

Can Workflow Search Be Amortized?

Demonstration-Guided Synthesis of LLM Agent Workflows

Shiyi Du¹, Jiayuan Liu^{1,2}, Weihua Du¹, Yue Huang³, Jiayi Li¹, Yingtao Luo¹,
Xiangliang Zhang³, Vincent Conitzer^{1,2}, Carl Kingsford^{1,*}

¹Carnegie Mellon University ²Foundations of Cooperative AI Lab (FOCAL)

³University of Notre Dame

{shiyid, jiayuan4, weihuad, lijiaiyi, yingtaol}@cs.cmu.edu
{yhuang37, xzhang33}@nd.edu {conitzer, carlk}@cs.cmu.edu

Abstract

Automated LLM workflow design often relies on per-task search, which is computationally expensive and does not explicitly reuse structural knowledge across tasks. We propose SWIFT (Synthesizing Workflows via Few-shot Transfer), which reuses prior search trajectories and demonstrations to synthesize target-task workflows without iterative target-task optimization. SWIFT distills compositional heuristics and output-interface contracts from source-task trajectories, then conditions a single LLM generation pass on these priors and cross-task workflow demonstrations to produce an executable workflow. Across 5 benchmarks, SWIFT outperforms the SOTA search-based baseline while reducing marginal per-task optimization cost by three orders of magnitude given an existing source bank. It also improves performance on 4 additional held-out benchmarks and transfers from GPT-4o-mini to Grok, Qwen, and Gemma execution backbones. These results suggest that prior workflow search can expose reusable structural priors that reduce optimization cost while preserving or improving accuracy on related tasks.

1 Introduction

LLM agent workflows, which are structured programs that orchestrate sequences of LLM calls, tool uses, and control flow, have become a common way to solve complex reasoning, coding, and generation tasks (Yao et al., 2023b; Wei et al., 2022). Manually designing such workflows is labor-intensive, motivating recent methods such as AFlow (Zhang et al., 2025c) and ADAS (Hu et al., 2025) to automate workflow design through iterative search. While effective, these approaches typically optimize each new task independently, repeatedly exploring a large discrete search space with limited explicit reuse of structural knowledge across tasks.

This repeated per-task search can be computationally expensive and may over-specialize to small validation sets (El et al., 2025; Wang et al., 2025a).

This raises a more targeted question: *when prior search has discovered useful workflows for related tasks, can those discoveries be reused to reduce target-task search?* We observe that optimized workflows in our math and code benchmarks often converge to a small family of domain-specific templates: math tasks favor multi-path reasoning with verification or extraction, while code tasks favor execution-feedback repair loops. This suggests that the effective workflow space can be more concentrated than a naive combinatorial view suggests. If such structural regularities transfer across tasks, then the cost of workflow discovery can be amortized rather than paid independently for each new task.

We instantiate this idea in SWIFT (Synthesizing Workflows via Few-shot Transfer; Figure 1), a framework that amortizes workflow design into reusable structural priors. SWIFT operates in two phases. **Offline**, it performs *contrastive trajectory distillation*: by analyzing both high- and low-performing workflows from prior search histories, it extracts compositional heuristics—actionable, operator-level routing rules that bound the generation space. It further distills *output contracts* (strict formatting constraints that ensure the output of one node can be seamlessly parsed by the next) from intermediate-scoring workflows that failed due to formatting mismatches, capturing interface specifications that ensure executability. **Online**, given only a brief description of a new target task, SWIFT synthesizes a complete, executable workflow for the target task in a single generation pass, conditioned on workflow examples from other tasks and a small set of automatically learned design rules. It involves no iterative search and no task-specific optimization.

To rigorously prevent information leakage, we

*Corresponding author.

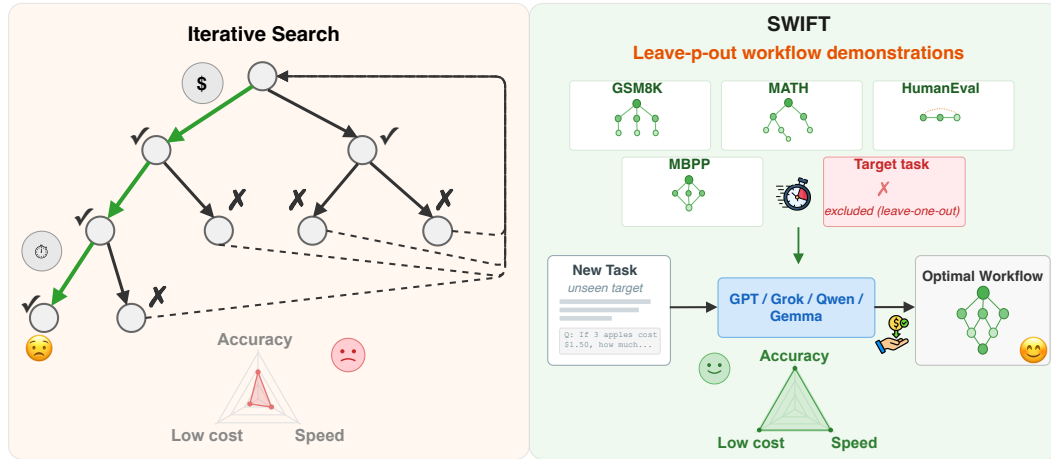


Figure 1: From per-task search to amortized workflow synthesis. Conventional methods optimize a fresh workflow for each target task; SWIFT reuses leave-one-out source workflows as design demonstrations to synthesize the held-out target workflow in one meta-generation call.

adopt a strict leave-one-out (LOO) protocol: when generating a workflow for any target task, all data from that task is entirely masked.

Our main contributions are as follows:

- **Amortized workflow synthesis.** We reframe automated workflow design from per-task iterative search to amortized synthesis. SWIFT distills reusable structural priors offline and synthesizes competitive workflows for tasks absent from the source demonstration pool in a single generation, reducing optimization cost by three orders of magnitude compared to the state-of-the-art search-based method.
- **Mechanistic understanding of cross-task transfer.** We show that contract-guided synthesis helps bridge the structural-functional gap, and that workflow demonstrations transfer structural and interface cues beyond human-readable operator names (Table 4).
- **Strong empirical performance and transfer boundaries.** SWIFT outperforms AFlow across all five main benchmarks (Table 1). It further improves performance on four benchmarks absent from trajectory distillation (Table 2) and transfers to three alternative execution backbones (Grok, Qwen, and Gemma) while also revealing failure modes from task mismatch and weaker instruction following (Table 5).

Source code for the method is available at <https://github.com/Kingsford-Group/swift>.

2 Related Work

Structured reasoning and multi-agent workflows. Single-agent prompting strategies, chain-of-thought (Wei et al., 2022), complexity-based prompting (Fu et al., 2023), self-consistency (Wang et al., 2023), tree-of-thought (Yao et al., 2023a), and graph-of-thought (Besta et al., 2024) all improve LLM performance through fixed reasoning patterns. Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and LATS (Zhou et al., 2024) further add iterative self-feedback or tree search at test time, but all operate per-instance. Hand-crafted multi-agent systems explore diverse collaboration patterns like inter-agent debate (Du et al., 2024), output ensembling via ranking and fusion (Jiang et al., 2023), intra-model persona simulation (Wang et al., 2024b), dynamic team selection and role-aware context routing (Liu et al., 2024, 2025), collaborative role adjustment (Chen et al., 2024b), and graph-structured agent networks (Qian et al., 2025), but generally operate within predefined agent architectures. Recent studies corroborate the primacy of topology: Zhou et al. (2025) find that communication topology matters more than prompt optimization, and G-Designer (Zhang et al., 2025b) learns task-specific topologies via GNNs. SWIFT shares this insight but *synthesizes* topology via demonstration-conditioned generation rather than manual design or training a separate model.

Automated workflow optimization. Recent work automates agent design through search and optimization. A large line of work focuses on

structure and workflow search, including AutoAgents (Chen et al., 2024a), GPTSwarm (Zhuge et al., 2024), and AgentSquare (Shang et al., 2025), which generate agent roles, computational graphs, or modular compositions via dynamic planning, graph optimization, or evolutionary search. Related approaches such as AFlow (Zhang et al., 2025c) and MaAS (Zhang et al., 2025a) further explore graph-based or probabilistic workflow spaces, though often requiring substantial per-task search cost. Another line of work focuses on prompt and program optimization, including DSPy (Khat-tab et al., 2024), OPRO (Yang et al., 2024), and TextGrad (Yuksekgonul et al., 2025), which optimize prompts or pipelines via compilation, black-box optimization, or gradient-like feedback. Finally, some approaches aim at dynamic or learned composition, such as DAAO (Su et al., 2026) and MAS-GPT (Ye et al., 2025), which generate query-specific agent structures, often relying on fine-tuning or learned policies. Despite these advances, prior work typically performs optimization independently for each task. In contrast, SWIFT amortizes structural knowledge across tasks, so that each new task benefits from prior optimization effort at near-zero marginal cost.

Demonstration-conditioned generation and cross-task transfer. LLMs can adapt to new tasks from few-shot demonstrations without parameter updates (Brown et al., 2020). Recent work studies what makes demonstrations effective (Min et al., 2022; Liu et al., 2022) and how to select them (Rubin et al., 2022). CrossICL (Gao et al., 2025) transfers demonstrations across NLP tasks via unsupervised alignment, and Wang et al. (2024a) synthesize target task demonstrations from similar source tasks. Agent Workflow Memory (Wang et al., 2025b) induces reusable workflow routines from past executions and retrieves them for new tasks. These methods transfer at the *instance level*, selecting or adapting input-output examples for a given task. SWIFT operates at the *program level*: it uses complete executable workflows as demonstrations to synthesize a novel workflow in a single generation pass, without directly executing a cached source workflow.

Amortized optimization and program synthesis. The idea of replacing per-instance optimization with a learned amortized mapping has deep roots in machine learning: amortized variational inference (Kingma and Welling, 2014; Rezende

et al., 2014) replaces per-datapoint ELBO optimization with a shared encoder, and neural program synthesis (Devlin et al., 2017; Balog et al., 2017) learns to predict programs from input-output specifications rather than searching over the program space, while Structured Agent Distillation (Liu et al., 2026) transfers reasoning and action behavior from teacher agents to smaller student models. SWIFT extends this amortization principle to agentic workflow design: rather than searching over graph topologies per task or training a student model, it directly synthesizes executable workflows conditioned on structural priors distilled from prior optimization trajectories.

