BayesFlow: A Probability Inference Framework for Meta-Agent Assisted Workflow Generation

Bo Yuan[1,3], Yun Zhou[2], Zhichao Xu[2]
Kiran Ramnath[2], Aosong Feng[2], Balasubramaniam Srinivasan[2,3]

[1]Georgia Institute of Technology
[2]Amazon Web Services AI Lab
[3]Correspondence to byuan48@gatech.edu, srbalasu@amazon.com

Abstract

Automatic workflow generation is the process of automatically synthesizing sequences of LLM calls, tool invocations, and post-processing steps for complex end-to-end tasks. Most prior methods cast this task as an optimization problem with limited theoretical grounding. We propose to cast workflow generation as Bayesian inference over a posterior distribution on workflows, and introduce **Bayesian Workflow Generation (BWG)**, a sampling framework that builds workflows step-by-step using parallel look-ahead rollouts for importance weighting and a sequential in-loop refiner for pool-wide improvements. We prove that, without the refiner, the weighted empirical distribution converges to the target posterior. We instantiate BWG as **BayesFlow**, a training-free algorithm for workflow construction. Across six benchmark datasets, BayesFlow improves accuracy by up to 9 percentage points over SOTA workflow generation baselines and by up to 65 percentage points over zero-shot prompting, establishing BWG as a principled upgrade to search-based workflow design.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable generality, often solving tasks with a single carefully engineered prompt (Wei et al., 2022a; Wang et al., 2022a; Shinn et al., 2023; Asai et al., 2024). However, as tasks become increasingly complex and demand multi-step reasoning, tool usage and memory, there has been a growing shift toward agentic systems, where multiple LLM agents collaborate through structured workflows, consistently achieving significant performance gains over simple prompt engineering. (Du et al., 2023; Zhao et al., 2023; Liang et al., 2023; Jiang et al., 2023; Madaan et al., 2023).

However, designing and refining workflows using frameworks such as AutoGen (Wu et al., 2024a), CAMEL (Li et al., 2023), and MetaGPT (Hong et al., 2023) remains a labor-intensive and expertise-driven process, often involving extensive trial and error. This manual bottleneck limits the scalability of LLM systems to new tasks, each demanding bespoke solutions which hinders the adaptability of existing solutions to evolving task requirements. (Tang et al., 2023; Qiao et al., 2024; Cemri et al., 2025). Consequently, **automatic workflow design** has emerged as a crucial research challenge to advance the capabilities and generality of LLM-based systems (Zhuge et al., 2024; Zhang et al., 2024a; Hu et al., 2024; Li et al., 2024b).

Most prior methods formulate automatic workflow design as optimization problems, maximizing validation performance via search heuristics such as Monte Carlo tree search (Zhang et al., 2024a), linear heuristic search (Hu et al., 2024), or evolutionary strategies (Li et al., 2025; Shang et al., 2024) via a meta optimizer LLM. However, these approaches typically lack rigorous theoretical foundations and yield a single high scoring solution with limited diversity. In contrast, our **sampling-based** Bayesian formulation provides principled guarantees and naturally produces a diverse set of high quality workflows via posterior sampling, an effect observed in controllable text generation (Qin et al., 2022), adversarial attacks (Guo et al., 2024), and machine translation (Faria et al., 2024).

In this work, we formulate workflow generation as a posterior sampling problem in structured **code** representations (Hu et al., 2024), where each workflow is composed of modular code chunks corresponding to steps. We adopt a Bayesian sampling perspective (Doucet et al., 2000), treating the meta optimizer LLM's internal knowledge as a prior distribution over plausible workflows, while incorporating external feedback signals (e.g. task-specific rewards or correctness) via an energy-based mechanism (Du and Mordatch, 2019). Energy-based mod-

Findings of the Association for Computational Linguistics: EACL 2026, pages 3151–3179

els (EBMs) model unnormalized probability distributions and take the form $\exp\big(R(s_{1:T})\big)$, where $s_{1:T}$ denotes a workflow with T steps and R is a predefined reward function.

More precisely, our objective is to sample a complete workflow $s_{1:T}$ from the unnormalized posterior.

$$q(s_{1:T}|s_0) \propto p(s_{1:T}|s_0)\ \exp\big(R(s_{1:T})\big) \quad (1)$$

where $p(s_{1:T}|s_0)$ is a prior induced by the meta optimizer LLM p and a query prompt s_0. For notational simplicity, we will use s and $s_{1:T}$ interchangeably and omit the dependency on s_0 when there is no confusion. It is worth noting that our posterior target distribution has rich connections to both reinforcement learning and inference scaling laws. Detailed analysis of connections is provided in Appendix A.

In this work, we introduce **Bayesian Workflow Generation (BWG)**, formalizing automatic workflow construction as Bayesian posterior sampling over workflows, replacing trajectory level code synthesis with fine-grained, step level generation via *parallel look-ahead rollouts* for importance weighting and *sequential in-loop refinement* for candidate improvement in Section 4. Furthermore, we prove that the weighted empirical distribution converges asymptotically to the target distribution under mild assumptions in Theorem 1. We instantiate BWG as **BayesFlow** in Section 5 which demonstrates consistent gains across various benchmarks and model families, as shown in Section 6.

In summary, our contributions are three-fold: (1) Motivated by connections to reinforcement learning and inference scaling laws, we formulate workflow generation as Bayesian inference rather than optimization, which naturally promotes diversity in generated workflows. We also provide rigorous theoretical guarantees for this Bayesian formulation. (2) We propose Bayesian Workflow Generation (BWG), a sampling-based framework that produces high-quality workflows via parallel look-ahead rollouts and sequential in-loop refinements. The parallel look-ahead step estimates the value of incomplete workflows without additional training or reliance on stronger closed-source models, while the in-loop refiner unifies prior methods under BWG and further improves workflow quality. (3) We instantiate BWG with **BayesFlow** and demonstrate consistent performance gains in six datasets on both closed-source and open-source LLMs. Consequently, BayesFlow improves accuracy by up to 9% over SOTA workflow generation baselines on the math reasoning dataset and yields an average gain of up to 4.6% across all benchmarks.

2 Related Work

Agentic workflows represent a fundamental paradigm in LLM applications, consisting of structured sequences of LLM invocations designed to solve complex tasks through predefined processes (Hong et al., 2024b; Zhang et al., 2024b). Unlike autonomous agents that make dynamic decisions based on environmental feedback, agentic workflows operate through predetermined static logical sequences (or with conditionals) that can be systematically designed and refined (Zhuge et al., 2023; Wang et al., 2023a). While effective, these manually designed workflows require substantial human expertise, limiting their scalability to new domains.

Recent research automates workflow optimization through prompt optimization (Fernando et al., 2023; Wang et al., 2023b; Yang et al., 2023; Yuksekgonul et al., 2024), hyperparameter tuning (Saad-Falcon et al., 2024), and workflow structure optimization (Li et al., 2024b; Hu et al., 2024; Zhuge et al., 2024). Training free methods leverage pretrained LLMs for iterative improvement without parameter updates. Component level methods like DSPy (Khattab et al., 2023) and TextGrad (Yuksekgonul et al., 2024) optimize prompting strategies within fixed structures. Structure generation methods including ADAS (Hu et al., 2024), GPTSwarm (Zhuge et al., 2024), and AFLOW (Zhang et al., 2024a) explore complete workflow architectures, though they struggle to leverage historical data and often perform no better than random sampling. The fundamental challenge lies in effectively navigating the vast search space of possible workflow configurations while maintaining computational efficiency and ensuring the generated workflows generalize across different tasks and domains. Moreover, there is limited theoretical discussion on the property of generated workflows. An extended discussion can be found in Appendix B.

3 Preliminaries

Workflow generation as step-level code generation We view automatic workflow generation as Bayesian posterior sampling in an energy-based model, and frame it concretely as a code generation task executed by an optimizer LLM. At the finest granularity, the optimizer LLM generates tokens,

yet throughout the rest of this paper, we operate at step level: tokens are grouped into meaningful code chunks s_t. To achieve so, we explicitly require the meta optimizer LLM to generate comments in the form of "Step <n>:" before every major block. See example workflows in Appendix H for each generated comment. Although a workflow is originally represented as a directed graph (Khattab et al., 2023), encoding it as code (Zhang et al., 2024a) imposes a convenient linear order greatly simplifies generation processes.

Math formulation A workflow with T steps is a discrete trajectory $s_{1:T} = (s_1, \ldots, s_T)$ drawn from the autoregressive prior of the meta optimizer LLM, $p(s_{1:T}) = p(s_1) \prod_{t=2}^{T} p(s_t \mid s_{1:t-1})$. We use a single reward that measures validation accuracy $R(s_{1:T}) = \mathrm{Acc}_{\mathrm{val}}(s_{1:T}) \in [0, 1]$. The posterior distribution over the workflows is then $\frac{1}{Z}\, p(s_{1:T}) \exp\big[R(s_{1:T})\big]$, where $Z = \sum_{s_{1:T}} p(s_{1:T}) \exp\big[R(s_{1:T})\big]$ is the normalization constant. For notational simplicity in mathematical expressions, we adopt a fixed horizon T. Workflows may terminate at different step counts; shorter sequences can be padded with empty strings, so that every trajectory admits a length representation T without loss of generality.

Challenges It is worth noting that sampling in this setting is nontrivial. The formulation exploits the autoregressive factorization of the LLM prior. Each conditional $p(s_t|s_{1:t-1})$ is produced step by step while operating under a regime *terminal-reward* in which the reward $R(s_{1:T})$ is observed only after the entire workflow has been generated. The absence of intermediate rewards poses a significant challenge in guiding prefix decisions. The posterior step distribution can be written as

$$p(s_t \mid s_{1:t-1}) \underbrace{\sum_{s_{t+1:T}} p(s_{t+1:T} \mid s_{1:t}) \exp\big[R(s_{1:T})\big]}_{\text{look-ahead value}}.$$

This look-ahead term couples the current choice with an intractable sum over all completions, making the exact sampling challenging.

One way to solve this problem is to draw N complete trajectories $s_{1:T}^{(i)} \sim p(s_{1:T})$ and then resample following unnormalized weights using only the reward: $w_i = \exp\big[R(s_{1:T}^{(i)})\big], \bar{w}_i = \frac{w_i}{\sum_{j=1}^{N} w_j}$. This method is simple but can suffer from *weight degeneracy* (Johansen, 2009), where a few $\bar{w}_i$ dominate and most samples contribute negligibly. A typical strategy to mitigate it is step-level resampling. This motivates us to propose parallel look-ahead rollouts in Section 4.1.

4 Bayesian Workflow Generation

To address the challenges in the terminal reward setting, we introduce **Bayesian Workflow Generation (BWG)**, a general Bayesian sampling framework shifting workflow construction from monolithic code synthesis to fine-grained step-level generation through **parallel loop-ahead rollouts** and **sequential in-loop refinements**. A class of inference-only methods treats workflow design as generating code at the trajectory level (Zhang et al., 2024a; Hu et al., 2024). Step-wise generation builds workflows incrementally, but generally relies on strong closed source to score partial workflows (Li et al., 2025) or an auxiliary learned value model to guide the search (Shang et al., 2024). In contrast, BWG assigns importance weights via parallel look-ahead rollouts that anticipate the downstream energy of each partial workflow. This look-ahead mechanism removes the reliance on external proprietary models and avoids expensive process-model training, while ensuring that the weighted empirical distribution converges asymptotically to the target distribution in Theorem 1.

