AdaptFlow: Adaptive Workflow Optimization via Meta-Learning

Anonymous ACL submission

Abstract

Recent advances in large language models (LLMs) have sparked growing interest in agentic workflows—structured sequences of LLM invocations designed to solve complex tasks. However, existing approaches often rely on static templates or manually designed workflows, which limit adaptability to diverse tasks and hinder scalability. We propose AdaptFlow, a natural language-based meta-learning framework inspired by model-agnostic meta-learning (MAML). AdaptFlow uses a bi-level optimization process: the inner loop performs task-specific adaptation via LLM-generated feedback, while the outer loop consolidates these refinements into a shared, generalizable initialization. Evaluated across question answering, code generation, and mathematical reasoning benchmarks, AdaptFlow consistently outperforms both manually crafted and automatically searched baselines, achieving state-of-the-art results with strong generalization across tasks and models. The source code and data are available at `https://anonymous.4open.science/r/AdaptFlow-FD17/`.

1 Introduction

In recent years, Large Language Models (LLMs) (Achiam et al., 2023; Guo et al., 2025) have rapidly advanced, achieving state-of-the-art results in tasks such as question answering (Rajpurkar et al., 2016; Yang et al., 2018), code synthesis (Chen et al., 2021; Nijkamp et al., 2023), and multi-turn dialogue (Zhang et al., 2020; Bai et al., 2022). With increasing model scale and training data, LLMs have also shown strong potential in dynamic reasoning and decision support (Shinn et al., 2023; Wei et al., 2022; Yao et al., 2023). In parallel, the concept of *Agentic Workflow* has gained traction—simulating agents with autonomous planning and execution capabilities to tackle complex tasks. Such frameworks perform particularly well in settings requiring multi-step reasoning (Yao et al., 2023; Creswell and Shanahan, 2023), long-term planning (Liu et al., 2023; Zhou et al., 2024b), and tool use (Schick et al., 2023; Qin et al., 2023).

While effective, manually designing agentic workflows is time-consuming and difficult to generalize across diverse tasks. To address this, recent work has explored automated workflow construction through prompt optimization, hyperparameter tuning, and structural search (Khattab et al., 2023; Chen et al., 2023; Li et al., 2024b; Liu et al., 2024; Song et al., 2024; Zhang et al., 2024a; Wang et al., 2025). However, many of these approaches (Liu et al., 2024; Zhang et al., 2024a) rely on graph-based representations with limited support for conditional states, which restricts the expressivity of the search space and hinders their applicability to complex tasks.

Recent frameworks such as ADAS (Hu et al., 2024) and AFLOW (Zhang et al., 2024b) adopt programmatic workflow representations to enable robust and flexible search. However, as noted by (Wang et al., 2025), these methods typically generate a single static workflow for the entire task set, limiting their ability to generalize across heterogeneous datasets with diverse problem types. In addition, ADAS performs coarse-grained workflow updates, resulting in redundant context accumulation and growing complexity that hinders convergence. AFLOW alleviates some of these issues using Monte Carlo Tree Search, but its reliance on discrete updates and early pruning can restrict the exploration of more expressive workflow candidates. These limitations underscore two challenges simultaneously: ***C1*. How to adaptively construct effective workflows for datasets containing diverse problems? *C2*. How to ensure convergent optimization in code search spaces?**

To tackle these challenges, we propose **AdaptFlow**—a meta-optimization framework that integrates principles from Model-Agnostic Meta-

Learning (Finn et al., 2017) into agentic workflow optimization. Unlike prior approaches that generate static workflows for entire task sets, AdaptFlow learns a shared workflow initialization that can rapidly adapt to diverse subtasks through updates guided by LLMs. It employs a *bi-level optimization scheme*, where the *inner loop* explores task-specific refinements via LLM-generated feedback, and the *outer loop* aggregates these improvements into a more generalizable workflow. This approach enables both effective subtask adaptation (addressing **C1**) and stable convergence in non-differentiable code spaces (addressing **C2**), yielding a scalable solution for general-purpose workflow construction.

Our key contributions are summarized as follows:

- We draw a novel analogy between neural network training and automated workflow optimization, which motivates AdaptFlow design.
- We propose AdaptFlow, a natural language-based meta-learning framework that incorporates MAML to enable rapid subtask adaptation.
- Experiments on benchmarks in question answering, code generation, and mathematical reasoning show that AdaptFlow outperforms both manual workflows and prior baselines, achieving state-of-the-art results with strong model-agnostic generalization.

2 Related Work

2.1 Agentic Workflow

Agentic workflows provide a structured alternative to autonomous agents for deploying large language models (LLMs). Instead of learning through environment interaction (Zhuge et al., 2023; Hong et al., 2024b), they execute static or semi-static sequences inspired by human reasoning (Zhang et al., 2024b), offering better interpretability and modularity.

Workflows can be general-purpose—featuring reusable patterns like chain-of-thought prompting, self-refinement, or role decomposition (Wei et al., 2022; Shinn et al., 2023)—or domain-specific, tailored for areas such as code generation (Hong et al., 2024c; Ridnik et al., 2024; Zhao et al., 2024), data analysis (Xie et al., 2024; Ye et al., 2024; Li et al., 2024a), mathematics (Zhong et al., 2024; Xu et al., 2024), and complex QA (Nori et al., 2023; Zhou et al., 2024a). While effective, manually designed workflows require significant human effort and lack adaptability, motivating automated optimization.

2.2 Agentic Workflow Optimization

Recent advances (Hu et al., 2024; Zhang et al., 2024b; Wang et al., 2025; Li et al., 2024b; Chen et al., 2023; Song et al., 2024; Hong et al., 2024c) have explored automating agentic workflows to improve LLM performance. Some methods focus on optimizing prompts or parameters within fixed workflows (Fernando et al., 2023; Guo et al., 2023; Khattab et al., 2023; Saad-Falcon et al., 2024), improving reasoning without altering the execution structure. In contrast, we optimize workflow structures directly, enabling broader adaptation across tasks.

Other approaches search over code-based workflows. ADAS (Hu et al., 2024) refines linear traces of executable code, while AFLOW (Zhang et al., 2024b) introduces compositional abstractions with MCTS. ScoreFlow (Wang et al., 2025) frames workflow generation as supervised prediction. However, these methods often produce static workflows and lack task-level adaptability. Our method, AdaptFlow, differs by performing bi-level meta-learning: it adapts workflows via LLM feedback at the subtask level and consolidates them into a generalizable initialization, supporting fast adaptation and robust generalization.

3 Preliminaries

3.1 Problem Formulation

The goal of automated agentic workflow optimization is to discover effective compositions of modular components—such as prompt templates, tool invocations, control logic, and reflection routines—that can guide large language models (LLMs) to solve complex tasks across diverse domains.

We formalize the agentic workflow design problem as a triplet $(\mathcal{S}, \mathcal{J}, \mathcal{A})$, where:

- $\mathcal{S}$ denotes the *search space*, encompassing all candidate workflows;
- $\mathcal{J} : \mathcal{S} \times \mathcal{T} \to \mathbb{R}$ is the *objective function* that quantifies the quality or utility of a workflow $\mathcal{W} \in \mathcal{S}$ when applied to a specific task $\mathcal{T}$;
- $\mathcal{A}$ represents the *search algorithm*, which explores $\mathcal{S}$ and generates candidate workflows guided by feedback from $\mathcal{J}$.

Given a task $\mathcal{T} \sim \mathcal{P}(\mathcal{T})$, the agent seeks to identify an optimal workflow through a task-conditioned search process:

$$\mathcal{W} = \mathcal{A}(\mathcal{S}, \mathcal{J}, \mathcal{T}), \tag{1}$$

$$\mathcal{W}^\star = \underset{\mathcal{W} \in \mathcal{S}}{\arg\max}\ \mathbb{E}_{\mathcal{T} \sim \mathcal{P}(\mathcal{T})} \left[\mathcal{J}(\mathcal{W}, \mathcal{T}) \right]. \tag{2}$$

Similar to prior efforts such as (Hu et al., 2024), our method adopts programmatic representations to define the workflow search space.

