# METAFLOW: A META APPROACH OF TRAINING LLMS INTO GENERALIZABLE WORKFLOW GENERATORS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large language models (LLMs) excel across a wide range of tasks, yet their instance-specific solutions often lack the structural consistency needed for reliable deployment. Workflows that encode recurring algorithmic patterns at the task level provide a principled framework, offering robustness across instance variations, interpretable traces for debugging, and reusability across problem instances. However, manually designing such workflows requires significant expertise and effort, limiting their broader application. While automatic workflow generation could address this bottleneck, existing methods either produce instance-specific solutions without learning task-level patterns, or cannot generalize beyond their training configurations. We present **MetaFlow**, which casts workflow generation as a meta-learning problem: given a task and an operator set, the model learns to compose solution strategies. MetaFlow trains in two stages—supervised fine-tuning on synthetic workflow data, followed by reinforcement learning with verifiable rewards (RLVR) that uses execution feedback across problem instances in the task to improve end-to-end success. The resulting model produces effective workflows for trained tasks and exhibits strong generalization to untrained tasks and novel operator sets. Across benchmarks in question answering, code generation, and mathematical reasoning, MetaFlow achieves performance comparable to state-of-the-art baselines on in-domain tasks with single inference, while demonstrating remarkable zero-shot generalization capabilities on out-of-domain tasks and operator sets.

## 1 INTRODUCTION

Large Language Models (LLMs) have demonstrated significant performance across a wide range of tasks, including code generation, question answering, and mathematical reasoning (Austin et al., 2021; Chen et al., 2021; Yang et al., 2018; Dua et al., 2019; Ding et al., 2024; Jiang et al., 2025; Cobbe et al., 2021; OpenAI, 2023; Zhu et al., 2024). However, because these models generate instance-specific solutions, they lack the structural consistency and transparency needed for reliable deployment, while also being difficult to adapt to similar tasks. *Workflows* that encode recurring algorithmic patterns provide a principled alternative, decomposing complex challenges into structured, manageable steps. However, manually designing such workflows requires significant expertise and effort, limiting their broader application.

To address this challenge, recent effort have focused on the automatic workflow generation (Khattab et al., 2023; Li et al., 2024; Song et al., 2024; Zhang et al., 2024a). Endowing LLMs with this strategy planning capability means lowering the barrier for complex task automation from requiring manual programming by experts to merely providing high-level task descriptions, thereby greatly liberating productivity. Nevertheless, representing the workflow as static graph (Zhuge et al., 2024) or neural network (Liu et al., 2024) in many of these methods limits the flexibility of generatable workflows.

A promising direction emerges from works like ADAS (Hu et al., 2024), AFlow (Zhang et al., 2024b), ScoreFlow (Wang et al., 2025) and FlowReasoner (Gao et al., 2025), which represent workflows as code (a structured combination of predefined *operators*), making the automatic generation of workflows more flexible and expressive, where *operator* is an encapsulation of common agentic

operations introduced by Zhang et al. (2024b). Within this code-based framework, current approaches adopt two different paradigms for workflow generation.

The first paradigm comprises **task-level** approaches, exemplified by ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2024b), which formulate workflow generation as a search problem within predefined task-operator set combination. Both methods employ iterative search strategies, with ADAS using evolutionary algorithms and AFlow using Monte Carlo Tree Search (MCTS), to discover high-performing workflows through repeated refinement. However, this search-based paradigm inherently constrains them to specific task and predetermined operator set. When encountering new tasks or operators, these methods require complete re-optimization from scratch, incurring substantial computational costs (Wang et al., 2025).

Conversely, the second paradigm consists of **instance-level** approaches, exemplified by ScoreFlow (Wang et al., 2025) and FlowReasoner (Gao et al., 2025), which generate workflows tailored to individual problem instances in the task. Both methods dynamically construct workflows at inference time, with ScoreFlow leveraging gradient-based optimization to refine agentic workflows, and FlowReasoner employing reasoning chains distilled from advanced models to design query-specific multi-agent systems. While these instance-level methods excel at tailoring workflows to specific problem instances, this granularity comes at the cost of reusability and deployment efficiency. They cannot capture task-level patterns that recur across similar problem instances, leading to redundant workflow generation for each query. In deployment scenarios, this approach foregoes the benefits of having optimized, reusable workflow templates that could be consistently applied to entire task.

The limitations of existing paradigms underscore two fundamental challenges that must be addressed to achieve truly general-purpose automatic workflow generation. **(1) How can we overcome the re-optimization requirement of task-level approaches when facing new domains (task-operator set combinations)? (2) How can we learn generalizable patterns that avoid redundant instance-level generation while adapting effectively to untrained domains?**

To systematically overcome the challenges, we propose **MetaFlow**, which formulates workflow generation as a meta-learning problem. Unlike search-based methods that require expensive re-optimization for new task-operator set combinations, **MetaFlow** learns to directly synthesize workflows from task descriptions and operator set specifications, enabling zero-shot generation through a single model inference.

To achieve robust zero-shot generalization, **MetaFlow** employs a two-stage training paradigm with diverse task-operator pairs. Adopting the code-based workflow representation from prior works(Hu et al., 2024; Zhang et al., 2024b; Wang et al., 2025; Gao et al., 2025), we first synthesize thousands of workflows using Qwen-Max (Team, 2024) across four tasks and a single operator set to finetune Qwen3-8B (Yang et al., 2025), establishing the foundation for understanding how tasks and operators relate to workflow structures. Subsequently, we apply online reinforcement learning with GRPO (Shao et al., 2024) across expanded domains, where execution feedback on problem instances directly optimizes the generation policy. This training ensures the model learns generalizable workflow construction principles rather than memorizing patterns. At inference, **MetaFlow** zero-shot generates effective workflows for novel configurations with only a single forward pass.