In addition to parallel sampling, BWG integrates a sequential in-loop refiner that improves complete rollouts: Given a set of existing full workflows, the refiner is expected to produce additional higher-quality workflows. This module coincides with the core optimization step in previous work - implemented by MCTS in (Zhang et al., 2024a; Li et al., 2025) or by linear heuristic search in (Hu et al., 2024). This perspective casts BWG as a unifying framework that subsumes existing workflow generation methods as special cases.

4.1 Parallel look-ahead rollouts

Unlike training a process reward model (Li et al., 2025), which is usually unstable (Xu et al., 2025) and sensitive to distribution shift (LeVine et al., 2023), we adopt the idea from Sequential Monte Carlo (SMC) (Doucet et al., 2001; Loula et al., 2025) and build the trajectory in the following round.

1. *Extend each prefix.* For each partial workflow $s_{1:t-1}^{(i)}$, sample one step from the prior $s_t^{(i)} \sim p(s_t|s_{1:t-1}^{(i)})$, forming the new prefix $s_{1:t}^{(i)}$.

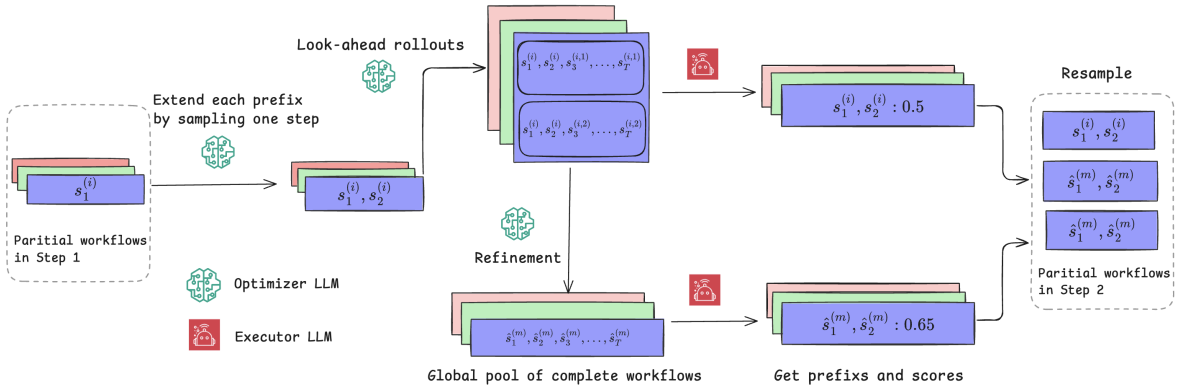

Figure 1: This diagram depicts Algorithm 1 in a setting with $N = 3$ candidate partial workflows, $K = 2$ look-ahead rollouts per candidate, and $M = 3$ refinement attempts. The optimizer LLM is invoked to (i) sample next-step expansions, (ii) generate parallel look-ahead rollouts to estimate downstream value, and (iii) refine the current pool; the executor LLM is invoked to score complete workflows. The figure shows the transition of partial workflows from Step 1 to Step 2.

2. *Look-ahead scoring.* For each new prefix $s_{1:t}^{(i)}$, draw K stochastic completions $\tilde{s}_{t+1:T}^{(i,k)} \sim p(s_{t+1:T}|s_{1:t}^{(i)})$ and set the importance weight $w^{(i)} = \frac{1}{K}\sum_{k=1}^{K} \exp\big[R\big(s_{1:t}^{(i)}, \tilde{s}_{t+1:T}^{(i,k)}\big)\big]$.
3. *Normalize and Resample.* Compute normalized weights $\bar{w}^{(i)} = w^{(i)} / \sum_{j=1}^{N} w^{(j)}$. Resample the prefixes $s_{1:t}^{(i)}$ according to $\bar{w}^{(i)}$; specifically, draw the N indices $a_1, \ldots, a_N$ with $\Pr(a_n = i) = \bar{w}^{(i)}$, thus replicating higher-weight prefixes and pruning lower-weight ones.

In each round, we progress from N partial workflows of length t to $t+1$. Iterating this round until T yields an approximation of the target posterior $p(s_{1:T}) \exp[R(s_{1:T})]$, mitigating weight degeneracy relative to pure prior reweighting. This mechanism leads to a tractable estimate of the intractable marginal and yields informative weights before resampling. We emphasize that the look-ahead rollouts are parallel across all partial workflows, incurring only negligible additional wall-clock time.

4.2 Sequential in-loop refinements

Even with informative look-ahead weights, exploration rollouts are drawn solely from the prior, limiting reach to workflows far from familiar patterns. More importantly, once a low-quality prefix is chosen, the parallel look-ahead has no mechanism to repair that early mistake: the prefix is fixed, and the trajectory tends to a poor terminal reward. Repeated resampling can partially mitigate this by reallocating mass to successful workflows, but a sample-only scheme remains highly reactive and underuses the self-reflection capabilities of the meta optimizer LLM (Renze and Guven, 2024).

Therefore, after computing the look-ahead scoring (Stage 2) in the parallel look-ahead mechanism, we augment the candidate pool *before* the final normalization and resampling (Stage 3) via sequential in-loop refinements as follows:

1. *Global refinement operator.* For each particle i with prefix $s_{1:t}^{(i)}$, let its K rollouts be $\mathcal{C}_t^{(i)} = \{\tilde{s}_{t+1:T}^{(i,k)}\}_{k=1}^{K}$ and its look-ahead weight be $w^{(i)}$. Define the pool that includes all complete rollouts of NK and their weights as $\mathcal{P}_t$. We introduce a general refinement operator $\mathcal{G}$ that generates M complete workflows based on the entire pool, i.e., $\{\hat{s}_{1:T}^{(m)}\}_{m=1}^{M} \sim \mathcal{G}(\cdot|\mathcal{P}_t)$. Note that refinement operates at the workflow level: It may revise existing prefixes instead of merely extending them, yielding globally improved trajectories, and thereby enabling the refinement of earlier low-quality steps.
2. *Score proposals.* For each workflow, set $\hat{w}^{(m)} = \exp\big[R(\hat{s}_{1:T}^{(m)})\big]$.
3. *Normalize and resample.* Normalize all the weights in the $\{w^{(i)}\}_{i=1}^{N} \cup \{\hat{w}^{(m)}\}_{m=1}^{M}$ and resample N prefixes from this enlarged pool.

The refinement operator is a core, commonly used component in existing trajectory-level methods. We adopt a sequential generate-and-insert schedule: starting from the current pool, we produce one complete workflow at a time, immediately score it, and insert it back into the pool before generating the next. Our contribution here is a conceptual framework in which various methods can

serve as a refiner. Concretely, previous work takes a pool of complete workflows with terminal scores and applies a search operator, e.g., linear heuristic expansion (Hu et al., 2024), Monte Carlo Tree Search (MCTS) (Zhang et al., 2024a), or evolutionary mutation (Shang et al., 2024). In our implementation, the refiner is MCTS largely adopted from AFlow (Zhang et al., 2024a) due to its superior performance across baselines in our experiments.

Building on parallel look-ahead rollouts and sequential global refinement, we arrive at the Algorithm 1. See Figure 1 for one concrete example. Its sole purpose in iteration t is to advance the prefix population from length t to $t{+}1$ by (i) exploiting parallel estimates of downstream value and (ii) injecting pool-wide refinements before a final resampling. This yields a transition operator preserving diversity, prioritizing promising prefixes, and preparing the pool for the next step.

4.3 Bayesian Workflow Generation as a general framework

BWG subsumes trajectory level workflow generation methods (Zhang et al., 2024a; Hu et al., 2024; Shang et al., 2024) as special cases. If the parallel look-ahead mechanism is disabled, BWG degenerates to pure refinement of complete workflows. In contrast, the removal of the refiner. i.e., $M = 0$, yields pure parallel exploration akin to SMC. In the general case , BWG strictly extends these extremes by inserting step-level stochastic exploration before refinement, exploiting informative prefixes that purely trajectory-level schemes cannot access.

4.4 Theoretical analysis

In this section, we present the convergence theorem that provides the theoretical analysis of Algorithm 1. To the best of our knowledge, this work provides the first theoretical analysis in the field of automatic workflow generation. The theorem shows that our parallel look-ahead design preserves asymptotic convergence to the original target distribution. It also implies that sampling from the target rather than the prior improves the expected reward of generated workflows. See Appendix C for a quick proof. It is worth noting that with the refinement mechanism enabled, the asymptotic convergence guarantee no longer holds. However, our ablation study demonstrates that refinement contributes to empirical gains.

Theorem 1. *Consider Algorithm 1 without the refiner (i.e., $M{=}0$) for T steps, and let $\widehat{\pi}_{N,T}$ be the empirical distribution of the final N complete workflows. Then, as $N \to \infty$[1], the empirical measure $\widehat{\pi}_{N,T}$ converges in probability to the target distribution q in equation 1. Moreover, $\mathbb{E}_q[R(s)] \geq \mathbb{E}_p[R(s)]$, where R is the reward function and p is the prior distribution of the meta optimizer LLM.*

Algorithm 1 StepUpdate

Require: Prefix pool $W_{t-1} := \{s^{(i)}_{1:t-1}\}_{i=1}^{N}$; rollout count K; number of refined workflows M

1: **Extend prefixes:** For each $i = 1, \ldots, N$, sample one step $s_t^{(i)} \sim p(s_t | s^{(i)}_{1:t-1})$ and set $s^{(i)}_{1:t} \leftarrow (s^{(i)}_{1:t-1}, s^{(i)}_t)$.

2: **Look-ahead scoring:** For each new prefix $s^{(i)}_{1:t}$, draw K completions $\{\tilde{s}^{(i,k)}_{t+1:T}\}_{k=1}^{K} \sim p(s_{t+1:T} | s^{(i)}_{1:t})$ and compute the score $w^{(i)} = \frac{1}{K}\sum_{k=1}^{K} \exp\big[R\big(s^{(i)}_{1:t}, \tilde{s}^{(i,k)}_{t+1:T}\big)\big]$.

3: **Global refinement:** Using the entire pool $\mathcal{P}_t$, generate M new complete workflows $\{\hat{s}^{(m)}_{1:T}\}_{m=1}^{M} \sim \mathcal{G}(\cdot | \mathcal{P}_t)$.

4: **Project and score proposals:** For each m, denote the first t steps as $\hat{s}^{(m)}_{1:t}$ and compute $\hat{w}^{(m)} = \exp\big[R\big(\hat{s}^{(m)}_{1:T}\big)\big]$.