3.2 Analogy: From Supervised Learning to Agentic Workflow Optimization

In traditional supervised learning, a model learns a parameterized function f_θ by minimizing the expected loss over labeled data $(x, y) \sim \mathcal{D}$:

$$\theta^\star = \underset{\theta}{\arg\min}\ \mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\mathcal{L}(f_\theta(x), y) \right], \tag{3}$$

$$\theta \leftarrow \theta - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(f_\theta(x_i), y_i), \tag{4}$$

where η is the learning rate and $\{(x_i, y_i)\}_{i=1}^N$ is a mini-batch of training examples. This process relies on differentiable loss functions and explicit ground-truth supervision, enabling gradient-based parameter updates in continuous space.

Analogously, agentic workflow optimization operates in a symbolic structure space defined over executable code (e.g., (Hu et al., 2024)). Given a task $\mathcal{T}$, the system executes a workflow $\mathcal{W}$ and obtains a task-level utility score from the objective function $\mathcal{J}(\mathcal{W}, \mathcal{T})$. The goal is to discover a workflow that maximizes the expected utility across a distribution of tasks:

$$\mathcal{W}^\star = \underset{\mathcal{W} \in \mathcal{S}}{\arg\max}\ \mathbb{E}_{\mathcal{T} \sim \mathcal{P}(\mathcal{T})} \left[\mathcal{J}(\mathcal{W}, \mathcal{T}) \right], \tag{5}$$

$$\mathcal{W} \leftarrow \mathcal{U}_1 \left(\mathcal{W}, \tilde{\nabla} \mathcal{J}(\mathcal{W}, \mathcal{T}) \right). \tag{6}$$

Here, $\mathcal{J}(\mathcal{W}, \mathcal{T})$ represents a task-level evaluation of the workflow's performance, expressed in natural language. This textual assessment serves as a form of *textual loss*, indicating the correctness of the workflow output. Based on this, $\tilde{\nabla}\mathcal{J}$ denotes a *textual gradient*—natural language feedback generated by an LLM that suggests possible improvements, identifies failure modes, or proposes structural alternatives. The update operator $\mathcal{U}_1$ then applies this feedback to revise the workflow in the code space, enabling updates in a discrete, non-differentiable setting. This process mirrors the gradient-based update in conventional learning, with a detailed analogy illustrated in Figure 1.

Unlike standard gradient descent, ADAS enables interpretable, feedback-driven optimization over program structures, thereby generalizing the concept of learning beyond parameter tuning.

3.3 Model-Agnostic Meta-Learning

Model-Agnostic Meta-Learning (MAML) (Finn et al., 2017) learns a model initialization that enables rapid adaptation to new tasks using only a few gradient steps. The core idea is to train the model not just to perform well on a set of tasks, but to be easily fine-tuned for any new task drawn from the same distribution.

Given a task distribution $\mathcal{T}$, each task $\mathcal{T}_i \sim \mathcal{T}$ is associated with a loss $\mathcal{L}_{\mathcal{T}_i}(\theta)$. MAML performs a bi-level optimization:

$$\theta_i' \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(\theta), \tag{7}$$

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim \mathcal{T}} \mathcal{L}_{\mathcal{T}_i}(\theta_i'). \tag{8}$$

In the inner loop, the model performs gradient descent on a given task to obtain adapted parameters θ_i'. In the outer loop, the original initialization θ is updated using the post-adaptation losses across multiple tasks. This procedure leverages second-order gradients and enables generalization to unseen tasks with minimal fine-tuning.

4 Methodology

4.1 Overview

We present **AdaptFlow**, a meta-optimization framework that integrates ideas from Model-Agnostic Meta-Learning (Finn et al., 2017) into the setting of Automated Design of Agentic Systems (Hu et al., 2024). As shown in Figure 1, AdaptFlow optimizes a single code-based workflow that can rapidly adapt to different subtasks, addressing (**C1**) by enabling structural reuse across related tasks. It follows a bi-level learning scheme: the inner loop performs **exploration**, adapting workflows using LLM-generated feedback within each subtask, while the outer loop performs **exploitation** by consolidating improvements into a shared initialization. This setup supports learning in non-differentiable code spaces without ground-truth supervision, directly addressing (**C2**), and leads to stable convergence and broad generalization. The complete AdaptFlow algorithm is formalized in

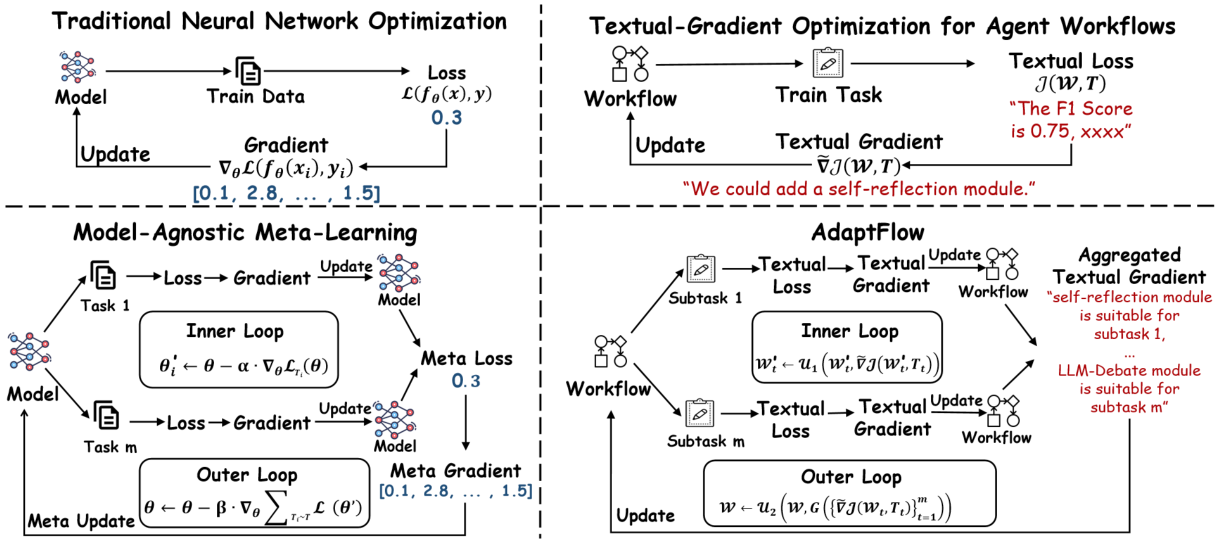

Figure 1: An analogy between Neural Network Optimization and Workflow Optimization, as well as between MAML and AdaptFlow.

Algorithm 1, and its procedural overview is visually depicted in Figure 2 to complement the formal description.

4.2 Task Clustering

Many tasks exhibit high internal diversity, making it difficult to optimize a single workflow across all instances. To address this, we first partition the training set $\mathcal{T}_{\text{train}}$ into m semantically coherent subtasks $\mathcal{T}_1, \ldots, \mathcal{T}_m$ using K-Means (MacQueen, 1967) clustering over instruction embeddings. The embeddings are obtained from the `all-MiniLM-L6-v2` model (Reimers and Gurevych, 2019). This decomposition enables subtask-specific workflow optimization and promotes more stable and effective learning.