3 Methodology

Search-based methods for automated LLM workflow generation are effective, but applying them independently to every new task can be expensive and may rediscover similar workflow structures across different tasks. We propose **SWIFT**, a framework that turns source-task search traces into reusable workflow priors for target-task synthesis. By conditioning workflow generation on cross-task demonstrations and distilled heuristics, SWIFT enables few-shot workflow synthesis via a single generation pass. The overall architecture of SWIFT is illustrated in Figure 2.

3.1 Problem Formulation: Amortized Workflow Synthesis

We represent a workflow w as an executable program that, given a task input x , orchestrates a sequence of operator calls to produce an output $y = w(x)$. Architecturally, w is represented as an operator compositional graph $G = (\mathcal{V}, \mathcal{E})$, realized as executable program following (Zhang et al., 2025c). Each node $v_i \in \mathcal{V}$ is an instantiation of a predefined operator from a typed library $\mathcal{O} = \{o_1, \dots, o_K\}$, where each operator encapsulates a specific capability such as LLM reasoning, code execution, or ensemble voting (Zhang et al., 2025c). Each instantiated operator is parameterized by a natural language instruction θ_i , and may encapsulate internal control flow such as conditional branching and retry loops. The edges $\mathcal{E}$ define the topological data flow and control logic between these operators.

Given a target task $\mathcal{T}_{target}$ and an evaluation dataset $\mathcal{D}$, the standard optimization objective seeks to find the best workflow w^* that maximizes

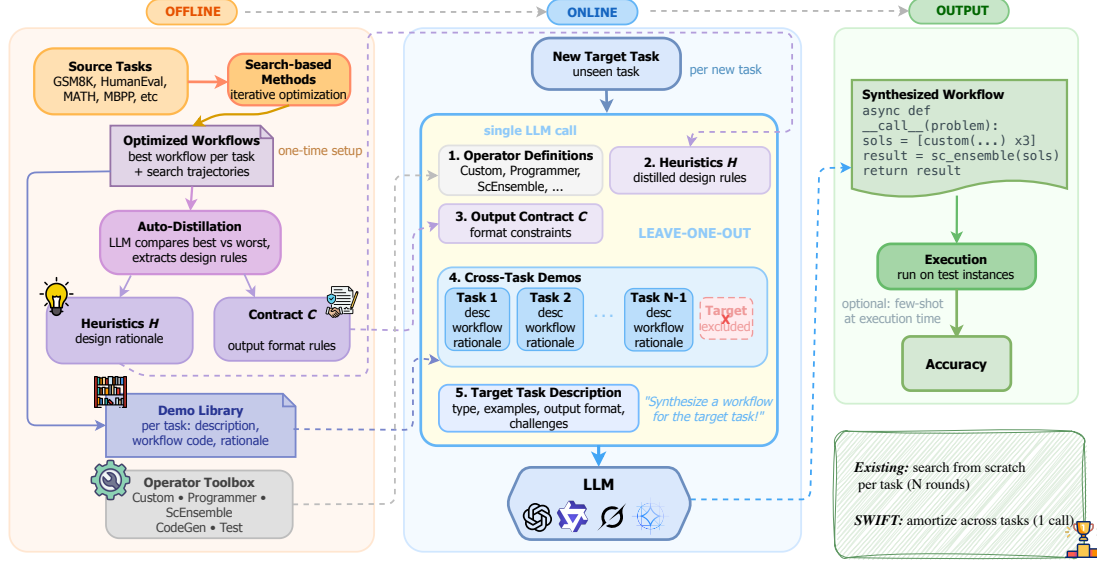


Figure 2: Overview of SWIFT. **Offline**: contrastive trajectory distillation extracts compositional heuristics ($\mathcal{H}$) and output contracts ($\mathcal{C}$) from source-task search histories. **Online**: a single LLM generation call synthesizes an executable workflow for a task absent from the source demonstration pool, conditioned on operator definitions, distilled priors, and leave-one-out demonstrations. The workflow is directly executed on test instances.

the expected task accuracy:

$$w^* = \arg \max_{w \in \mathcal{W}} \mathbb{E}_{x \sim \mathcal{D}} [\text{Acc}(w(x), \mathcal{T}_{\text{target}})].$$

Because the search space $\mathcal{W}$ is discrete, non-differentiable, and combinatorially vast, solving this from scratch per task via iterative algorithms can be expensive (each candidate workflow must be generated, executed, and scored). If N candidates are evaluated and each evaluation costs C , target-task optimization cost scales with $N \cdot C$. SWIFT instead instantiates an amortized synthesis map through in-context generation with a frozen LLM: $w_{\text{target}} \leftarrow f_{\phi}(\mathcal{O}, \mathcal{H}, \mathcal{C}, \mathcal{D}_{\text{demo}}, \mathcal{T}_{\text{target}})$. This produces one executable candidate workflow in a single meta-generation call, without evaluating and refining multiple workflows on the target task.

3.2 Output Contracts for Executable Workflow Synthesis

A synthesized workflow w can be logically and topologically sound in its reasoning process (Structural) but still fail because its intermediate or final outputs violate the strict downstream interfaces (Functional). We call this failure mode the *structural-functional gap*. For example, a mathematically sound workflow might compute the correct numerical answer but return it embedded within conversational text, causing an exact-match evaluation parser to register a failure.

In a standalone LLM call, output formatting can often be handled by appending a format instruction. In a multi-node workflow, however, formatting errors cascade across node interfaces: an intermediate node may output semantically correct but syntactically incompatible text (e.g., “The answer is 42” instead of “42”), which downstream nodes, ensemble voters, extraction operators, or evaluation parsers, cannot parse. Each node individually follows its local instruction, yet the inter-node handoff fails. Moreover, during amortized synthesis, the generator must produce a complete, novel workflow in a single generation pass without the opportunity for per-node prompt tuning. Without explicit interface constraints, the synthesis LLM may compensate for this uncertainty by generating overly complex topologies: adding redundant ensemble voting, multi-candidate generation, and error recovery branches as hedging mechanisms. Our ablation confirms this effect empirically: removing contracts causes the synthesizer to produce workflows with substantially more nodes and higher per-sample cost, yet lower accuracy (Table 4; see Appendix H for a detailed topology comparison).

We resolve this phenomenon via **contract-guided synthesis**. To automatically construct contracts, we inspect source-task workflows that achieved partial success but failed final extraction. From these specific failure patterns, we distill a set

of **Output Contracts** ($\mathcal{C}$). These contracts act as lightweight interface constraints injected into the synthesis process. They specify expected answer formats, code-only outputs, boxed-answer requirements, etc.

3.3 Demonstration-Conditioned Workflow Synthesis

Equipped with the heuristics $\mathcal{H}$ and contracts $\mathcal{C}$, target workflow synthesis becomes a demonstration-conditioned workflow generation problem. For the unseen target task $\mathcal{T}_{target}$, we construct a structured meta-prompt P that encapsulates the available action space (the operator library $\mathcal{O}$), the distilled structural and functional constraints ($\mathcal{H} \cup \mathcal{C}$), and a retrieved set of source-task workflow demonstrations $\mathcal{D}_{demo} \subset \{w_1^*, \dots, w_N^*\}$.

Under our LOO protocol, the target task’s examples, search trajectories, error logs, and optimized workflows are excluded from the meta-prompt ($\mathcal{D}_{demo}$ is strictly disjoint from $\mathcal{T}_{target}$). These cross-task demonstrations provide two complementary forms of evidence: (1) **Executable Syntax**, showing how operators are instantiated, connected, and called in Python; and (2) **Strategy Transfer**, exposing reusable workflow-level patterns that can guide synthesis when source and target tasks are structurally related.

By conditioning on P , the meta-generator executes $w_{target} \leftarrow f_\phi(P)$, producing one executable candidate workflow without iterative target-task search. The complete procedure is formally described in Algorithm 1 (Appendix B).

4 Experiments

To evaluate the efficacy of SWIFT as an amortized workflow synthesis framework, we design our experiments to move beyond standard performance benchmarking to empirically answer the following research questions:

- **RQ1 (Efficacy & Task Transfer):** Can single-pass, demonstration-conditioned synthesis match or exceed computationally expensive iterative search? How well do source-derived workflow priors transfer to benchmarks absent from trajectory distillation?
- **RQ2 (Efficiency & Amortization):** Can workflow structures discovered on source tasks reduce the cost of designing workflows for held-out targets, and how does this comparison change under marginal reuse versus cold-start accounting?

- **RQ3 (Workflow Transfer Signals):** What design signals are transferred from cross-task workflow demonstrations, and how do heuristics, contracts, domain match, code order, and operator names affect synthesis?

- **RQ4 (Execution-Model Portability):** Do synthesized workflows remain effective when executed by different LLM backbones, and when can weaker instruction following make transferred workflows harmful?

We provide complementary analyses of semantic demonstration retrieval, interactive execution, generator–executor coupling, synthesis variability, environment-related failures, and retrieval-based workflow reuse in Appendix I.

Datasets & Leave-One-Out Protocol. We evaluate SWIFT on several source-pool benchmarks: GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021); and five held-out benchmarks: MultiArith (Roy and Roth, 2015), SVAMP (Patel et al., 2021), AIME 2024–2025 (Zhang and Math-AI, 2024, 2025), AQuA (Ling et al., 2017), and BigCodeBench (Zhuo et al., 2025) (Table 6 in Appendix A). For the execution-backbone portability analysis in Section 4.4, we additionally include DROP (Dua et al., 2019) and HotpotQA (Yang et al., 2018).

The source-pool benchmarks are the tasks for which we have workflow demonstrations and search trajectories. We apply task-level leave-one-out within this pool: when a source-pool benchmark is used as the target, its own training examples, error logs, search trajectories, and optimized workflow are excluded from the meta-prompt, while demonstrations from the remaining source-pool tasks may still be used. The held-out benchmarks are never included in trajectory distillation or demonstration construction. When evaluating a held-out benchmark, SWIFT receives only the target-task description together with priors and demonstrations derived from the source-pool benchmarks. Thus, these held-out results test transfer beyond the demonstration pool.