5: **Augment pool:** Form $\mathcal{P}_t^{\text{aug}} = \{(s^{(i)}_{1:t}, w^{(i)})\}_{i=1}^{N} \cup \{(\hat{s}^{(m)}_{1:t}, \hat{w}^{(m)})\}_{m=1}^{M}$.

6: **Resample prefixes:** Draw N prefixes from $\mathcal{P}_t^{\text{aug}}$ with probabilities proportional to their scores $\{w^{(i)}\} \cup \{\hat{w}^{(m)}\}$;

7: **return** the resulting set as $W_t = \{\hat{s}^{(j)}_{1:t}\}_{j=1}^{N}$ and all generated complete workflows $\mathcal{P}_t^{complete} = \{s^{(i)}_{1:t}, \tilde{s}^{(i,k)}_{t+1:T}\} \cup \{\hat{s}^{(m)}_{1:T}\}$

Theorem 2 formalizes an "exploration without drift" property of BayesFlow with refinement. We show that refinement increasing diversity and encouraging broader exploration of the workflow space, yet it does so in a controlled manner. The TV bound quantifies this control: if each refinement step perturbs the induced complete workflow distribution by at most ε_t, then the refined prefix flow stays close to the baseline flow, with deviation growing only additively across steps. In particular, under a uniform bound ε, we obtain $\mathrm{TV}(\pi_T, q) \leq (T-1)\varepsilon$, meaning that refinement can expand the support of sampled workflows while remaining provably close to the target distribution. See Appendix D for proof.

[1] Standard assumptions for SMC are taken to hold; see Doucet et al. (2000).

Algorithm 2 BayesFlow

Require: Rollouts count K; number of partial workflows in each round N, number of refined workflows M, maximum step count T

Initialize *empty* prefix pool $W_0 \leftarrow \varnothing$
$t \leftarrow 0$
while $t \leq T$ **do**
    Run StepUpdate(W_t, K, M), and get W_{t+1} and $\mathcal{P}_{t+1}^{complete}$
    $t \leftarrow t + 1$
end while
$\mathcal{C} \leftarrow \bigcup_{t=1}^{T} \mathcal{P}_t^{complete}$
return $\mathcal{C}$, $\arg\max_{s \in \mathcal{C}} R(s)$

Theorem 2. *Consider Algorithm 1 with the refiner enabled (i.e., $M > 0$) for T steps. Let π_T denote the limiting distribution over complete workflows. Then π_T admits the mixture decomposition $\pi_T = \alpha\, q + (1-\alpha)\, \pi_{\text{rest}}$ for some $\alpha \in (0, 1]$, where q is the target distribution, and π_{rest} is a distribution supported on workflows that undergo refinement at least once.*

Moreover, let μ_t^0 and μ_t^1 denote the distributions over prefixes at step t for Algorithm 1 without and with the refiner, respectively. Assume that at each step t, for any prefix law, the distributions over complete workflows before and after refinement differ by at most ε_t in total variation distance. Then the prefix drift is bounded as

$$\mathrm{TV}(\mu_t^1, \mu_t^0) \;\le\; \sum_{k=1}^{t-1} \varepsilon_k, \qquad t = 1, \ldots, T.$$

In particular, if $\varepsilon_k \le \varepsilon$ for all k, then $\mathrm{TV}(\mu_t^1, \mu_t^0) \le (t-1)\varepsilon$, and hence $\mathrm{TV}(\pi_T, q) \le (T-1)\varepsilon$.

5 BayesFlow

Building on Bayesian workflow generation, we instantiate **BayesFlow**, a flexible, inference-only method for automatic workflow generation.

Design space Inspired by the design space in Nie et al. (2025), BayesFlow avoids the predefined agentic modules used in previous work (Zhang et al., 2024a; Hu et al., 2024; Zhang et al., 2025) (e.g., Ensemble, Revise, Programmer), which can constrain the design space of the workflow. Instead, we specify only a lightweight workflow interface and expose generic helper primitives—LLM calls and code execution—leaving prompts, sampling hyperparameters, and control logic entirely to the meta optimizer, thereby encouraging innovation and adaptability. Concretely, we provide two helper functions: `chat_completion`, which is a single LLM call given a system role, instructions, and a message history to return one or more candidate continuations; and `exec_code`, which executes generated code against public unit tests and reports pass/fail outcomes with error diagnostics. The `exec_code` is only provided on coding datasets with public test sets. This minimal interface is model-agnostic, easy to extend, and decouples high-level workflow design from implementation details. See Appendix G for detailed signatures. We also present the generated workflows in Appendix H.

Prior distribution We instantiate the prior distribution p of the meta optimizer LLM via a carefully designed prompt that instructs the model to produce workflow code in a fixed rule-based format. The prompt requires the agent to annotate each substantive step. If the resulting workflow fails to execute, we trigger a self-correction routine: the meta LLM is re-invoked with the same prompt augmented by the captured error log and asked to emit a corrected, runnable version. In this self-correction stage, we include the explicit prompt: *"Do not use any try-except blocks. Fix the root cause rather than catching it."* This mitigates reward hacking through exception handling and improves the success rate of self-correction.

Refiner implementation. Our refiner is inspired by the selection logic of MCTS in AFlow (Zhang et al., 2024a) but uses a simpler pool-based procedure. From the current pool of complete workflows with associated validation rewards, we select the top-C candidates (C=3 in all experiments), then sample one workflow with probabilities given by a softmax over rewards with temperature 0.1. We then pass the selected workflow to the meta optimizer LLM with the text gradient procedure of Yuksekgonul et al. (2024). We design a prompt that requests one consequential edit while preserving the prescribed code format. The edited workflow is inserted back into the pool, and the process repeats for the next refinement step.

BayesFlow algorithm We note that, in expectation, the mean reward for complete workflows does not decrease between steps, so the optimum population would be reached at the final step T. However,

in practice, under finite sampling with a limited number of workflows per step, the highest-scoring workflow may appear earlier. Therefore, instead of restricting selection to the last step T, we maintain a global pool of all candidates generated between steps and return as the final output the single workflow with the highest validation score observed throughout the run, as shown in Algorithm 2.

6 Experiments

6.1 Experiments setup

Datasets and models We benchmark BayesFlow on six datasets across three domains, i.e., math reasoning: MATH (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021), question answering: DROP (Dua et al., 2019), HotpotQA (Yang et al., 2018), and expert knowledge: GPQA (Rein et al., 2024), MMLU-Pro (Wang et al., 2024a). Workflows are drafted with Claude 3.5-Sonnet (Anthropic, 2024) (meta optimizer LLM) and executed with Claude 3.7-Sonnet (Anthropic, 2025) or Qwen2.5-7B-Instruct (Team, 2024). All models are accessed via APIs. We set the temperature to 0 for the meta optimizer LLM. For MATH, we follow (Hong et al., 2023) and select 617 level-5 problems spanning four categories: Combinatorics & Probability, Number Theory, Pre-algebra, and Pre-calculus. For datasets with more than 1000 examples, we randomly sample 1000 instances. Then we adopt the standard 1:4 validation–test split used in prior work. All datasets are distributed with licenses that permit academic research use.

Metrics For GSM8K and MATH we report the solve rate, the percentage of problems fully solved. For HotpotQA and DROP we use F1 score. For MMLU-Pro and GPQA, we measure accuracy, checking if the final selected option is correct.

Baselines We evaluated six baselines in experiments. Three are prompt-based methods: (i) zero-shot prompts (IO), (ii) zero-shot chain-of-thought (CoT) (Wei et al., 2022a), and (iii) CoT with self-consistency (CoT-SC) (Wang et al., 2022a), where each question is answered three times. The prompts are from the official repository of ADAS (Hu et al., 2024). We also compare against three automatic workflow generation baselines: ADAS, AFlow (Zhang et al., 2024a) and MaAS (Zhang et al., 2025). For ADAS, we follow the original protocol with 30 iterations; for AFlow, we run 20 iterations and evaluate each generated workflow five times on the validation set; and for MaAS, we adopt the default hyperparameters from the official repository. We base our implementations on the official repositories where feasible. For datasets not originally supported, we extend the baselines by revising dataset-specific prompts while keeping the remaining components unchanged.

6.2 Main results

Each experiment was repeated three times with different random seeds and each selected workflow is evaluated on the test set five times (more hyperparameters shown in Appendix E). Table 1 reports the mean and standard deviation of accuracy. For the closed-source model, Claude 3.7-sonnet, BayesFlow achieves the highest average accuracy (80.8 %) across the six benchmarks, outperforming the next-best method (AFlow, 76.2 %) by 4.6 percentage points. Its advantage is most pronounced on the challenging MATH and GPQA datasets, where it exceeds AFlow by 9.3 percentage points and 3.9 percentage points, respectively. Standard deviations for **BayesFlow** are uniformly modest across the data sets, indicating a more stable behavior compared to other workflow baselines. For example, AFlow's high variance on HotpotQA is due to occasional failures to output effective workflows. BayesFlow also shows generalizability to the smaller-sized model, Qwen 2.5-7B-Instruct, achieving the top average accuracy (63.4%), with a 3.5 percentage points margin over AFlow (59.9%) in the six tested datasets.

In addition to the six main datasets, we evaluated AFlow and BayesFlow on the coding benchmark MBPP (Austin et al., 2021) using the Claude-3.7-Sonnet executor. Because code generation requires stricter interfaces, we modify the optimizer's meta-prompt to (i) expose a helper function `exec_code` that executes the function on public test cases and (ii) enforce an exact output format to minimize syntax errors. Notably, BayesFlow achieves pass@1 at 85.0% in MBPP, which largely exceeds AFlow's 75.5%, further demonstrating the superior performance of BayesFlow[2].

The final MBPP workflow consists of six test-centered steps (see the complete workflow in Figure 5). After generating candidate solutions, it performs early exit if all public tests pass; otherwise, it triggers self-correction guided by captured error logs, produces a revised implementation, and

[2]We had issues reproducing results for the other two baselines, thus omit MBPP from Table 1 now.