4.3 Bi-Level Workflow Optimization

Inner Loop (Exploration) For each subtask $\mathcal{T}_t$, the workflow $\mathcal{W}_t'$ is iteratively refined using LLM-generated textual feedback. At each step, we evaluate the current utility $\mathcal{J}(\mathcal{W}_t', \mathcal{T}_t)$ and apply the symbolic update:

$$\mathcal{W}_t' \leftarrow \mathcal{U}_1\left(\mathcal{W}_t', \tilde{\nabla}\mathcal{J}(\mathcal{W}_t', \mathcal{T}_t)\right). \tag{9}$$

To ensure stable and meaningful exploration, we define a binary continuation signal $\delta_t \in 0, 1$ as:

$$\delta_t = \mathbb{I}\left[\mathcal{J}(\mathcal{W}_t, \mathcal{T}_t) - \mathcal{J}(\mathcal{W}_t', \mathcal{T}_t) > \epsilon\right], \tag{10}$$

where $\mathcal{W}_t$ denotes the best workflow found so far. Here, $\mathcal{J}$ evaluates a workflow's performance on task $\mathcal{T}_t$ via textual assessment, serving as a form of task-level *textual loss*. Based on this evaluation, the LLM generates a *textual gradient* $\tilde{\nabla}\mathcal{J}$ that reflects potential improvements or corrections. The update operator $\mathcal{U}_1$ applies this feedback to revise the workflow $\mathcal{W}_t'$ in the code space. The inner loop continues only if $\delta_t = 1$, indicating that the update yields a non-trivial gain. This continuation signal acts as a local convergence criterion, mitigating instability from long-context accumulation and ensuring effective symbolic refinement. The prompt design for $\mathcal{U}_1$ is detailed in Section A.3.1.

Outer Loop (Exploitation) After inner-loop optimization across all subtasks, we aggregate the resulting feedback to update the global workflow. Each $\tilde{\nabla}\mathcal{J}(\mathcal{W}_t, \mathcal{T}_t)$ denotes a textual gradient—natural language feedback from the LLM that suggests workflow improvements based on subtask performance. The aggregation function G merges these gradients into a unified signal, which is then applied via the update operator $\mathcal{U}_2$:

$$\mathcal{W} \leftarrow \mathcal{U}_2\left(\mathcal{W}, G\left(\left\{\tilde{\nabla}\mathcal{J}(\mathcal{W}_t, \mathcal{T}_t)\right\}_{t=1}^{m}\right)\right). \tag{11}$$

This meta-level update integrates subtask-specific insights into a generalizable workflow. The prompt design for $\mathcal{U}_2$ is provided in Section A.3.2.

To further improve robustness, we apply a **reflection** step after the update. The updated workflow is re-executed on each subtask to identify remaining failure cases. The agent then generates refinement suggestions, which are used to perform a secondary symbolic update. This reflection-enhanced outer

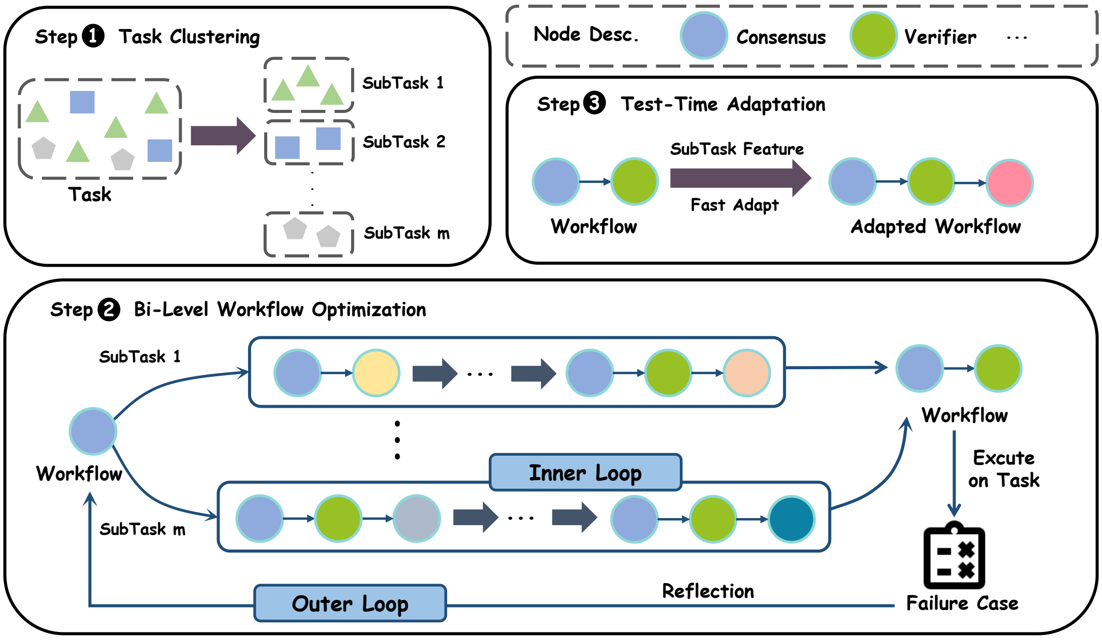

Figure 2: Illustration of the AdaptFlow framework, consisting of three stages. (1) **Task Clustering**: training tasks are grouped into semantically coherent subtasks. (2) **Bi-Level Workflow Optimization**: a bi-level optimization process is applied—inner loop explores workflow variants using LLM-generated feedback; outer loop aggregates updates into a generalizable initialization. (3) **Test-Time Adaptation**: the learned workflow is adapted to unseen tasks based on subtask-level descriptions generated from input questions. The detailed mechanism of inner and outer updates is shown in Figure 1.

loop helps address blind spots and improve generalization.

4.4 Test-Time Adaptation

To evaluate generalization, we apply the learned initialization $\mathcal{W}$ to a set of unseen test tasks $\mathcal{T}$test. Following the same procedure as in training, we partition $\mathcal{T}$test into n subtasks $\mathcal{T}'_1, \ldots, \mathcal{T}'_n$ using instruction-level clustering.

For each subtask $\mathcal{T}'_t$, we randomly sample a subset $\tilde{\mathcal{T}}'_t \subset \mathcal{T}'_t$ and prompt a language model to generate a high-level description $\mathcal{F}(\tilde{\mathcal{T}}'_t)$ based solely on the input questions from the sampled tasks—without access to answers or solutions. This representation captures the subtask's semantic intent and guides adaptation.

We then apply a symbolic update operator $\mathcal{U}_3$ to specialize the global workflow based on this subtask description:

$$\mathcal{W} \leftarrow \mathcal{U}_3 \left(\mathcal{W}, \mathcal{F}(\tilde{\mathcal{T}}'_t) \right). \tag{12}$$

Here, $\mathcal{U}_3$ modifies the workflow to better suit the characteristics of the new subtask using natural language cues. The prompt design for $\mathcal{U}_3$ is detailed in Section A.3.4. The resulting specialized workflow $\mathcal{W}$ is then evaluated on the full subtask $\mathcal{T}'_t$, enabling effective adaptation to previously unseen task distributions.

5 Experiment Setup

Datasets We evaluate our method on eight public datasets across three domains: question answering, code generation, and mathematical reasoning. For HUMANEVAL (Chen et al., 2021) and MBPP (Austin et al., 2021), we use the full datasets. Following AFLOW (Zhang et al., 2024b), we sample 1,319 examples from the GSM8K test split (Cobbe et al., 2021). For MATH (Hendrycks et al., 2021), we follow (Hong et al., 2024a) and select level-5 problems from four categories: Combinatorics and Probability, Number Theory, Prealgebra, and Pre-calculus. We also include two advanced math benchmarks: AIME (OpenAI, 2023) and OLYMPIADBENCH (Zhu et al., 2024). For DROP (Dua et al., 2019) and HOTPOTQA (Yang et al., 2018), we follow prior work (Shinn et al., 2023; Zhang et al., 2024b; Wang et al., 2025) and randomly sample 1,000 instances each. All datasets are split into validation and test sets with a 1:4 ratio. See Table 6 for full statistics.