Our main contributions are:

- **Meta-learning Framework**: We introduce **MetaFlow**, a novel approach that reformulates workflow generation from discrete search within fixed configurations to continuous learning across diverse task-operator set combinations. By conditioning workflow generation on task descriptions and operator specifications, our framework achieves strong zero-shot generalization to unseen domains without any re-optimization, reducing computational cost from thousands of API calls to a single model inference.

- **Scalable Training Pipeline**: We design a two-stage training framework combining supervised learning with online reinforcement learning, utilizing diverse domains to ensure robust generalization to untrained domains.

- **Comprehensive Evaluation**: Extensive experiments demonstrate that **MetaFlow** achieves competitive performance on in-domain benchmarks while exhibiting remarkable zero-shot generalization to out-of-domain task classes and operator sets, including solving program-

ming problems with novel operator combinations never seen during training and solving question answering problem using the vector database search operator.

## 2 Related Works

### 2.1 Agentic Workflow

Agentic workflows decompose complex tasks into structured steps through predefined operators and dependencies (Zhang et al., 2024b; Wang et al., 2025; Gao et al., 2025). Unlike autonomous agents that learn through environment interaction (Zhuge et al., 2024; Hong et al., 2024), workflows provide interpretable and consistent execution patterns. Recent works adopt code-based representations for superior expressiveness (Hu et al., 2024; Zhang et al., 2024b; Wang et al., 2025; Gao et al., 2025), supporting applications in code generation, question answering, and mathematical reasoning (Austin et al., 2021; Chen et al., 2021; Yang et al., 2018; Dua et al., 2019; Ding et al., 2024; Jiang et al., 2025; Cobbe et al., 2021; OpenAI, 2023; Zhu et al., 2024). However, manual workflow design remains a significant bottleneck requiring deep expertise.

### 2.2 Automatic Workflow Generation

Recent advances have explored automating workflow generation to improve LLM performance (Chen et al., 2023; Zhang et al., 2024b; Wang et al., 2025; Li et al., 2024; Song et al., 2024). While some methods optimize prompts within fixed workflows (Guo et al., 2023; Khattab et al., 2023), we focus on optimizing workflow structures directly.

Current structural optimization follows two paradigms. Task-level approaches like ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2024b) search for optimal workflows through evolutionary algorithms or MCTS, but require complete re-optimization for new domains. Instance-level methods including ScoreFlow (Wang et al., 2025) and FlowReasoner (Gao et al., 2025) generate query-specific workflows but fail to extract reusable patterns.

Our Methods, **MetaFlow** reformulates workflow generation as meta-learning over diverse task-operator combinations during training. Through two-stage optimization combining supervised learning with reinforcement learning, it achieves true zero-shot generation—producing effective workflows for novel domains via single model inference, eliminating both re-optimization and adaptation overhead.

## 3 Problem Definition

Existing works often formulate automatic workflow generation as optimization problems (Xu et al., 2025; Li et al., 2025), requiring separate optimizations for each task. We elevate this perspective by reformulating it as a Meta-learning problem (Finn et al., 2017; Franceschi et al., 2018). To ground this formulation, we first define our core concepts: a Problem Instance $\mathsf{p}$ is a single, concrete problem to be solved. A Task $\mathsf{C}$ is a family of such instances with a shared structure. The available operators are defined by an Operator Set $\mathsf{Ops}$, a collection of fundamental, reusable operations (Zhang et al., 2024b). Together, the (Task, Operator Set) $(\mathsf{C}, \mathsf{Ops})$ constitutes a complete domain.

Within this meta-learning framework, an LLM serves as the meta-learner (the planner, $\pi_\theta$). Its core responsibility is to learn a meta-strategy that enables fast adaptation. Specifically, given any $(\mathsf{C}, \mathsf{Ops})$ pair, the planner can rapidly generate an efficient and reusable Workflow $\mathsf{W}$. This workflow is a task-level strategy, formalized as a structured sequence of operators from $\mathsf{Ops}$. When applied to any specific Problem Instance $\mathsf{p}$ within Task $\mathsf{C}$, this workflow guides the execution process to produce a high-quality Solution $\mathsf{s}$.

Unlike traditional meta-learning approaches such as Model-Agnostic Meta-Learning (MAML) (Finn et al., 2017), which rely on gradient updates for adaptation, our method achieves fast adaptation through the synthesis of workflows without requiring any gradient-based fine-tuning.

The differences between our method and traditional approachs can be illustrated as 1

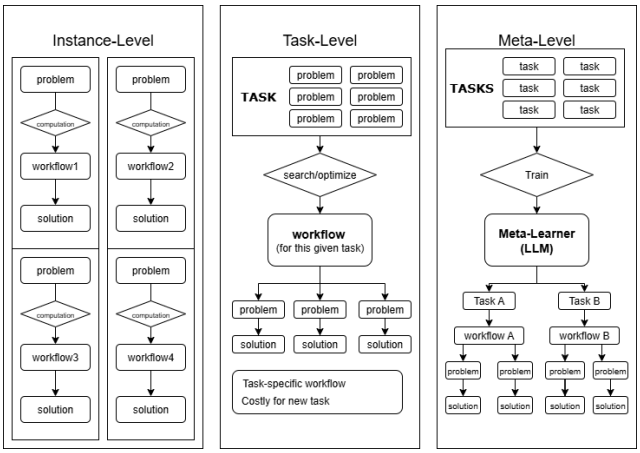


Figure 1: Three methods

As a meta-learning task, our goal is to optimize the meta-parameters $\theta$ of the planner LLM ($\pi_\theta$). Specifically, we seek the optimal $\theta^*$, such that the workflow $\mathsf{W}$ generated by $\pi_\theta$ maximizes the expected reward when applied to any domain $(\mathsf{C}, \mathsf{Ops})$ from the (task, operators set) distribution $\mathcal{D}_{(\mathsf{C},\mathsf{Ops})}$. We adopt the classic optimization strategy of meta-learning, namely the **bi-level optimization** structure Franceschi et al. (2018), which can be rigorously formulated as:

