresses this challenge by treating workflows as reusable assets and proposing a two-stage framework: (i) *workflow pool construction.* Discovering a diverse set of operator compositions $\mathcal{W} = \{w_1, \ldots, w_K\}$ that can be reused across queries, and (ii) *query-level workflow selection.* Learning a selection policy $\pi : \mathcal{Q} \to \mathcal{W}$ that routes each query to the best workflow in the pool.

3.2. Stage 1: Workflow Pool Construction

The first stage constructs a diverse pool of candidate workflows $\mathcal{W} = \{w_1, \ldots, w_K\}$ that serves as the arm set for the subsequent bandit selection. The pool construction is guided by two properties:

- **Validity**: $\forall w \in \mathcal{W}, \exists q \in \mathcal{Q}_{\text{train}}$ such that $r(q, w) \geq \tau$.
- **Completeness**: $\forall q \in \mathcal{Q}_{\text{train}}, \exists w \in \mathcal{W}$ such that $r(q, w) \geq \tau$.

The workflow pool can be populated from multiple sources: human-designed workflows, domain-validated workflows, or automatically discovered workflows. For fair comparison, we use automatically discovered workflows generated as follows. Given a seed set $\mathcal{Q}_{\text{seed}} \subseteq \mathcal{Q}_{\text{train}}$, we generate workflows through three steps (hyperparameters in Appendix A):

- **Initial Generation.** For each $q \in \mathcal{Q}_{\text{seed}}$, prompt an LLM to generate an initial workflow $w_0^{(q)} = \text{GENERATE}(q, \mathcal{O}, \mathcal{E})$, where $\mathcal{O}$ is the operator library and $\mathcal{E}$ contains examples. Successful workflows ($r(q, w) \geq \tau$) are added to $\mathcal{W}$.
- **Execution-Guided Evolution.** If the initial workflow fails, evolve it using execution feedback:

$$w_{j+1}^{(q)} = \text{EVOLVE}(w_j^{(q)}, q, \text{trace}_j, \text{error}_j) \quad (2)$$

  where trace_j captures operator-level execution traces and error_j contains error messages. Repeat until success or reaching the maximum evolution rounds.
- **Cross-Query Transfer.** For queries where evolution fails, retrieve similar successful queries using embedding similarity:

$$\mathcal{N}_k(q) = \underset{(q',w') \in \mathcal{S}}{\overset{(k)}{\arg\max}} \cos(\phi(q), \phi(q')) \quad (3)$$

  where $\mathcal{S}$ is the set of successful (query, workflow) pairs. Test if retrieved workflows directly solve q; otherwise, evolve them further.

See Appendix A.11 for complete prompts.

3.3. Stage 2: Contextual Bandit Selection

The second stage formulates workflow selection as a **Contextual Bandit** problem: the **context** is the query embedding $\phi(q) \in \mathbb{R}^d$ capturing semantic and structural properties; the **arms** are workflows $\mathcal{W}$, each represented by an embedding $\phi(w) \in \mathbb{R}^d$; and the **reward** is bounded feedback $r \in [0, 1]$ based on answer quality. The embedding function ϕ is defined in Section 3.3.2. We adopt the contextual bandit framework to achieve a principled balance between sample efficiency and performance under uncertain workflow outcomes.

We propose a Hybrid Neural-Linear UCB algorithm that decouples reward prediction from uncertainty quantification: a neural network captures the complex, non-linear reward surface, while a linear UCB component provides stable, interpretable exploration bonuses with well-understood theoretical properties. This separation allows each component to focus on its strength—expressiveness for the neural network, principled uncertainty for the linear model—rather than requiring a single model to do both.

3.3.1. Bandit Formulation and Assumptions

We formalize the interaction protocol as follows. At each round t:

1. The environment presents a query $q_t \sim \mathcal{Q}_{\text{train}}$.
2. The learner observes the context $\phi(q_t)$ and selects a workflow (arm) $w_t \in \mathcal{W}$.
3. The learner receives a bounded reward $r_t \in [0, 1]$ measuring answer quality.

We make the following assumptions for contextual bandits:

Assumption 3.1 (Semantic Embedding)**.** The embedding function ϕ preserves semantic similarity: queries requiring similar reasoning patterns have similar embeddings, and workflows implementing similar problem-solving strategies have similar embeddings. Formally, if q_1 and q_2 require similar skills to solve, then $\|\phi(q_1) - \phi(q_2)\|$ is small; if w_1 and w_2 implement similar operator compositions, then $\|\phi(w_1) - \phi(w_2)\|$ is small.

Assumption 3.2 (Reward Smoothness)**.** The expected reward function is Lipschitz continuous in the embedding space: semantically similar (query, workflow) pairs yield similar expected rewards. Formally, there exists $L > 0$ such that $|\mathbb{E}[r|q_1, w_1] - \mathbb{E}[r|q_2, w_2]| \leq L\|x(q_1, w_1) - x(q_2, w_2)\|$.

Assumption 3.3 (Realizability)**.** There exists a function $f^* : \mathbb{R}^{2d} \to [0, 1]$ such that $\mathbb{E}[r|q, w] = f^*(x(q, w))$, where $x(q, w) = [\phi(q); \phi(w)] \in \mathbb{R}^{2d}$ is the joint feature representation (see Section 3.3.2).

Empirical validation of these assumptions is provided in Appendix D.

3.3.2. Feature Representation

For a (query, workflow) pair (q, w), we construct a joint feature representation by concatenating their L_2-normalized embeddings:

$$x(q, w) = [\phi(q); \phi(w)] \in \mathbb{R}^{2d}, \quad \phi(\cdot) = \frac{\text{Embed}(\cdot)}{\|\text{Embed}(\cdot)\|_2} \quad (4)$$

We use a single frozen text embedding model to encode both query text and workflow source code. The workflow code serves as a semantic summary of the problem-solving strategy, capturing operator sequences and composition structure. The concatenated representation $x(q, w)$ encodes the compatibility between query requirements and workflow capabilities. Implementation details (e.g., context length handling) are provided in Appendix A.

3.3.3. Hybrid Neural-Linear Architecture

Our model decouples reward prediction from uncertainty quantification through two components:

Neural Network Component. We employ a 3-layer MLP with progressive dimensionality reduction to predict rewards:

$$f_\theta(x) = \sigma(\text{MLP}(x)) \in [0, 1] \quad (5)$$

Algorithm 1 CB-Orchestrator Training

Require: Workflow pool $\mathcal{W}$, training queries $\mathcal{Q}_{\text{train}}$, exploration rounds T_{explore}, total rounds T, exploration weight α, regularization λ, batch size B, training epochs E

Initialize f_θ, $A \leftarrow \lambda I$, $\mathcal{D} \leftarrow \emptyset$, $\mathcal{H} \leftarrow \emptyset$
for batch $b = 1$ to $\lceil T/B \rceil$ **do**
  $\mathcal{B} \leftarrow \emptyset$ *current batch*
  for $i = 1$ to B **do**
    $t \leftarrow (b-1) \cdot B + i$; **if** $t > T$ **then break**
    Sample $q \sim \mathcal{Q}_{\text{train}}$
    if $t \leq T_{\text{explore}}$ **then**
      $w \sim \text{Uniform}(\mathcal{W} \setminus \{w' : (q, w') \in \mathcal{H}\})$
    else
      $w \leftarrow \arg\max_{w' \in \mathcal{W} \setminus \{w'' : (q,w'') \in \mathcal{H}\}} \text{UCB}(q, w')$
    end if
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(q, w)\}$; $\mathcal{H} \leftarrow \mathcal{H} \cup \{(q, w)\}$
  end for
  $\{r_i\} \leftarrow \text{ParallelExec}(\mathcal{B})$ *Execute batch in parallel*
  for $(q, w, r) \in \mathcal{B}$ **do**
    $x \leftarrow [\phi(q); \phi(w)]$; $A \leftarrow A + xx^\top$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x, r)\}$
  end for
  Reset f_θ; train on $\mathcal{D}$ for E epochs
end for
return (f_θ, A)

where σ is the sigmoid output. Architecture details are provided in Appendix A.

Linear UCB Component. Following LinUCB (Li et al., 2010), we maintain a precision matrix $A = \lambda I + \sum_{t=1}^{T} x_t x_t^\top$ and compute uncertainty as:

$$u(x) = \sqrt{x^\top A^{-1} x} \tag{6}$$

In our hybrid design, $u(x)$ primarily measures *feature-space novelty/coverage* rather than directly estimating $\text{Var}[r \mid x]$; under the smoothness assumption, it serves as a practical proxy for exploration. This enables closed-form uncertainty quantification, making exploration behavior predictable via the parameter α. Computational details are in Appendix B.

Hybrid UCB Score. The final selection score combines exploitation (neural network prediction) with exploration (linear uncertainty):

$$\text{UCB}(q, w) = f_\theta(x(q, w)) + \alpha \cdot u(x(q, w)) \tag{7}$$

where $\alpha > 0$ controls the exploration-exploitation trade-off. At each round, we select:

$$w^* = \arg\max_{w \in \mathcal{W}} \text{UCB}(q, w) \tag{8}$$

3.3.4. Training Algorithm

Algorithm 1 presents the complete training procedure. The training consists of two phases, both using batch-parallel execution to reduce wall-clock time:

Random Exploration Phase. For the first T_{explore} rounds, we uniformly sample (query, workflow) pairs to collect diverse training data under the *no-duplicate* constraint. In each round, we sample $q \sim \mathcal{Q}_{\text{train}}$ and then sample $w \sim \text{Uniform}(\mathcal{W} \setminus \{w' : (q, w') \in \mathcal{H}\})$, execute to observe reward r, update $A \leftarrow A + xx^\top$, and add (x, r) to the training buffer.