Algorithm 1: AdaptFlow Algorithm

Input: train tasks $\mathcal{T}_{\text{train}}$, test tasks $\mathcal{T}_{\text{test}}$, inner iterations n_{inner}, outer iterations n_{outer}

Cluster $\mathcal{T}_{\text{train}}$ into m subtasks $\{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$;
Initialize global workflow $\mathcal{W} = \mathcal{W}_1 = \ldots = \mathcal{W}_m$;
`// Outer loop`
for $i \leftarrow 1$ **to** n_{outer} **do**
  foreach $\mathcal{T}_t \in \{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$ **do**
    Initialize $\mathcal{W}'_t \leftarrow \mathcal{W}$; $j \leftarrow 0$;
    `// Inner loop`
    while $\mathcal{J}(\mathcal{W}'_t, \mathcal{T}_t) < \mathcal{J}(\mathcal{W}_t, \mathcal{T}_t) - \epsilon$ ***and*** $j < n_{inner}$ **do**
      Execute $\mathcal{W}'_t$ on $\mathcal{T}_t$, obtain $\tilde{\nabla}\mathcal{J}$;
      $\mathcal{W}'_t \leftarrow \mathcal{U}_1\left(\mathcal{W}'_t, \tilde{\nabla}\mathcal{J}\right)$;
      $j \leftarrow j + 1$;
    end
    $\mathcal{W}_t \leftarrow \mathcal{W}'_t$;
  end
  $\mathcal{W} \leftarrow \mathcal{U}_2\left(\mathcal{W}, G\left(\left\{\tilde{\nabla}\mathcal{J}(\mathcal{W}_t, \mathcal{T}_t)\right\}_{t=1}^{m}\right)\right)$
end
Cluster $\mathcal{T}_{\text{test}}$ into n subtasks $\{\mathcal{T}'_1, \ldots, \mathcal{T}'_n\}$;
foreach $\mathcal{T}'_t$ **do**
  $\mathcal{W}' \leftarrow \mathcal{U}_3\left(\mathcal{W}, \mathcal{T}'_t\right)$;
  Evaluate $\mathcal{W}^*$ on $\mathcal{T}'_t$;
end

Baselines We compare our method against two categories of baselines: manually designed workflows and automatically optimized workflows for large language models (LLMs). **Manual Workflows** include widely used prompting strategies and agent-based methods: Vanilla prompting, Chain-of-Thought (CoT) (Wei et al., 2022), Reflexion (Shinn et al., 2023), LLM Debate (Du et al., 2023), Step-back Abstraction (Zhou et al., 2022), Quality-Diversity (QD) (Wang et al., 2023), and Dynamic Role Assignment (Qian et al., 2023). These approaches are constructed using fixed templates or heuristics without task-specific adaptation. **Automatically Optimized Workflows** are derived through workflow optimization or search. We include ADAS (Hu et al., 2024) and AFLOW (Zhang et al., 2024b), which learn or search for agentic workflow structures in a data-driven manner to improve LLM performance across tasks.

Implementation Details We use a decoupled architecture separating optimization and execution. GPT-4.1 (OpenAI, 2024a) serves as the optimizer, while executors include DeepSeekV2.5 (DeepSeek, 2024), GPT-4o-mini (OpenAI, 2024b), Claude-3.5-Sonnet (Anthropic, 2024), and GPT-4o (OpenAI, 2024c). All models are accessed via public APIs with a fixed temperature of 0.5. The outer loop runs for 3 iterations, and the inner loop allows up to 6 updates per subtask.

Metrics We adopt task-specific evaluation metrics tailored to each dataset category. For mathematics benchmarks, including GSM8K, MATH, AIME, and OLYMPIADBENCH, we use the **Solve Rate**—the proportion of correctly solved problems—as the primary metric. For code generation tasks (HUMANEVAL and MBPP), we report **pass@1**, following the evaluation protocol of Chen et al. (Chen et al., 2021), which measures the correctness of the top-1 generated solution. For question-answering datasets such as HOTPOTQA and DROP, we adopt the **F1 Score** to evaluate the overlap between predicted and ground-truth answers.

6 Results and Analysis

6.1 Main Results

As shown in Table 1, our method delivers consistently strong performance across three distinct domains—question answering, code generation, and mathematics—achieving the highest overall average score of **68.5**. This suggests that our unified framework generalizes well to tasks with varying structures and reasoning demands. In particular, the substantial gains on mathematics benchmarks demonstrate the framework's strength in handling complex symbolic and multi-step reasoning.

These results highlight the advantage of learning workflows in a task-adaptive and optimization-aware manner. Compared to existing baselines, including both manually designed strategies and automatically optimized methods, our approach achieves more balanced improvements across domains, underscoring its robustness and scalability. The consistent lead over ADAS (Hu et al., 2024) and AFLOW (Li et al., 2024b), which operate in a similar code-based search space, further supports the effectiveness of meta-level adaptation in building generalizable agentic workflows.

Method	QA		Coding		MATH				Average
	HOTPOTQA	DROP	HUMANEVAL	MBPP	GSM8K	MATH	AIME	OLYMPIAD	
Vanilla	70.7	79.6	87.0	71.8	92.7	48.2	12.4	25.0	60.9
COT	69.0	78.8	90.8	72.5	91.3	49.9	10.1	26.4	61.1
Reflexion	68.3	79.5	86.3	72.4	92.4	49.3	10.5	25.9	60.6
LLM Debate	68.5	79.3	90.8	73.3	93.8	52.7	13.7	29.8	62.7
Step-back Abstraction	67.9	79.4	87.8	71.9	90.0	47.9	4.8	19.3	58.6
Quality Diversity	69.3	79.7	88.5	72.5	92.3	50.5	9.4	28.8	61.4
Dynamic Assignment	67.9	76.8	89.3	71.5	89.2	50.7	12.7	27.6	60.7
ADAS	64.5	76.6	82.4	53.4	90.8	35.4	10.4	21.2	54.3
AFlow	73.5	80.6	**94.7**	83.4	93.5	56.2	17.4	28.5	65.6
Ours	**73.8**	**82.4**	**94.7**	**84.0**	**94.6**	**61.5**	**22.6**	**34.4**	**68.5**

Table 1: Performance comparison across three domains: question answering, code generation, and mathematics. Best results are shown in **bold**, and second-best results are underlined. In our method, GPT-4.1 is used for workflow refinement, while GPT-4o-mini-0718 is responsible for workflow execution.

6.2 Ablation Study

Ablation on Reflection To evaluate the impact of the reflection module in the outer loop, we conduct an ablation study on the MATH dataset. We use GPT-4.1 for workflow updates and GPT-4o-mini-0718 for workflow execution. In the ablated setting, denoted as `w/o` reflection, we remove the reflection step where the model samples and revises failed cases after the initial outer-loop update. As shown in Table 2, incorporating reflection consistently leads to better performance across iterations, with a final accuracy of 61.5 compared to 60.2 without reflection. This highlights the importance of targeted self-correction in enhancing workflow robustness and adaptability.

Outer Loop Iteration	1	2	3
`w/o` reflection	56.7	58.2	60.2
ours	**57.2**	**58.6**	**61.5**

Table 2: Performance comparison across iterations on the MATH dataset. `w/o` reflection denotes the setting without the reflection component, while **ours** includes it.

Subtask	w/o adaptation	ours
Prealgebra	73.1	**76.4**
Precalculus	20.8	**21.4**
Counting & Probability	61.9	**63.1**
Number Theory	68.3	**73.9**
Overall	58.0	**61.5**

Table 3: Ablation results on math subtasks with and without test-time adaptation. `w/o` adaptation disables test-time adaptation.