$$\theta^* = \arg\max_{\theta} \underbrace{E_{(\mathsf{C},\mathsf{Ops})\sim\mathcal{D}_{(\mathsf{C},\mathsf{Ops})}}}_{\text{Outer Loop (Meta-Optimization)}} \underbrace{[E_{\mathsf{W}\sim\pi_\theta(\cdot|\mathsf{C},\mathsf{Ops},\mathcal{O})}[E_{\mathsf{p}\sim P(\mathsf{p}|\mathsf{C})}[R(\text{Exec}(\mathsf{W},\mathsf{p}))]]]}_{\text{Inner Loop (Task-Specific Adaptation \& Evaluation)}}$$

Where

- $(\mathsf{C}, \mathsf{Ops}) \sim \mathcal{D}_{(\mathsf{C},\mathsf{Ops})}$: sampling a task from the (task, operators set) distribution $\mathcal{D}_{(\mathsf{C},\mathsf{Ops})}$.
- $\mathsf{W} \sim \pi_\theta(\cdot \mid \mathsf{C}, \mathsf{Ops}, \mathcal{O})$: the planner $\pi_\theta$ as the meta-learner, performing fast adaptation under the given specific task description $\mathsf{C}$, available operator set $\mathsf{Ops}$, and optional contextual information $\mathcal{O}$ (such as system prompt), to generate a workflow $\mathsf{W}$ customized for the task.
- $\mathsf{p} \sim P(\mathsf{p} \mid \mathsf{C})$: sampling a specific problem instance $\mathsf{p}$ from the instance distribution of task $\mathsf{C}$ for evaluating the generated workflow.
- $\text{Exec}(\mathsf{W}, \mathsf{p})$ represents the process of applying the workflow $\mathsf{W}$ to solve the problem instance $\mathsf{p}$ in the executor environment, producing the final solution.
- $R(\cdot)$ is a reward function used to evaluate the quality of the final solution (e.g., code test pass rate, answer accuracy).

The **outer loop** optimizes not the performance on a single instance or single problem family, but the average performance of workflows across the entire (task, operators set) distribution $\mathcal{D}_{(\mathsf{C},\mathsf{Ops})}$, which directly drives the meta-learner $\pi_\theta$ to learn strategies with cross-domain generalization capabilities. In contrast, the optimization objectives of works such as ScoreFlow(Wang et al., 2025) and ComfyUI-R1 (Xu et al., 2025) can be formalized as maximizing the expected reward on a single problem instance $\mathsf{p}$, i.e., $\arg\max E_{\mathsf{p}\sim P(\mathsf{p}|\mathsf{C})}[R(\text{Exec}(\mathsf{W}_\mathsf{p}, \mathsf{p}))]$. Our framework elevates the optimization plane from the **instance-level** to the **meta-level**, thereby explicitly learning general workflow strategies $\mathsf{W} \sim \pi_\theta(\cdot \mid \mathsf{C}, \mathsf{Ops})$ that can generalize across entire task and operator combinations (Zhang et al., 2024b; Li et al., 2025).

## 4 METHODOLOGY

Our **MetaFlow** framework aims to train a large language model through a two-stage learning process to automatically generate efficient and reusable workflows for given problem families. This section elaborates on our system architecture, and core training algorithm.

### 4.1 METAFLOW ARCHITECTURE

The core architecture of **MetaFlow** includes a planner LLM and an execution-evaluation environment.

**Workflow Representation**: A workflow is represented as a structured script based on the MetaGPT framework (Hong et al., 2023), composed of a series of predefined operator calls. Following the practices of AFlow and ScoreFlow, we design a set of general, semantically rich atomic operators, including `Generate` (for content generation), `Summarize` (for information compression), `Revise` (for iterative improvement), and `Ensemble` (for result aggregation) (Zhang et al., 2024b; Li et al., 2025). These operators form the basic building blocks of workflows.

**Input-Output**: Building on this, we further clarify the complete input-output structure of the model. The MODEL INPUT consists of two parts: OPERATOR DESCRIPTIONS, which define the functions, parameters, and input-output formats of each available operator (whether preset or user-defined); TASK TYPE DESCRIPTION, which elucidates the domain characteristics of the target problem family and the input-output formats of each belonging problem instance. The OUTPUT requires a structured WORKFLOW aimed at efficiently solving all problem instances under the task type using the given operator set. This design paradigm allows users to conveniently introduce new tools and new task types through natural language or semi-structured descriptions during testing. At the same time, it ensures that these new tools and tasks can be quickly understood and effectively generalized by the model, achieving high flexibility and scalability.