Baselines. We compare SWIFT against: (1) *Single-agent paradigms* (e.g., CoT (Wei et al., 2022), SC (Wang et al., 2023)); (2) *Hand-crafted multi-agent workflows* (e.g., DyLAN (Liu et al., 2024), AgentVerse (Chen et al., 2024b)); and (3)

Table 1: Test accuracy (%) on five benchmarks (GPT-4o-mini). **Bold** = best, underline = runner-up. Baselines from Zhang et al. (2025a). [†]Avg. over all five. *MultiArith is not in the source demonstration pool.

	Method	GSM8K	MATH	HumanEval	MBPP	MultiArith*	Avg. [†]
Single	Vanilla	87.45	46.29	87.08	71.83	96.85	77.90
	CoT (Wei et al., 2022)	87.10	46.40	88.13	71.83	96.31	77.95
	ComplexCoT (Fu et al., 2023)	86.89	46.53	87.49	72.36	96.70	77.99
	SC (CoT×5) (Wang et al., 2023)	87.57	47.91	88.60	73.60	96.58	78.85
Hand-craft	MultiPersona (Wang et al., 2024b)	87.50	45.43	88.32	73.19	97.49	78.39
	LLM-Debate (Du et al., 2024)	89.47	48.54	88.68	70.29	97.33	78.86
	LLM-Blender (Jiang et al., 2023)	88.35	46.92	88.80	77.05	97.29	79.68
	DyLAN (Liu et al., 2024)	89.98	48.63	90.42	77.30	97.12	80.69
	AgentVerse (Chen et al., 2024b)	89.91	47.35	89.29	74.28	97.50	79.67
	MacNet (Qian et al., 2025)	87.95	45.18	84.57	65.28	96.03	75.80
Automated	AutoAgents (Chen et al., 2024a)	87.69	45.32	87.64	71.95	96.42	77.80
	GPTSwarm (Zhuge et al., 2024)	89.14	47.88	89.32	77.43	96.79	80.11
	ADAS (Hu et al., 2025)	86.12	43.18	84.19	68.13	96.02	75.53
	AgentSquare (Shang et al., 2025)	87.62	48.51	89.08	78.46	97.77	80.29
	AFlow (Zhang et al., 2025c)	91.16	51.28	90.93	81.67	96.22	82.25
	MaAS (Zhang et al., 2025a)	<u>92.30</u>	<u>51.82</u>	<u>92.85</u>	<u>82.17</u>	98.80	<u>83.59</u>
	SWIFT (ours)	94.31	55.14	95.42	83.28	<u>98.54</u>	85.34

Automated test-time search (e.g., AFlow (Zhang et al., 2025c), MaAS (Zhang et al., 2025a)). AFlow serves as our primary search-based baseline because it directly optimizes executable workflow graphs through iterative search.

Implementation Details. For our primary evaluations, we use `gpt-4o-mini` as both the meta-generator f_ϕ and the underlying execution engine. To assess portability across execution backbones, we additionally evaluate SWIFT using Grok, Qwen, and Gemma backbones. The trajectory distillation (Phase 1) is performed once entirely offline. Online synthesis (Phase 2) requires exactly one generation pass. The synthesis call uses temperature $T = 0.0$. Reported accuracies follow the baseline evaluation protocol; we therefore interpret small differences cautiously and emphasize consistent trends across tasks.

4.1 Main Results: Amortized Synthesis Outperforms Iterative Search (RQ1)

Table 1 shows that SWIFT can synthesize strong workflows for held-out target tasks without iterative optimization on those targets. Under task-level LOO, SWIFT receives no target-task workflow code, search trajectory, or error log. SWIFT achieves the best average accuracy among the compared methods and improves over AFlow on all five benchmarks. These results suggest that, once source-task demonstrations are available, repeated target-task search is not always necessary.

Regularization from source-derived templates.

One possible explanation for the gains is that source-derived templates regularize workflow synthesis toward simple structures that have worked across related tasks. Compared with selecting a new graph independently for each benchmark, this reusable-template bias can reduce highly task-specific design choices while preserving the main structure needed for the task family (Appendix J). MultiArith is absent from the demonstration pool. Yet SWIFT achieves performance matching MaAS on MultiArith. This suggests that templates distilled from source-pool math workflows can transfer to related arithmetic word problems without requiring MultiArith-specific demonstrations or target-task workflow search.

Held-out Transfer and Failure Modes. Table 2 reveals that transfer efficacy varies systematically by domain. Within mathematics, gains are substantial: SVAMP and AQuA show the largest improvements, suggesting that multi-path ensemble

Table 2: Held-out benchmark results. These benchmarks are not part of the source demonstration pool. Results therefore measure transfer from source-pool workflows to new benchmarks.

Dataset	Vanilla	SWIFT	Δ
SVAMP	81.0	95.8	+14.8
AIME 24–25	8.3	14.6	+6.3
AQuA	52.9	79.9	+27.0
BigCodeBench	31.6	34.3	+2.7

Table 3: Efficiency on MATH with GPT-4o-mini. **Bold** = best, shaded = runner-up. SWIFT’s optimization cost includes distillation over available source trajectories and target-workflow synthesis.

Method	Optimization		Execution		Total		Acc. (%)
	Cost (\$)	Time	Cost (\$)	Time	Cost (\$)	Time	
LLM-Debate	—	—	6.76	92 min	6.76	92 min	48.54
DyLAN	13.01	508 min	2.89	39 min	15.90	547 min	48.63
MacNet	—	—	2.35	47 min	2.35	47 min	45.18
GPTSwarm	7.02 $1755\times$	129 min	0.93	30 min	7.95	159 min	47.88
AFlow	22.50 $5625\times$	184 min	1.66	23 min	24.16 $14\times$	207 min $30\times$	51.28
MaAS	3.38 $845\times$	53 min	0.42	19 min	3.80 $2\times$	72 min $10\times$	51.82
SWIFT (Ours)	0.004[†]	0.13 min[†]	1.73	7 min	1.73	7 min	55.14

[†]Includes one-time contrastive distillation and single-pass optimization.

structures are reusable across several math-style benchmarks, with AQUA showing that SWIFT can synthesize multiple-choice extraction steps despite having no multiple-choice workflows in its demonstration pool (Appendix D). AIME gains come primarily from code-assisted verification, but the vast majority of failures reflect the base model’s reasoning ceiling rather than workflow limitations (Appendix D). BigCodeBench shows the smallest gain and failure analysis attributes most errors to missing Python dependencies, not workflow topology, as detailed in Appendix D.

4.2 Efficiency: Reusing Structural Priors Across Tasks (RQ2)

We use MATH as a representative case study, as it is one of the most computationally demanding benchmarks in this paper. SWIFT reduces optimization cost by over three orders of magnitude compared to search-based methods (Table 3), completing workflow synthesis in under 10 seconds versus 184 minutes for AFlow. When accounting for the full pipeline (optimization *plus* execution on the test set), SWIFT remains $14\times$ cheaper and $29\times$ faster than AFlow, and $2\times$ cheaper and $10\times$ faster than MaAS, while achieving the highest accuracy. On the evaluated benchmarks, SWIFT achieves a better cost–accuracy trade-off than the SOTA search-based baseline, suggesting that prior search trajectories can provide useful structural priors for new workflow synthesis. Detailed per-benchmark cost breakdowns are provided in Appendix F.2.

The scaling structure of amortized synthesis.

The cost structures of per-task search and amortized synthesis differ not in degree but in *kind*. Search-based methods pay the full optimization cost independently for every new task: n tasks

cost $n \times C_{\text{search}}$, and no reusable asset is produced. SWIFT instead converts prior optimization traces into persistent structural priors, reducing each subsequent task to a single generation pass at cost $C_{\text{synth}} \approx \$0.004$.

A fair cost comparison depends on how source trajectories are accounted for. If they are treated as a reusable source bank, SWIFT incurs only the marginal synthesis cost for each held-out target. If instead the full upstream search budget is charged to SWIFT, the one-time source cost is approximately $C_{\text{source}} \approx \$112.50$, yielding a break-even point of about five downstream tasks relative to per-task search (Appendix G). Thus, SWIFT is most cost-effective in settings where optimized workflows or search traces are already available, or where many related tasks will be optimized over time.

This accounting also clarifies what the target-dataset leave-one-out protocol tests: it prevents direct reuse of the held-out task’s workflow or search traces, while evaluating whether structural priors learned from other source tasks can support workflow synthesis for that target. It does not rule out transfer through broader domain-level regularities, which we treat as part of the amortization signal.

By achieving higher accuracy at substantially lower marginal optimization cost, SWIFT suggests that per-task search often rediscovers reusable structural patterns. Rather than treating each task’s optimized workflow as an isolated artifact, amortized synthesis converts prior search trajectories into persistent priors that can guide workflow design for related new tasks.

Table 4: Component ablation. $\mathcal{H}$ = compositional heuristics, $\mathcal{C}$ = output contracts. Zero-shot = no demos, Shuffled = randomized code lines, Cross-domain = mismatched-category demos, Random-ops = random operator names. Shaded column : the Rand-ops results show that human-readable operator names are not very essential, although topology, interfaces, and usage patterns remain available to the model.

Dataset	Full SWIFT	Component		Demonstration			
		w/o $\mathcal{H}$	w/o $\mathcal{C}$	Zero-shot	Shuffled	Cross-dom.	Rand-ops
GSM8K	94.31	88.82 -5.5	89.86 -4.5	91.19 -3.1	89.38 -4.9	89.29 -5.0	89.67 -4.6
MATH	55.14	49.59 -5.6	50.41 -4.7	2.47 -52.7	0.00 -55.1	47.74 -7.4	49.59 -5.6
HumanEval	95.42	93.13 -2.3	90.84 -4.6	84.73 -10.7	89.31 -6.1	87.02 -8.4	91.60 -3.8
MBPP	83.28	81.53 -1.8	80.65 -2.6	72.43 -10.9	69.50 -13.8	70.38 -12.9	77.51 -5.8
Avg.	82.04	78.27 -3.8	77.94 -4.1	62.71 -19.3	62.05 -20.0	73.61 -8.4	77.09 -4.9

4.3 Ablation: Sources of Transfer in Workflow Synthesis (RQ3)

To identify which inputs support transfer, we ablate the priors and demonstrations used by the meta-generator (Table 4).