UCB-Guided Optimization Phase. After exploration, we switch to UCB-guided selection. For each remaining round, we sample $q \sim \mathcal{Q}_{\text{train}}$, compute UCB scores using Equation 7, select $w^* = \arg\max_w \text{UCB}(q, w)$, execute and observe reward r, update A, and add (x, r) to the buffer.

Training details. We train the selector with batch-parallel workflow execution and a low-noise, no-duplicate evaluation protocol, retraining the reward predictor on the accumulated buffer after each batch; full details and discussion (including extensions to high-noise settings) are deferred to Appendix A.

3.4. Stage 3: Inference with Adaptive Selection

At inference time, given a new query $q \in \mathcal{Q}_{\text{test}}$, CB-Orchestrator selects and executes a workflow as follows:

Basic Inference. Compute embeddings and select the workflow with highest predicted reward:

$$w^* = \arg\max_{w \in \mathcal{W}} f_\theta(x(q, w)) \tag{9}$$

Note that we use pure exploitation (no exploration bonus) during inference, as the goal is to maximize performance rather than gather information.

Wilson Score Filtering. To improve robustness, we filter out unreliable workflows using Wilson score confidence intervals, retaining only workflows with sufficient trials and high success rates. Details are provided in Appendix A.

Online Learning Mode. CB-Orchestrator also supports an optional online learning mode that enables model adaptation to distribution shifts and dynamic pool expansion when encountering queries outside the current pool's coverage. Details are provided in Appendix A.4.

4. Experiments

4.1. Experimental Setup

Datasets. We conduct experiments on five diverse benchmarks: GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) for math reasoning, HumanEval (Chen, 2021)

Table 1. The performance of CB-Orchestrator compared to baseline methods on math reasoning and code generation benchmarks. Backbone is GPT-4o-mini for all experiments. CB-Orchestrator consistently achieves the best performance across all benchmarks.

Method	Workflow-based	MATH	GSM8K	MBPP	HumanEval	Avg.
Basic Prompting						
Vanilla	✗	46.29	87.45	71.83	87.08	73.16
CoT (Wei et al., 2022)	✗	46.40	87.10	71.83	88.13	73.37
CoT SC (5-shot) (Wang et al., 2022)	✓	47.91	87.57	73.60	88.60	74.42
Fixed Multi-Agent Systems						
LLM-Debate (Du et al., 2024)	✗	48.54	89.47	70.29	88.68	74.25
LLM-Blender (Jiang et al., 2023)	✗	46.92	88.35	77.05	88.80	75.28
DyLAN (Liu et al., 2024)	✗	48.63	89.98	77.30	90.42	76.58
AgentVerse (Chen et al., 2023)	✗	47.35	89.91	74.28	89.29	75.21
Autonomous Multi-agent Systems						
AutoAgents (Chen et al., 2024)	✗	45.32	87.69	71.95	87.64	73.15
GPTSwarm (Zhuge et al., 2024)	✓	47.88	89.14	77.43	89.32	75.94
ADAS (Hu et al., 2025)	✓	43.18	86.12	68.13	84.19	70.41
AgentSquare (Shang et al., 2025)	✓	48.51	87.62	78.46	89.08	75.92
AFlow (Zhang et al., 2025e)	✓	51.28	91.16	81.67	90.93	78.76
MaAS (Zhang et al., 2025b)	✓	51.82	92.30	82.17	92.85	79.79
CB-Orchestrator	✓	**53.87**	**93.19**	**84.05**	**94.35**	**81.37**

and MBPP (Austin et al., 2021) for code generation, and the comprehensive GAIA (Mialon et al., 2023) dataset for evaluation on real-world tasks requiring reasoning, multi-modality, and tool-use proficiency. For GSM8K, MATH and GAIA, we report the Solve Rate (%); for HumanEval and MBPP, we report pass@1. Detailed statistics regarding the datasets are presented in Appendix A.

Baselines. We compare our proposed CB-Orchestrator against a wide range of competitive baselines: (1) Basic Prompting methods, including Vanilla zero-shot prompting, Chain-of-Thought (CoT) (Wei et al., 2022), and CoT Self-Consistency (CoT SC) (Wang et al., 2022) which enhance robustness through majority voting. (2) Fixed Multi-Agent Systems, which employ static collaboration structures, including LLM-Debate (Du et al., 2024), LLM-Blender (Jiang et al., 2023), DyLAN (Liu et al., 2024), AgentVerse (Chen et al., 2023), TapeAgent (Bahdanau et al., 2024) and Sibyl(Wang et al., 2024). (3) Autonomous Multi-Agent Systems, which automatically generate agent systems. This group includes AutoAgents (Chen et al., 2024), GPTSwarm (Zhuge et al., 2024) that models agents as optimizable graphs, AgentSquare (Shang et al., 2025) for modular search, and recent high-performing frameworks AFlow (Zhang et al., 2025e) and MaAS (Zhang et al., 2025b).

Implementation Details. We employ GPT-4o-mini-0718 (Hurst et al., 2024) as the backbone LLM across all stages for both our algorithm and baselines, with Llama-3.1-70B (Grattafiori et al., 2024) used for cross-model generalization analysis. We employ the Qwen3-4B embedding model (Zhang et al., 2025f). For Stage 1, workflow generation uses temperature 1.0 for diversity, while all other LLM calls use temperature 0 for determinism. For Stage 2, total training rounds are $T = 500$ for HumanEval, $T = 3000$ for GAIA, and $T = 1200$ for other datasets. During inference, we run five independent evaluations per test query; for each benchmark, we compute the overall metric (Solve Rate / pass@1) per run and report the average across runs. See Appendix A for complete hyperparameters and dataset statistics.

4.2. Performance Analysis

Table 1 summarizes the performance of CB-Orchestrator compared to baseline methods on math reasoning and code generation benchmarks. The results demonstrate that **CB-Orchestrator consistently achieves the best performance across all benchmarks**, reaching a leading average score of 81.37%. This surpasses the strongest autonomous baselines, MaAS and AFlow, by 1.58% and 2.61% respectively, and significantly outperforms other baselines.

On the GAIA benchmark (Table 14 in Appendix C.3), **CB-Orchestrator achieves the highest average score (21.0%)**, with particularly strong performance on L1 tasks (28.6% solve rate).

4.3. Cost Efficiency and Training Dynamics

A key advantage of CB-Orchestrator is its exceptional sample efficiency during the training phase. By formulating workflow selection as a Contextual Bandit problem, we

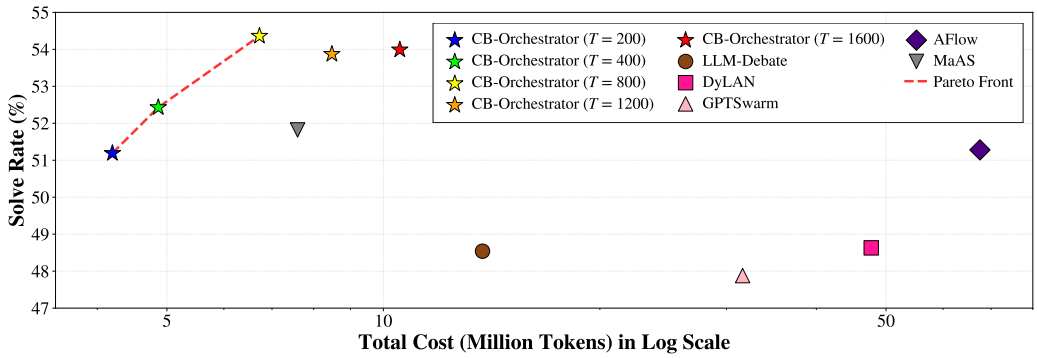

Figure 2. Cost-Efficiency comparison on the MATH benchmark.

effectively bridge the gap between exploration and exploitation. Figure 2 illustrates the relations between the solve rate (%) and the total cost (including training and test phase) on the MATH benchmark for CB-Orchestrator (with different T) and baselines. The results show that CB-Orchestrator forms a superior Pareto front. Specifically, CB-Orchestrator ($T = 400$) achieves a solve rate of 52.43%, higher than all baselines, with a cost of approximately 3.69 million tokens. Furthermore, **CB-Orchestrator** ($T = 800$) **significantly increases the performance ceiling to 54.36% while substantially reducing the total token cost** and keeping test-time overhead relatively low. As the workflow pool is reusable, if we do not count the cost during workflow pool construction step, CB-Orchestrator ($T = 800$) **reduces training token cost by no less than 40.55%** compared to all the baselines (cost accounting details in Appendix A.6).

In addition, Figure 2 shows CB-Orchestrator converges before $T = 800$. As illustrated in Figure 4 (Appendix A.8), CB-Orchestrator demonstrates rapid convergence at around Round 650. In practice, **we can easily observe the convergence from the trend of cumulative average reward and stop training at the appropriate time**. That is to say, the above-mentioned cost reduction figures can be stably achieved.

4.4. Ablation Study and Strategic Analysis

To investigate the contribution of each component in CB-Orchestrator, we conduct an ablation study on the MATH and MBPP benchmarks. We first define the experimental variants to clarify the impact of each design choice: PureRandom selects workflows randomly from the pool to establish a baseline. RandomNN selects samples randomly for training the neural network. GreedyNN removes the exploration bonus to evaluate the necessity of the uncertainty based exploration term. LinUCB (Li et al., 2010) and NeuralUCB (Zhou et al., 2020) represent standard linear and neural bandit respectively. WithoutEvo removes the evolutionary search phase and performs selection over a static initial pool of workflows. The results are summarized in Figure 3. Detailed numerical results are provided in Table 13 (Appendix C.2).