**Execution & Evaluation Environment**: To achieve end-to-end optimization, we build an automated environment. This environment receives a candidate workflow $\mathsf{W}_i$ and a set of $N$ problem instances $\mathsf{p}_1, \ldots, \mathsf{p}_N$ from a specific problem family $\mathsf{C}$ as input. In the execution phase, the environment uses the `metagpt` framework to orchestrate the workflow (Hong et al., 2023). Each operator in the workflow (such as `Generate`) completes its specific subtask by calling an external lightweight language model API (qwen-turbo) and processes each problem instance. Upon completion, an automated evaluator module verifies the correctness of each execution result, for example, by running unit tests or comparing outputs with standard answers. Based on the evaluation results, the environment computes a quantified, verifiable reward score $R(\mathsf{W}_i)$ for the workflow $\mathsf{W}_i$. This score directly reflects the generalization capability of the workflow, calculated as the average success rate over $N$ instances:

$$R(\mathsf{W}_i) = \frac{1}{N} \sum_{j=1}^{N} \mathbb{I}(\text{is correct}(\text{Exec}(\mathsf{W}_i, \mathsf{p}_j)))$$

where $\mathbb{I}(\cdot)$ is the indicator function. This reward score $R(\mathsf{W}_i)$ is then passed to the training algorithm as the basis for policy updates.

### 4.2 TRAINING ALGORITHM

Our training process is divided into two stages: **supervised learning-based initialization** and **reinforcement learning-based end-to-end optimization**. This combination aims to fully leverage the guiding role of supervised data while retaining the ability of reinforcement learning to explore superior strategy spaces.

#### 4.2.1 PHASE ONE: SFT INITIALIZATION

To address the cold start problem caused by the excessively large exploration space in reinforcement learning from scratch, we first perform supervised fine-tuning on the base LLM. We construct a high-quality dataset containing pairs of (problem family-operator group description, high-quality workflow). Through standard autoregressive language model training on this dataset, the model $\pi_\theta$ learns the initial ability to generate syntactically correct and structurally reasonable workflows. The goal of this phase is to inject prior knowledge about effective strategy patterns into the model, providing the generated candidate workflows with a reasonable starting point, thereby significantly narrowing the search space in the subsequent RL phase and accelerating convergence.

#### 4.2.2 PHASE TWO: RLVR OPTIMIZATION

After SFT, we employ policy gradient algorithms to perform end-to-end self-improvement on the planner $\pi_\theta$. The core of this phase is the **RLVR (Reinforcement Learning with Verifiable Reward)** loop, with the specific process as follows:

1. **Policy Sampling**: For a problem family $\mathsf{C}$ sampled from the training set, the planner $\pi_\theta$ generates a batch of $k$ candidate workflows $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_k$.

2. **Execution & Reward Calculation**: Each candidate workflow $\mathsf{W}_i$ is tested in the execution and evaluation environment described in Section 3.2, obtaining its corresponding reward score $R(\mathsf{W}_i)$ that reflects generalization capability.

3. **Policy Update**: We use this batch of '(workflow, reward)' pairs to update the parameters of the planner $\pi_\theta$. In particular, we adopt the **Group Relative Policy Optimization (GRPO)** algorithm (Shao et al., 2024). The core idea of GRPO is to use the average performance within the group as a baseline to estimate advantages, thereby avoiding training an independent value network. We compute the advantage as follows:

$$\hat{A}(\mathsf{W}_i) = R(\mathsf{W}_i) - \mu_R$$

where $\mu_R = \frac{1}{k}\sum_{j=1}^{k} R(\mathsf{W}_j)$ is the average reward obtained by the group of $k$ workflows. This calculation based on "group-relative advantages" makes the optimization signal derive from whether the workflow performance is better or worse than the current batch's average level, rather than an absolute, potentially noisy value estimate.

4. **Variance Reduction: Evaluation Based on Common Random Numbers**

The effectiveness of policy gradients largely depends on the accuracy of the advantage function $\hat{A}(\mathsf{W}_i)$ estimation. In our GRPO method, the advantage is computed relative to the batch average reward $\mu_R$. If each workflow $\mathsf{W}_i$ in the batch is evaluated on a set of independently and randomly sampled problem instances, the variance of the reward $R(\mathsf{W}_i)$ will include not only policy randomness (i.e., the quality of the workflow itself) but also environmental randomness (i.e., the difficulty of problem instances). This additional variance propagates to $\mu_R$ and $\hat{A}(\mathsf{W}_i)$, producing noisier gradients and reducing learning efficiency.

To address this issue, we adopt a classic variance reduction technique—**Common Random Numbers (CRN)** (Kleijnen, 1975). In specific implementation, we **ensure that all $k$ candidate workflows $\mathsf{W}_1, \ldots, \mathsf{W}_k$ in the same training batch (group) are evaluated on the exact same set of problem instances** $\{\mathsf{p}_1, \ldots, \mathsf{p}_N\}$. By fixing the random variable of problem instances when comparing workflows, we eliminate noise arising from differences in problem sampling. This keeps the expected value of $R(\mathsf{W}_i) - R(\mathsf{W}_j)$ unchanged but significantly reduces its variance. Ultimately, this ensures that our computed relative advantages more accurately reflect the intrinsic performance differences between workflows, leading to a more stable policy update direction and accelerating model convergence.

## 5 EXPERIMENTS

This section presents systematic experiments designed to evaluate our proposed **MetaFlow** framework. The experimental design aims to answer several core research questions: (1) Can workflows generated by **MetaFlow** achieve comparable performance to current state-of-the-art methods while requiring only single inference for enhanced efficiency? (2) Does the framework exhibit strong out-of-distribution (OOD) generalization to new tasks and operators? (3) What are the individual contributions of key components such as supervised fine-tuning initialization and reinforcement learning with verifiable rewards? (4) Can **MetaFlow** effectively enhance the performance of small to medium-sized language models on complex tasks? We first introduce the experimental configuration, followed by main results, ablation studies, and representative case analyses.

### 5.1 Experimental Setup

To ensure fairness and reproducibility, we construct a standardized experimental environment.