Demonstrations help more on complex workflows. Zero-shot synthesis has little effect on GSM8K but collapses MATH to 2.47%, suggesting that harder tasks require demonstrations to specify multi-step structure and formatting conventions. Figure 3 shows a similar pattern: GSM8K saturates with one demonstration, while MATH improves after more demonstrations.

Operator names are not the only signal. Randomizing operator names degrades performance but preserves much of the full system’s accuracy (Table 4). This does not establish topology-only transfer, since interfaces, argument patterns, and local code context remain visible; it shows instead that human-readable operator names are not so necessary for useful transfer.

Transfer is stronger within related task families. Cross-domain demonstrations reduce performance, especially on MATH and MBPP (Table 4), indicating that demonstrations convey task-family-specific motifs rather than syntax alone. These motifs transfer better when source and target tasks share similar verification structure.

Output contracts improve executability. Removing output contracts ($-\mathcal{C}$) consistently lowers accuracy (Table 4). Contracts specify intermediate and final output formats; without them, workflows can fail through parser errors or incompatible node handoffs. Appendix C discusses a related code-generation failure mode, supporting an interface-aware view of workflow synthesis.

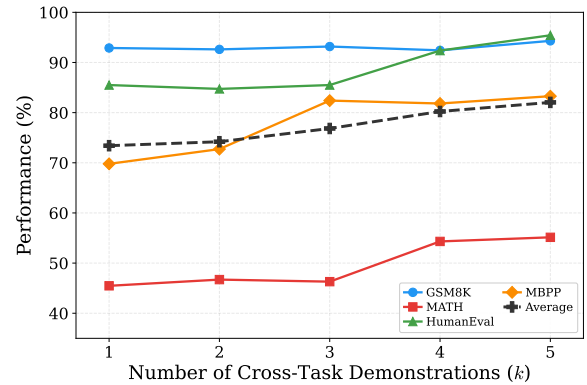


Figure 3: Effect of demo count (k) on performance. GSM8K saturates at $k=1$; MATH jumps at $k=4$, suggesting complex tasks need more demos to disambiguate workflow topology.

4.4 Portability Across Execution Backbones (RQ4)

We evaluate whether SWIFT-generated workflows transfer across execution models by running the same synthesized workflows with Grok-4.1-fast, Qwen3-8B (Team, 2025), and Gemma-3-12B-IT (Team et al., 2025) (Table 5). This experiment isolates the portability of the synthesized workflow structure: the graph and operator instructions are held fixed, while the LLM used to execute each operator changes. SWIFT improves accuracy on 16 of 18 benchmark-model pairs, with the largest gains on MATH and MBPP. The gains vary across models, suggesting that workflow transfer can be affected by the execution backbone’s ability to follow multi-step instructions reliably.

Table 5: Generality across model families. Bold = best per backbone-benchmark pair.

Backbone	Method	GSM8K	MATH	HumanEval	MBPP	DROP	HotpotQA	Avg.
Grok-4.1-fast	Vanilla	92.23	44.03	87.02	64.29	68.91	55.15	68.61
	SWIFT (Ours)	93.93	59.05	91.60	85.30	71.17	78.00	79.84
	Δ	+1.70	+15.02	+4.58	+21.01	+2.26	+22.85	+11.24
Qwen3-8B	Vanilla	88.44	61.11	85.50	66.28	49.69	65.49	69.42
	SWIFT (Ours)	92.51	66.46	88.55	72.73	83.21	72.68	79.36
	Δ	+4.07	+5.35	+3.05	+6.45	+33.52	+7.19	+9.94
Gemma-3-12B-IT	Vanilla	91.85	58.23	84.73	71.55	63.13	71.19	73.45
	SWIFT (Ours)	93.46	48.35	87.02	75.95	63.00	73.36	73.52
	Δ	+1.61	-9.88	+2.29	+4.40	-0.13	+2.17	+0.08

5 Conclusion

In this work, we introduced SWIFT, a framework that reframes LLM workflow optimization from costly, iterative test-time search to amortized synthesis guided by demonstrations. By conditioning a meta-generator on cross-task programmatic demonstrations, compositional heuristics, and output contracts, SWIFT deduces optimized solver architectures in a single generation pass. Empirically, SWIFT not only matches or exceeds state-of-the-art search baselines like AFlow across 5 benchmarks, but does so while reducing marginal per-task optimization cost by three orders of magnitude given an existing source bank. The ablations further suggest that demonstrations, output contracts, and task-family alignment all contribute to transfer, while human-readable operator names are not the only source of useful signal. Overall, SWIFT provides an efficient, scalable way for the automated design of autonomous AI agents.

Limitations

SWIFT is evaluated primarily on static benchmarks with automatic correctness signals, so the results do not establish effectiveness in stochastic, interactive, or irreversible environments. The method also assumes access to source workflows; under cold-start deployment, the upstream cost of constructing this source bank would be accounted for (Appendix G).

Acknowledgments

This work was supported in part by the US National Science Foundation [III-2232121] and the US National Institutes of Health [R01HG012470]. Shiyi Du was partially supported by a SoftBank Group / Arm Ph.D. Fellowship. Jiayuan Liu and Vincent Conitzer thank the Cooperative AI Foundation, Macroscopic Ventures (formerly Polaris

Ventures / the Center for Emerging Risk Research) and Jaan Tallinn’s donor-advised fund at Founders Pledge for financial support. Weihua Du was supported in part by SoftBank Group Corp. and Arm, Inc. Yue Huang was supported by a JetStream2 Fellowship. Jiayi Li was partially supported by a gift from Arm, Inc. Conflict of Interest: Carl Kingsford is a co-founder of Ellumigen, Inc.

References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *5th International Conference on Learning Representations*.
- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems* 33, pages 1877–1901.
- Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. 2024a. [Autoagents: A framework for automatic agent generation](#). In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 22–30. International Joint Conferences on Artificial Intelligence Organization.

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. 2024b. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.
- Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pages 990–998. PMLR.
- Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*, pages 11733–11763.
- Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378.
- Batu El, Mert Yuksekgonul, and James Zou. 2025. Inefficiencies of meta agents for agent design. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 20815–20824. Association for Computational Linguistics.
- Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2023. Complexity-based prompting for multi-step reasoning. In *The Eleventh International Conference on Learning Representations*.
- Jinglong Gao, Xiao Ding, Lingxiao Zou, Bing Qin, and Ting Liu. 2025. Crossicl: Cross-task in-context learning via unsupervised demonstration transfer. *arXiv preprint arXiv:2505.24143*.
- 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. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.
- Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*.
- Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14165–14178.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.
- Diederik P. Kingma and Max Welling. 2014. Auto-encoding variational Bayes. In *2nd International Conference on Learning Representations, Banff, AB, Canada*.
- Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 158–167.
- Jiachang Liu, Dinghan Shen, Yizhe Zhang, William B Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for gpt-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114.
- Jun Liu, Zhenglun Kong, Peiyan Dong, Changdi Yang, Tianqin Li, Yanyue Xie, Yifan Gong, Xuan Shen, Pu Zhao, Hao Tang, et al. 2026. Structured agent distillation for large language model agents. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems*, pages 3676–3685.
- Jun Liu, Zhenglun Kong, Changdi Yang, Fan Yang, Tianqi Li, Peiyan Dong, Joannah Nanjeyk, Hao Tang, Geng Yuan, Wei Niu, et al. 2025. Rcr-router: Efficient role-aware context routing for multi-agent llm systems with structured memory. *arXiv preprint arXiv:2508.04903*.
- Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2024. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36*, pages 46534–46594.

- Sewon Min, Xinxu Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. [Rethinking the role of demonstrations: What makes in-context learning work?](#) In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.
- Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 2080–2094.
- Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2025. Scaling large language model-based multi-agent collaboration. In *The Thirteenth International Conference on Learning Representations*.
- Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286. PMLR.
- Subhro Roy and Dan Roth. 2015. Solving general arithmetic word problems. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1743–1752.
- Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671.
- Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2025. Agentsquare: Automatic llm agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*.
- 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.
- Jinwei Su, Qizhen Lan, Yinghui Xia, Lifan Sun, Weiyou Tian, Tianyu Shi, and Lewei He. 2026. Difficulty-aware agentic orchestration for query-specific multi-agent workflows. In *Proceedings of the ACM Web Conference 2026*, pages 2060–2070.
- Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.
- Qwen Team. 2025. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.
- Dingzirui Wang, Xuanliang Zhang, Qiguang Chen, Longxu Dou, Xiao Xu, Rongyu Cao, Yingwei Ma, Qingfu Zhu, Wanxiang Che, Binhua Li, et al. 2024a. In-context transfer learning: Demonstration synthesis by transferring similar tasks. *arXiv preprint arXiv:2410.01548*.
- Ningning Wang, Xavier Hu, Pai Liu, He Zhu, Yue Hou, Heyuan Huang, Shengyu Zhang, Jian Yang, Jiaheng Liu, Ge Zhang, et al. 2025a. Efficient agents: Building effective agents while reducing cost. *arXiv preprint arXiv:2508.02694*.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.
- 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.
- Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2025b. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 63897–63911.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.
- Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.
- John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854.
- Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380.

- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36*, pages 11809–11822.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.
- Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Shiheng Chen, and Jing Shao. 2025. MAS-GPT: Training LLMs to build LLM-based multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 72063–72090. PMLR.
- Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. 2025. [Optimizing generative AI by backpropagating language model feedback](#). *Nature*, 639(8055):609–616.
- Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025a. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 75834–75852.
- Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. 2025b. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 76678–76692.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025c. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*.
- Yifan Zhang and Team Math-AI. 2024. American invitational mathematics examination (aime) 2024.
- Yifan Zhang and Team Math-AI. 2025. American invitational mathematics examination (aime) 2025.
- 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 *Proceedings of the 41st International Conference on Machine Learning*, pages 62138–62160.
- Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Serkan Ö Arık. 2025. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*.
- 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*, pages 62743–62767.
- Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pages 66602–66656.

A Dataset Statistics

Table 6 details the statistics of the nine primary evaluation benchmarks used in our experiments. We partition the datasets into in-distribution tasks, which serve as leave-one-out demonstrations, and held-out tasks that are absent from the demonstration pool.