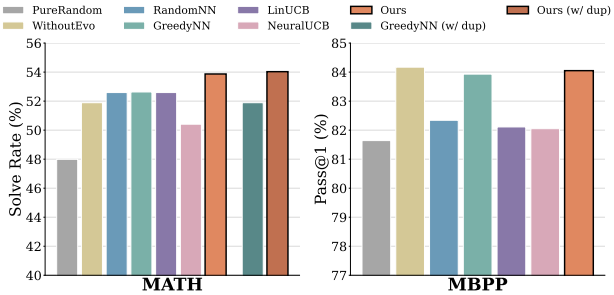

Figure 3. Ablation results on MATH and MBPP benchmarks. We compare different bandit selection strategies and evaluate the impact of the evolution process. Methods with black borders indicate our proposed approaches.

As indicated in Figure 3, **the workflow evolution mechanism provides a complementary benefit to the overall system**. Even the Without Evolution variant which performs bandit selection over a static initial workflow pool already achieves a strong average score of 68.03%. By integrating the iterative evolution process, CB-Orchestrator further boosts the performance to 68.96%. This result suggests that, while bandit-based selection serves as the primary driver of task adaptation, the evolution process acts as an effective optimization tool that refines workflow structures and further lifts the performance ceiling of the framework.

Our HybridUCB exploration strategy performs better than all baseline selection strategies. As shown in Figure 3, all bandit-based variants significantly surpass the PureRandom baseline. This confirms that leveraging workflow and query features to workflow selection is essential for performance. Interestingly, GreedyNN achieves a high score of 68.28%, which exceeds the performance of RandomNN, LinUCB, and NeuralUCB. This competitive performance of GreedyNN is due to our training protocol where re-exploring the same query-workflow pairs is disallowed.

This constraint forces the model to maintain a degree of exploration capability even without an explicit bonus term.

The underperformance of LinUCB and NeuralUCB highlights the complexity of the adaptive selection task. LinUCB is 1.61% lower than ours. It is hard for LinUCB to capture the intricate relationships between different dimensions of the embedding vectors. This limitation causes the model to settle on a single fixed workflow and prevents it from making adaptive selections based on the query context. NeuralUCB yields the lowest score among trained variants because its exploration assumption is too strong for our specific scenario. This leads to unreasonable exploration behavior that hinders stable learning. By contrast, our HybridUCB effectively balances exploitation via neural reward modeling with a reasonable exploration bonus to achieve best results.

The exploration bonus becomes more critical when duplicate sampling is allowed. We further conduct ablation experiments on MATH where the same (query, workflow) pairs can be evaluated multiple times during training. As shown in Figure 3, when duplicate sampling is allowed (w/ dup), Ours achieves 54.03% while GreedyNN drops to 51.89%. This 2.14% performance gap is larger than the 1.24% gap observed in the no-duplicate setting, demonstrating that the exploration bonus in HybridUCB plays a more important role when the training protocol does not inherently enforce exploration through the no-duplicate constraint.

4.5. Generalizability and Scalability Analysis

We further investigate the generalizability and scalability of CB-Orchestrator across different dimensions.

Cross-Domain Transfer. We conduct a cross-domain transferability analysis between the MATH and GSM8K benchmarks using two distinct settings: **(i) Source Pool & Source NN:** This configuration represents a pure zero-shot transfer, where both the evolved workflow structures and the neural reward model optimized on the source domain are applied directly to the target domain without any further refinement. **(ii) Source Pool & Retrained NN:** In this setting, we maintain the workflow pool from the source domain while training the neural reward model on the target domain using our HybridUCB method with $T = 1200$. As summarized in Table 2, our framework demonstrates remarkable generalizability. Even in the pure zero-shot Source Pool & Source NN setting, CB-Orchestrator outperforms many baseline methods. Furthermore, by simply retraining the reward model, the performance on MATH increases to 54.94%, significantly surpassing the strongest baseline and even our method trained on MATH; we hypothesize this may be partly due to GSM8K providing more training queries and thus richer feedback for learning the selector.

LLM Backbone Scalability. To verify the scalability of CB-Orchestrator across different LLM backbones, we evaluate its performance using Llama-3.1-70b as the backbone. We conduct experiments on the MATH and HumanEval benchmarks, because the strongest baseline, MaAS (Zhang et al., 2025b), reported results on these two datasets for this backbone. As summarized in Table 12, CB-Orchestrator achieves an average score of 64.24%, outperforming the Vanilla baseline by a large margin and demonstrating an overall performance slightly better than MaAS. These results demonstrate that our CB-Orchestrator scales effectively to open-source LLMs.

Table 2. Cross-domain transferability analysis. M and G denote MATH and GSM8K benchmarks, respectively. We evaluate the performance of CB-Orchestrator compared to baseline methods.

Method	**M → G**	**G → M**
Baseline Methods		
GPTSwarm	89.96	45.18
AFlow	91.95	49.39
MaAS	92.80	51.02
CB-Orchestrator (Ours)		
Source Pool & Source NN	89.72	53.62
Source Pool & Retrained NN	92.75	54.94

4.6. Case Study

To illustrate how CB-Orchestrator adaptively selects workflows based on query characteristics, we analyze the workflow selection patterns on the MATH test set. CB-Orchestrator primarily utilizes 6 workflows. Notably, different problem types trigger different workflow selections. Detailed examples with specific queries and workflow implementations are provided in Appendix E.

5. Conclusion

We introduced CB-Orchestrator, a two-stage framework for adaptive workflow optimization for LLM agents. We decouple workflow construction from adaptive selection. Stage 1 builds a diverse workflow pool via execution-guided evolutionary search; Stage 2 casts selection as a contextual bandit and applies Hybrid Neural-Linear UCB for sample-efficient exploration over query-workflow pairs.

Experiments across five benchmarks demonstrate that CB-Orchestrator consistently outperforms strong baselines while reducing training-phase token overhead by at least 40.55% and end-to-end training overhead by at least 29.73%. Ablation studies confirm that both the evolutionary pool construction and the Hybrid UCB exploration contribute to performance gains. Furthermore, CB-Orchestrator exhibits strong cross-domain transferability, scales to unified multi-task settings, and generalizes effectively to different LLM

backbones. These results, together with the framework's natural ability to incorporate prior knowledge into the workflow pool, suggest that CB-Orchestrator offers a practical path toward deploying versatile, cost-efficient LLM agents across diverse real-world applications.

Impact Statement

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

References

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

Bahdanau, D., Gontier, N., Huang, G., Kamalloo, E., Pardinas, R., Piché, A., Scholak, T., Shliazhko, O., Tremblay, J. P., Ghanem, K., et al. Tapeagents: a holistic framework for agent development and optimization. *arXiv preprint arXiv:2412.08445*, 2024.

Chen, G., Dong, S., Shu, Y., Zhang, G., Sesay, J., Karlsson, B., Fu, J., and Shi, Y. Autoagents: A framework for automatic agent generation. In *IJCAI International Joint Conference on Artificial Intelligence*, pp. 22 – 30, 2024.

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

Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Chan, C.-M., Yu, H., Lu, Y., Hung, Y.-H., Qian, C., et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023.

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

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of Machine Learning Research*, volume 235, pp. 11733 – 11763, Vienna, Austria, 2024.

Gao, H., Liu, Y., He, Y., Dou, L., Du, C., Deng, Z., Hooi, B., Lin, M., and Pang, T. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Ho, C.-T., Gong, J., Yao, X., Bai, Y., Akkur, A. B., and Ren, H. Polymath: A self-optimizing agent with dynamic hierarchical workflow. *arXiv preprint arXiv:2508.02959*, 2025.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. In *13th International Conference on Learning Representations, ICLR 2025*, pp. 80648 – 80681, Singapore, Singapore, 2025.

Hurst, A., Lerer, A., Goucher, A. P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A., Welihinda, A., Hayes, A., Radford, A., et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Jiang, D., Ren, X., and Lin, B. Y. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. volume 1, pp. 14165 – 14178, Toronto, ON, Canada, 2023. URL http://dx.doi.org/10.18653/v1/2023.acl-long.792.

Li, G., Hammoud, H., Itani, H., Khizbullin, D., and Ghanem, B. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

Li, L., Chu, W., Langford, J., and Schapire, R. E. A contextual-bandit approach to personalized news article recommendation. pp. 661 – 670, Raleigh, NC, United states, 2010.

Li, S., Hasson, H., and Ghosh, J. Omac: A broad optimization framework for llm-based multi-agent collaboration. *arXiv preprint arXiv:2505.11765*, 2025.

Li, Z., Xu, S., Mei, K., Hua, W., Rama, B., Raheja, O., Wang, H., Zhu, H., and Zhang, Y. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*, 2024.

Liu, S., Fang, J., Zhou, H., Wang, Y., and Meng, Z. Sew: Self-evolving agentic workflows for automated code generation. *arXiv preprint arXiv:2505.18646*, 2025.

Liu, Z., Zhang, Y., Li, P., Liu, Y., and Yang, D. A dynamic llm-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. volume 36, New Orleans, LA, United states, 2023.

Mialon, G., Fourrier, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Nie, F., Feng, L., Ye, H., Liang, W., Lu, P., Yao, H., Alahi, A., and Zou, J. Weak-for-strong: Training weak meta-agent to harness strong executors. *arXiv preprint arXiv:2504.04785*, 2025.

Niu, B., Song, Y., Lian, K., Shen, Y., Yao, Y., Zhang, K., and Liu, T. Flow: Modularized agentic workflow automation. *arXiv preprint arXiv:2501.07834*, 2025.

Shang, Y., Li, Y., Zhao, K., Ma, L., Liu, J., Xu, F., and Li, Y. Agentsquare: Automatic llm agent search in modular design space. In *13th International Conference on Learning Representations, ICLR 2025*, pp. 58955 – 58979, Singapore, Singapore, 2025.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. volume 36, New Orleans, LA, United states, 2023.

Su, J., Xia, Y., Lan, Q., Song, X., Jingsong, Y., He, L., and Shi, T. Difficulty-aware agent orchestration in llm-powered workflows. *arXiv e-prints*, pp. arXiv–2509, 2025.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. 2022. URL http://dx.doi.org/10.48550/arXiv.2203.11171.