#### 5.1.1 Training Configuration

**MetaFlow** employs a two-stage training paradigm.

**Stage 1: Supervised Fine-Tuning (SFT) Initialization.** To alleviate the cold-start problem in reinforcement learning, we first conduct supervised fine-tuning on the base model Qwen3-8B. We construct an expert dataset containing 1,300 high-quality samples, where each sample is a (task class description, high-quality workflow) pair. These workflows cover typical strategy patterns including sequential, branching, looping, and parallel execution, providing the model with a robust initial policy.

**Stage 2: Reinforcement Learning with Verifiable Rewards (RLVR) Optimization.** Following SFT initialization, we employ RLVR for end-to-end optimization of the Planner. To ensure training efficiency, this stage focuses on four selected domains: GSM8K, DROP, MBPP, and HumanEval. During this stage, the model is restricted to using only four basic operators. Training employs the Grouped Relative Policy Optimization (GRPO) algorithm for 137 steps. The reward curve shows consistent and stable policy improvement throughout the training process. To control training costs, the Executor in the training loop calls a lightweight API—Qwen-Turbo.

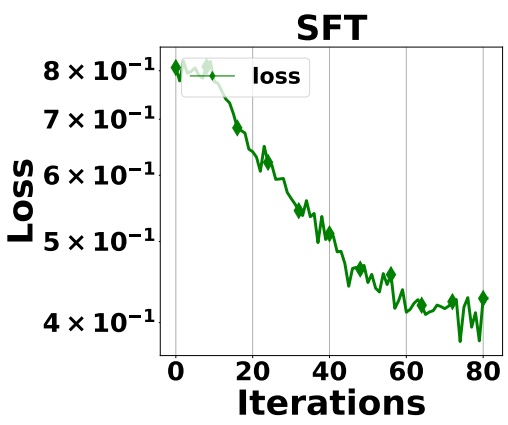


Figure 2: SFT loss curve

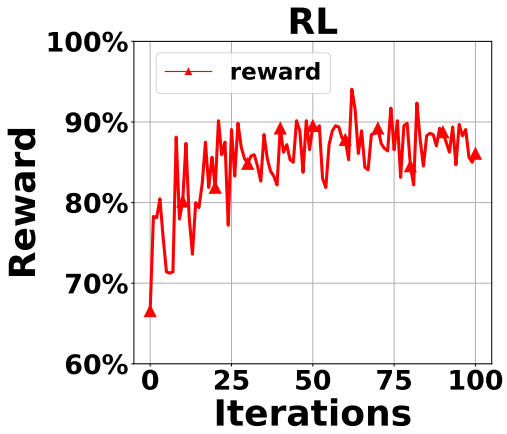


Figure 3: RL running reward

Figure 4: Trainning curves

We adopt the experimental settings from ScoreFlow, with evaluation covering three major domains: question answering, code generation, and mathematical reasoning.

- **Question Answering:** 1,000 instances sampled from each of HotpotQA and DROP.
- **Code Generation:** Full sets of MBPP.and Humaneval.
- **Mathematical Reasoning:** 1,000 questions sampled from GSM8K and Level-5 problems from the MATH dataset.

Consistent with ScoreFlow's approach, all datasets except HumanEval are deterministically split into train and test sets at a 1:4 ratio. We did not test the scores on the HumanEval dataset, as its small size makes it difficult to complete the subsequent validation and testing steps.

#### 5.1.2 Baseline Methods

We compare **MetaFlow** against two categories of methods:

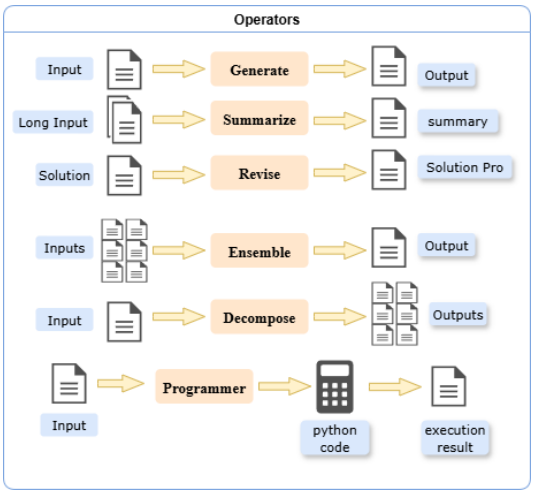


Figure 5: Operators

- **Manually Designed Workflows:** Including existing prompting strategies such as Vanilla Prompting(IO), Chain-of-Thought (CoT), Reflexion, LLM Debate, Step-back Abstraction, Quality-Diversity (QD), and Dynamic Role Assignment.
- **Automatically Optimized Workflows:** Including current state-of-the-art methods such as ADAS, AFLOW, and Scoreflow.

#### 5.1.3 Operator Design

We predefine a set of general atomic operators: Generate, Summarize, Revise, and Ensemble. A key feature of our framework is that the Planner can dynamically rewrite any operator's prompt to customize its behavior. To rigorously test OOD generalization, we introduce two novel operators unseen during training at the evaluation stage: Decompose and Programmer. These operators' functionalities are provided through natural language descriptions via in-context injection. The Decompose operator is responsible for breaking complex problems into subproblem lists, while the Programmer operator requires the LLM to write code snippets and return execution results.

#### 5.1.4 Evaluation Protocol

**Inference Process:** For each task, **MetaFlow** generates 20 candidate workflows. We employ a best-of-20 strategy: all candidate workflows execute on an independent validation set containing 50 problem instances, and the workflow with the highest average reward is selected for final testing.