Method	GSM8K	MATH	HotpotQA	DROP	MMLU-Pro	GPQA	Average
Optimizer: Claude-3.5-Sonnet, Executor: Claude-3.7-Sonnet							
IO	90.0$_{\pm 0.2}$	40.2$_{\pm 0.2}$	57.4$_{\pm 12.0}$	75.1$_{\pm 0.1}$	57.8$_{\pm 0.7}$	44.8$_{\pm 0.5}$	60.9$_{\pm 2.0}$
CoT	95.2$_{\pm 0.2}$	43.2$_{\pm 0.4}$	45.1$_{\pm 17.9}$	81.5$_{\pm 0.3}$	77.4$_{\pm 1.4}$	65.6$_{\pm 1.7}$	68.0$_{\pm 3.0}$
CoT-SC	96.0$_{\pm 0.2}$	45.0$_{\pm 0.2}$	21.8$_{\pm 0.0}$	49.9$_{\pm 0.5}$	24.3$_{\pm 0.7}$	65.8$_{\pm 1.2}$	50.5$_{\pm 0.3}$
ADAS	96.0$_{\pm 0.2}$	44.1$_{\pm 2.4}$	73.3$_{\pm 3.4}$	81.2$_{\pm 1.2}$	80.1$_{\pm 1.1}$	64.0$_{\pm 1.5}$	73.1$_{\pm 0.8}$
MaAS	96.4$_{\pm 0.2}$	41.3$_{\pm 0.5}$	76.3$_{\pm 0.6}$	84.2$_{\pm 0.4}$	82.0$_{\pm 0.6}$	55.2$_{\pm 2.0}$	72.6$_{\pm 0.4}$
AFlow	**96.5**$_{\pm 0.2}$	60.1$_{\pm 1.5}$	63.7$_{\pm 18.4}^{\dagger}$	89.2$_{\pm 0.3}$	**82.3**$_{\pm 0.7}$	65.3$_{\pm 1.7}$	76.2$_{\pm 3.1}$
BayesFlow (Ours)	96.0$_{\pm 0.3}$	**69.4**$_{\pm 4.1}$	**77.5**$_{\pm 0.7}$	**90.8**$_{\pm 1.5}$	81.8$_{\pm 0.8}$	**69.2**$_{\pm 1.8}$	**80.8**$_{\pm 0.8}$
Optimizer: Claude-3.5-Sonnet, Executor: Qwen 2.5-7B-Instruct							
IO	24.7$_{\pm 0.2}$	16.6$_{\pm 0.1}$	21.5$_{\pm 0.1}$	27.7$_{\pm 0.1}$	41.1$_{\pm 0.1}$	33.1$_{\pm 0.4}$	27.4$_{\pm 0.1}$
CoT	**90.6**$_{\pm 0.3}$	17.6$_{\pm 0.2}$	21.9$_{\pm 0.4}$	32.2$_{\pm 0.3}$	53.9$_{\pm 0.4}$	34.4$_{\pm 1.4}$	41.8$_{\pm 0.3}$
CoT-SC	90.0$_{\pm 0.3}$	15.6$_{\pm 0.1}$	27.1$_{\pm 0.7}$	12.3$_{\pm 0.1}$	42.3$_{\pm 1.5}$	34.0$_{\pm 0.7}$	36.9$_{\pm 0.3}$
ADAS	86.9$_{\pm 0.5}$	21.6$_{\pm 1.7}$	68.3$_{\pm 0.8}$	75.1$_{\pm 1.6}$	53.4$_{\pm 1.0}$	34.1$_{\pm 2.2}$	56.6$_{\pm 0.6}$
MaAS	88.3$_{\pm 0.1}$	47.1$_{\pm 0.2}$	56.3$_{\pm 0.1}$	48.7$_{\pm 0.8}$	54.6$_{\pm 0.2}$	34.6$_{\pm 0.5}$	54.9$_{\pm 0.2}$
AFlow	90.4$_{\pm 1.1}$	47.0$_{\pm 0.7}$	67.6$_{\pm 0.7}$	69.9$_{\pm 1.3}$	51.0$_{\pm 3.6}$	33.5$_{\pm 1.7}$	59.9$_{\pm 0.7}$
BayesFlow (Ours)	**90.6**$_{\pm 0.6}$	**49.1**$_{\pm 2.7}$	**70.1**$_{\pm 1.4}$	**78.6**$_{\pm 1.7}$	**56.2**$_{\pm 1.3}$	**35.7**$_{\pm 2.2}$	**63.4**$_{\pm 0.7}$

Table 1: Accuracy (mean $\pm$ std.) of different methods across six datasets. We compare with both prompting baselines and agentic workflow methods. The best result for each dataset is in **bold**, and the second-best is underlined. The high variance† on AFlow for HotpotQA stems from occasional failures.

re-tests to verify fixes. A final fallback attempt provides an additional chance for recovery. Taken together, these steps form a reasonable and interpretable procedure that explicitly leverages public tests to drive refinement.

6.3 Experiment analysis

BayesFlow is more token-efficient To further demonstrate the strength of BayesFlow, we compare it against AFlow, the second-best approach in terms of final accuracy in our experiments. As shown in Table 2, BayesFlow is more efficient than AFlow in most settings, achieving higher accuracy at lower cost. We emphasize that while Monte Carlo rollouts are generally more token-intensive, in automatic workflow generation the total token cost is driven not only by how many workflows are produced during training, but also by the cost of evaluating each workflow. This evaluation typically requires running the executor LLM over the full validation set, which can dominate overall usage.

We attribute BayesFlow's efficiency to two key engineering choices. First, rather than relying on heavyweight, pre-defined operators such as LLM debate or self-consistency, we use a lightweight helper interface, substantially reducing overhead. Second, AFlow evaluates each workflow multiple times to obtain a stable estimate, whereas our im-

Metric	**AFlow**	**BayesFlow**
MMLU-Pro, Optimizer: Claude Haiku 4.5		
Input tokens	38,557,324	**5,290,460**
Output tokens	10,062,070	**6,585,667**
Accuracy	0.432	**0.571**
DROP, Optimizer: Claude Haiku 4.5		
Input tokens	79,170,565	**14,527,334**
Output tokens	15,907,446	**3,644,159**
F1	0.735	**0.780**
MMLU-Pro, Optimizer: Claude Sonnet 3.7		
Input tokens	63,200,652	**45,630,330**
Output tokens	**7,114,185**	32,351,702
Accuracy	0.535	**0.573**
DROP, Optimizer: Claude Sonnet 3.7		
Input tokens	114,158,960	**26,616,937**
Output tokens	19,722,922	**9,447,793**
F1	0.755	**0.765**

Table 2: **AFlow vs. BayesFlow: optimizer-side token usage and task performance.** We report optimizer token usage during workflow search using Claude Haiku 4.5 or Claude Sonnet 3.7 as the optimizer LLM and Qwen2.5-7B-Instruct as the executor LLM. **Bold** marks the better value within each dataset–optimizer block: lower token usage and higher task performance.

Dataset	N=5	N=10	N=15
GSM8K	96.3$_{\pm 0.2}$	96.0$_{\pm 0.3}$	96.5$_{\pm 0.3}$
MATH	70.4$_{\pm 1.9}$	69.4$_{\pm 4.1}$	71.1$_{\pm 0.9}$
HotpotQA	77.6$_{\pm 0.1}$	77.5$_{\pm 0.7}$	77.0$_{\pm 0.7}$
DROP	90.8$_{\pm 0.2}$	90.8$_{\pm 1.5}$	90.7$_{\pm 0.3}$
MMLU-Pro	82.8$_{\pm 0.3}$	81.8$_{\pm 0.8}$	82.1$_{\pm 0.4}$
GPQA	69.7$_{\pm 1.6}$	69.2$_{\pm 1.8}$	70.4$_{\pm 1.2}$

Table 3: Ablation over the number of partial workflows per round (N) with a fixed per round budget. Numbers are mean $\pm$ standard derivation.

plementation evaluates each workflow only once.

In terms of wall-clock runtime, BayesFlow can be further accelerated when sufficient API rate limits are available, since our rollouts can be generated and evaluated fully in parallel. In contrast, most search-based algorithms rely on sequential optimization steps that remain inherently serial, and thus cannot be sped up to the same extent even under sufficient rate limits.

BayesFlow is robust to the number of look-ahead rollouts. We study the effect of the ratio between the number of look-ahead rollouts N and refined workflows M. To isolate the ratio,we fix the proposal budget $N + M$ per round. In the main experiments, the budget is 20, and here we vary $N \in \{5, 10, 15\}$, adjusting M accordingly. The results in Table 3 using Claude-3.7-Sonnet executor show that BayesFlow is robust to the choice of N. This shows that our implementation is stable across exploration–exploitation settings. We also show that the expected reward almost monotonically increases over rounds in Appendix F which supports the effectiveness of our look-ahead mechanism.

BayesFlow demonstrates stronger workflow-level inference-time scaling. Unlike the standard inference-time scaling which repeatedly samples answers from a single LLM, we scale *number of workflows* and aggregate their outputs. We report three simple metrics: **Best@L** counts an example correct if at least one of the L workflows produces the right answer; **Mean@L** averages the fraction of correct answers among the L outputs; and **Majority@L** takes a vote on the L answers and is correct only if the most frequent answer matches the ground truth.

In this experiment, we select the top L workflows generated by both AFlow and BayesFlow. Workflows are produced using Claude-3.5-Sonnet as the optimizer LLM and Claude-3.7-Sonnet as the executor model. To demonstrate transferability, we then evaluated these fixed workflows with Claude-3.5-Sonnet on a randomly sampled 100-example subset of MATH. Across $L \in \{1, 2, 4, 8\}$, BayesFlow consistently outperforms AFlow, with especially pronounced gains in Best@L, as shown in Table 4. This pattern indicates that BayesFlow does not just find a single high-quality workflow; it produces a set of high-quality and diverse workflows, increasing the chance that at least one solves each problem.

L	Majority@L	Mean@L	Best@L
1	**74**/45	**74**/45	**74**/45
2	**74**/45	**72.5**/27	**80**/46
4	**74**/46	**74**/30.75	**85**/65
8	**78**/69	**70.88**/47.12	**86**/81

Table 4: Accuracy (%) shown as **BayesFlow**/AFlow, exemplified on MATH. The better value in each cell is bold.

7 Conclusion

We introduced Bayesian workflow generation, a principled framework that casts workflow synthesis as particle-based posterior sampling with look-ahead rollouts and text-gradient refinement. Across seven benchmarks and two executor families, BayesFlow delivers systematic gains: Surpassing the strongest baseline by 4.6 percentage points on Claude-3.7-Sonnet and by 3.5 percentage points on Qwen-2.5-7B on average. Taken together, these findings indicate that BWG is a promising direction compared to optimization-centric approaches. By emphasizing posterior sampling with lightweight refinement rather than heavy bespoke optimization, BayesFlow achieves competitive accuracy gains.

Limitations

The expressivity of BayesFlow partly stems from parallel look-ahead rollouts, which increase inference-time cost. A promising direction is a principled controller that dynamically tunes hyperparameters from on-line signals to reduce computation without hurting accuracy. Our current implementation prunes most rollout traces without long-term reuse. A stronger memory mechanism that retains and prioritizes high-value trajectories could better exploit complete look-ahead evidence and eliminate redundant sampling. Our evaluation

focuses on two executor families in seven benchmarks but can be extended further; future work will extend the analysis to recent high-reasoning models and broader settings, e.g., multimodality or agents, to more rigorously assess the portability of sampling-based design and its budget–accuracy trade-offs.

Ethical considerations

We will release the full codebase to enable reproduction of our results. All datasets used are publicly available under their respective licenses; no human-subjects data, personally identifiable information, or proprietary assets are involved. We used AI tools only for minor language refinement (grammar, clarity, and formatting) during final manuscript preparation. No AI was used for idea generation, analysis, or text drafting, and all authors take full responsibility for the content.

References

Anthropic. 2024. Introducing Claude 3.5 Sonnet. Accessed 8 Aug 2025.

Anthropic. 2025. Claude 3.7 sonnet and claude code. Accessed 8 Aug 2025.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

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

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, and 1 others. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*.

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

Arnaud Doucet, Nando De Freitas, and Neil Gordon. 2001. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer.