Wang, Y., Shen, T., Liu, L., and Xie, J. Sibyl: Simple yet effective agent framework for complex real-world reasoning. *arXiv preprint arXiv:2407.10718*, 2024.

Wang, Y., Yang, L., Li, G., Wang, M., and Aragam, B. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, New Orleans, LA, United states, 2022.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation. 2023. URL http://dx.doi.org/10.48550/arXiv.2308.08155.

Xu, S., Zhang, J., Di, S., Luo, Y., Yao, L., Liu, H., Zhu, J., Liu, F., and Zhang, M.-L. Robustflow: Towards robust agentic workflow generation. *arXiv preprint arXiv:2509.21834*, 2025.

Ye, R., Tang, S., Ge, R., Du, Y., Yin, Z., Chen, S., and Shao, J. Mas-gpt: Training llms to build llm-based multi-agent systems. volume 267, pp. 72063 – 72090, Vancouver, BC, Canada, 2025.

Zhang, G., Chen, K., Wan, G., Chang, H., Cheng, H., Wang, K., Hu, S., and Bai, L. Evoflow: Evolving diverse agentic workflows on the fly. *arXiv preprint arXiv:2502.07373*, 2025a.

Zhang, G., Niu, L., Fang, J., Wang, K., Bai, L., and Wang, X. Multi-agent architecture search via agentic supernet. In *Proceedings of Machine Learning Research*, volume 267, pp. 75834 – 75852, Vancouver, BC, Canada, 2025b.

Zhang, G., Yue, Y., Sun, X., Wan, G., Yu, M., Fang, J., Wang, K., Chen, T., and Cheng, D. G-designer: Architecting multi-agent communication topologies via graph neural networks. volume 267, pp. 76678 – 76692, Vancouver, BC, Canada, 2025c.

Zhang, J., Cai, K., Zeng, Q., Liu, N., Fan, S., Chen, Z., and Wang, K. Failure-driven workflow refinement. *arXiv preprint arXiv:2510.10035*, 2025d.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X.-H., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. Aflow: Automating agentic workflow generation. In *13th International Conference on Learning Representations, ICLR 2025*, pp. 100351 – 100388, Singapore, Singapore, 2025e.

Zhang, Y., Li, M., Long, D., Zhang, X., Lin, H., Yang, B., Xie, P., Yang, A., Liu, D., Lin, J., et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025f.

Zhang, Y., Lin, C., Tang, S., Chen, H., Zhou, S., Ma, Y., and Tresp, V. Swarmagentic: Towards fully automated agentic system generation via swarm intelligence. *arXiv preprint arXiv:2506.15672*, 2025g.

Zhang, Y., Liu, X., and Xiao, C. Metaagent: Automatically constructing multi-agent systems based on finite state machines. volume 267, pp. 75667 – 75694, Vancouver, BC, Canada, 2025h.

Zheng, C., Chen, J., Lyu, Y., Ng, W. Z. T., Zhang, H., Ong, Y.-S., Tsang, I., and Yin, H. Mermaidflow: Redefining agentic workflow generation via safety-constrained evolutionary programming. *arXiv preprint arXiv:2505.22967*, 2025.

Zhou, D., Li, L., and Gu, Q. Neural contextual bandits with ucb-based exploration. In *International conference on machine learning*, pp. 11492–11502. PMLR, 2020.

Zhou, H., Wan, X., Sun, R., Palangi, H., Iqbal, S., Vulić, I., Korhonen, A., and Arık, S. Ö. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*, 2025.

Zhu, R., Jiang, B., Mei, L., Yang, F., Wang, L., Gao, H., Bai, F., Zhao, P., Lin, Q., Rajmohan, S., et al. Adaptflow: Adaptive workflow optimization via meta-learning. *arXiv preprint arXiv:2508.08053*, 2025.

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., and Schmidhuber, J. Gptswarm: Language agents as optimizable graphs. In *Proceedings of Machine Learning Research*, volume 235, pp. 62743 – 62767, Vienna, Austria, 2024.

A. Implementation Details

A.1. Dataset Statistics

Table 3 summarizes the datasets used in our experiments. We use the validation split for training (Stage 1 workflow pool construction and Stage 2 bandit training) and the test split for evaluation (Stage 3 inference).

Table 3. Dataset statistics...

Dataset	Domain	Train	Test	Workflows	Operators
MATH	Math reasoning	119	486	68	Custom, ScEnsemble, Programmer
GSM8K	Math reasoning	264	1055	72	Custom, ScEnsemble, Programmer
HumanEval	Code generation	33	131	31	Custom, CustomCodeGenerate, ScEnsemble, Test
MBPP	Code generation	86	341	81	Custom, CustomCodeGenerate, ScEnsemble, Test
GAIA	Multi-tool QA	93	301	36	Custom, Reasoning, Decompose, Search, FileAnalysis, Programmer, ScEnsemble, FinalAnswer

GSM8K Training Subset for Stage 1. GSM8K has a relatively large training split. For workflow pool construction (Stage 1), we sample 100 training queries from the full GSM8K training split to generate candidate workflows and to compute pool coverage statistics.

A.2. Hyperparameters

Table 4 summarizes all hyperparameters used in CB-Orchestrator.

Table 4. Hyperparameters for CB-Orchestrator.

Parameter	Sym.	Val.	Parameter	Sym.	Val.
Stage 1: Workflow Pool Construction					
Evolution rounds	N_{evolve}	2	Transfer rounds	N_{transfer}	2
Gen. temperature	-	1.0	Exec. temperature	-	0
Similar queries	k	5			
Stage 2: Bandit Training					
Embedding dim	d	2560	Learning rate	η	10^{-5}
Hidden dim 1	h_1	2048	Weight decay	-	10^{-3}
Hidden dim 2	h_2	64	Gradient clip	-	1.0
Exploration coef.	α	0.1	Epochs/batch	E	50
Regularization	λ	0.5	NN batch size	-	16
Total rounds	T	See below	Explore rounds	T_{explore}	100
Eval batch size	B	25	Random seed	-	42
Stage 3: Inference					
Wilson confidence	z	3.891	Min samples	$n_{\min}$	5
Keep ratio	-	0.5			

Training Rounds per Dataset. We use $T = 1200$ for MATH, GSM8K, and MBPP, $T = 3000$ for GAIA, and $T = 500$ for HumanEval.

A.3. Architecture and Feature Representation Details

Neural Network Architecture. The reward prediction network is a 3-layer MLP with pre-norm design and progressive dimensionality reduction:

$$h_0 = \text{LN}(x) \in \mathbb{R}^{2d} \tag{10}$$

$$h_1 = \text{GELU}(\text{LN}(W_1 h_0 + b_1)) \in \mathbb{R}^{d_1} \tag{11}$$

$$h_2 = \text{GELU}(\text{LN}(W_2 h_1 + b_2)) \in \mathbb{R}^{d_2} \tag{12}$$

$$f_\theta(x) = \sigma(W_3 h_2 + b_3) \in [0, 1] \tag{13}$$

where LN denotes layer normalization, GELU is the activation function, and σ is the sigmoid output. We use $d = 2560$ (embedding dimension), $d_1 = 2048$, and $d_2 = 64$, progressively compressing the representation from $2d = 5120$ to 1.

Feature Representation. Workflow source code serves as a stable semantic summary of the problem-solving strategy. Each workflow contains: (1) the sequence of operators invoked (e.g., `Programmer`, `ScEnsemble`, `Test`), (2) their composition structure (sequential, branching, iterative), and (3) operator-specific parameters and prompts.

Wilson Score Filtering. At inference time, we filter workflows using Wilson score confidence intervals to improve robustness. For each workflow w with n trials and s successes during training, the Wilson lower bound is:

$$\text{Wilson}(w) = \frac{s + \frac{z^2}{2}}{n + z^2} - \frac{z}{n + z^2}\sqrt{\frac{s(n-s)}{n} + \frac{z^2}{4}} \tag{14}$$

where $z = 3.891$ corresponds to 99.99% confidence. We require $n_{\min} = 5$ samples per workflow and keep only the top 50% of workflows ranked by Wilson score, filtering out unreliable workflows with insufficient trials or consistently poor performance.

A.4. Online Learning Mode

A natural concern is that the workflow pool $\mathcal{W}$, though diverse, is finite and constructed from training queries. When encountering queries that fall outside the pool's coverage—requiring reasoning patterns not captured by any existing workflow—the system can only select the "least bad" option. To address this limitation, CB-Orchestrator supports an optional *online learning mode* that enables both model adaptation and dynamic pool expansion.

Model Adaptation. After each inference, the model can continue learning from observed rewards:

$$A \leftarrow A + xx^\top, \quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(x, r)\} \tag{15}$$

Periodic retraining of f_θ adapts the selection policy to distribution shifts in the query stream, improving performance on recurring query patterns.

Coverage Gap Detection. For each query q, we compute a coverage score reflecting the pool's ability to handle it:

$$\text{Coverage}(q) = \max_{w \in \mathcal{W}} f_\theta(x(q, w)) \tag{16}$$

A low coverage score (below threshold τ_{gap}) combined with high uncertainty $u(x(q, w))$ across all workflows signals that q likely requires a strategy absent from the current pool. We define the gap indicator:

$$\text{Gap}(q) = \mathbb{I}\left[\text{Coverage}(q) < \tau_{\text{gap}} \wedge \min_{w \in \mathcal{W}} u(x(q, w)) > u_{\text{thresh}}\right] \tag{17}$$

On-Demand Workflow Generation. When $\text{Gap}(q) = 1$, we invoke the same generation mechanism from Stage 1 using q as the seed query: (1) generate an initial workflow tailored to q's requirements, (2) evolve using execution feedback if the initial attempt fails, and (3) add successful workflows to the pool. The new workflow w_{new} is integrated into the bandit:

$$\mathcal{W} \leftarrow \mathcal{W} \cup \{w_{\text{new}}\} \tag{18}$$

Since w_{new} has no prior observations, its uncertainty $u(x(q, w_{\text{new}}))$ is naturally high, encouraging exploration during subsequent queries. This mechanism enables the system to continuously expand its capabilities rather than being constrained to the initial pool.