**Executor and Judge:** To ensure fair comparison, during evaluation, all methods (including baselines) use GPT-4o-mini as the Executor. For question answering tasks, we adopt the LLM-as-a-Judge paradigm, also using GPT-4o-mini as the judge.

#### 5.1.5 Implementation Details

The Planner is implemented based on Qwen3-8B, with workflow execution orchestrated by the MetaGPT framework.

### 5.2 Main Results and Analysis

#### 5.2.1 Overall Performance Comparison

As shown in Table 1, **MetaFlow** achieves an average performance score of 78.8, demonstrating competitive results. Notably, its performance is comparable to ScoreFlow, Aflow and ADAS, which requires resource-intensive per-instance optimization. Additionally, **MetaFlow** matches or exceeds

| Method | DROP | MBPP | GSM8K | MATH | Avg |
|---|---|---|---|---|---|
| IO | 81.6 | 69.5 | 89.1 | 52.2 | 73.1 |
| CoT (Wei et al., 2022) | 83.2 | 70.4 | 88.3 | 53.4 | 73.8 |
| CoT SC (Wang et al., 2022) | 83.2 | 71.3 | 88.6 | 53.8 | 74.2 |
| MedPrompt (Nori et al., 2023) | 83.0 | 69.2 | 88.1 | 53.7 | 73.5 |
| MultiPersona (Wang et al., 2024) | 81.3 | 70.4 | 89.8 | 51.9 | 73.4 |
| Self Refine (Madaan et al., 2024) | 82.5 | 70.0 | 87.5 | 50.0 | 72.5 |
| ADAS (Hu et al., 2024) | 81.3 | 68.7 | 90.5 | 51.7 | 73.1 |
| AFlow (Zhang et al., 2024b) | 83.5 | 82.9 | 90.8 | 55.8 | 78.3 |
| ScoreFlow (Wang et al., 2025) | 86.2 | 84.7 | 94.6 | 64.4 | 82.5 |
| **Ours** | **82.8** | **77.5** | **93.8** | **61.0** | **78.8** |

other automated workflow generation methods that rely on computationally expensive search processes on downstream tasks. All baseline model scores in the table are sourced from the ScoreFlow (Wang et al., 2025) paper.

#### 5.2.2 Out-of-Distribution (OOD) Generalization

**Adaptation to Novel Operators:** Our results demonstrate that **MetaFlow** can effectively integrate new operators into generated workflows based solely on their natural language descriptions. On MATH,MBPP and HotpotQA A.0.1 in Appendix A, the model autonomously generated workflows incorporating the unseen Programmer operator. This capability enables performance that not only significantly surpasses manually designed workflows but also rivals automated search methods that have access to the operator during training or search phases.

**Adaptation to New Domains:** The framework exhibits strong generalization to task domains not encountered during RLVR optimization such as MATH. Using only high-level task descriptions provided in prompts, **MetaFlow** generates effective workflows with performance comparable to state-of-the-art methods specifically trained or designed for these domains. This highlights the framework's rapid zero-shot adaptation capability, a core objective of our meta-learning paradigm.

## 6 Conclusion

We introduce MetaFlow, which employs a two-stage training paradigm with diverse task-operator pairs. Across benchmarks in question answering, code generation, and mathematical reasoning, MetaFlow achieves performance comparable to state-of-the-art baselines on in-domain tasks with single inference, while demonstrating remarkable zero-shot generalization capabilities on out-of-domain tasks and operator sets.

## References

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

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa,

Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

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

Hongxin Ding, Yue Fang, Runchuan Zhu, Xinke Jiang, Jinyang Zhang, Yongxin Xu, Xu Chu, Junfeng Zhao, and Yasha Wang. 3ds: Decomposed difficulty data selection's case study on llm medical domain adaptation. *arXiv preprint arXiv:2410.10901*, 2024.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *NAACL-HLT (1)*, pp. 2368–2378. Association for Computational Linguistics, 2019.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 1126–1135. PMLR, 2017.

Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 1568–1577. PMLR, 2018.

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

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023.

Ilgee Hong, Zichong Li, Alexander Bukharin, Yixiao Li, Haoming Jiang, Tianbao Yang, and Tuo Zhao. Adaptive preference scaling for reinforcement learning with human feedback. *Advances in Neural Information Processing Systems*, 37:107249–107269, 2024.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Zhang, Zijuan Gui, et al. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

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

Bowen Jiang, Runchuan Zhu, Jiang Wu, Zinco Jiang, Yifan He, Junyuan Gao, Jia Yu, Rui Min, Yinfan Wang, Haote Yang, et al. Evaluating large language model with knowledge oriented language specific simple question answering. *arXiv preprint arXiv:2505.16591*, 2025.

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

Jack P. C. Kleijnen. Antithetic variates, common random numbers and optimal computer time allocation in simulation. *Management Science*, 21(10):1176–1185, 1975.

Chenyang Li, Ziqiang Wang, Dong Zhang, Xue Zhao, Cheng Wang, Xingwu Wang, Yuan Wang, Haifeng Zhang, and Wenwu Zhu. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

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

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

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

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

Zhihong Shao, Peiyi Yuan, Hongsheng Li, Yizhe Wang, Yubo Xu, Xiaoke Sun, Ke Liu, Yuanhan Lin, Chunyan Yue, Kun Chen, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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

Qwen Team. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *The Eleventh International Conference on Learning Representations*, 2022.

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

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

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

Zhenran Xu, Yiyu Wang, Xue Yang, Longyue Wang, Weihua Luo, Kaifu Zhang, Baotian Hu, and Min Zhang. Comfyui-r1: Exploring reasoning models for workflow generation. *arXiv preprint arXiv:2506.09790*, 2025.