Arnaud Doucet, Simon Godsill, and Christophe Andrieu. 2000. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and computing*, 10(3):197–208.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*.

Yilun Du and Igor Mordatch. 2019. Implicit generation and modeling with energy based models. *Advances in neural information processing systems*, 32.

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

Gonçalo Faria, Sweta Agrawal, António Farinhas, Ricardo Rei, José de Souza, and André Martins. 2024. QUEST: Quality-aware metropolis-hastings sampling for machine translation. *Advances in Neural Information Processing Systems*, 37:89042–89068.

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

Xingang Guo, Fangxu Yu, Huan Zhang, Lianhui Qin, and Bin Hu. 2024. Cold-attack: Jailbreaking llms with stealthiness and controllability. *arXiv preprint arXiv:2402.08679*.

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

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiangtao Lu, and 6 others. 2024a. Data interpreter: An LLM agent for data science. *CoRR*, abs/2402.18679.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024b. Metagpt: Meta programming for A multi-agent collaborative framework. In *ICLR*. OpenReview.net.

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

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

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *CoRR*, abs/2312.13010.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. *arXiv preprint arXiv:2306.02561*.

Adam Johansen. 2009. A tutorial on particle filtering and smoothing: Fifteen years later.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, and 1 others. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Will LeVine, Benjamin Pikus, Anthony Chen, and Sean Hendryx. 2023. A Baseline Analysis of Reward Models' Ability To Accurately Analyze Foundation Models Under Distribution Shift. *arXiv preprint arXiv:2311.14743*.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to SQL: are we fully ready? *Proc. VLDB Endow.*, 17(11):3318–3331.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, Fengli Xu, and Yong Li. 2025. Agentswift: Efficient llm agent design via value-guided hierarchical search. *arXiv preprint arXiv:2506.06017*.

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*.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, and 1 others. 2025. Syntactic and semantic control of large language models via sequential monte carlo. *arXiv preprint arXiv:2504.13139*.

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

Skander Moalla, Andrea Miele, Daniil Pyatko, Razvan Pascanu, and Caglar Gulcehre. 2024. No representation, no trust: connecting representation, collapse, and trust issues in ppo. *Advances in Neural Information Processing Systems*, 37:69652–69699.

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

Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolò Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, Renqian Luo, Scott Mayer McKinney, Robert Osazuwa Ness, Hoifung Poon, Tao Qin, Naoto Usuyama, Chris White, and Eric Horvitz. 2023. Can generalist foundation models outcompete special-purpose tuning? case study in medicine. *CoRR*, abs/2311.16452.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. 2024. Benchmarking agentic workflow generation. *arXiv preprint arXiv:2410.07869*.

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. 2022. Cold decoding: Energy-based constrained text generation with langevin dynamics. *Advances in Neural Information Processing Systems*, 35:9538–9551.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2024. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*.

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

Paul B Rohrbach and Robert L Jack. 2022. Convergence of random-weight sequential Monte Carlo methods. *arXiv preprint arXiv:2208.12108*.

Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, and 1 others. 2024. Archon: An architecture search framework for inference-time techniques. *arXiv preprint arXiv:2409.15254*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

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

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.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471.

Nan Tang, Chenyu Yang, Ju Fan, Lei Cao, Yuyu Luo, and Alon Halevy. 2023. Verifai: verified generative ai. *arXiv preprint arXiv:2307.02796*.

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

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. 2023b. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*.

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

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

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*.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, and 1 others. 2024a. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290.

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

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

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

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024a. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024b. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. Haichart: Human and AI paired visualization system. *Proc. VLDB Endow.*, 17(11):3178–3191.

Peiran Xu, Zhuohao Li, Xiaoying Xing, Guannan Zhang, Debiao Li, and Kunyu Shi. 2025. Hybrid Reward Normalization for Process-supervised Non-verifiable Agentic Tasks. *arXiv preprint arXiv:2509.25598*.

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

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

Yilin Ye, Jianing Hao, Yihan Hou, Zhan Wang, Shishi Xiao, Yuyu Luo, and Wei Zeng. 2024. Generative AI for visualization: State of the art and future directions. *Vis. Informatics*, 8(1):43–66.

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

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*.

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

Jiayi Zhang, Chuang Zhao, Yihan Zhao, Zhaoyang Yu, Ming He, and Jianping Fan. 2024b. Mobileexperts: A dynamic tool-enabled agent team in mobile devices. *CoRR*, abs/2407.03913.

Qinlin Zhao, Jindong Wang, Yixuan Zhang, Yiqiao Jin, Kaijie Zhu, Hao Chen, and Xing Xie. 2023. Competeai: Understanding the competition behaviors in large language model-based agents. *CoRR*.

Qihuang Zhong, Kang Wang, Ziyang Xu, Juhua Liu, Liang Ding, Bo Du, and Dacheng Tao. 2024. Achieving> 97% on gsm8k: Deeply understanding the problems makes llms perfect reasoners. *arXiv preprint arXiv:2404.14963*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*.

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

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

A Connection to reinforcement learning and inference scaling law

In reinforcement learning, a typical objective for an unknown distribution q is $\max_q \mathbb{E}_q\big[R(s)\big] - \mathrm{KL}\big(q(s) \,\|\, p(s)\big)$ (Schulman et al., 2017; Ouyang et al., 2022) which is to maximize the reward while maintaining the KL divergence with respect to the prior distribution to mitigate model collapse (Moalla et al., 2024) and reward hacking (Skalse et al., 2022). One can show that the optimizer is exactly the target distribution in equation 1; see Theorem 3 for a detailed proof. On the inference side, weighted majority voting (Wu et al., 2024b) for LLM reasoning has been shown to be a scalable and effective way to improve performance; in essence, it corresponds to drawing and re-weighting samples in accordance with the same goal. These insights naturally lead to a **principled** posterior sampling view of workflow generation.

Optimal solution in Reinforcement learning as posterior sampling.

Theorem 3. *Let $p(s_{1:T})$ be a prior distribution, and let R be a measurable reward function. Consider the optimization over distributions q:*

$$\max_q \; \mathcal{J}(q) \;:=\; \mathbb{E}_q\big[R(s_{1:T})\big] \;-\; \mathrm{KL}(q \,\|\, p)\,,$$

where $\mathrm{KL}(q\|p) = \mathbb{E}_q\Big[\log \frac{q}{p}\Big]$. *Then the unique maximizer is*

$$q^\star(s_{1:T}) \;=\; \frac{p(s_{1:T})\,\exp\big(R(s_{1:T})\big)}{Z}$$

where

$$Z \;=\; \mathbb{E}_p[\exp\big(R(s_{1:T})\big)]$$

Proof. Introduce a Lagrange multiplier λ for the normalization constraint $\int q = 1$ and form

$$\mathcal{L}(q,\lambda) = \int q(s)\,R(s)\,ds \;-\; \int q(s)\log\frac{q(s)}{p(s)}\,ds + \lambda\Big(1 - \int q(s)\,ds\Big)\,.$$

where we write $s \equiv s_{1:T}$ for brevity. Taking the first variation w.r.t. q and setting it to zero yields, for almost every s,

$$R(s) - \Big(\log q(s) - \log p(s) + 1\Big) - \lambda = 0$$

Hence, we have

$$\log q(s) \;=\; R(s) + \log p(s) - 1 - \lambda,$$

Exponentiating gives

$$q(s) \propto p(s)\, e^{R(s)}.$$

□

Weighted majority voting as posterior sampling. Let the target posterior over workflows be $q(s) \propto p(s)\exp(R(s))$, where p is a meta-optimizer prior and R is a reward oracle. Draw N i.i.d. samples $\{s^{(i)}\}_{i=1}^{N} \sim p$ and form importance weights $w_i \propto \exp(R(s^{(i)}))$, normalized so that $\sum_{i=1}^{N} w_i = 1$. Define the weighted empirical measure

$$\hat{q}_N = \sum_{i=1}^{N} w_i\, \delta_{s^{(i)}}.$$

If $\mathbb{E}_p\left[e^{R(s)}\right] < \infty$, then by the law of large numbers for the sampling of self-normalized importance, as N grows, the empirical distribution $\hat{q}_N$ concentrates on the target q, and sampling by multinomial resampling indices with probabilities $\{w_i\}$ yields draws asymptotically distributed as q.

B Extended Related Work

B.1 Agentic Workflows

Agentic workflows represent a fundamental paradigm in LLM applications, consisting of structured sequences of LLM invocations designed to solve complex tasks through predefined processes (Hong et al., 2024b; Zhang et al., 2024b). Unlike autonomous agents that make dynamic decisions based on environmental feedback, agentic workflows operate through static and predetermined logical sequences (or with conditionals) that can be systematically designed and refined (Zhuge et al., 2023; Wang et al., 2023a). This structured approach enables workflows to leverage existing human domain expertise and iterative refinement processes, making them particularly suitable for tasks where consistent, reproducible outcomes are desired. The landscape of agentic workflows can be broadly categorized into general and domain-specific approaches.

Generalist Agentic Workflows: General workflows focus on universal problem-solving methodologies that can be applied across diverse domains. Notable examples include Chain-of-Thought prompting (Wei et al., 2022b), which guides LLMs through step-by-step reasoning processes; Self-Consistency (Wang et al., 2022b), which generates multiple reasoning paths and selects the most consistent answer; Self-Refine (Madaan et al., 2023), which iteratively improves outputs through self-critique; and MultiPersona approaches (Wang et al., 2024b), which leverage different perspectives to enhance problem-solving capabilities. Additionally, MedPrompt (Nori et al., 2023) demonstrates how structured prompting strategies can achieve expert-level performance in specialized domains through carefully designed workflow components.

Domain-Specific Agentic Workflows: Domain-specific workflows are tailored to address the unique challenges and requirements of particular application areas. In code generation, workflows often incorporate testing, debugging, and iterative refinement processes, with notable examples including AlphaCodium (Ridnik et al., 2024), which transitions from prompt engineering to flow engineering, and AgentCoder (Huang et al., 2023), which employs multi-agent collaboration for iterative code optimization. Data analysis workflows typically combine data exploration, statistical analysis, and visualization components (Hong et al., 2024a; Xie et al., 2024; Li et al., 2024a; Ye et al., 2024). Mathematical problem-solving workflows emphasize systematic reasoning, calculation verification, and solution validation (Zhong et al., 2024). Question-answering workflows often integrate information retrieval, fact verification, and answer synthesis processes, utilizing techniques such as Language Agent Tree Search to unify reasoning, acting, and planning (Zhou et al., 2024). While these manually designed workflows have demonstrated significant effectiveness in their respective domains, their development requires substantial human expertise and domain knowledge, limiting their scalability and adaptability to new problem areas.