Practical Considerations. On-demand generation incurs additional latency and cost, so it should be triggered judiciously. In practice, we recommend: (1) batching gap queries and generating workflows offline during low-traffic periods, (2) setting conservative thresholds ($\tau_{\text{gap}} \approx 0.3$, u_{thresh} at the 90th percentile of training uncertainties) to avoid unnecessary generation, and (3) periodically pruning redundant workflows using the Wilson score filtering mechanism. This design ensures that the pool grows adaptively while maintaining computational efficiency.

A.5. Workflow Pool Coverage and Training Subset

Pool Coverage. Table 5 reports the coverage of our constructed workflow pools. In all benchmarks, the evaluator provides binary feedback $r \in \{0, 1\}$ (success/failure), so we use a fixed success threshold $\tau = 1$ without additional tuning. Here, "Train Queries" refers to the subset of training queries used in Stage 1 workflow pool construction for that dataset (e.g., we sample 100 training queries for GSM8K). Coverage is defined as the fraction of these Stage 1 training queries for which at least one workflow in the pool achieves success ($r \geq \tau$). High coverage indicates that the pool contains effective workflows for diverse query types.

Table 5. Workflow pool coverage across datasets.

Dataset	Train Queries	Covered	Coverage	Pool Size
MATH	119	86	72.3%	68
GSM8K	100	98	98.0%	72
HumanEval	33	31	93.9%	31
MBPP	86	82	95.3%	81

The lower coverage on MATH (72.3%) reflects the inherent difficulty of competition-level mathematics problems. Despite this, our bandit policy learns to select effective workflows for the covered queries.

Training Workflow Subset. For Stage 2 bandit training, we sample a subset of $K \approx 30$ workflows from the full pool rather than using all workflows. This design choice is motivated by several observations:

- **Empirical sufficiency**: Our experiments show that 30 workflows provide sufficient diversity to achieve strong performance, as many workflows in the pool have similar strategies and near-identical success patterns.
- **Training efficiency**: Fewer arms reduce the sample complexity of bandit learning. With K arms and T training rounds, each arm receives approximately T/K samples on average; using fewer arms increases per-arm sample count, improving reward estimation.
- **Inference filtering**: The Wilson score filtering in Stage 3 (Section 3.4) further reduces the active workflow set, so training on a subset does not limit final performance.

In practice, the number of training workflows can be adjusted based on the training budget and pool diversity. For datasets with smaller pools (e.g., HumanEval with 31 workflows), we use all available workflows.

A.6. Cost Breakdown and Accounting

To ensure fair comparison, we provide a detailed cost breakdown for all methods on the MATH benchmark. Table 6 reports token consumption across different phases.

Cost Accounting Principles. We apply consistent accounting across all methods:

- **All LLM calls counted**: Training, inference, workflow generation, and any internal evaluator calls are included.
- **Failed executions**: Executions that fail or raise exceptions still incur token costs; we count all tokens consumed before failure.
- **Input and output tokens**: Both prompt tokens and completion tokens are summed. Table 7 provides the input/output breakdown.

Table 6. Detailed cost breakdown on MATH benchmark (all values in millions of tokens).

Method	Pool Gen.	Training	Inference	Total	Solve %
LLM-Debate	–	–	13.73	13.73	48.54
DyLAN	–	38.30	9.39	47.68	48.63
GPTSwarm	–	27.70	3.89	31.59	47.88
AFlow	–	62.88	4.66	67.54	51.28
MaAS	–	5.43	2.16	7.60	51.82
CB-Orch (T=200)	0.59	0.78	2.24	3.61	51.19
CB-Orch (T=400)	0.59	1.53	2.16	4.28	52.43
CB-Orch (T=800)	0.59	3.23	2.32	6.14	54.36
CB-Orch (T=1200)	0.59	5.00	2.30	7.89	53.87
CB-Orch (T=1600)	0.59	6.91	2.45	9.95	53.99

Table 7. Input/output token breakdown on MATH (millions of tokens).

Method	Train Input	Train Output	Infer Input	Infer Output
LLM-Debate	–	–	3.28	10.46
DyLAN	22.15	16.15	6.08	3.30
GPTSwarm	21.33	6.37	3.11	0.79
AFlow	33.83	29.05	2.51	2.15
MaAS	3.05	2.38	1.31	0.85
CB-Orch (T=1200)	3.24	1.76	1.60	0.70

Workflow Pool Reusability. The workflow pool generation cost (0.59M tokens for MATH) is a one-time expense that can be amortized across multiple deployments. If we exclude pool generation, CB-Orchestrator (T=800) achieves 54.36% solve rate with only 5.55M tokens, representing a 26.97% total token cost (including training and test phase) reduction compared to MaAS.

A.7. Attribution

Parts of the code used in our experiments are adapted from the open-source implementation of AFlow (Zhang et al., 2025e) and MaAS (Zhang et al., 2025b)

A.8. Training Dynamics

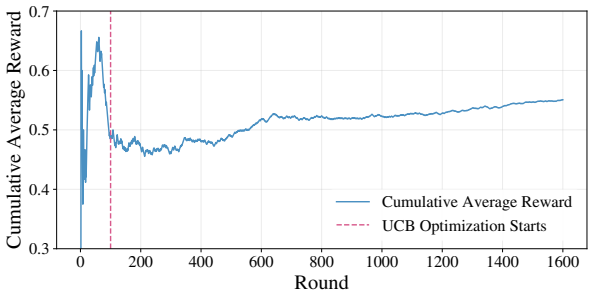

Figure 4. Cumulative average reward during bandit training on MATH. The curve shows rapid convergence at around Round 650, indicating that the bandit policy has learned effective workflow selection.

A.9. Operator Library

Following AFlow (Zhang et al., 2025e), we define operators as reusable atomic building blocks that encapsulate common LLM interaction patterns. Each operator has a standardized interface with typed inputs and outputs, enabling modular composition. We design domain-specific operator libraries for different task types. Table 8 shows operators for math reasoning (MATH, GSM8K), Table 9 for code generation (HumanEval, MBPP), and Table 10 for multi-tool QA (GAIA).

Table 8. Operators for math reasoning benchmarks (MATH, GSM8K).

Operator	Description & Interface
`Custom`	General-purpose LLM call with customizable prompts. Flexible for planning, reflection, decomposition, formatting, etc. `(input: str, instruction: str) -> {response: str}`
`ScEnsemble`	Self-consistency voting across multiple solutions. Requires 3+ solutions for effective voting. `(solutions: List[str], problem: str) -> {response: str}`
`Programmer`	Generates and executes Python code in a sandbox. Auto-retries on errors. Returns computed answer. `(problem: str, analysis: str) -> {code: str, output: str}`

Table 9. Operators for code generation benchmarks (HumanEval, MBPP).

Operator	Description & Interface
`Custom`	General-purpose LLM call for reasoning about code structure and logic. `(input: str, instruction: str) -> {response: str}`
`CustomCodeGenerate`	Specialized code generation with entry point awareness. `(problem: str, entry_point: str, instruction: str) -> {response: str}`
`ScEnsemble`	Self-consistency voting for code solutions. `(solutions: List[str], problem: str) -> {response: str}`
`Test`	Executes test cases and iteratively refines code. Retries up to 3 times on failures. `(problem: str, solution: str, entry_point: str) -> {result: bool, solution: str}`

A.10. Workflow Example

Below is an example workflow generated for MATH problems. The workflow uses a three-step structure: decomposition, code-based solution, and LaTeX formatting.

Table 10. Operators for multi-tool QA benchmark (GAIA).

Operator	Description & Interface
`Custom`	General-purpose LLM call for flexible reasoning and formatting. `(input: str, instruction: str) -> {response: str}`
`Reasoning`	Multi-step structured reasoning. Can receive context from other operators. `(question: str, context: str) -> {thought: str, answer: str}`
`Decompose`	Breaks complex questions into sub-tasks with dependency tracking. `(question: str, context: str) -> {analysis: str, subtasks: List, final_aggregation: str}`
`Search`	Iterative web search with query generation, page fetching, and answer extraction. `(question: str) -> {answer: str}`
`FileAnalysis`	Extracts information from attachments (PDF, Excel, CSV, images, text, JSON). `(question: str, file_path: str) -> {file_info: str}`
`Programmer`	Python code execution for computational tasks. `(question: str, context: str) -> {code: str, output: str}`
`ScEnsemble`	Self-consistency voting across multiple solutions. `(solutions: List[str], problem: str) -> {response: str}`
`FinalAnswer`	Mandatory final step for answer formatting per QA evaluation rules. `(question: str, analysis: str) -> {thought: str, final_answer: str}`

Example Workflow for MATH (Python)

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

    async def __call__(self, problem: str):
        # Step 1: Decompose the problem
        decomposition = await self.custom(
            input=problem,
            instruction="Break down the problem into steps...")

        # Step 2: Solve using code execution
        solution = await self.programmer(
            problem=decomposition['response'],
            analysis="Write a solve() function...")

        # Step 3: Format answer in LaTeX
        formatted = await self.custom(
            input=f"Solution: {solution['output']}",
            instruction="Format as \\boxed{answer}...")

        return formatted['response'], self.llm.get_cost()
```

This workflow demonstrates the typical operator composition pattern: `Custom` handles flexible text processing (decomposition, formatting), while `Programmer` executes code for numerical computation.

A.11. Workflow Generation Prompts

We provide the complete prompts used in Stage 1 for workflow generation and evolution. Variables in {`braces`} are filled at runtime.

A.11.1. Initial Generation System Prompt

WORKFLOW_GENERATE_SYSTEM

```
You are building a Graph to solve {type} problems.
Please construct a complete and correct workflow graph.