B Algorithm

Algorithm 1 formalizes the complete end-to-end execution pipeline of SWIFT. The procedure consists of an offline phase that contrastively distills structural heuristics from historical failure and success trajectories, followed by an online phase that leverages these distilled priors for efficient, single-pass workflow synthesis.

C Limitations of ScEnsemble for code generation tasks

An important early finding was that ScEnsemble (self-consistency ensemble via text-based majority voting) is unreliable as the terminal correctness criterion for code generation tasks. ScEnsemble selects the most frequent answer among multiple sampled solutions by comparing their textual representations. This is effective for math and QA tasks where the final answer is a short, canonical string (e.g., “42”, “Paris”), but is poorly suited to code because:

1. **Surface-form diversity.** Functionally equivalent programs can differ vastly in variable names, control flow structure, whitespace, and style. Two correct implementations of the same function will almost never be textually identical.
2. **No semantic equivalence check.** Text-based voting treats each character difference as a “disagreement,” so even trivially equivalent programs (e.g., `for i in range(n)` vs. `for i in range(0, n)`) are counted as distinct solutions.
3. **Chimera outputs.** When no clear majority exists, the ensemble prompt may produce a “compromise” snippet that combines fragments from multiple solutions, resulting in code that does not compile or execute.

Observation. Across our experiments, we found that synthesized workflows which rely solely on ScEnsemble for code tasks consistently underperform those that incorporate a **test-driven retry** pattern (generate code → execute against test cases → if tests fail, regenerate with error feedback).

Execution-based validation assesses *functional* correctness rather than surface-form similarity.

This finding also generalizes beyond SWIFT: any workflow system that applies self-consistency voting as the sole correctness criterion for code will encounter the same limitation. Execution-based evaluation is necessary for tasks where output correctness is defined by behavior rather than surface form.

Clarification. Some synthesized workflows (e.g., BigCodeBench, Supplementary Section J) still include ScEnsemble as a pre-filtering step before test execution. This is not contradictory: when ensemble voting serves only as a heuristic candidate selector (with execution-based testing as the final arbiter of correctness), the surface-form similarity issue is mitigated. The failure mode we identify here is using text-based voting as the *terminal* correctness check with no subsequent execution validation.

D Out-of-distribution Failure Analysis

We analyze failure modes on several out-of-distribution (OOD) benchmarks—datasets unseen during demonstration construction. SWIFT synthesizes workflows for these datasets using only in-distribution demonstrations, making failures particularly informative: they reveal where cross-task transfer is helpful or limited. Figure 4 summarizes the OOD performance overview.

D.1 SVAMP: adversarial word problems

Score: 95.8%. **Gain over vanilla:** +14.8 points.

Workflow: 5×Custom (solve) → ScEnsemble (vote) → Programmer (code verification).

SVAMP is designed to test robustness against surface-form variation of grade-school math problems. The gain over vanilla GPT-4o-mini demonstrates strong cross-task transfer from GSM8K/MATH demonstrations: the synthesized ensemble+verification pipeline provides systematic error correction that vanilla prompting lacks.

The 34 remaining failures cluster into two categories:

- **Contradictory/adversarial premises** (18 cases, 53%): Problems where the stated numbers are logically inconsistent, designed to confuse models.
- **Ambiguous referents** (16 cases, 47%): Problems where pronouns or phrases like “how many” can refer to different quantities.

Algorithm 1 Synthesizing Workflows via Few-shot Transfer (SWIFT)

Require: All available tasks $\mathcal{T}_{all}$, Target task $\mathcal{T}_{target}$, Operator library $\mathcal{O}$.

Require: Historical trajectories $\{\tau_{\mathcal{T}}\}_{\mathcal{T} \in \mathcal{T}_{all}}$, Threshold γ .

Require: Meta-generator LLM f_{ϕ} , Contrastive reflection LLM f_{ψ} .

```
1: % Definition: Source tasks  $\mathcal{T}_{src} \leftarrow \mathcal{T}_{all} \setminus \{\mathcal{T}_{target}\}$       {Target-dataset leave-one-out constraint}
2: % Phase 1: Contrastive Trajectory Distillation (Offline)
3: Initialize global heuristics  $\mathcal{H} \leftarrow \emptyset$ , output contracts  $\mathcal{C} \leftarrow \emptyset$ 
4: for each task  $\mathcal{T} \in \mathcal{T}_{src}$  do
5:    $\tau \leftarrow \tau_{\mathcal{T}}$ 
6:    $w_{best} \leftarrow \arg \max_{w \in \tau} \text{Acc}(w)$       {Select the best validation workflow}
7:    $\tau_{low} \leftarrow \{w \in \tau \mid 0 < \text{Acc}(w) < \gamma \cdot \text{Acc}(w_{best})\}$       {Low-scoring workflows ( $\gamma=0.6$ )}
8:    $\tau_{zero} \leftarrow \{w \in \tau \mid \text{Acc}(w) = 0\}$       {Collapsed workflows}
9:    $w_{low} \leftarrow \arg \min_{w \in \tau_{low}} \text{Acc}(w)$ ;  $w_{zero} \leftarrow \text{first}(\tau_{zero})$       {Hard negatives}
10:  Extract execution traces and parsing error logs  $E_{low}, E_{zero}$  from  $w_{low}, w_{zero}$ 
11:  % Distill topological priors and functional contracts via contrastive reflection
12:   $\mathcal{H}_{\mathcal{T}}, \mathcal{C}_{\mathcal{T}} \leftarrow f_{\psi}(w_{best}, w_{low}, E_{low}, w_{zero}, E_{zero})$ 
13:   $\mathcal{H} \leftarrow \mathcal{H} \cup \mathcal{H}_{\mathcal{T}}$ ,  $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}_{\mathcal{T}}$ 
14: end for
15: % Phase 2: Few-Shot Amortized Synthesis (Online)
16: % Retrieve programmatic demonstrations exclusively from source tasks (OOD)
17:  $\mathcal{D}_{demo} \subset \{w_{best}^{(\mathcal{T})} \mid \mathcal{T} \in \mathcal{T}_{src}\}$ 
18: Construct meta-prompt  $P = (\mathcal{O}, \mathcal{H}, \mathcal{C}, \mathcal{D}_{demo}, \mathcal{T}_{target})$ 
19: % End-to-end Contract-Guided Generation (Single generation pass)
20:  $w_{target} \leftarrow f_{\phi}(P)$       {Workflow natively adheres to  $\mathcal{H}$  and  $\mathcal{C}$ }
21: return Executable workflow  $w_{target}$ 
```

Table 6: Dataset statistics. *Unseen* denotes datasets not included in the demonstration pool during workflow generation. [†] AIME combines AIME 2024 and AIME 2025.

Dataset	Task Type	Test	Val	Unseen
<i>In-Distribution (used as leave-one-out demonstrations)</i>				
GSM8K	Math (Elementary)	1,055	264	
MATH	Math (Competition)	486	119	
HumanEval	Code Generation	131	33	
MBPP	Code Generation	341	86	
<i>Out-of-Distribution (entirely absent from demonstration pool)</i>				
MultiArith	Math (Elementary)	480	120	✓
SVAMP	Math (Elementary)	800	200	✓
AIME [†]	Math (Competition)	48	12	✓
AQUA	Math (Multiple Choice)	204	50	✓
BigCodeBench	Code Generation	912	228	✓
Total		4,457	1,112	

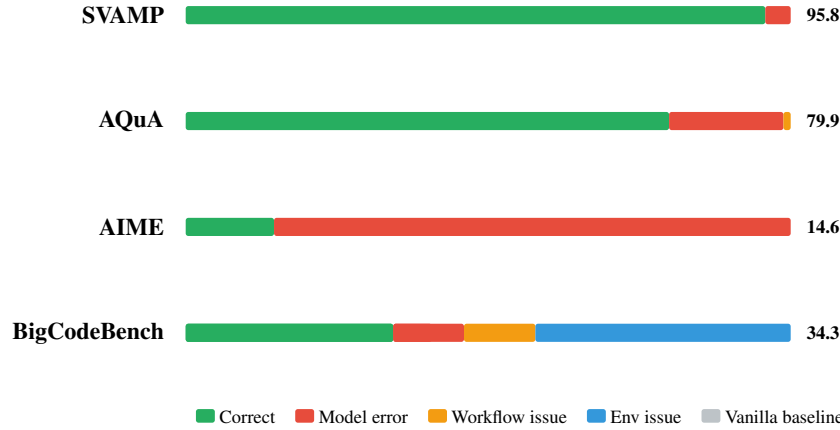


Figure 4: OOD performance overview. Each SWIFT bar decomposes the full test set into correct (green), execution model errors (red), workflow design issues (orange), and infrastructure errors (blue). AIME is dominated by model errors (competition-level math). BigCodeBench is the only benchmark where infrastructure issues (missing Python modules) constitute the largest failure category.

Case 1: Contradictory premise (score = 0.0).

Problem: “A waiter had 3 customers. After some left he still had 4 customers. How many more customers stayed behind than those that left?”

Expected: 5.0 **Predicted:** 2

Analysis. Starting with 3 and “still having 4” after some leave is impossible. All five reasoning paths produced answers based on the literal (contradictory) text, and ScEnsemble converged on 2. Even the Programmer verification step faithfully computed from the inconsistent numbers. **Root cause:** SVAMP deliberately includes adversarially constructed problems. The workflow has no mechanism to flag or reinterpret contradictory premises.

Case 2: Distractor numbers (score = 0.0).

Problem: “There were 61 parents and 177 pupils at the school concert. The program could seat 44 people. How many people were present?”

Expected: 238.0 **Predicted:** 44

Analysis. The question asks “how many people were present” ($61 + 177 = 238$), but the model latched onto the distractor number (seating capacity = 44). The ensemble reinforced this because all five paths were confused by the same irrelevant detail. **Root cause:** SVAMP’s adversarial construction places distractors that systematically mislead all paths simultaneously.

D.2 AIME 2024–2025: competition mathematics

Score: 14.6%. **Gain over vanilla:** +6.3 points.

Workflow: Programmer (code solve) → Custom (interpret) → 3×Custom (alternatives) → ScEnsemble → Custom (extract `\boxed{ }`).