B.2 Automatic Agentic Workflow Generation

The manual design of effective agentic workflows requires substantial human expertise and domain knowledge, creating a significant bottleneck for the widespread adoption of LLM-based systems across diverse applications. To address this limitation, recent research has focused on automating the discovery and optimization of agentic workflows, aiming to reduce human intervention while maintaining or improving performance. This emerging field encompasses various approaches that can be broadly categorized into three dimensions: automated prompt optimization (Fernando et al., 2023;

Wang et al., 2023b; Yang et al., 2023; Yuksekgonul et al., 2024), hyperparameter tuning (Saad-Falcon et al., 2024), and complete workflow structure optimization (Li et al., 2024b; Hu et al., 2024; Zhuge et al., 2024). The fundamental challenge lies in effectively navigating the vast search space of possible workflow configurations while maintaining computational efficiency and ensuring the generated workflows generalize across different tasks and domains. These automated approaches can be further distinguished by their optimization methodology: training free methods that rely on iterative refinement through LLM based search, and training based methods that learn to generate optimal workflows through supervised or reinforcement learning techniques.

Training Free Approaches: Training free automated workflow optimization methods leverage the inherent capabilities of pre trained LLMs to iteratively improve workflow designs without requiring parameter updates. Early approaches have primarily focused on optimizing individual components within fixed workflow structures. DSPy (Khattab et al., 2023) provides a programming model that separates workflow logic from prompt engineering, allowing for systematic optimization of prompting strategies while maintaining workflow structure. TextGrad (Yuksekgonul et al., 2024) treats prompt optimization as a form of automatic differentiation through text, enabling gradient-based optimization of prompting strategies. PromptBreeder (Fernando et al., 2023) introduces self-referential improvement mechanisms that evolve prompts through iterative refinement processes. These component-level optimization methods have demonstrated significant improvements in reasoning performance, but remain constrained by their focus on predefined workflow templates and limited exploration of alternative structural configurations.

More recent work has attempted to address the limitations of component-level optimization by exploring complete workflow structure generation. ADAS (Hu et al., 2024) represents workflows using code structures and employs linear heuristic search algorithms to discover effective configurations, though it faces efficiency challenges due to the accumulation of irrelevant information and coarse workflow storage mechanisms. GPTSwarm (Zhuge et al., 2024) uses graph-based representations combined with reinforcement learning techniques for workflow optimization, but struggles with conditional states due to the limitations of graph structures in expressing complex logical relationships. AFLOW addresses these representational limitations by reformulating workflow optimization as a Monte Carlo Tree Search problem over code-represented workflows, where LLM-invoking nodes are connected by flexible edges that can express complex relationships including conditional logic, loops, and parallel execution (Zhang et al., 2024a). However, training free methods fundamentally suffer from their inability to effectively leverage historical optimization data and environmental feedback, often performing no better than random workflow sampling and struggling with convergence to optimal solutions within limited iterations.

Training Based Approaches: Training based automated workflow generation and optimization methods address the limitations of training-free approaches by learning to generate optimal workflows through explicit parameter optimization. These methods can accumulate knowledge from previous optimization attempts and adapt their generation strategies based on empirical feedback. Weak-for-Strong Harnessing (W4S) introduces a novel paradigm where a smaller, trainable meta-agent is optimized via reinforcement learning to design workflows that effectively harness stronger language models (Nie et al., 2025). The approach formulates workflow optimization as a multi-turn Markov Decision Process and employs Reinforcement Learning for Agentic Workflow Optimization (RLAO), enabling the meta-agent to learn from both successful and failed workflow attempts through reward-weighted regression. This training based approach demonstrates superior performance compared to training free methods while maintaining cost efficiency by training smaller models rather than fine-tuning large ones.

ScoreFlow takes a different training based approach by leveraging preference optimization techniques adapted for workflow generation (Wang et al., 2025). The framework introduces Score DPO, a novel variant of Direct Preference Optimization that incorporates quantitative evaluation scores directly into the optimization process, addressing the variance and inaccuracies inherent in preference data that can slow convergence in standard preference optimization methods. ScoreFlow generates multiple workflows per task, evaluates their performance using quantitative metrics, and uses these scores to construct preference pairs for training the workflow generator through gradient-

based optimization. This approach enables adaptive workflow generation where different workflows can be optimized for different types of problems, improving scalability compared to methods that optimize a single workflow for entire task sets. Both training based approaches demonstrate significant advantages over their training free counterparts, achieving superior performance while enabling smaller models to outperform larger ones through optimized workflow design, highlighting the potential of learned optimization strategies in automated agentic system development.

C Quick proof of Theorem 1

Proof. With $M{=}0$, Algorithm 1 is an SMC sampler for the target $p(s_{1:T})\exp[R(s_{1:T})]$. In step t, the mutation kernel is $p(s_t|s_{1:t-1})$ and the weight is $\mathbb{E}_{p(s_{t+1:T}|s_{1:t})}\exp(R(s_{1:T}))$. BWG uses the Monte Carlo estimator $w^{(i)} = \frac{1}{K}\sum_{k=1}^{K}\exp\left(R(s_{1:t},\tilde{s}^{(k)}_{t+1:T})\right)$ with $\tilde{s}^{(k)}_{t+1:T} \sim p(\cdot|s_{1:t})$ i.i.d. Hence, the weight $w^{(i)}$ is positive and unbiased. Then we can apply Theorem 3.5 in Rohrbach and Jack (2022) to prove the convergence of the equivalence distribution. To prove the inequality, let $Z(\beta) = \mathbb{E}_p[\exp(\beta R)]$ and the normalized distribution $q_\beta = \exp(\beta R)p/Z(\beta)$. Then $\frac{d}{d\beta}\mathbb{E}_{q_\beta}[R] \geq 0$. Thus $\mathbb{E}_{q_\beta}[R]$ is non-decreasing in β, and for $\beta > 0$, $\mathbb{E}_{q_\beta}[R] \geq \mathbb{E}_{q_0}[R] = \mathbb{E}_p[R]$. □

D Proof of Theorem 2

Proof. For each final workflow, either it was never selected from a refined candidate at any step, or it was selected from a refined candidate at least once. Conditioning on the event that refinement is never selected, the resampling procedure reduces to the no-refiner algorithm, so the conditional distribution is exactly q. Therefore, by the law of total probability, the final law admits the decomposition $\pi_T = \alpha q + (1-\alpha)\pi_{\text{rest}}$.

Let μ_t^0 and μ_t^1 be the prefix distributions at step t without or with the refiner. Fix a step t. Consider the one-step update that maps a prefix distribution to the next-prefix distribution. Introduce an intermediate distribution $\widetilde{\mu}_{t+1}$ defined as: start from prefixes distributed as μ_t^1, but perform the *no-refiner* update for one step. Then by the triangle inequality, $\mathrm{TV}(\mu_{t+1}^1, \mu_{t+1}^0) \leq \mathrm{TV}(\mu_{t+1}^1, \widetilde{\mu}_{t+1}) + \mathrm{TV}(\widetilde{\mu}_{t+1}, \mu_{t+1}^0)$.

For the first term, by assumption, for any prefix law the distributions over *complete* workflows before vs. after refinement differ by at most ε_t in total variation. The next-prefix distribution is obtained from a complete-workflow distribution by applying the projection map that keeps only the first $t{+}1$ steps. By pushforward contraction of total variation, $\mathrm{TV}(\mu_{t+1}^1, \widetilde{\mu}_{t+1}) \leq \varepsilon_t$.

For the second term, $\widetilde{\mu}_{t+1}$ and μ_{t+1}^0 are obtained by applying the *same* randomized update rule to inputs μ_t^1 and μ_t^0, respectively. Total variation cannot increase under applying the same post-processing, hence $\mathrm{TV}(\widetilde{\mu}_{t+1}, \mu_{t+1}^0) \leq \mathrm{TV}(\mu_t^1, \mu_t^0)$. Combining the two bounds yields the recursion

$$\mathrm{TV}(\mu_{t+1}^1, \mu_{t+1}^0) \leq \varepsilon_t + \mathrm{TV}(\mu_t^1, \mu_t^0).$$

Hence

$$\mathrm{TV}(\mu_t^1, \mu_t^0) \leq \sum_{k=1}^{t-1}\varepsilon_k, \qquad t = 1, \ldots, T.$$

If $\varepsilon_k \leq \varepsilon$ for all k, then $\mathrm{TV}(\mu_t^1, \mu_t^0) \leq (t-1)\varepsilon$. □

E Hyperparameters

Across all main experiments, we use a fixed set of hyperparameters: rollout count $K = 1$, number of partial workflows per round $N = 10$, number of refined workflows $M = 10$, and maximum steps $T = 5$. Note that the optimal configuration can vary by dataset: In MATH we observe consistent gains through $T = 5$ rounds, whereas on other datasets three steps typically suffice to reach the best workflow. In addition, each complete workflow is evaluated on the validation set only once, while AFLOW evaluates each workflow five times; although this increases variance, the sampling-based nature of our Bayesian framework largely mitigates it.

F More experiment analysis

Expected reward almost monotonically increases over rounds. To assess the effectiveness of the parallel look-ahead mechanism, we plot the maximal, minimal, and average rewards at each round in Fig. 2. To isolate the rollout mechanism, we disable sequential refinement by setting $M{=}0$ in these experiments. We observe that the mean validation accuracy improves monotonically or near-monotonically across rounds, indicating that look-ahead exploration alone is indeed effective. The greatest gain occurs between rounds 1 and 2 suggesting that early pruning of low-quality workflows accounts for most of the improvement. The

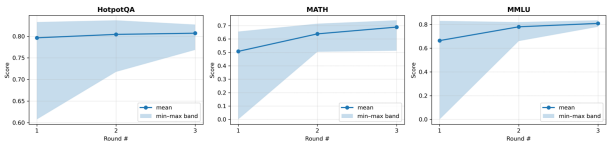

Figure 2: Evolution of validation accuracy across three sampling rounds. Solid lines show the mean over $K = 20$ samples; the shaded region spans the minimum and maximum run in each round.

min–max band also narrows markedly for MMLU-Pro, showing that the workflow pool becomes more homogeneous as unpromising branches are discarded. Overall, Fig. 2 demonstrates that parallel look-ahead can reliably guide step-level generation without the need of a process reward model.