**IMPORTANT: When initializing operators in __init__, you must pass self.llm as the
first argument.  Check the 'initialization' field in operator_description for the
correct syntax.**

**CRITICAL - Operator Call Syntax:**
- Operators are callable objects (they implement __call__).  Call them DIRECTLY on the
instance:
  - CORRECT: `await self.search(question=problem)`,
`await self.file_analysis(question=problem, file_path=file_path)`
  - WRONG: `await self.search.search(...)`,
`await self.file_analysis.file_analysis(...)`
- This applies to ALL operators:  custom, reasoning, decompose, search,
file_analysis, programmer, sc_ensemble, final_answer

**CRITICAL - Async Result Handling:**
- You CANNOT access dict keys directly after await in one expression!
- WRONG: `return await self.final_answer(...)['final_answer']` (causes "coroutine was
never awaited" error)
- CORRECT: First store the result, then access keys:
  result = await self.final_answer(...)
  return result['final_answer'], ...

Here's an example of using the `custom` method in graph:
  CUSTOM_PROMPT = "Your instruction here"
  response = await self.custom(input=problem, instruction=CUSTOM_PROMPT)
In custom, the input and instruction are directly concatenated (instruction+input).
You may construct `input` by concatenating problem and strings previously generated
by other operators, such as problem statement, previous reasoning, candidate
solutions, test feedback, etc.  using f-strings.
The Custom operator is very flexible and can be used multiple times with different
prompts.  The function of Custom method can be Planning, Reflection, Decomposition,
Synthesis and whatever you want.
**IMPORTANT - Do not use Programmer unless necessary.**

**IMPORTANT: ScEnsemble requires MULTIPLE DIFFERENT solutions (typically 3+) to the
SAME problem for voting.  Do NOT use ScEnsemble with only one solution.**

**IMPORTANT - Answer Format Standardization:**
- **For math problems**:  MUST add a final Custom node to format the answer using
proper LaTeX format:
  FORMAT_PROMPT = '''Given the problem and solution, extract only the final answer
and format it as \boxed{{answer}}.
  IMPORTANT formatting rules:
  - Use LaTeX \frac{{a}}{{b}} for fractions, NOT a/b
  - Use LaTeX \sqrt{{x}} for square roots, NOT sqrt(x)
  - Use \pi for pi, NOT decimal approximations'''
- **For code problems**:  Test operator already returns properly formatted code, so
DO NOT add formatting after Test.

**Under no circumstances should Graph output None for any field.**
```

A.11.2. Initial Generation User Prompt

WORKFLOW_GENERATE_INPUT

```
Here is a problem example from the {type} domain:
<problem>
{problem}
</problem>

Available operators:
{operator_description}

Please generate a complete workflow graph that can solve {type} problems.
**CRITICAL: Design the workflow to be generalizable across the {type} problem
category, NOT just tailored to this specific instance.**

The graph must follow this exact template structure:

from typing import Literal
import asyncio
import workspace_bandit.{dataset}.workflows.template.operator as op
from scripts.async_llm import create_llm_instance
from scripts.evaluator import DatasetType

class Workflow:
  def __init__(self, name:  str, llm_config, dataset:  DatasetType) -> None:
    self.name = name
    self.dataset = dataset
    self.llm = create_llm_instance(llm_config)
    # All operators pre-initialized:
    {operator_init}
  async def __call__(self, problem:  str{entry_point_param}):
    # Your workflow logic here
    return answer, self.llm.get_usage_summary()["total_cost"]

Output ONLY the complete Python code in <graph></graph> tags.  DO NOT use markdown
code blocks.
```

A.11.3. Evolution System Prompt

WORKFLOW_EVOLVE_SYSTEM

```
You are an expert at debugging and improving workflow graphs for {type} problems.

A workflow failed to solve a problem.  Your task is to analyze the failure and
create an IMPROVED workflow.

####################################
NOTICE:
**VERY CRITICAL ANTI-OVERFITTING RULES:**
1.  You MUST NOT copy or embed any problem-specific tokens, values, strings, or
names into your prompts/logic/code
2.  You MUST NOT hardcode constants that come from this specific problem (e.g.,
specific numbers, names, or strings from the problem)
3.  Your fix must work for ALL similar problems, not just this one
4.  Focus on improving the GENERAL strategy, not solving this specific instance
####################################

**IMPORTANT: When initializing operators in __init__, you must pass self.llm as the
first argument.**

**Improvement Approach:**
Analyze the execution trace and error information to identify the root cause.
Based on your analysis, decide what changes are needed - this could include:
- **Modifying prompts**
- Adjusting workflow structure
- Adding/removing operators

**Under no circumstances should Graph output None for any field.**
```

A.11.4. Evolution User Prompt (Math/General)

WORKFLOW_EVOLVE_INPUT_MATH

```
## Previous Workflow (Failed):
```python
{previous_workflow}
```
Problem That Failed:
<problem>
{problem}
</problem>
Execution Trace (Operator Inputs/Outputs):
{execution_trace}
Error Information:
{error_info}
Expected Answer:
{expected_output}
Workflow's Predicted Answer:
{actual_output}
Available Operators:
{operator_description}
```

A.11.5. Evolution User Prompt (Code)

WORKFLOW_EVOLVE_INPUT_CODE

```
## Previous Workflow (Failed):
```python
{previous_workflow}
```
Problem That Failed:
<problem>
{problem}
</problem>
Execution Trace (Operator Inputs/Outputs):
{execution_trace}
Error Information:
{error_info}
Correct Code (Reference Solution):
```python
{correct_code}
```
Wrong Code (Workflow Generated):
```python
{wrong_code}
```
Test Case Results:
{test_case_results}
Available Operators:
{operator_description}
```

A.11.6. EVOLUTION USER PROMPT (QA)

WORKFLOW_EVOLVE_INPUT_QA

```
## Previous Workflow (Failed):
```python
{previous_workflow}
```
Problem That Failed:
<problem>
{problem}
</problem>
Execution Trace (Operator Inputs/Outputs):
{execution_trace}
Error Information:
{error_info}
Expected Answer:
{expected_output}
Workflow's Predicted Answer:
{actual_output}
Available Operators:
{operator_description}
QA-Specific Guidance:
- If file_path is provided, call FileAnalysis FIRST - it returns 'file_info'
(extracted info, NOT an answer)
- FileAnalysis -> Reasoning (with file_info as context) -> FinalAnswer is the
standard file-based pipeline
- For web-based questions, use Search (check 'answer' field - empty string means
search failed)
- For computational questions on file data, use FileAnalysis -> Programmer (pass
file_info as context)
- For factual questions, prefer Search over Reasoning when possible
- Always use FinalAnswer as the LAST step to format the answer correctly
```

A.11.7. EVOLUTION TASK PROMPT

WORKFLOW_EVOLVE_TASK

```
---
**Your Task:**
1. Analyze WHY the workflow failed
2. Design an IMPROVED workflow that fixes the issue
3. **CRITICAL**: Ensure your fix is GENERALIZABLE - it must work for similar
problems, not just this one

**CRITICAL - Operator Call Syntax:**
- Operators are callable objects. Call them directly: `await self.search(...)`, NOT
`await self.search.search(...)`
- WRONG: `await self.file_analysis.file_analysis(...)`, `await
self.reasoning.reasoning(...)`
- CORRECT: `await self.file_analysis(...)`, `await self.reasoning(...)`

The graph must follow this exact template structure:

from typing import Literal
import asyncio
import workspace_bandit.{dataset}.workflows.template.operator as op
from scripts.async_llm import create_llm_instance
from scripts.evaluator import DatasetType

class Workflow:
  def __init__(self, name: str, llm_config, dataset: DatasetType) -> None:
    self.name = name
    self.dataset = dataset
    self.llm = create_llm_instance(llm_config)
    # Initialize operators using op.OperatorName(self.llm)

  async def __call__(self, problem: str{entry_point_param}):
    # Your improved workflow implementation
    return answer, self.llm.get_usage_summary()["total_cost"]