Ablation on Test-Time Adaptation To assess the effectiveness of our test-time adaptation strategy, we conduct an ablation study on four mathematical reasoning subtasks: Prealgebra, Precalculus, Counting & Probability, and Number Theory. As shown in Table 3, removing the adaptation module results in a consistent drop in performance across all subtasks. Notably, the largest improvement is observed in Number Theory, where accuracy increases from 68.3 to 73.9, suggesting that adaptation plays a crucial role in handling complex symbolic reasoning. The overall average accuracy improves by 3.5 points, confirming that test-time refinement enhances the generalization of the global workflow to previously unseen problems.

6.3 Convergence Analysis

We analyze the convergence behavior of both inner and outer loops on the MATH dataset, as shown in Figure 3. The inner loop exhibits noticeable fluctuations due to the accumulation of long-context dependencies and the large workflow search space, a challenge also observed in ADAS (Hu et al., 2024). Despite this, our constrained update mechanism helps maintain reasonable performance at each step. In contrast, the outer loop shows steady improvement, as it only aggregates the best-performing workflows from each subtask, leading to more stable and reliable updates at the meta level. These results demonstrate that our method effectively ensures convergence throughout the optimization process, addressing the core challenge of **C2**.

6.4 Model Agnostic Analysis

To assess generality, we evaluate our method on the MATH dataset using four LLMs: GPT-4o-mini, GPT-4o, Claude-3.5-Sonnet, and DeepSeek-V2.5. As shown in Table 4, our method consistently achieves the best performance, demonstrat-

Model	Method							
	Vanilla	COT	Reflexion	LLM debate	Step-back Abstraction	Quality Diversity	Role Assignment	Ours
GPT-4o-mini	48.2	49.9	49.3	52.7	47.9	50.5	50.7	**61.5**
GPT-4o	53.8	53.7	54.2	55.1	53.3	56.6	53.3	**63.6**
claude-3-5-sonnet	20.4	22.6	22.6	23.8	20.7	21.4	20.1	**27.8**
DeepSeek-V2.5	52.6	52.0	53.3	54.1	52.8	55.1	53.5	**61.1**

Table 4: Model-agnostic performance comparison across various workflow optimization methods on the MATH dataset. **Ours** consistently achieves the highest accuracy across all LLM backbones.

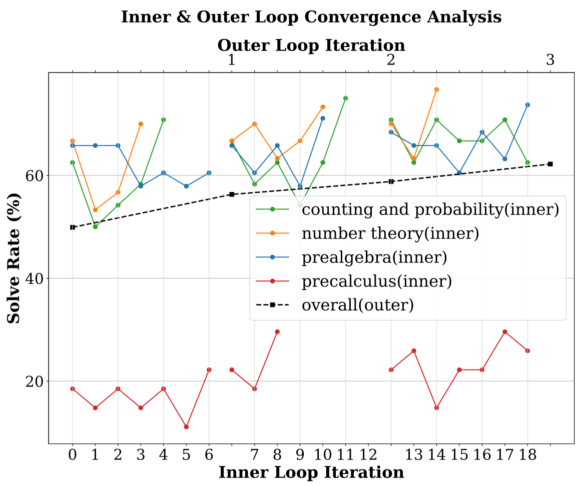

Figure 3: Convergence behavior of the inner and outer optimization loops on the MATH dataset. The inner loop (solid lines) shows fluctuations in solve rate across iterations for each subtask, with a maximum of 6 iterations per subtask, while the outer loop (dashed line) steadily improves overall performance by aggregating the best workflows per subtask.

Module	All	PreC	PreA	NT	C&P
DA	✓	✓	✓	✓	✓
AE	✓	✓	✓	✓	✓
CS	✓	✓	✓	✓	✓
VF	✓	✓	✗	✗	✗
CL	✓	✗	✗	✗	✗
SY	✓	✓	✓	✓	✓
VT	✗	✗	✗	✓	✗
AD	✗	✗	✓	✗	✗

Table 5: Module usage across subtasks on the MATH dataset. Each column represents a workflow configuration: **All** denotes the final workflow obtained after the third round of outer-loop optimization, while the others reflect the best inner-loop workflows before aggregation. Subtask abbreviations: **PreC** = Precalculus, **PreA** = Prealgebra, **NT** = Number Theory, **C&P** = Counting & Probability. Module abbreviations: **DA** = Diverse Agents, **AE** = Answer Extraction, **CS** = Consensus, **VF** = Verifier, **CL** = Clarifier, **SY** = Synthesis, **VT** = Value Tracker, **AD** = Approximation Detector. ✓ indicates module is used; ✗ indicates not used.

ing strong robustness and generalization.

While absolute performance varies across LLMs, our method consistently outperforms all baselines. The lower accuracy of Claude-3.5-Sonnet may stem from its weaker handling of structured outputs like JSON, which are central to our answer extraction pipeline. Nonetheless, our approach remains effective across model families without requiring model-specific customization.

6.5 Case Study

We present a case study to illustrate how the outer loop aggregates subtask-specific workflows into a unified workflow (Table 5). The **All** column represents the workflow obtained after outer-loop aggregation, while other columns correspond to the best inner-loop workflows for each subtask.

Shared Front-End. All workflows include three core modules: **DA** (Diverse Agents), **AE** (Answer Extraction), and **CS** (Consensus). These ensure solution diversity, consistent answer formats, and stable outputs, forming a robust foundation applicable across domains.

Task-Specific Modules. Additional modules are selectively introduced based on subtask characteristics. For example, **AD** (Approximation Detector) in Prealgebra handles rounding mismatches, while **VT** (Value Tracker) in Number Theory tracks intermediate values in multi-step reasoning.

This modular design supports both generalization and specialization, enabling high performance across diverse mathematical tasks.

7 Conclusion

We introduced AdaptFlow, a bi-level meta-optimization framework that learns adaptable agentic workflows via LLM-guided symbolic feedback. Across eight benchmarks, AdaptFlow outperforms both manual and automated baselines, with components like reflection and test-time adaptation enhancing robustness. Overall, it offers a scalable, model-agnostic solution for automating workflow design.

Limitations

While AdaptFlow achieves strong generalization, it has two primary limitations. First, the quality of symbolic updates depends on LLM-generated textual feedback, which can be vague or insufficiently detailed for complex failure cases. More structured or fine-grained feedback could improve update precision. Second, the optimization process requires repeated LLM queries, leading to non-trivial computational costs. Reducing query overhead through more efficient adaptation strategies is an important direction for future work.

References

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

Anthropic. 2024. Claude 3.5 sonnet. Available at https://www.anthropic.com/index/claude-3-5.

Jacob Austin, Augustus Odena, Maxwell Nye, and 1 others. 2021. Program synthesis with large language models. In *NeurIPS*.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, and et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.

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

Mark Chen, Jerry Tworek, Heewoo Jun, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Karl Cobbe and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2104.03235*.

Antonia Creswell and Murray Shanahan. 2023. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2305.05642*.

DeepSeek. 2024. Deepseek-v2.5. Available at https://deepseek.com.

Yixin Du and 1 others. 2023. The devil is in the debate: On the utility of argumentative dialogue agents for reasoning. *arXiv preprint arXiv:2305.14325*.

Dheeru Dua and 1 others. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *NAACL*.

Bas Fernando, Azalia Mirhoseini, Andrew Dai, and Quoc Le. 2023. Promptbreeder: Towards the automatic evolution of prompts. *arXiv preprint arXiv:2309.00680*.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Jiahao Guo, Zifan Wang, Sheng Zha, Xiaodong Wang, Xin Jin, and Dacheng Tao. 2023. Evoprompt: Language model guided genetic prompt optimization. *arXiv preprint arXiv:2309.07932*.