G Helper Functions

We present the function signatures of two helper functions which are exposed to the meta-optimizer LLM for workflow generation.

```python
def call_llm(
    messages: List[str],
    temperature: float,
    num_of_response: int,
    agent_role: str,
    instructions: str
) -> List[str]:
    """
    Call the ChatCompletion API and return a list of text responses.
    ----------
    messages : List[str]
        A list of input messages
    temperature : float
        Sampling temperature. Higher values (e.g., 0.8) yield more random outputs;
        lower values (e.g., 0.2) make the output more focused and deterministic.
    num_of_response : int
        Number of responses to generate from the model.
    agent_role : str
        The role/persona for the assistant, e.g., "helpful teacher". Included in the system prompt.
    instructions : str
        Additional task-specific guidance. Appended to the system prompt.
    Returns
    -------
    List[str]
        A list of generated response strings (one per requested completion).
    """
```

Figure 3: Helper function on chat completion

```python
def exec_code(solution: str, entry_point: str) -> Union[str, List[Dict], Dict]:
    """
    This function tests a Python function implementation against public test cases, providing ↩
        ↪ feedback about failures and execution errors.

    Args:
        solution (str): Complete Python code containing the function implementation.

        entry_point (str): Name of the function to test.

    Returns:
        Union[str, List[Dict], Dict]:
            - If ALL tests pass: returns the string "no error".
            - If some tests fail (AssertionError): returns List[Dict], each with:
                {
                  "test_case": str, # the failing assertion
                  "error_type": "AssertionError",
                  "error_message": str, # assertion message
                  "traceback": List[str] # full traceback for debugging
                }
            - If code execution fails (syntax/runtime): returns Dict with:
                { "exec_fail_case": str } # message describing the execution failure
    """
```

Figure 4: Helper function on public test sets

H Workflows from BayesFlow

```python
class Workflow:
    def __init__(
        self,
        name: str,
        llm_config: dict,
        dataset: DatasetType,
    ) -> None:
        """
```

```
        Initialize the workflow with name, LLM configuration and dataset type.

        Args:
            name: Name of the workflow
            llm_config: Configuration for the LLM
            dataset: Type of dataset to use
        """
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)

    async def __call__(self, problem: str, entry_point: str) -> Tuple[str, float]:
        """
        Execute the workflow to generate and test solutions.

        Args:
            problem: Problem description
            entry_point: Name of the function to implement

        Returns:
            Tuple containing the solution code and total cost
        """
        # Step 1: Generate initial solution
        messages = [f"Write a Python function named {entry_point} that solves this problem:\n{problem}"]
        solutions = await self.llm.call_llm(
            messages=messages,
            temperature=0.2,
            num_of_response=2,
            agent_role="expert Python programmer",
            instructions="Write clean, efficient code that exactly matches the function name and parameters specified. Return only the function implementation without explanations."
        )

        # Step 2: Test solutions
        test_results = []
        for solution in solutions:
            result = exec_code(solution, entry_point)
            test_results.append((solution, result))

        # Step 3: If any solution passes tests, return it
        for solution, result in test_results:
            if result == "no error":
                return solution, self.llm.get_usage_summary()["total_cost"]

        # Step 4: If no solution passes, try to fix the best solution
        best_solution = solutions[0]
        messages = [
            f"The following solution failed:\n{best_solution}\n\nError details:\n{test_results[0][1]}\n\nPlease fix the code to pass the tests.",
        ]
        fixed_solutions = await self.llm.call_llm(
            messages=messages,
            temperature=0.1,
            num_of_response=1,
            agent_role="expert Python debugger",
            instructions="Return only the corrected function implementation without explanations."
        )

        # Step 5: Test fixed solution
        for fixed_solution in fixed_solutions:
            result = exec_code(fixed_solution, entry_point)
            if result == "no error":
                return fixed_solution, self.llm.get_usage_summary()["total_cost"]

        # Step 6: If still no success, try one final attempt with more context
        messages = [
            f"Write a function named {entry_point} that solves this problem:\n{problem}\n\n"
            f"Previous attempts failed. Here are the test results:\n{test_results}\n\n"
```

```
            "Please implement a correct solution that passes all tests."
        ]
        final_solutions = await self.llm.call_llm(
            messages=messages,
            temperature=0.3,
            num_of_response=1,
            agent_role="senior Python developer",
            instructions="Focus on passing the test cases. Return only working code."
        )

        # Return the final attempt, whether it works or not
        return final_solutions[0], self.llm.get_usage_summary()["total_cost"]
```

Figure 5: MBPP workflow

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

    async def __call__(self, problem: str) -> tuple[str, float]:
        """
        Implements a workflow for DROP dataset questions requiring discrete reasoning.
        Returns a tuple containing the final answer and total API cost.
        """
        # Step 1: Parse the question and identify key information
        parse_prompt = [
            f"Given this DROP dataset question, identify the type of reasoning required and key information:\n{problem}\n"
            "Focus on identifying whether it requires numerical reasoning, entity tracking, or date manipulation."
        ]
        analysis = await self.llm.call_llm(
            messages=parse_prompt,
            temperature=0.2,
            num_of_response=1,
            agent_role="analytical assistant",
            instructions="Analyze the question type and key information needed."
        )

        # Step 2: Generate detailed reasoning steps
        reasoning_prompt = [
            f"Question: {problem}\n"
            f"Analysis: {analysis[0]}\n"
            "Solve this step by step, showing your work clearly."
        ]
        reasoning_responses = await self.llm.call_llm(
            messages=reasoning_prompt,
            temperature=0.3,
            num_of_response=2,
            agent_role="mathematical expert",
            instructions="Show detailed step-by-step reasoning. Be precise with calculations and logic."
        )

        # Step 3: Cross-validate answers and extract final result
        validation_prompt = [
            f"Question: {problem}\n"
            f"Reasoning Path 1: {reasoning_responses[0]}\n"
            f"Reasoning Path 2: {reasoning_responses[1]}\n"
```

```
            "Compare these solutions and provide the final answer in its simplest form - just the number, name, or date without explanation."
        ]
        final_answers = await self.llm.call_llm(
            messages=validation_prompt,
            temperature=0.1,
            num_of_response=1,
            agent_role="precise evaluator",
            instructions="Extract only the final answer in its simplest form - no explanations or additional text."
        )

        # Step 4: Format the answer according to requirements
        answer = final_answers[0].strip()
        # Remove any explanatory text or prefixes
        if "answer is" in answer.lower():
            answer = answer.split("answer is")[-1]
        answer = answer.strip(" .,:")

        # Return the final answer and total cost
        return answer, self.llm.get_usage_summary()["total_cost"]
```

Figure 6: DROP workflow

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

    async def __call__(self, problem: str) -> tuple[str, float]:
        """
        Implements a workflow for GPQA multiple-choice questions.
        Returns:
        - answer: str - Single capital letter (A-D) representing the chosen option
        - cost: float - Total cost of LLM API calls
        """
        # Step 1: Domain-specific knowledge retrieval with emphasis on quantitative aspects
        knowledge_prompt = f"""
        For this graduate-level science question:
        {problem}

        1. Identify the specific scientific domain and sub-topic
        2. List the key scientific principles and mathematical relationships
        3. Write out ALL relevant formulas, equations, and units
        4. Specify numerical ranges, constants, or threshold values
        5. Note critical assumptions and boundary conditions
        6. Identify potential calculation pitfalls or unit conversion issues

        Structure your response with clear headers and numbered equations.
        """
        knowledge_responses = await self.llm.call_llm(
            messages=[knowledge_prompt],
            temperature=0.2,
            num_of_response=2,
            agent_role="domain expert",
            instructions="Provide comprehensive quantitative knowledge relevant to the question."
        )

        # Step 2: Mathematical analysis and calculations
        math_prompt = f"""
        Using the established knowledge bases:
```

```
        Knowledge Base 1: {knowledge_responses[0]}
        Knowledge Base 2: {knowledge_responses[1]}

        For the question:
        {problem}

        1. Set up the mathematical framework:
           - Define all variables and their units
           - List equations to be used
           - Identify given values and unknowns
        2. Perform step-by-step calculations:
           - Show all work clearly
           - Include unit conversions
           - Note intermediate results
        3. Calculate numerical results for each option if applicable
        4. Estimate reasonable ranges for answers

        Structure as a detailed mathematical solution.
        """
        math_responses = await self.llm.call_llm(
            messages=[math_prompt],
            temperature=0.2,
            num_of_response=2,
            agent_role="mathematical physicist",
            instructions="Show detailed mathematical work and calculations."
        )

        # Step 3: Scientific reasoning with quantitative support
        analysis_prompt = f"""
        Based on the mathematical analysis:
        Calculation Path 1: {math_responses[0]}
        Calculation Path 2: {math_responses[1]}

        For the question:
        {problem}

        1. Evaluate how each calculation approach supports/contradicts the options
        2. Consider numerical accuracy and significant figures
        3. Check for mathematical consistency with physical principles
        4. Identify which calculations most definitively distinguish between options
        """
        analysis_responses = await self.llm.call_llm(
            messages=[analysis_prompt],
            temperature=0.3,
            num_of_response=2,
            agent_role="expert scientist",
            instructions="Analyze calculations and their implications for each option."
        )

        # Step 4: Generate answers with quantitative justification
        solution_prompt = f"""
        Using the mathematical results and analysis:
        Analysis 1: {analysis_responses[0]}
        Analysis 2: {analysis_responses[1]}

        Choose the correct answer (A, B, C, or D) and explain:
        1. How the calculations support this choice
        2. Why the numerical results rule out other options
        3. Any mathematical constraints that confirm this answer

        Question: {problem}
        """
        solution_responses = await self.llm.call_llm(
            messages=[solution_prompt],
            temperature=0.2,
            num_of_response=3,
            agent_role="scientific expert",
            instructions="Select and justify answer based on quantitative evidence."
        )
```

```
        # Step 5: Final verification with emphasis on mathematical consistency
        verification_prompt = f"""
        Review these solutions:
        Solution 1: {solution_responses[0]}
        Solution 2: {solution_responses[1]}
        Solution 3: {solution_responses[2]}

        Question: {problem}

        Verify:
        1. Mathematical correctness
        2. Consistency with physical laws
        3. Proper handling of units and significant figures

        Output only a single capital letter (A, B, C, or D) representing the final answer.
        """
        final_response = await self.llm.call_llm(
            messages=[verification_prompt],
            temperature=0.1,
            num_of_response=1,
            agent_role="scientific reviewer",
            instructions="Output only a single capital letter (A, B, C, or D) as the final answer."
        )

        # Extract the single letter answer
        answer = final_response[0].strip()
        if len(answer) > 1:
            answer = ''.join(char for char in answer if char in 'ABCD')[:1]

        return answer, self.llm.get_usage_summary()["total_cost"]
```

Figure 7: GPQA workflow

```
import re
import statistics
class Workflow:
    def __init__(
        self,
        name,
        llm_config,
        dataset: DatasetType,
    ) -> None:
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)

    async def __call__(self, problem: str) -> tuple[str, float]:
        """
        Implements a multi-step workflow for solving GSM8K math problems.
        Returns the final numeric answer in LaTeX boxed format and total API cost.
        """
        # Step 1: Generate detailed step-by-step solutions
        solution_prompt = [
            f"Solve this math problem step by step:\n{problem}\n"
            "Show your work clearly, and end with the final numeric answer only (no units or ↩
                ↪ symbols)."
        ]
        solutions = await self.llm.call_llm(
            messages=solution_prompt,
            temperature=0.2, # Low temperature for precise calculations
            num_of_response=2, # Generate two solutions as verification
            agent_role="math teacher",
            instructions="Break down the problem into clear steps. Verify calculations carefully."
        )

        # Step 2: Extract numeric answers from solutions
```

```
        numbers = []
        for solution in solutions:
            matches = re.findall(r'-?\d*\.?\d+', solution.split('\n')[-1])
            if matches:
                numbers.append(float(matches[0]))

        # Step 3: Verify answers with a different approach
        verify_prompt = [
            f"Question: {problem}\n"
            "Solve this problem using a different method than before. "
            "Give only the final numeric answer without any units."
        ]
        verification = await self.llm.call_llm(
            messages=verify_prompt,
            temperature=0.3,
            num_of_response=1,
            agent_role="math expert",
            instructions="Focus on accuracy. Double-check your calculations."
        )

        # Step 4: Extract verification number
        verify_num = None
        if verification:
            matches = re.findall(r'-?\d*\.?\d+', verification[0].split('\n')[-1])
            if matches:
                verify_num = float(matches[0])
                numbers.append(verify_num)

        # Step 5: Determine final answer through consensus
        if not numbers:
            # Fallback if no numbers extracted
            final_prompt = [
                f"What is the final numeric answer to this problem? {problem}\n"
                "Give ONLY the number, no explanation or units."
            ]
            final_attempt = await self.llm.call_llm(
                messages=final_prompt,
                temperature=0.1,
                num_of_response=1,
                agent_role="math expert",
                instructions="Provide only the final numeric answer."
            )
            matches = re.findall(r'-?\d*\.?\d+', final_attempt[0])
            final_answer = float(matches[0]) if matches else 0.0
        else:
            # Use median to handle outliers
            final_answer = statistics.median(numbers)

        # Step 6: Format answer in LaTeX boxed notation
        formatted_answer = f"\\boxed{{{int(final_answer) if final_answer.is_integer() else ↩
            ↪ final_answer}}}"

        return formatted_answer, self.llm.get_usage_summary()["total_cost"]
```