AIME is the hardest OOD benchmark, with an 85.4% failure rate. Even the most elaborate synthesized workflow (code+4-path ensemble+extraction) cannot compensate for GPT-4o-mini’s limited mathematical reasoning on competition-level problems. The +6.3 gain over vanilla shows that the workflow provides *some* benefit (primarily from the code-verification step), but the ceiling is low.

Case 1: Combinatorial geometry (score = 0.0).

Problem: “Find the number of rectangles that can be formed inside a fixed regular dodecagon where each side lies on a side or diagonal.”

Expected: 315 **Predicted:** `\boxed{36}`

Analysis. The Programmer operator generated code enumerating only axis-aligned rectangles (36), missing the rotated ones. The three Custom paths also produced underestimates. **Root cause:** Competition-level geometry exceeds GPT-4o-mini’s capability, and the code-verification step implements an incomplete enumeration.

Case 2: Number theory (score = 0.0).

Problem: “Let A be the set of positive integer divisors of 2025. Let B be a randomly selected subset of A . The probability that B is a nonempty set with LCM equal to 2025 is m/n . Find $m+n$.”

Expected: 237 **Predicted:** `\boxed{32777}`

Analysis. The predicted answer is two orders of magnitude off, indicating a fundamental misapplication of inclusion-exclusion. **Root cause:** Advanced number theory exceeds the execution model’s reasoning depth.

Case 3: Walking speed optimization (score = 0.0).

Problem: “Aya walks at different speeds during a 9-block walk. Blocks 1–3 at s ft/s, blocks 4–6 at $2s$ ft/s, blocks 7–9 at a specified pattern. She takes 25% more time than walking all 9 blocks at s ft/s. Find s^2 .”

Expected: 204 **Predicted:** `\boxed{180}`

Analysis. The Programmer code solved for s^2 but set up the time equation incorrectly, computing 180 instead of 204. **Root cause:** Multi-step algebraic word problems with fraction arithmetic compound errors through the pipeline.

E Dataset statistics and representative examples

This section supplements the benchmark overview in the main paper with problem-level statistics. Table 7 reports the problem length distribution for all benchmarks, and Figure 6 visualizes the full min–mean–max range.

E.1 MATH benchmark: topic distribution

The MATH test set (Level 5, highest difficulty) spans four topic areas (Figure 7).

F Reproducibility details

F.1 Operator library

SWIFT reuses the operator library introduced by AFlow (Zhang et al., 2025c). Table 8 defines the six operators available to the workflow synthesizer.

F.2 Computational cost

Tables 9 and 10 detail the computational cost and time required for workflow optimization and benchmark execution using the GPT-4o-mini API. Notably, our approach is highly cost-effective, incurring a total execution cost of only \$6.08 across all nine benchmarks.

G Cost Accounting and Amortized Scaling

We provide a formal accounting of the cost structure underlying SWIFT, addressing how the efficiency comparison in Table 3 changes when upstream source-trajectory construction is charged to SWIFT.

Source-bank accounting. Under the target-dataset leave-one-out protocol, the trajectories available to SWIFT when synthesizing a workflow for a target task $\mathcal{T}_{\text{target}}$ are produced exclusively from the optimization of other source tasks $\mathcal{T}_{\text{src}} = \mathcal{T}_{\text{all}} \setminus \{\mathcal{T}_{\text{target}}\}$. For example, when the target is GSM8K, the trajectory pool excludes all GSM8K workflows and search traces, and is constructed from the remaining source tasks. This prevents direct reuse of the target task’s workflow or search trace, but the upstream cost of constructing the source bank should still be considered under cold-start deployment.

Scaling law comparison. Let $C_{\text{search}}(T)$ denote the optimization cost of a search-based method on task T , and let C_{synth} denote SWIFT’s marginal

AIME: Per-Problem Results (48 problems)

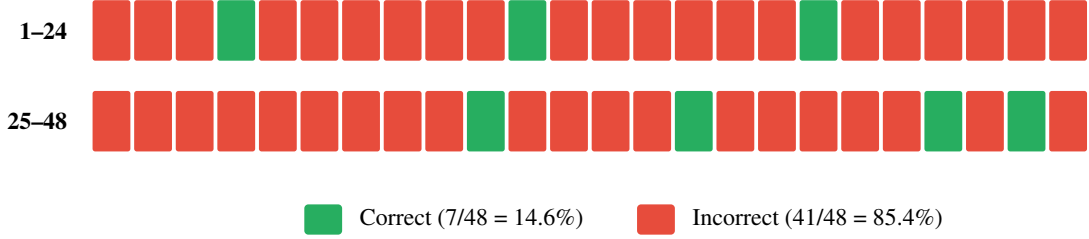


Figure 5: AIME per-problem results. Each cell represents one problem; green = correct, red = incorrect. Only 7 of 48 problems are solved correctly (scattered across problem indices), suggesting that failures are not clustered in “hard” problem positions but reflect execution-model limitations across diverse mathematical topics.

Table 7: Problem length statistics for the evaluation benchmarks.

Benchmark	Task Type	Test	Problem Length (words)		
			Mean	Min	Max
GSM8K	Math	1,055	46	15	164
MATH	Math	486	40	2	211
HumanEval	Code	131	71	18	249
MBPP	Code	341	19	9	81
DROP	QA	800	188	54	931
HotpotQA	QA	800	16	6	46
MultiArith	Math	480	32	23	45
SVAMP	Math	800	32	15	57
AQuA	Math	204	45	9	92
AIME	Math	48	55	18	115
BigCodeBench	Code	912	148	43	580

target-task design cost once source trajectories are available, including distillation over those trajectories and workflow synthesis. For n target tasks, the two paradigms incur:

$$C_{\text{total}}^{\text{search}} = \sum_{i=1}^n C_{\text{search}}(T_i) \approx n \cdot C_{\text{search}}, \quad (1)$$

$$C_{\text{total}}^{\text{amortized}} = \underbrace{C_{\text{source}}}_{\text{one-time}} + n \cdot C_{\text{synth}}, \quad (2)$$

where $C_{\text{source}} = \sum_{T \in \mathcal{T}_{\text{src}}} C_{\text{search}}(T)$ is the total cost of collecting source-task trajectories. In our setting, $C_{\text{synth}} \approx \$0.004$, while $C_{\text{search}} \approx \$22.50$ for the representative search-based baseline in Table 3.

Break-even analysis. Even under a conservative accounting that charges C_{source} entirely to SWIFT, amortized synthesis becomes cheaper when

$$C_{\text{source}} + n \cdot C_{\text{synth}} < n \cdot C_{\text{search}}, \quad (3)$$

yielding a break-even point of

$$n^* = \frac{C_{\text{source}}}{C_{\text{search}} - C_{\text{synth}}} \approx \frac{112.50}{22.50 - 0.004} \approx 5. \quad (4)$$

That is, if the full source-task budget is attributed to SWIFT, amortized synthesis becomes cheaper after roughly five downstream tasks. If a source bank is already available, the relevant cost is instead the marginal target-task design cost. Thus, SWIFT is most cost-effective when optimized workflows or search traces are already available, or when many related tasks will be optimized over time.

From cost reduction to structural insight. The cost results also reflect a structural property of the evaluated workflow design problems. Per-task search often discovers related structures across tasks, such as multi-path ensembles for math and test-driven retry loops for code. SWIFT uses these repeated structures as reusable priors for target-task synthesis. From this perspective, the target-dataset leave-one-out protocol prevents direct reuse

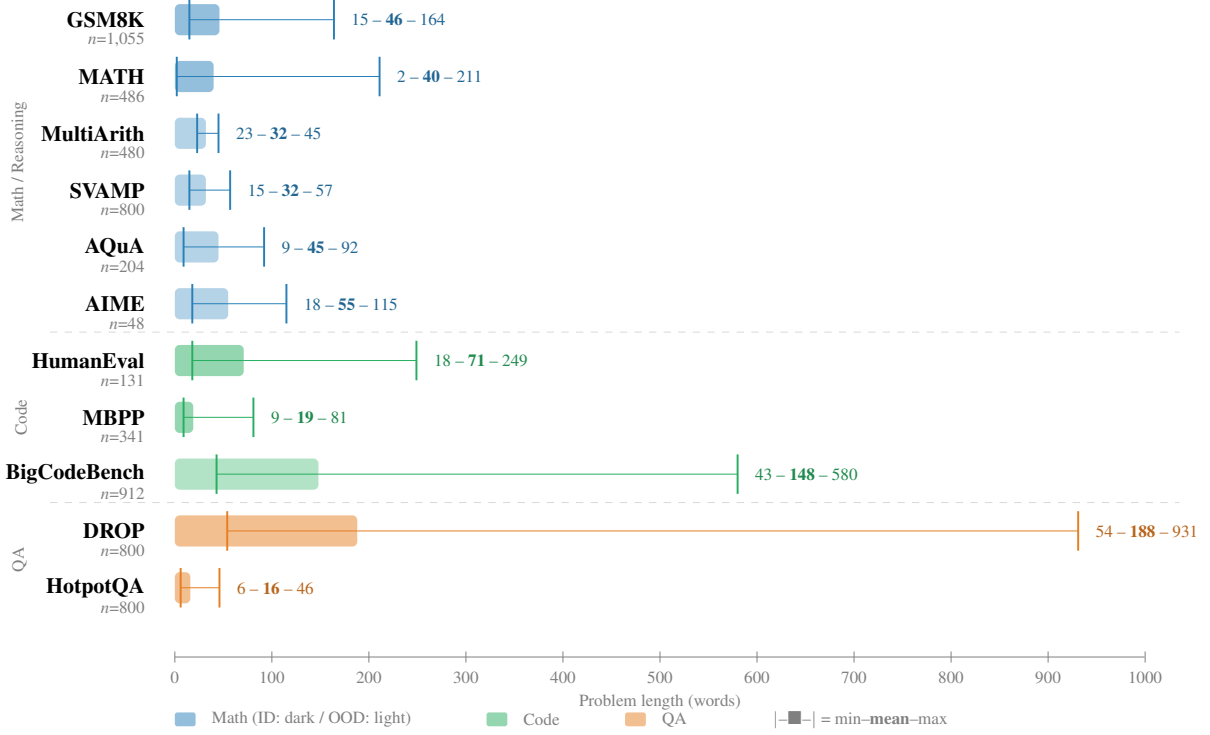


Figure 6: Problem length distribution (words) for all 11 benchmarks. Filled bars show the mean; whiskers extend to min and max. DROP has the longest problems (passages averaging 188 words, max 931). BigCodeBench is second (code docstrings averaging 148 words, max 580). MATH has the widest variance among math benchmarks (2–211 words), reflecting diverse problem types. HotpotQA and MBPP have the shortest prompts (mean 16 and 19 words, respectively).

of the held-out task while testing whether source-task structures can guide workflow design for a new target.