Dan Hendrycks and 1 others. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Chenxing Wei, Danyang Li, Jiaqi Chen, Jiayi Zhang, and 1 others. 2024a. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*.

Yujia Hong, Junyang Wu, Fan Zhang, and Shuo Zhang. 2024b. Adaptive agents with code and memory for solving math word problems. *arXiv preprint arXiv:2403.01290*.

Yujia Hong, Junyang Wu, Fan Zhang, and Shuo Zhang. 2024c. Sweagent: Code generation via structured workflow execution with llms. *arXiv preprint arXiv:2403.01290*.

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

Omar Khattab, Bhanukiran Vinzamuri Akula, and 1 others. 2023. Dspy: Expressive, modular prompting for language models. *arXiv preprint arXiv:2310.01348*.

Tao Li, Jiacheng Liu, Yichi Zhang, and 1 others. 2024a. Autoda: Towards data analysis automation with large language models. *arXiv preprint arXiv:2403.18270*.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024b. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*.

Zhen Liu, Wentao Wu, Yuke Zhu, and Zhiwei Steven Ling. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. *arXiv preprint arXiv:2303.13455*.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2024. A dynamic llm-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*.

J. MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297. University of California Press.

Erik Nijkamp, Ziyang Tu, Zexue Lin, and et al. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

Harsha Nori, Michael R. King, Scott M. McKinney, and 1 others. 2023. Capabilities of gpt-4 on medical challenge problems. *arXiv preprint arXiv:2303.13375*.

OpenAI. 2023. Aime benchmark for mathematical reasoning. *https://openai.com/research*. Accessed 2024.

OpenAI. 2024a. Gpt-4.1 overview. Available at https://openai.com/index/gpt-4-1/.

OpenAI. 2024b. Gpt-4o-mini-0718. Available via OpenAI API.

OpenAI. 2024c. Introducing gpt-4o. Available at https://openai.com/index/hello-gpt-4o/.

Yujia Qian and 1 others. 2023. Role-play prompting for multi-agent collaboration with llms. *arXiv preprint arXiv:2305.14325*.

Chenyan Qin, Kang Liu, Yaqing Zhang, and 1 others. 2023. Toolllm: Facilitating large language models to master 160+ tools. *arXiv preprint arXiv:2307.16789*.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *EMNLP*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992. Association for Computational Linguistics.

Tomer Ridnik, Yuval Shalev, Achiya Noy, and 1 others. 2024. Coding agents: Interactive code generation via llm planning and execution. *arXiv preprint arXiv:2402.03345*.

Javier Saad-Falcon, Ren Liu, Anthony Chan, and Allen Lin. 2024. Hyperparameter optimization in agentic llm pipelines. *arXiv preprint arXiv:2401.04903*.

Timo Schick, Ananya Dwivedi-Yu, Hinrich Schütze, and Peter Prettenhofer. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

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

Linxin Song, Jiale Liu, Jieyu Zhang, Shaokun Zhang, Ao Luo, Shijian Wang, Qingyun Wu, and Chi Wang. 2024. Adaptive in-conversation team building for language model agents. *arXiv preprint arXiv:2405.19425*.

Yichi Wang and 1 others. 2023. Large language models as optimizers for quality-diversity search. *arXiv preprint arXiv:2303.05832*.

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

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

Jinyuan Xie, Yang Yang, Ziyang Zhang, and 1 others. 2024. Autonova: Generating llm pipelines for data science via prompt evolution. *arXiv preprint arXiv:2402.04002*.

Yicheng Xu, Wei Zhang, Tao Li, and Shuo Zhang. 2024. Towards generalizable agents for mathematical reasoning. *arXiv preprint arXiv:2402.09399*.

Zhilin Yang and 1 others. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Qinyuan Ye, Xiao Liu, Yichong Zhou, and Shuo Zhang. 2024. Llm-dp: Towards autonomous data processing agents. *arXiv preprint arXiv:2403.05923*.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. 2024a. G-designer: Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*.

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

Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, and Bill Dolan. 2020. Dialogpt: Large-scale generative pre-training for conversational response generation. *ACL*.

Yilun Zhao, Yuan Yang, Zecheng Hu, and 1 others. 2024. Agentcoder: Integrating planning and execution for code generation agents. *arXiv preprint arXiv:2403.14260*.

Licheng Zhong, Yiding Li, Yichong Zhou, and Shuo Zhang. 2024. Mathagent: Math reasoning with retrieval-augmented code agents. *arXiv preprint arXiv:2402.03620*.

Xuezhi Zhou and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Yichong Zhou, Bowen Zhang, Yujia Hong, and Shuo Zhang. 2024a. Reasoning agents: Llm-as-policy for compositional task solving. *arXiv preprint arXiv:2404.01865*.

Yifan Zhou, Yecheng Li, Runzhe Xu, and 1 others. 2024b. Llm+p: Empowering large language models with planning for complex tasks. *arXiv preprint arXiv:2403.03031*.

Zeyu Zhu and 1 others. 2024. Olympiadbench: A benchmark for mathematical reasoning at the olympiad level. *arXiv preprint arXiv:2402.00000*.

Yujia Zhuge, Qinyuan Ye, Xiao Liu, Shujie Wang, and Furu Wei. 2023. Gptswarm: Multi-agent collaboration via llm-based swarm intelligence. *arXiv preprint arXiv:2311.16688*.

A Appendix

A.1 Dataset Details

Our experiments span eight public benchmarks across three major domains: question answering, code generation, and mathematical reasoning. Table 6 summarizes the dataset statistics, including the number of validation/test instances and the number of subtasks for each dataset. Each subtask represents a semantically or structurally coherent group of problems, enabling more focused workflow specialization during meta-optimization.

For question answering, we use subsets of HotpotQA and DROP, each containing 1,000 examples in total, with a 1:4 split for validation and testing. The examples are clustered into six subtasks based on instruction similarity. Similarly, in the coding domain, HumanEval and MBPP are divided into three and four subtasks, respectively, reflecting different code generation patterns.

In the mathematics domain, the datasets exhibit more diverse task structures. For GSM8K and AIME, we apply instruction-level clustering to derive six distinct subtasks per dataset, capturing variations in reasoning complexity and problem format.

Notably, two datasets—MATH and OlympiadBench—come with predefined topic categories, and thus do not undergo clustering. The MATH dataset contains high school-level math problems and is partitioned into four canonical categories: **Prealgebra**, **Precalculus**, **Number Theory**, and **Counting & Probability**, following the protocol introduced by Hendrycks et al. (2021). These categories capture distinct types of mathematical reasoning, from basic arithmetic to combinatorial logic.

Likewise, OlympiadBench is sourced from competitive mathematics exams and is naturally divided into four topics: **Algebra**, **Combinatorics**, **Geometry**, and **Number Theory**, as defined in the original benchmark by Zhu et al. (2024). These topics correspond to challenging mathematical reasoning tasks requiring manipulation, multi-step derivation, and rigorous abstraction.

Overall, our dataset setup provides a rich and heterogeneous landscape for evaluating workflow generalization, supporting both cluster-derived and taxonomy-preserving subtask definitions across domains.

A.2 Analogy Explanation

Figure 2 visualizes the analogy between neural network optimization and workflow optimization, which forms the conceptual foundation for our method. Here, we detail the core correspondences both at the structure level (parameters, updates, gradients) and at the algorithmic level (meta-learning procedure).