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

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

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

Jiayi Zhang, Yihang Xiang, Chao Wang, Amina Zhou, Jiaqi Lu, Di Chen, Chaowei He, Yanshuai Wang, Bin Ding, Dacheng Gao, et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024b.

Zeyu Zhu et al. Olympiadbench: A benchmark for mathematical reasoning at the olympiad level. *arXiv preprint arXiv:2402.00000*, 2024.

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

# A CASE STUDY: WORKFLOW DESIGN TRANSFORMATION AFTER TRAINING

To better understand the effectiveness of our training paradigm, we conduct a critical ablation study analyzing changes in model behavior before and after training. We compare two randomly selected workflows generated by the untrained base model (Qwen3-8B) versus the fully trained **MetaFlow** model after our two-stage training.

### A.0.1 IMPACT OF TRAINING ON WORKFLOW GENERATION BEHAVIOR

**Before Training (Base Model Qwen3-8B):** Without SFT and RLVR optimization, the base model generates workflows with multi-step logic following a linear process: extract function name → summarize → generate → revise → ensemble.

```python
class Workflow:
    # ... (initialization code omitted) ...
    async def run_workflow(self):
        # Step 1: Extract function name from test cases
        func_name = await self.generate(
            instruction="Extract the function name from the test cases.
                         Return ONLY the function name, nothing else.",
            context=self.problem_text
        )
        # Step 2: Summarize problem description and test cases
        problem_summary = await self.summarize(
            instruction="Summarize the problem description and test cases
                         to create a clear problem statement.",
            context=self.problem_text
        )
        # Step 3: Generate code based on problem summary
        generated_code = await self.generate(
            instruction=f"Write a Python function named '{func_name}'...
                          Return ONLY the function definition, nothing
                              else.",
            context=problem_summary
        )
        # Steps 4 & 5: Revise and ensemble
        revised_code = await self.revise(...)
        final_code = await self.ensemble(contexts=[generated_code,
            revised_code])
        return final_code
```

Listing 1: Workflow generated by the untrained base model.

**Analysis:** While structured, this workflow has potential weaknesses:

1. **Information Bottleneck:** The `summarize` step, intended to condense information, risks losing critical details or constraints from the original problem description.
2. **Instruction Ambiguity:** Vague instructions like "Return ONLY the function definition" can be misinterpreted, leading to incomplete code that lacks necessary import statements or helper functions.

**After Training (MetaFlow Model):** In contrast, the **MetaFlow** model optimized through our two-stage training paradigm generates a workflow that, while maintaining complexity, is significantly more robust and methodical in its approach, and has learned the self-consistency operator unseen during training.

```python
% --- DO NOT IMPORT HERE ---
class Workflow:
    def __init__(self, config, problem) -> None:
        # --- DO NOT MODIFY THIS SECTION ---
        self.config = config
        self.problem_text = problem
        self.llm = create(config)
```

```
        self.generate = operator.Generate(self.llm, self.problem_text)
        self.revise = operator.Revise(self.llm, self.problem_text)
        self.summarize = operator.Summarize(self.llm, self.problem_text)
        self.ensemble = operator.Ensemble(self.llm, self.problem_text)
        self.selfconsistency = operator.SelfConsistency(self.llm, self.
            problem_text)

    async def run_workflow(self):
        """
        Implement the core problem-solving logic here.
        Remember:
        - Use detailed, comprehensive instructions
        - Dynamic instruction construction is powerful
        - All operators expect (instruction: str, context: str) except
            Ensemble which takes contexts: List[str]
        """
        import asyncio

        # Step 1: Extract function name from test cases
        function_name = await self.generate(
            instruction="""
            Analyze the test cases provided and extract the function name
                .
            The function name should match the naming convention used in
                the test cases.
            Return ONLY the function name, nothing else.
            """,
            context=self.problem_text
        )

        # Step 2: Extract input types and expected outputs from test
            cases
        input_types_output = await self.generate(
            instruction="""
            Analyze the test cases provided and extract the input types
                and expected outputs.
            Format the response as a dictionary where keys are the
                parameter names and values are the types.
            Also, identify the expected output format.
            Return ONLY the structured information, nothing else.
            """,
            context=self.problem_text
        )

        # Step 3: Generate multiple candidate implementations
        implementations = await asyncio.gather(
            self.generate(
                instruction=f"""
                Given the problem description: {self.problem_text}
                Generate a Python function that satisfies the test cases.
                Ensure that the function name is '{function_name}'.
                Include all necessary imports at the beginning of the
                    function.
                Use the extracted input types and expected outputs: {
                    input_types_output}.
                """,
                context=self.problem_text
            ),
            # ... (additional parallel generate calls omitted for brevity
                ) ...
            self.generate(
                instruction=f"""
                Given the problem description: {self.problem_text}
```

```
                Generate another Python function that satisfies the test cases.
                Ensure that the function name is '{function_name}'.
                Include all necessary imports at the beginning of the function.
                Use the extracted input types and expected outputs: {input_types_output}.
                """,
                context=self.problem_text
            )
        )

        # Step 4: Select the best implementation using SelfConsistency
        best_implementation = await self.selfconsistency(
            instruction=f"""
            Evaluate the following candidate implementations:
            {implementations[0]}
            {implementations[1]}

            Select the best implementation based on the following criteria:
            - Consistency with the test cases
            - Logical correctness
            - Efficient and clean code
            - Proper use of imports and function signature

            Return ONLY the best implementation, nothing else.
            """,
            context=self.problem_text
        )

        return best_implementation
```

Listing 2: Workflow generated by the trained **MetaFlow** model.