Table 10 provides a detailed per-benchmark breakdown of distillation, synthesis, and execution costs.

H Effect of Output Contracts on Synthesized Workflow Topology

To empirically demonstrate that output contracts influence the *structural design* of synthesized workflows, not merely the terminal output format, we compare the workflow topologies generated with and without contracts ($\mathcal{C}$) across four benchmarks. Table 11 summarizes the structural differences across all four datasets.

Key observations. (1) On GSM8K and HumanEval, removing contracts causes the synthesizer to triple the node count by adding multi-candidate generation and ensemble voting as hedging mechanisms, yet accuracy *drops* by 4–5 points. (2) On MATH, both versions use ensemble voting, but the contract version selects a more suitable operator (Custom vs. AnswerGenerate) and

generates more candidates (5 vs. 3). (3) MBPP shows near-identical topology in both conditions, suggesting that contracts primarily reshape architectures for tasks with complex output interfaces (math extraction, code validation) rather than tasks where the operator library already enforces output structure (e.g., Test operator in code tasks).

I Additional Experiments and Analyses

This section provides additional analyses of demonstration selection, interactive execution, generator–executor coupling, synthesis variability, environment-related failures, and retrieval-based workflow reuse.

I.1 Semantic Demonstration Retrieval

We evaluate target-conditioned semantic retrieval on the four held-out benchmarks reported in the main text. For each target, a frozen semantic encoder represents only its task-level metadata. Historical workflow demonstrations are ranked by cosine similarity, and the top- k workflows are selected from the same demonstration pool used by SWIFT. The target task is excluded from the candi-

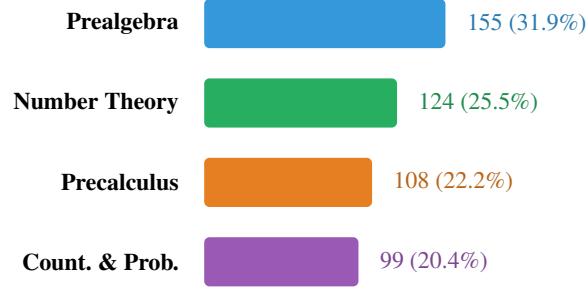


Figure 7: Topic distribution in the MATH test set (486 Level-5 problems). All four categories are roughly balanced, with Prealgebra slightly overrepresented.

Table 8: Operator library specification. Each operator is a reusable building block with a fixed interface. The workflow synthesizer composes these operators into task-specific pipelines.

Operator	Input	Output	Description
Custom	<i>text + instruction</i>	<i>text response</i>	General-purpose text generation with custom instruction
AnswerGenerate	<i>question text</i>	<i>thought + answer</i>	Structured reasoning with thought chain
Programmer	<i>problem description</i>	<i>code + output</i>	Generates and executes Python code
CustomCodeGenerate	<i>problem + entry point</i>	<i>code string</i>	Generates code matching a function signature
ScEnsemble	<i>list of solutions</i>	<i>best solution</i>	Majority voting / selection
Test	<i>code + test cases</i>	<i>pass/fail + output</i>	Executes code against test cases

date pool before retrieval. Neither target examples nor labels are used, and all other synthesis and execution settings remain unchanged.

Table 12 shows that semantic retrieval achieves the highest macro-average accuracy at $k = 1$. The effect is target-dependent: $k = 1$ performs best on SVAMP and AIME, whereas $k = 2$ performs best on AQUA. None of the semantic variants exceeds the original SWIFT result on BigCodeBench. Thus, semantic similarity provides a useful selection signal, but retrieving more demonstrations is not monotonically beneficial and may introduce negative transfer.

I.2 Interactive Execution on InterCode-Python

We evaluate whether a workflow synthesized and frozen before evaluation can operate in a multi-turn, feedback-conditioned setting using InterCode-Python (Yang et al., 2023). Each task is evaluated

as exactly one interactive episode. The model submits an action, receives execution feedback and a scalar reward, and conditions its subsequent action on the updated observation, for up to seven turns. The synthesized workflow remains fixed across all tasks and turns, although its execution may contain conditional branches and bounded feedback or retry loops.

We report two metrics. *First-turn success* evaluates the initial action before environment feedback is available. *Episode success* counts a task as solved if any of its feedback-conditioned turns succeeds.

As shown in Table 13, SWIFT is numerically higher under both measures. We interpret these modest gains as preliminary evidence of interactive extensibility: a fixed workflow can condition its execution path and subsequent actions on dynamically updated observations. These results do not establish broad generality across interactive,

Table 9: Computational cost breakdown per benchmark during execution. All costs are for GPT-4o-mini API calls. The total experiment cost across all 9 benchmarks is approximately \$6.08. Workflow synthesis cost (one-time, per seed) is negligible (<\$0.01 per benchmark).

Benchmark	Problems	LLM Calls per Problem	Avg Cost per Problem	Total Cost
In-Distribution				
GSM8K	1,055	3	\$0.0011	\$1.20
MATH	486	7	\$0.0036	\$1.73
HumanEval	131	2–3	\$0.0002	\$0.03
MBPP	341	3–4	\$0.0001	\$0.04
Out-of-Distribution				
MultiArith	480	2	\$0.0002	\$0.08
SVAMP	800	7	\$0.0010	\$0.80
AQuA	204	5	\$0.0012	\$0.24
AIME	48	7	\$0.0030	\$0.14
BigCodeBench	912	5–6	\$0.0020	\$1.82
Total (all benchmarks)				\$6.08

Table 10: Detailed cost and time breakdown for the four main benchmarks. **Optimization** includes one-time distillation and synthesis; **Execution** is running the synthesized workflow on the full test set. All costs are GPT-4o-mini API calls. Optimization accounts for less than 0.6% of total expenditure and completes in under 11 seconds per benchmark.

Benchmark	Optimization (one-time)			Execution		Total (\$)
	Distill (\$)	Synth (\$)	Time	Cost (\$)	Time	
GSM8K	0.001	0.003	10.6 s	1.199	35.1 min	1.203
MATH	0.001	0.003	8.1 s	1.729	6.8 min	1.733
HumanEval	0.001	0.003	7.8 s	0.032	1.5 min	0.036
MBPP	0.001	0.003	7.0 s	0.044	1.8 min	0.048
Total	0.005	0.012	33.5 s	3.004	45.2 min	3.021

stochastic, or irreversible environments.

I.3 Generator–Executor Coupling

The primary experiments use GPT-4o-mini as both the workflow meta-generator and execution model. To examine the generator side directly, we additionally evaluate Qwen→GPT: Qwen3-8B synthesizes the workflow, which is then executed by GPT-4o-mini. Together with GPT→GPT and GPT→Qwen, this comparison distinguishes changing the generator while holding the GPT executor fixed from changing the executor while retaining GPT-generated workflows.

Table 14 shows that, with the GPT executor held fixed, Qwen-generated workflows remain relatively

close to GPT-generated workflows on GSM8K, HumanEval, and MBPP. The larger degradation on MATH indicates that synthesis quality and model–task compatibility still matter. These results support partial, task-dependent portability instead of generator-independent workflow synthesis.

I.4 Variability Across Meta-Generation Attempts

We conduct three independent meta-generation attempts on two representative held-out targets spanning mathematical reasoning and code generation: AQuA and BigCodeBench. Table 15 reports the accuracy of each attempt, the mean and sample standard deviation across attempts, and the number

Table 11: Structural comparison of synthesized workflows with and without output contracts ($\mathcal{C}$). “Nodes” counts total operator invocations; “Depth” counts sequential levels. Accuracy from Table 4.

Dataset	With $\mathcal{C}$		Without $\mathcal{C}$		Δ	
	Nodes	Acc	Nodes	Acc	Nodes	Acc
GSM8K	2	94.3	5	89.9	+3	−4.4
MATH	7	55.1	5	50.4	−2	−4.7
HumanEval	2	95.4	6	90.8	+4	−4.6
MBPP	3	83.3	3	80.7	± 0	−2.6

Table 12: Accuracy (%) of target-conditioned semantic demonstration retrieval on four held-out benchmarks. The original SWIFT configuration is compared with semantic retrieval using $k \in \{1, 2, 3\}$.

Dataset	Vanilla	SWIFT	SWIFT Semantic $k = 1$	SWIFT Semantic $k = 2$	SWIFT Semantic $k = 3$
SVAMP	81.0	95.8	95.9	94.5	95.5
AIME 24–25	8.3	14.6	16.7	8.3	4.2
AQuA	52.9	79.9	82.8	84.3	82.4
BigCodeBench	31.6	34.3	31.9	33.7	33.4
Macro average	43.5	56.1	56.8	55.2	53.9

Table 13: Success rates on InterCode-Python. Each task is evaluated using one episode with up to seven turns.

Method	First-turn success	Episode success (up to 7 turns)
Vanilla	0.40	0.48
SWIFT	0.43	0.49

of structurally distinct executable workflows.

All three attempts produce structurally distinct executable workflows on each target. BigCodeBench performance varies little across these attempts, whereas AQuA exhibits greater variation.

I.5 Environment-Normalized BigCodeBench Evaluation

The BigCodeBench failure analysis identifies unavailable sandbox dependencies as an important source of execution failures. We retain the original benchmark score for comparability and additionally report an environment-normalized score as a supplementary diagnostic of environment-related failures.

The environment-normalized score is 55.04%, compared with the original score of 34.32% (Table 16). This difference highlights the contribution of execution-environment limitations to the observed errors.

I.6 Matched Workflow-Reuse Baselines

We compare SWIFT with two non-trained, semantic-retrieval-based workflow-reuse controls.