Output a brief analysis in <analysis></analysis> tags (root cause + fix strategy),
then the complete Python code in <graph></graph> tags.
```

B. Theoretical Discussion

B.1. Design Rationale and Theoretical Positioning

We begin with an honest assessment of our Hybrid Neural-Linear UCB design. Our approach is a **practical design choice** that prioritizes stability and computational efficiency over theoretical optimality. Unlike NeuralUCB (Zhou et al., 2020) or Neural-LinUCB, which compute uncertainty in the neural network's learned representation space and enjoy formal regret guarantees under specific assumptions (e.g., the neural tangent kernel regime), our method decouples reward prediction from uncertainty quantification by using the *original embedding space* for exploration bonuses.

This decoupling means we cannot directly inherit the $\tilde{O}(\sqrt{T})$ regret bounds from neural bandit theory. However, we argue that this design is well-motivated for our application, and we provide intuitive justification grounded in our smoothness assumption.

B.2. Intuitive Justification via Lipschitz Smoothness

The key insight is that **under Assumption 3.2 (reward smoothness), feature-space novelty serves as a reasonable proxy for reward uncertainty**.

Lipschitz Continuity Implies Bounded Prediction Error. Assumption 3.2 states that the expected reward function $f^*(x) = \mathbb{E}[r|x]$ is L-Lipschitz continuous:

$$|f^*(x_1) - f^*(x_2)| \leq L \cdot \|x_1 - x_2\| \tag{19}$$

This means that for any new context x, if we have observed a nearby context x' with known reward, the prediction error is bounded by $L \cdot \|x - x'\|$.

Linear UCB Uncertainty Captures Feature-Space Novelty. The linear UCB uncertainty $u(x) = \sqrt{x^\top A^{-1} x}$ measures how "novel" the feature vector x is relative to previously observed features. Geometrically, $u(x)$ is large when x lies in directions that have been under-explored, and small when x aligns with well-covered directions in the feature space.

Connecting Novelty to Reward Uncertainty. Under Lipschitz smoothness, feature-space novelty translates to reward uncertainty: if x is far from all observed contexts (high $u(x)$), then by Lipschitz continuity, our reward prediction for x may have high error. Conversely, if x is close to many observed contexts (low $u(x)$), the Lipschitz bound ensures our prediction is reliable. Thus, while $u(x)$ does not directly estimate $\mathrm{Var}[r|x]$, it provides a *conservative upper bound* on how much our reward prediction might be wrong—which is precisely what UCB-style exploration requires.

Informal Regret Intuition. Let $\epsilon_t = |f_\theta(x_t) - f^*(x_t)|$ denote the neural network's prediction error at round t. Under Lipschitz smoothness, we expect ϵ_t to decrease as the training set grows and covers more of the feature space. The UCB exploration bonus $\alpha \cdot u(x)$ encourages visiting under-explored regions, which:

1. Provides training data to reduce ϵ_t in those regions (improving exploitation);
2. Ensures we do not miss high-reward arms in unexplored areas (avoiding linear regret).

While we do not prove a formal regret bound, this intuition explains why the algorithm converges to good performance in practice.

B.3. Comparison with Neural Bandits

Our approach differs fundamentally from NeuralUCB (Zhou et al., 2020) and Neural-LinUCB in how uncertainty is quantified.

NeuralUCB computes uncertainty using the gradient of the neural network's output:

$$u_{\text{NeuralUCB}}(x) = \sqrt{g(x)^\top Z^{-1} g(x)}, \quad g(x) = \nabla_{\theta_L} f_\theta(x) \tag{20}$$

where $Z = \lambda I + \sum_t g(x_t) g(x_t)^\top$. This approach has formal regret guarantees under the neural tangent kernel (NTK) regime, but requires: (1) computing and storing gradients for each sample, (2) maintaining a $p \times p$ matrix where p is the last-layer dimension, and (3) assuming the network operates in the NTK regime (wide networks, small learning rates).

Our Hybrid Approach uses the original feature space for uncertainty:

$$u_{\text{Hybrid}}(x) = \sqrt{x^\top A^{-1} x}, \quad A = \lambda I + \sum_t x_t x_t^\top \tag{21}$$

Trade-offs. Our design sacrifices theoretical guarantees for practical benefits:

- **Computational efficiency**: No gradient computation or storage; the precision matrix A is updated with simple outer products.
- **Training flexibility**: The neural network can be trained with any optimizer, learning rate, or architecture without affecting uncertainty estimates.
- **Stability**: Linear UCB uncertainty is well-behaved regardless of neural network training dynamics, avoiding gradient explosion or vanishing issues.
- **Interpretability**: The uncertainty directly reflects feature-space coverage, making exploration behavior predictable and tunable via the single parameter α.

The cost is that our uncertainty estimate may be less accurate when the reward function is highly non-linear in the original feature space. However, our empirical results (Table 13) show that NeuralUCB underperforms our hybrid approach on this task, suggesting that the stability benefits outweigh the theoretical advantages of gradient-based uncertainty for workflow orchestration.

B.4. Sample Complexity

The sample efficiency of CB-Orchestrator stems from two factors:

Finite Arm Set. Unlike continuous-armed bandits, our workflow pool $\mathcal{W}$ contains K discrete arms. This reduces the exploration requirement from covering a continuous space to evaluating K discrete options. With n queries and K workflows, the total number of (query, workflow) combinations is nK, but the smoothness assumption (Assumption 3.2) allows generalization across similar queries, effectively reducing the sample complexity.

Embedding-Based Generalization. The joint embedding $x(q, w) = [\phi(q); \phi(w)]$ enables transfer learning across queries. Observing the reward for (q_1, w) provides information about (q_2, w) when $\phi(q_1) \approx \phi(q_2)$. This is formalized by the precision matrix A: after observing x_1, the uncertainty $u(x_2)$ decreases for all x_2 correlated with x_1.

Under the realizability assumption (Assumption 3.3) and in an idealized setting where the effective dimension $d \ll n$, the sample requirement to achieve ϵ-optimal selection can scale like $\tilde{O}(dK/\epsilon^2)$ rather than $\tilde{O}(nK/\epsilon^2)$. We use this as intuition for why embedding-based generalization can reduce the required number of workflow evaluations, which is consistent with the cost reduction observed in our experiments (Figure 2).

B.5. Computational Complexity and Scalability

We analyze the computational costs of our Hybrid Neural-Linear UCB and discuss scalability to larger settings.

Current Setting. With embedding dimension $2d = 5120$, the precision matrix $A \in \mathbb{R}^{5120\times5120}$ requires $5120^2 \times 4 \approx$ 100MB (float32). We avoid explicit matrix inversion by using Cholesky decomposition: decompose $A = LL^\top$, then solve $Ly = x$ for y, giving $u(x) = \|y\|_2$. This reduces per-query complexity from $O(d^3)$ (matrix inversion) to $O(d^2)$ (triangular solve). The Cholesky factor L is recomputed once per batch after updating A, amortizing the $O(d^3)$ decomposition cost across B queries. Table 11 summarizes the per-operation costs.

Table 11. Computational complexity of key operations.

Operation	Time	Frequency
Matrix update: $A \leftarrow A + xx^\top$	$O(d^2)$	Per sample
Cholesky decomposition: $A = LL^\top$	$O(d^3)$	Per batch
Uncertainty: $u(x) = \sqrt{x^\top A^{-1} x}$	$O(d^2)$	Per (query, workflow)
Neural network forward pass	$O(d \cdot h)$	Per (query, workflow)

In our experiments ($d = 2560$, $K \approx 30$, $T \approx 1200$, $B = 25$), the Cholesky decomposition takes ∼0.5 seconds and is performed $T/B \approx 48$ times. Per-round UCB computation for K arms takes ∼10ms. These costs are negligible compared to workflow execution (seconds to minutes per query).

Scaling Analysis. We consider two scaling dimensions:

(1) Larger workflow pools ($K \to$ hundreds). The precision matrix A is independent of K—it depends only on embedding dimension d. Larger K increases per-round UCB computation linearly ($O(K \cdot d^2)$), but this remains fast: even with $K = 500$, computing UCB scores for all arms takes ∼150ms, still negligible compared to LLM inference.

(2) Higher embedding dimensions ($d \to$ larger). Memory scales as $O(d^2)$ and Cholesky decomposition as $O(d^3)$. However, for practical workflow orchestration, modern embedding models (e.g., OpenAI, Cohere) typically use dimensions in the range of 1,024–8,192, well within our current implementation's efficient operating regime. Our experiments with $d = 2560$ demonstrate that this dimension provides sufficient representational capacity for accurate reward prediction while maintaining computational tractability.

C. Additional Results

C.1. LLM Backbone Scalability Results

Table 12 presents the scalability results using the Llama-3.1-70B backbone.

Table 12. Scalability analysis using Llama-3.1-70B as the backbone LLM. We compare CB-Orchestrator against Vanilla and MaAS baselines.

Method	MATH	HumanEval	Avg.
Vanilla	31.93	80.06	56.00
MaAS	42.97	**85.26**	64.12
CB-Orchestrator	**44.36**	84.12	**64.24**

C.2. Ablation Study Results

Table 13 provides the detailed numerical results for the ablation study presented in Figure 3. We also include results for the duplicate sampling setting (w/ dup) on MATH, where the same (query, workflow) pairs can be evaluated multiple times during training.

Table 13. Ablation results on MATH and MBPP benchmarks. We compare different bandit selection strategies and evaluate the impact of the evolution process. The (w/ dup) variants allow duplicate (query, workflow) pair sampling during training.

Method	MATH	MBPP	Avg.
Framework Variants			
PureRandom	47.99	81.64	64.81
WithoutEvo	51.89	**84.17**	68.03
Exploration Strategy Variants			
RandomNN	52.59	82.34	67.47
GreedyNN	52.63	83.93	68.28
LinUCB	52.59	82.11	67.35
NeuralUCB	50.41	82.05	66.23
Ours	53.87	84.05	**68.96**
Duplicate Sampling Variants (MATH only)			
GreedyNN (w/ dup)	51.89	–	–
Ours (w/ dup)	**54.03**	–	–

C.3. GAIA Benchmark Results

Table 14 reports the results on the GAIA benchmark, categorized into three levels: L1 (fundamental reasoning with ≤ 5 steps or 1 tool), L2 (multi-step orchestration with diverse tools), and L3 (long-horizon tasks requiring arbitrary tool-use).

D. Empirical Validation of Assumptions

We validate the three assumptions underlying CB-Orchestrator through experiments on the MATH dataset (486 test problems, 68 workflows, and $T = 1200$ bandit training rounds).

D.1. Assumption 3.1: Semantic Embedding

We verify that the embedding function ϕ preserves semantic similarity through systematic analysis across all pairs, not just cherry-picked examples.

Query Embeddings. We compute pairwise L2 distances for all 486 MATH problems (117,855 pairs). The distance distribution spans $[0.32, 1.42]$ with mean 1.22. To systematically validate semantic preservation, we analyze the *top-5 closest* and *top-5 farthest* pairs—these represent the strongest test cases for the assumption.

The results show a clear pattern: **all 5 closest pairs** share problem templates or reasoning patterns, while **all 5 farthest pairs** span entirely different mathematical domains. This is not cherry-picking—we report *every* pair at the distribution extremes.

Workflow Embeddings. We compute pairwise distances for all 68 workflows (2,278 pairs). The distance range is

Table 14. The performance of CB-Orchestrator compared to baseline methods on the GAIA benchmark. Backbone is GPT-4o-mini for all experiments.

Method	**L1**	**L2**	**L3**	**Avg.**
Basic Prompting				
GPT-4o-mini	7.5	4.4	0	4.7
GPT-4	9.7	1.9	2.1	4.1
Fixed Multi-Agent Systems				
TapeAgent	23.7	14.5	10.2	16.6
Sibyl	21.5	15.7	4.1	15.6
Autonomous Multi-agent Systems				
AutoAgents	16.1	0	0	5.2
GPTSwarm	23.7	16.4	2.0	16.3
AgentSquare	22.6	15.7	6.3	16.3
AFlow	10.8	8.8	4.1	8.0
MaAS	25.9	22.0	6.3	20.7
CB-Orchestrator	28.6	21.1	5.3	**21.0**

Table 15. All top-5 closest query pairs. Every pair shares problem structure or reasoning pattern.

Rank	**Dist.**	**Shared Structure**
1	0.32	Both ask "Bekah has [N] brass digits... how many distinct numbers" (identical template, different N)
2	0.44	Both ask circular seating of Senate committee with party constraints
3	0.45	Both ask "put 4 balls in 3 boxes" with distinguishability variants
4	0.46	Both ask area of isosceles right triangle given altitude to hypotenuse
5	0.47	Both show identical triangle diagrams (Asymptote code), ask for x

$[0.13, 0.93]$ with mean 0.51. We analyze the top-3 closest and top-3 farthest pairs:

The pattern is consistent: close workflow pairs share the same operator composition with minor variations (e.g., number of solutions, loop vs. sequential calls), while far pairs use fundamentally different strategies (e.g., with vs. without ScEnsemble, different operator orderings).

D.2. Assumption 3.2: Reward Smoothness

We verify that semantically similar workflows yield similar rewards on the same query using two methods.

Method 1: KNN Reward Prediction. For each (query, workflow) pair in the training data, we predict its reward using the k nearest workflows (by embedding distance) that were also evaluated on the same query. If the smoothness assumption holds, nearby workflows should have correlated rewards.

The KNN predictor achieves 84.75% accuracy—significantly higher than the 51% random baseline—demonstrating that nearby workflows indeed yield similar rewards. The strong correlation ($r = 0.73$, $p < 10^{-6}$) further confirms that embedding distance is predictive of reward similarity.

Method 2: Success Workflow Clustering. We analyze whether successful workflows on a given query cluster together in embedding space. For each query with both successful and failed workflow trials, we compute:

- Mean distance between successful workflows (intra-success)
- Mean distance between successful and failed workflows (cross-distance)

Successful workflows are on average 20% closer to each other than to failed workflows, indicating that success is not randomly distributed but clusters in embedding space—consistent with reward smoothness.

Table 16. All top-5 farthest query pairs. Every pair involves completely different mathematical domains.

Rank	Dist.	Query A Domain	Query B Domain
1	1.42	Parametric line equations	Circular permutations
2	1.42	Triangle angle trisectors	Counting odd 5-digit numbers
3	1.41	Simplify $(\sqrt{6}+\sqrt{24})^2$	Probability of consecutive seats
4	1.41	3-digit palindrome multiples of 3	Simplify radical expression
5	1.40	Polygon perimeter geometry	Product divisibility probability

Table 17. Workflow pairs at distance extremes. Close pairs share operator sequences; far pairs differ fundamentally.

Type	Dist.	Workflow A	Workflow B
Close	0.13	Custom→Programmer(1×)→ScEnsemble→Format	Custom→Programmer(3×)→ScEnsemble→Format
Close	0.13	Custom→Programmer(3× loop)→ScEnsemble→Format	Custom→Programmer(3×)→ScEnsemble→Format
Close	0.14	Custom→Programmer(3×)→ScEnsemble→Format	Custom→Custom(3×)→ScEnsemble→Format
Far	0.93	Custom→ScEnsemble→Programmer	Programmer only (hardcoded)
Far	0.93	Custom→Programmer(3×)→ScEnsemble→ Programmer→Format	Custom→ScEnsemble→Programmer
Far	0.93	Custom→Programmer(3×)→ScEnsemble→Format	Custom→Custom→Custom→Programmer (no ensemble)

D.3. Assumption 3.3: Realizability

The realizability assumption states that there exists a function f^* such that $\mathbb{E}[r|q,w] = f^*(x(q,w))$. While we cannot directly verify the existence of such a function, we provide indirect evidence through the learning dynamics:

- **Learning Improvement**: During training, the success rate improves from 44% (exploration phase, rounds 1–100) to 49.4% (exploitation phase, rounds 101–1200), indicating that the neural network learns to predict rewards better than random selection.
- **KNN Prediction Success**: The strong performance of KNN prediction (84.75% accuracy) suggests that the expected reward is indeed a smooth function of the embedding, which is a weaker but related condition.
- **Final Model Performance**: The trained model achieves state-of-the-art results on all benchmarks (Table 1), demonstrating that the learned function f_θ successfully approximates the true reward structure.

Together, these results suggest that while the assumptions may not hold perfectly, they hold sufficiently well in practice to enable effective learning and generalization.

E. Case Study: Adaptive Workflow Selection

To demonstrate how CB-Orchestrator adaptively selects workflows based on query characteristics, we analyze the selection patterns on the MATH test set (486 problems) and present concrete examples where different problem types activate different workflows.

E.1. Workflow Selection Distribution

During inference on the MATH test set, CB-Orchestrator selects from 68 available workflows, but primarily utilizes 6 workflows with distinct problem-solving strategies:

E.2. Case Study Examples

We present three representative examples where CB-Orchestrator selects different workflows for different problem types.

Table 18. KNN reward prediction results ($k = 3$). High accuracy and correlation confirm reward smoothness.

Metric	Value	Interpretation
KNN Prediction Accuracy	84.75%	Correct prediction of success/failure
Random Baseline (majority)	51.08%	Baseline without smoothness
Pearson Correlation (r)	0.73	Strong positive correlation
p-value	$< 10^{-6}$	Statistically significant

Table 19. Success workflow clustering analysis. Successful workflows cluster together.

Distance Type	Mean Distance	Sample Size
Success–Success (intra)	0.40	2,138 pairs
Success–Failure (cross)	0.50	1,066 pairs

E.2.1. Example 2: Geometry Problem → Workflow 7

Problem: *In the diagram, the three concentric circles have radii of 4, 6, and 7. Three regions are labeled X, Y, or Z. Of these three regions, what is the difference between the area of the region with the greatest area and the area of the region with the smallest area?*

Selected Workflow: workflow_7 (Decomposition with validation)

Workflow Strategy:

Workflow 7: Key Logic