Structure-Level Analogy. In traditional supervised learning, model training involves continuous optimization of parameters θ using gradients $\nabla_\theta L$ derived from a differentiable loss. In contrast, our workflow optimization operates in a discrete, space, where the workflow $\mathcal{W}$ is updated through textual feedback generated by LLMs. The following table presents the one-to-one mapping:

Meta-Learning Analogy: MAML vs. AdaptFlow. At the algorithmic level, AdaptFlow is inspired by Model-Agnostic Meta-Learning (MAML), but adapted to the setting. While MAML learns a parameter initialization θ that can rapidly adapt via gradient updates, AdaptFlow learns a

	QA		Coding		MATH			
	HOTPOTQA	DROP	HUMANEVAL	MBPP	GSM8K	MATH	AIME	OLYMPIADBENCH
Validation Size	200	200	33	86	264	119	91	51
Val. Subtasks	6	6	3	4	6	4	6	4
Test Size	800	800	131	341	1055	486	373	212
Test Subtasks	6	6	3	4	6	4	6	4

Table 6: Dataset statistics for each domain and subtask. Validation/test sizes represent the number of instances used for evaluation, and subtask numbers denote the total distinct subtasks grouped under each benchmark.

Neural Network Optimization	**Workflow Optimization (AdaptFlow)**
Model parameters θ	Workflow structure $\mathcal{W}$
Loss function $L(f_\theta(x), y)$	Utility function $J(\mathcal{W}, T)$
Gradient $\nabla_\theta L$	Textual gradient $\widetilde{\nabla} J$ (LLM feedback)
Gradient descent update $\theta \leftarrow \theta - \eta \nabla_\theta L$	Symbolic update $\mathcal{W}' \leftarrow \mathcal{U}_1(\mathcal{W}, \widetilde{\nabla} J)$
Batch of examples $\{(x_i, y_i)\}$	Batch of tasks or subtask data $\mathcal{T}_t$

Table 7: Structure-level analogy between differentiable model optimization and discrete workflow optimization.

generalizable workflow $\mathcal{W}$ that adapts via LLM-generated updates. The table below compares the two approaches step-by-step:

Together, these analogies highlight how AdaptFlow generalizes the principles of meta-learning to the domain of agentic workflow optimization in spaces.

A.3 Prompt Templates

A.3.1 Inner Loop Workflow Optimization Prompt

```
# Overview
You are an expert machine learning researcher
    testing various agentic systems. Your
    objective is to design building blocks such
    as prompts and control flows within these
    systems to solve complex tasks. Your aim is
    to design an optimal agent performing well
    on the MATH dataset, which evaluates
    mathematical problem-solving abilities
    across various mathematical domains
    including algebra, counting and probability,
     geometry, intermediate algebra, number
    theory, prealgebra and precalculus.

## An example question from MATH:

**instruction (Not Given)**: Solve the following
     problem and provide a detailed solution.
    Present the final answer using the \boxed{}
    format.

**question**: question

**solution (Not Given)**: solution

# Discovered architecture archive
Here is the archive of the discovered
    architectures:

[ARCHIVE]

The fitness value is defined as the accuracy on
    a validation question set. Your goal is to
    maximize this fitness. You should use your
    own judgment to decide whether to optimize
    on the latest architecture, as its
    performance may not necessarily be better.

# Output Instruction and Example:
The first key should be ("thought"), and it
    should capture your thought process for
    designing the next function. In the "thought
    " section, first reason about what should be
     the next interesting agent to try, then
    describe your reasoning and the overall
    concept behind the agent design, and finally
     detail the implementation steps.
The second key ("name") corresponds to the name
    of your next agent architecture.
Finally, the last key ("code") corresponds to
    the exact âĂforward()âĂİ function in Python
    code that you would like to try. You must
    write a COMPLETE CODE in "code": Your code
    will be part of the entire project, so
    please implement complete, reliable,
    reusable code snippets.

Here is an example of the output format for the
    next agent architecture:

[EXAMPLE]

You must use the exact function interface used
    above. You need to specify the instruction,
    input information, and the required output
    fields for various LLM agents to do their
    specific part of the architecture. Also, it
    could be helpful to set the LLMâĂŹs role and
     temperature to further control the LLMâĂŹs
    response. Note that the LLMAgentBase() will
    automatically parse the output and return a
    list of âĂInfosâĂİ. You can get the content
    by Infos.content. DO NOT FORGET the taskInfo
     input to LLM if you think it is needed,
    otherwise LLM will not know about the task.
```

MAML (Finn et al., 2017)	AdaptFlow (Ours)
Model initialization θ	Workflow initialization $\mathcal{W}$
Task-specific adaptation via $\theta' \leftarrow \theta - \alpha \nabla_\theta L_T$	Subtask-specific refinement via $\mathcal{W}' \leftarrow \mathcal{U}_1(\mathcal{W}, \widetilde{\nabla} J)$
Compute outer gradient from θ'	Aggregate textual feedback from refined workflows $\{\widetilde{\nabla} J_t\}$
Outer update: $\theta \leftarrow \theta - \beta \nabla_\theta \sum L_{T_i}(\theta'_i)$	Meta update: $\mathcal{W} \leftarrow \mathcal{U}_2(\mathcal{W}, G(\{\widetilde{\nabla} J_t\}))$
Adaptation via differentiable gradient	Adaptation via textual feedback
Few-shot generalization to new tasks	Test-time adaptation via $\mathcal{W}^* \leftarrow \mathcal{U}_3(\mathcal{W}, \mathcal{F}(T'_t))$

Table 8: Algorithm-level comparison between MAML and AdaptFlow.

```
# Your task
You are deeply familiar with LLM prompting
    techniques and LLM agent works from the
    literature. Your goal is to maximize "
    fitness" by proposing interestingly new
    agents.
Observe the discovered architectures carefully
    and think about what insights, lessons, or
    stepping stones can be learned from them.
Please focus on the architecture with the
    optimal fitness, and based on that, propose
    what you believe is the most likely next
    agent architecture. Note that each
    optimization step can involve adding one or
    two new modules to the current best solution
    , or proposing an entirely novel solution.
    However, it's important to ensure that each
    change remains relatively simple and not
    overly complex.
```

A.3.2 Outer Loop Workflow Optimization Prompt

```
# Overview
You are an expert machine learning researcher
    testing various agentic systems. Your
    objective is to design building blocks such
    as prompts and control flows within these
    systems to solve complex tasks. Your aim is
    to design an optimal agent performing well
    on the MATH dataset, which evaluates
    mathematical problem-solving abilities
    across various mathematical domains
    including algebra, counting and probability,
     geometry, intermediate algebra, number
    theory, prealgebra and precalculus.

## An example question from MATH:

**instruction (Not Given)**: Solve the following
     problem and provide a detailed solution.
    Present the final answer using the \\boxed{}
     format.

**question**: question

**solution (Not Given)**: solution

Note: We divide the overall MATH task into seven
     distinct subtasks. Below is the performance
     of the Discovered Architecture Archive on
    each of these seven subtasks.
Discovered Architecture Archive
The following presents the archive of the
    discovered architectures on seven subtasks
    as well as the full MATH task:

[ARCHIVE_LIST]

The fitness value is defined as the accuracy on
    a validation question set. Your goal is to
    identify an architecture that either
    maximizes fitness across the seven subtasks
    or can quickly evolve toward that goal. Note
     that you should not limit yourself to only
    the most recently generated
    architecturesâĂŤyour objective is to
    maximize this fitness.
# Output Instruction and Example:
The first key should be ("thought"), and it
    should capture your thought process for
    designing the next function. In the "thought
    " section, first reason about what should be
     the next interesting agent to try, then
    describe your reasoning and the overall
    concept behind the agent design, and finally
     detail the implementation steps.
The second key ("name") corresponds to the name
    of your next agent architecture.
Finally, the last key ("code") corresponds to
    the exact âĂforward()âĂİ function in Python
    code that you would like to try. You must
    write a COMPLETE CODE in "code": Your code
    will be part of the entire project, so
    please implement complete, reliable,
    reusable code snippets.

Here is an example of the output format for the
    next agent architecture:

[EXAMPLE]

You must use the exact function interface used
    above. You need to specify the instruction,
    input information, and the required output
    fields for various LLM agents to do their
    specific part of the architecture.
Also, it could be helpful to set the LLMâĂŹs
    role and temperature to further control the
    LLMâĂŹs response. Note that the LLMAgentBase
    () will automatically parse the output and
    return a list of âĂIInfosâĂİ. You can get the
     content by Infos.content.
DO NOT FORGET the taskInfo input to LLM if you
    think it is needed, otherwise LLM will not
    know about the task.

## WRONG Implementation examples:
Here are some mistakes you may make:
```

```
1. This is WRONG: ```
feedback, correct = critic_agent([taskInfo,
    thinking, answer], critic_instruction, i)
feedback_info = verifier_agent([taskInfo, Info('
    feedback', 'Critic Agent', thinking, 0)],
    verification_instruction)
```
It is wrong to use "Info('feedback', 'Critic
    Agent', thinking, 0)". The returned "
    feedback" from LLMAgentBase is already Info.