Figure 8: GSM8K workflow

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

```python
    async def __call__(self, problem: str) -> tuple[str, float]:
        """
        Implements a multi-step workflow for HotpotQA with reasoning verification.
        Returns:
        - answer: str - The final answer only, without reasoning or explanation
        - cost: float - Total cost of LLM API calls
        """
        # Step 1: Initial analysis to identify key information and question type
        analysis_prompt = f"Analyze this multi-hop question and identify the key entities and relationships needed: {problem}"
        analysis = await self.llm.call_llm(
            messages=[analysis_prompt],
            temperature=0.3,
            num_of_response=1,
            agent_role="analytical assistant",
            instructions="Identify the key entities, relationships, and the type of information needed to answer this question."
        )

        # Step 2: Extract and validate supporting facts from Wikipedia context
        facts_prompt = f"Question: {problem}\nAnalysis: {analysis[0]}\nIdentify the specific Wikipedia sentences that contain the essential information needed to answer this question. For each fact, explain why it's necessary for the multi-hop reasoning chain."
        supporting_facts = await self.llm.call_llm(
            messages=[facts_prompt],
            temperature=0.4,
            num_of_response=3,
            agent_role="evidence finder",
            instructions="Extract only the most relevant sentences from the Wikipedia context. Ensure each fact is necessary for bridging the reasoning steps. Identify any missing information."
        )

        # Step 3: Generate multiple reasoning paths using validated supporting facts
        reasoning_paths = []
        for facts in supporting_facts:
            reasoning_prompt = f"Question: {problem}\nRelevant Facts: {facts}\nProvide a step-by-step reasoning path that connects these supporting facts to answer the question."
            paths = await self.llm.call_llm(
                messages=[reasoning_prompt],
                temperature=0.7,
                num_of_response=2,
                agent_role="logical reasoner",
                instructions="Create a clear reasoning chain that shows how the supporting facts connect to reach the answer. Each step should be grounded in the provided facts."
            )
            reasoning_paths.extend(paths)

        # Step 4: Generate candidate answers based on each reasoning path
        answers = []
        for path in reasoning_paths:
            answer_prompt = f"Question: {problem}\nReasoning: {path}\nProvide only the final answer in its simplest form without explanation."
            candidate = await self.llm.call_llm(
                messages=[answer_prompt],
                temperature=0.2,
                num_of_response=1,
                agent_role="precise answerer",
                instructions="Give only the exact answer in its simplest form - just the name, number, or fact requested. No explanations."
            )
            answers.extend(candidate)

        # Step 5: Verify and select the final answer
        verification_prompt = f"Question: {problem}\nSupporting Facts: {supporting_facts}\nCandidate answers: {answers}\nSelect the most accurate answer that is fully supported by the facts and simplify it to its most basic form."
```

```
        final_answer = await self.llm.call_llm(
            messages=[verification_prompt],
            temperature=0.1,
            num_of_response=1,
            agent_role="answer validator",
            instructions="Choose and simplify the answer to its most basic form, ensuring it is fully supported by the extracted Wikipedia facts. For names, give just the name. For yes/no questions, respond only with Yes or No. For measurements, give just the number and unit. Remove any explanations or additional context."
        )

        # Return the simplified final answer and total cost
        return final_answer[0], self.llm.get_usage_summary()["total_cost"]
```

Figure 9: HotpotQA workflow

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

    async def __call__(self, problem: str) -> tuple[str, float]:
        """
        Implements a workflow for solving MATH problems with multiple verification steps.
        Returns:
        - answer: str - Final answer in LaTeX boxed format
        - cost: float - Total cost of LLM API calls
        """
        # Step 1: Plan solution approach and identify key concepts
        plan_instructions = """
        Analyze this math problem and create a solution plan:
        1. Identify the mathematical concepts and theorems needed
        2. Break down the problem into clear logical steps
        3. Note any potential pitfalls or edge cases to consider
        4. Outline the sequence of calculations needed
        Do not solve the problem yet - focus on planning the approach.
        """
        solution_plans = await self.llm.call_llm(
            messages=[problem],
            temperature=0.4,
            num_of_response=2,
            agent_role="mathematical strategist",
            instructions=plan_instructions
        )

        # Step 2: Execute solution with detailed reasoning using the plans
        solve_instructions = f"""
        Solve this math problem following the solution plans provided:
        Plan 1: {solution_plans[0]}
        Plan 2: {solution_plans[1]}

        Show your work step by step, following these guidelines:
        1. Use the identified concepts and theorems correctly
        2. Follow the planned logical steps
        3. Watch for the noted pitfalls
        4. Double-check calculations
        5. Format the final answer using LaTeX boxed{{}}
        """
        solutions = await self.llm.call_llm(
            messages=[f"Problem: {problem}"],
            temperature=0.3,
```

```
            num_of_response=2,
            agent_role="expert mathematician",
            instructions=solve_instructions
        )

        # Step 3: Verify and validate solutions
        verify_instructions = """
        Verify these solutions for correctness:
        1. Check mathematical logic and steps
        2. Verify calculations
        3. Ensure answer format is correct (LaTeX boxed{})
        4. Compare the two solutions
        Return only the correct boxed answer. If solutions differ, explain why one is correct.
        """
        verification = await self.llm.call_llm(
            messages=[f"Problem: {problem}\nSolution 1: {solutions[0]}\nSolution 2: {solutions[1]}"],
            temperature=0.2,
            num_of_response=1,
            agent_role="mathematical reviewer",
            instructions=verify_instructions
        )

        # Step 4: Format check and extraction
        format_instructions = """
        Extract and format the final answer following these rules:
        1. Answer must be in LaTeX boxed{} format
        2. For fractions, use \frac{n}{d}
        3. For multiple values, separate with commas inside boxed{}
        4. For vectors/matrices, use proper LaTeX notation
        5. Remove any explanation text, keep only the boxed{} answer
        Return only the formatted answer.
        """
        formatted_answer = await self.llm.call_llm(
            messages=[verification[0]],
            temperature=0.1,
            num_of_response=1,
            agent_role="LaTeX formatter",
            instructions=format_instructions
        )

        # Extract just the boxed answer
        answer = formatted_answer[0]
        if "\\boxed{" not in answer:
            # Final fallback to ensure boxed format
            answer = f"\\boxed{{{answer}}}"

        return answer, self.llm.get_usage_summary()["total_cost"]
```

Figure 10: MATH workflow

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

    async def __call__(self, problem: str) -> tuple[str, float]:
        """
        Implements a workflow for MMLU multiple choice questions.
        Returns:
        - answer: str - A single capital letter (A-J) representing the chosen option
        - cost: float - Total cost of LLM API calls
```

```
        """
        # Step 1: Structured decomposition of the question
        decomposition_prompt = f"""
        Break down this multiple choice question into its core components:
        {problem}

        1. Core Concept: What is the main topic or principle being tested?
        2. Given Information: What key facts or conditions are provided?
        3. Required Reasoning: What logical steps are needed to connect the given information to ↩
            ↪ the answer?
        4. Relationships: How do different elements in the question relate to each other?
        5. Answer Criteria: What specific characteristics must the correct answer satisfy?

        Provide a structured analysis addressing each point above.
        """
        decomposition_responses = await self.llm.call_llm(
            messages=[decomposition_prompt],
            temperature=0.2,
            num_of_response=2,
            agent_role="analytical expert",
            instructions="Break down the question systematically, focusing on logical relationships ↩
                ↪ and key components."
        )

        # Step 2: Detailed analysis incorporating decomposition insights
        analysis_prompt = f"""
        Using this structural analysis:
        {decomposition_responses[0]}
        {decomposition_responses[1]}

        Evaluate the multiple choice question:
        {problem}

        For each option (A-J):
        1. How well does it align with the core concept identified?
        2. Does it satisfy the answer criteria established?
        3. Is it supported by the logical relationships we identified?

        After analysis, select the most appropriate answer choice (A-J).
        """
        analysis_responses = await self.llm.call_llm(
            messages=[analysis_prompt],
            temperature=0.3,
            num_of_response=2,
            agent_role="expert academic advisor",
            instructions="Use the structural analysis to systematically evaluate each option and ↩
                ↪ clearly indicate your final answer as a single capital letter A-J."
        )

        # Step 3: Cross-check against decomposition framework
        verification_prompt = f"""
        Question: {problem}

        Consider how each answer option aligns with our structural analysis:
        {decomposition_responses[0]}

        Previous answer suggestions: {', '.join(analysis_responses)}

        Verify the answer by:
        1. Testing it against each component of our decomposition
        2. Checking for logical consistency with identified relationships
        3. Confirming it meets all answer criteria

        Which option (A-J) best satisfies these requirements?
        """
        verification_responses = await self.llm.call_llm(
            messages=[verification_prompt],
            temperature=0.2,
            num_of_response=1,
            agent_role="subject matter expert",
```

```
            instructions="Systematically verify the answer against our structural analysis framework."
        )

        # Step 4: Final consensus with strict constraints
        all_responses = analysis_responses + [verification_responses[0]]
        consensus_prompt = f"""
        Question: {problem}

        Previous analyses suggested these answers: {', '.join(all_responses)}

        Based on our complete structural analysis and verification, determine the final answer.
        You must output EXACTLY one capital letter A-J, nothing else.
        """

        final_response = await self.llm.call_llm(
            messages=[consensus_prompt],
            temperature=0.1,
            num_of_response=1,
            agent_role="decision maker",
            instructions="Output only a single capital letter A-J as the final answer, with no ↩
                ↪ additional text."
        )

        # Step 5: Extract and validate the answer
        answer = re.findall(r'[A-J]', final_response[0])[0]

        return answer, self.llm.get_usage_summary()["total_cost"]
```

Figure 11: MMLU-Pro workflow