```
# Step 1: Decompose the problem
decomposition = await self.custom(input=problem,
    instruction="Decompose the problem into smaller, "
                "manageable parts")

# Step 2: Generate solution with Programmer
response = await self.programmer(problem=problem,
                                 analysis=decomposition['response'])

# Step 3: Validate calculations (unique to this workflow)
validated_response = await self.custom(
    input=f"Problem: {problem}\n\nCalculated: {final_solution}",
    instruction="Validate the calculation logic based on "
                "the problem and solution.")
```

Why This Selection: Geometry problems involving area calculations require careful verification of formulas and arithmetic. The validation step in workflow_7 specifically addresses this need by double-checking the calculation logic, reducing errors in multi-step area computations.

Result: Correctly predicted $\boxed{7\pi}$ (difference between annulus areas).

E.2.2. Example 3: Combinatorics Problem → Workflow 102

Problem: *Given a right triangle whose side lengths are all integer multiples of 8, how many units are in the smallest possible perimeter of such a triangle?*

Selected Workflow: workflow_102 (Probability decomposition + Programmer)

Workflow Strategy:

Table 20. Workflow usage distribution on MATH test set (486 problems).

Workflow	Count	%	Primary Strategy
workflow_102	168	34.6%	Probability decomposition + Programmer
workflow_119	111	22.8%	Symmetry-aware analysis + ScEnsemble
workflow_50	72	14.8%	Detailed analysis + Code generation
workflow_7	71	14.6%	Decomposition + Validation step
workflow_99	58	11.9%	Core decomposition + ScEnsemble
workflow_105	6	1.2%	Multi-solution generation

Workflow 102: Key Logic

```
# Step 1: Decompose with structured analysis
DECOMPOSE_PROMPT = "Break down the following probability "
    "problem into clear steps and calculations: "
decomposition = await self.custom(input=problem,
                                  instruction=DECOMPOSE_PROMPT)

# Step 2: Generate systematic solution approach
SOLUTION_PROMPT = "From the analysis provided, establish "
    "the systematic approach to determining..."
solution = await self.custom(
    input=problem + '\n' + decomposition['response'],
    instruction=SOLUTION_PROMPT)

# Step 3: Code generation for computation
programming_result = await self.programmer(
    problem=problem, analysis=decomposition['response'])
```

Why This Selection: This problem requires systematic enumeration of Pythagorean triples scaled by 8. Workflow_102's emphasis on "probability problem" decomposition (which generalizes to counting/enumeration problems) combined with code execution is well-suited for finding the minimum perimeter through systematic search.

Result: Correctly predicted $\boxed{96}$ (perimeter of 24-32-40 triangle).

E.3. Analysis: Query-Workflow Matching

The examples demonstrate that CB-Orchestrator learns meaningful query-workflow associations:

- **Number theory** problems (prime factorization, divisibility) → Workflows with systematic decomposition and ensemble verification
- **Geometry** problems (area, angle calculations) → Workflows with explicit validation steps
- **Counting/Combinatorics** problems (enumeration, optimization) → Workflows emphasizing code-based systematic search

This adaptive selection emerges naturally from the contextual bandit training, where the neural network learns to predict which workflow strategies succeed for which problem types based on the semantic similarity captured in the embeddings.