Your task
You are well-versed in LLM prompting techniques
    and agent-based frameworks from the
    literature. You are tasked with designing a
    new agent architecture based on the best-
    performing solutions from each subtask of
    the MATH benchmark. The goal is for this new
     architecture to satisfy at least one of the
     following criteria:

It effectively integrates key modules and
    features from the optimal solutions of
    individual subtasks, resulting in a
    generalizable and adaptable architecture
    that performs well across all subtasks;

Alternatively, the architecture should exhibit
    strong adaptability and rapid update
    capabilities, allowing it to quickly evolve
    and converge toward the optimal solution for
     each specific subtask.
However, you should ensure that the newly
    generated frameworks is not significantly
    more complex than the original one, and you
    may also remove some redundant LLM
    invocation code.
```

### A.3.3 Reflection Prompt

```
We noticed that the current agent is prone to
    making mistakes when handling the following
    cases:
[CASE_LIST]

Please analyze the reasons for these mistakes
    and propose improvements.

Your response should be organized as follows:

"reflection": Provide your thoughts on the
    mistakes in the implementation, and suggest
    improvements.

"thought": Revise your previous proposal or
    propose a new architecture if necessary,
    using the same format as the example
    response.

"name": Provide a name for the revised or new
    architecture. (Don't put words like "new" or
     "improved" in the name.)

"code": Provide the corrected code or an
    improved implementation. Make sure you
    actually implement your fix and improvement
    in this code.
```

A.3.4 Test-Time Adaptation Workflow Optimization Prompt

```
# Overview
You are an expert machine learning researcher
    testing various agentic systems. Your
    objective is to design building blocks such
    as prompts and control flows within these
    systems to solve complex tasks. Your goal is
     to design an optimal agent that performs
    well on the MATH dataset. You may analyze
    the characteristics of these problems and
    then design an agent capable of effectively
    solving them.

[TASK_DSC]

Note: Your goal is to design an improved agent
    based on the previous agent, tailored to the
     characteristics of the current task. We aim
     to rapidly enhance the performance of the
    current agent.

# Output Instruction and Example:
The first key should be ("thought"), and it
    should capture your thought process for
    designing the next function. In the "thought
    " section, first reason about what should be
     the next interesting agent to try, then
    describe your reasoning and the overall
    concept behind the agent design, and finally
     detail the implementation steps.
The second key ("name") corresponds to the name
    of your next agent architecture.
Finally, the last key ("code") corresponds to
    the exact âĂforward()âĂİ function in Python
    code that you would like to try. You must
    write a COMPLETE CODE in "code": Your code
    will be part of the entire project, so
    please implement complete, reliable,
    reusable code snippets.

Here is an example of the output format for the
    next agent architecture:

[EXAMPLE]

You must use the exact function interface used
    above. You need to specify the instruction,
    input information, and the required output
    fields for various LLM agents to do their
    specific part of the architecture.
Also, it could be helpful to set the LLMâĂŹs
    role and temperature to further control the
    LLMâĂŹs response. Note that the LLMAgentBase
    () will automatically parse the output and
    return a list of âĂInfosâĂİ. You can get the
     content by Infos.content.
DO NOT FORGET the taskInfo input to LLM if you
    think it is needed, otherwise LLM will not
    know about the task.

# Your task
You are well-versed in LLM prompting techniques
    and agent-based frameworks from the
    literature. You are tasked with designing a
    new agent architecture based on the previous
     agent to solve the current task.
```
```

A.4 Workflow Case

To provide a concrete illustration of our system's output, we present the workflow code generated in the final outer-loop iteration on the MATH dataset. This example reflects the culmination of iterative refinement across subtasks and highlights the integration of shared and task-specific modules.

```
def forward(self, taskInfo):
    import re
    from collections import Counter

    def extract(text):
        for p in [r'\\boxed{([^}]*)}', r'\(([^)
            ]+)\)', r'\\frac{[^}]*}{[^}]*}', r'(\
            d+)\s*$']:
            m = re.search(p, text)
            if m: return m.group(0).strip()

    roles = ['Math Professor', 'Grade School
        Teacher', 'Math Enthusiast', 'Math
        Olympiad Student', 'Helpful Assistant']
    agents = [LLMAgentBase(['thinking', 'solution
        '], f'A{i}', role=r, temperature=0.7 +
        0.1*i) for i, r in enumerate(roles)]
    sols = [a([taskInfo], "Please think step by
        step and solve.", i) for i, a in
        enumerate(agents)]

    ext_agent = LLMAgentBase(['extracted_answer
        '], 'Extractor', role='Answer Extractor',
         temperature=0.1)
    answers, amap = [], {}
    for i, (t, s) in enumerate(sols):
        ans = ext_agent([taskInfo, s], "Extract
            ONLY final boxed answer.", i)[0].
            content.strip() or extract(s.content)
        if ans: answers.append(ans); amap.
            setdefault(ans, (t, s))

    top = Counter(answers).most_common()
    if top:
        top_answers = [a for a, c in top if c ==
            top[0][1]]
        if len(top_answers) == 1:
            _, sol = amap[top_answers[0]]
        else:
            inputs = [taskInfo] + sum((list(amap[
                a]) for a in top_answers), []) +
                [Info('extracted_answer', '', a,
                -1) for a in top_answers]
            sol = LLMAgentBase(['thinking', '
                solution'], 'Final Decider',
                temperature=0.1)(inputs, "Choose
                best answer.")[1]
    else:
        inputs = [taskInfo] + sum(([t, s] for t,
            s in sols), [])
        sol = LLMAgentBase(['thinking', 'solution
            '], 'Fallback Decider', temperature
            =0.1)(inputs, "Choose among all.")[1]

    verifier = LLMAgentBase(['feedback', 'correct
        '], 'Verifier', role='Checker',
        temperature=0.1)
    clarifier = LLMAgentBase(['clarification'], '
        Clarifier', role='Solver', temperature
        =0.4)
    synthesizer = LLMAgentBase(['thinking', '
        solution'], 'Synth', temperature=0.3)

    for i in range(2):
        ext = ext_agent([taskInfo, sol], "Extract
             ONLY final boxed answer.", 100+i)[0]
        fb, ok = verifier([taskInfo, sol, ext], "
            Check correctness.", i)
        if ok.content == 'True': return sol

        clar, = clarifier([taskInfo, sol, fb], "
            Respond to critique.", i)
        fb2, ok2 = verifier([taskInfo, sol, ext,
            clar], "Recheck solution.", 100+i)
        if ok2.content == 'True': return sol

        syn_inputs = [taskInfo, sol, fb2, clar] +
             sum(sols, []) + [Info('
            extracted_answer', '', a, -1) for a
            in answers if a]
        sol = synthesizer(syn_inputs, "Revise or
            synthesize.")[1]

    return sol
```