**Analysis:** The trained model's workflow exhibits several key improvements contributing to its robustness:

1. **From Compression to Structured Extraction:** The model avoids the potentially lossy `summarize` step. Instead, it performs targeted extractions to parse unstructured text into structured data (e.g., function name, I/O types). This approach preserves information integrity and provides a more reliable foundation for subsequent steps.
2. **Parallel Exploration for Increased Robustness:** Rather than following a single generation-and-revision path, the model generates multiple, independent candidate solutions in parallel using `asyncio.gather`. This multi-path exploration inherently increases the robustness of the process by diversifying the solution space and reducing the risk of settling on a single, flawed implementation.
3. **More Explicit and Context-Aware Instructions:** The instructions are dynamically constructed using f-strings to include context from prior steps (e.g., 'function_name', 'input_types_output'). This makes the prompts highly specific and unambiguous, directly mitigating the weaknesses of the base model and ensuring that each generative step is precisely guided.

# A Case Study: Learning a Novel Search Operator for Multi-Hop QA

Our operator-centric framework is designed for extensibility, allowing external tools to be seamlessly integrated as new operators. This case study demonstrates how a model learns to utilize a novel 'Search' tool—specifically 'operator.VectorSearch'—which was not seen during its primary training phase. The following workflow code was implemented to solve problems from the HotpotQA dataset, a task that requires multi-hop reasoning over multiple documents.

```

class Workflow:
    def __init__(self, config, problem) -> None:
        # --- DO NOT MODIFY THIS SECTION ---
        self.config = config
        self.problem_text = problem
        self.llm = create(config)

        self.generate = operator.Generate(self.llm, self.problem_text)
        self.revise = operator.Revise(self.llm, self.problem_text)
        self.summarize = operator.Summarize(self.llm, self.problem_text)
        self.ensemble = operator.Ensemble(self.llm, self.problem_text)
        self.vector_search = operator.VectorSearch(self.llm, self.
            problem_text)

    async def run_workflow(self):
        """
        Implement the core problem-solving logic here.
        Remember:
        - Use detailed, comprehensive instructions
        - Dynamic instruction construction is powerful
        - All operators expect (instruction: str, context: str) except
            Ensemble which takes contexts: List[str]
        """
        import asyncio

        # Step 1: Extract the question and identify the type of reasoning
             required
        question_extraction_instruction = """
        Analyze the provided problem text and extract the following:
        1. The exact question being asked
        2. Identify whether the question requires:
           - Bridge reasoning (connecting entities across documents)
           - Comparison reasoning (comparing properties across documents)
           - Compositional reasoning (combining multiple facts)
        3. Provide a brief justification for your reasoning type
            classification
        """
        question = await self.generate(instruction=
            question_extraction_instruction, context=self.problem_text)

        # Step 2: Retrieve relevant documents based on the question
        doc_retrieval_instruction = """
        Given the extracted question, retrieve relevant documents from
            the knowledge base that provide information related to the
            question.
        Focus on documents that contain entities, relationships, or facts
             that could help answer the question.
        """
        retrieved_docs = await self.vector_search(
            instruction=doc_retrieval_instruction,
            context=question,
            top_k=5
        )

        # Step 3: Analyze the retrieved documents to identify bridge
            entities and reasoning chains
        analysis_instruction = f"""
        Based on the following information:
        - Question: {question}
        - Retrieved Documents: {retrieved_docs}

        Perform the following steps:
```

```
        1. Identify all entities, key facts, and relationships across the documents that are relevant to answering the question.
        2. Determine which entity or concept serves as the "bridge" connecting information across documents.
        3. Construct an explicit reasoning chain that connects the information from the documents to form a complete answer.
        4. Ensure the reasoning chain is precise and follows a logical flow.
        """
        reasoning_chain = await self.generate(instruction=analysis_instruction, context=retrieved_docs)

        # Step 4: Extract the final answer from the last document in the reasoning chain
        final_answer_extraction_instruction = """
        From the reasoning chain provided, extract the final answer to the question.
        Ensure the answer is:
        - A short text span (entity or phrase)
        - Precise and unambiguous
        - Directly supported by the reasoning chain
        """
        final_answer = await self.generate(
            instruction=final_answer_extraction_instruction,
            context=reasoning_chain
        )

        # Step 5: Critique and refine the extracted answer if necessary
        refinement_instruction = """
        Review the extracted answer and refine it based on the following criteria:
        - Ensure it is factually accurate
        - Ensure it is precise and concise
        - Ensure it directly addresses the question
        """
        refined_answer = await self.revise(instruction=refinement_instruction, context=final_answer)

        return refined_answer
```

Listing 3: A workflow where the model learns to use the 'VectorSearch' operator to retrieve documents for multi-hop reasoning.

### Analysis and Performance

The workflow in Listing 3 demonstrates the successful integration and application of a new tool. In Step 2, the model dynamically constructs a search query from its initial analysis and invokes the VECTORSEARCH operator, effectively performing active information retrieval. When evaluated on the HotpotQA downstream task, this approach achieved a **60% search accuracy**. This result is significant as it confirms that our operator framework enables models to learn and effectively utilize unseen tools.

### Note on Comparability with Baselines

It is crucial to highlight a fundamental difference between our evaluation and that of many previous works on HotpotQA. Our methodology requires the model to **actively perform a search** to find relevant information. In contrast, prior baselines are often provided with the ground-truth supporting documents as part of their input, thereby bypassing the challenging information retrieval step entirely. Because our system solves a more complete and realistic version of the task that includes an explicit search phase, a direct comparison of end-to-end accuracy with such baselines is not meaningful.