Direct-Reuse retrieves the most similar source-task workflow and executes it unchanged. **Explicit-Adapt** uses one target-conditioned LLM call to adapt the retrieved workflow before execution.

As shown in Table 17, SWIFT exceeds Direct-Reuse and Explicit-Adapt by 6.05 and 3.48 percentage points in macro-average accuracy, respectively. Explicit adaptation improves the macro average over direct reuse but is not uniformly beneficial: it improves GSM8K, MATH, and MBPP, while reducing HumanEval accuracy. Direct-Reuse remains competitive on HumanEval, and Explicit-Adapt matches SWIFT on MBPP. Thus, simple nearest-workflow reuse or one-call adaptation does not fully account for the aggregate gains of SWIFT, but the benefit of synthesis remains task-dependent.

J Workflow Examples

Below we show the codes of the workflow examples. Boilerplate (imports, `__init__`) is omitted for clarity.

GSM8K. Three independent reasoning paths are generated and aggregated via SCENSEMBLE (self-consistency voting), followed by a dedicated extraction step.

MATH. Same multi-path + ensemble structure as GSM8K, with a $\text{\LaTeX} \backslash \boxed{\}$ extraction step tailored to the MATH answer format

HumanEval. A test-driven retry loop: generate code, run unit tests, and if the tests fail, analyze the

Table 14: Accuracy (%) under different generator–executor configurations. GPT denotes GPT-4o-mini, and Qwen denotes Qwen3-8B. Arrows indicate workflow generator → execution model. Macro averages are computed over the four benchmarks shown.

Benchmark	GPT→GPT	GPT→Qwen	Qwen→GPT
GSM8K	94.31	92.51	92.61
MATH	55.14	66.46	45.19
HumanEval	95.42	88.55	93.89
MBPP	83.28	72.73	82.70
Macro average	82.04	80.06	78.60

Table 15: Accuracy (%) across three independent meta-generation attempts. Mean $\pm$ SD summarizes performance across attempts. “Distinct” reports the number of structurally distinct executable workflows among the three attempts.

Target	Domain	Attempt 1	Attempt 2	Attempt 3	Mean $\pm$ SD	Distinct
AQuA	Math	79.90	83.33	75.98	79.74 $\pm$ 3.68	3/3
BigCodeBench	Code	34.32	33.88	34.32	34.17 $\pm$ 0.25	3/3

Table 16: Original and environment-normalized BigCodeBench accuracy (%). The normalized score supplements, rather than replaces, the original benchmark result.

Method	Original	Environment-normalized
SWIFT	34.32	55.04

error and regenerate, up to two repair attempts.

MultiArith. A simple two-step workflow: detailed step-by-step reasoning followed by numerical extraction. Despite having no MultiArith demonstrations, SWIFT synthesizes a clean “solve then extract” pattern transferred from the math-reasoning source demos.

AIME. A hybrid workflow combining PROGRAMMER (code-assisted computation) with multi-path reasoning and ensemble. This is the most complex OOD workflow, reflecting the difficulty of competition-level math.

BigCodeBench. Three candidate implementations voted via SCENSEMBLE, validated by unit tests, with a single retry on failure. The test-driven pattern is transferred from the HumanEval and MBPP source demos.

K Use of LLM Disclosure

Large language models were used in this work for (1) evaluation of the proposed methods (as described in the experimental setup) and (2) improving the clarity of the writing.

Table 17: Accuracy (%) of SWIFT and matched workflow-reuse baselines under task-level leave-one-out evaluation. Direct-Reuse executes the retrieved source workflow unchanged; Explicit-Adapt uses one target-conditioned LLM call to adapt it before execution.

Dataset	SWIFT	Direct-Reuse	Explicit-Adapt
GSM8K	94.31	85.12	89.76
MATH	55.14	41.77	49.59
HumanEval	95.42	94.66	91.60
MBPP	83.28	82.40	83.28
Macro average	82.04	75.99	78.56

Listing 1: GSM8K workflow.

```

1 class Workflow:
2     def __init__(self, name, llm_config, dataset):
3         self.name = name
4         self.dataset = dataset
5         self.llm = create_llm_instance(llm_config)
6
7         # Operators
8         self.custom = Custom(self.llm)
9         self.sc_ensemble = ScEnsemble(self.llm)
10
11     async def __call__(self, problem: str):
12         # Step 1: Generate 3 independent reasoning paths
13         solutions = []
14         for _ in range(3):
15             reasoning = await self.custom(
16                 input=problem,
17                 instruction=(
18                     "Break down the problem step by step, "
19                     "extract the necessary numbers, perform "
20                     "the calculations, and provide only the "
21                     "final answer as a single number."
22                 )
23             )
24             solutions.append(reasoning['response'])
25
26         # Step 2: Self-consistency ensemble voting
27         ensemble_result = await self.sc_ensemble(
28             solutions=solutions,
29             problem=problem
30         )
31
32         # Step 3: Extract clean numerical answer
33         final = await self.custom(
34             input=(
35                 f"Final answer based on reasoning: "
36                 f"{ensemble_result['response']}"
37             ),
38             instruction=(
39                 "Extract ONLY the final numerical answer. "
40                 "No units, no explanation."
41             )
42         )
43
44         return final['response'], self.llm.cost_manager.total_cost

```

Listing 2: MATH workflow.

```

1 class Workflow:
2     def __init__(self, name, llm_config, dataset):
3         self.name = name
4         self.dataset = dataset
5         self.llm = create_llm_instance(llm_config)
6
7         # Operators
8         self.custom = Custom(self.llm)
9         self.sc_ensemble = ScEnsemble(self.llm)
10
11     async def __call__(self, problem: str):
12         # Step 1: Generate 3 independent solutions
13         solutions = []
14         for _ in range(3):
15             solution = await self.custom(
16                 input=problem,
17                 instruction=(
18                     "Solve the problem step by step "
19                     "and show all calculations."
20                 )
21             )
22             solutions.append(solution['response'])
23
24         # Step 2: Self-consistency ensemble voting
25         final_solution = await self.sc_ensemble(
26             solutions=solutions,
27             problem=problem
28         )
29
30         # Step 3: Extract answer in LaTeX \boxed{} format
31         formatted_answer = await self.custom(
32             input=(
33                 f"Final answer: "
34                 f"{final_solution['response']}"
35             ),
36             instruction=(
37                 "Extract ONLY the final answer "
38                 "in \\boxed{} format."
39             )
40         )
41
42         return (
43             formatted_answer['response'],
44             self.llm.cost_manager.total_cost
45         )

```

Listing 3: HumanEval workflow (Part 1: Attempt 1 and 2).

```

1 class Workflow:
2     def __init__(self, name, llm_config, dataset):
3         self.name = name
4         self.dataset = dataset
5         self.llm = create_llm_instance(llm_config)
6         # Operators
7         self.custom = Custom(self.llm)
8         self.custom_code_generate = CustomCodeGenerate(self.llm)
9         self.test = Test(self.llm, dataset=self.dataset)
10    async def __call__(self, problem: str, entry_point: str):
11        # ---- Attempt 1: initial generation ----
12        initial_solution = await self.custom_code_generate(
13            problem=problem,
14            entry_point=entry_point,
15            instruction=(
16                "Generate a Python function that meets "
17                "the specifications in the docstring."
18            )
19        )
20        test_result = await self.test(
21            problem=problem,
22            solution=initial_solution['response'],
23            entry_point=entry_point
24        )
25        if test_result['result']:
26            return (
27                initial_solution['response'],
28                self.llm.cost_manager.total_cost
29            )
30
31        # ---- Attempt 2: analyze failure and fix ----
32        analysis = await self.custom(
33            input=(
34                f"Problem: {problem}\n"
35                f"Failed code: "
36                f"{initial_solution['response']}\n"
37                f"Error: {test_result['solution']}"
38            ),
39            instruction=(
40                "Analyze why this code failed "
41                "and how to fix it"
42            )
43        )
44
45        improved = await self.custom_code_generate(
46            problem=problem,
47            entry_point=entry_point,
48            instruction=(
49                f"Fix the code based on: "
50                f"{analysis['response']}"
51            )
52        )
53
54        test_result2 = await self.test(
55            problem=problem,
56            solution=improved['response'],
57            entry_point=entry_point
58        )
59
60        if test_result2['result']:
61            return (
62                improved['response'],
63                self.llm.cost_manager.total_cost
64            )

```

Listing 4: HumanEval workflow (continued; Part 2: Attempt 3).

```
65     # ---- Attempt 3: regenerate from scratch ----
66     final = await self.custom_code_generate(
67         problem=problem,
68         entry_point=entry_point,
69         instruction=(
70             f"Previous attempts failed. "
71             f"Error: {test_result2['solution']}. "
72             f"Generate a correct solution."
73         )
74     )
75
76     return (
77         final['response'],
78         self.llm.cost_manager.total_cost
79     )
```

Listing 5: MultiArith workflow.

```

1 class Workflow:
2     def __init__(self, name, llm_config, dataset):
3         self.name = name
4         self.dataset = dataset
5         self.llm = create_llm_instance(llm_config)
6
7         # Operators
8         self.custom = Custom(self.llm)
9
10    async def __call__(self, problem: str):
11        # Step 1: Detailed step-by-step reasoning
12        detailed_response = await self.custom(
13            input=problem,
14            instruction=(
15                "You are a skilled mathematician. Please "
16                "solve the following arithmetic word "
17                "problem step-by-step:\n"
18                "1. Read and extract all relevant "
19                "numerical information and operations.\n"
20                "2. Clearly state the mathematical "
21                "operations in a logical sequence.\n"
22                "3. Perform the calculations step-by-step, "
23                "showing your work clearly.\n"
24                "4. Provide a concise final answer as a "
25                "single numerical value without any "
26                "additional text or units."
27            )
28        )
29
30        # Step 2: Extract clean numerical answer
31        final_answer = await self.custom(
32            input=detailed_response['response'],
33            instruction=(
34                "Given the detailed calculations and the "
35                "answer derived from the arithmetic "
36                "problem above, please provide ONLY the "
37                "numerical answer without any additional "
38                "text or units."
39            )
40        )
41
42        return (
43            final_answer['response'],
44            self.llm.cost_manager.total_cost
45        )

```

Listing 6: AIME workflow.

```

1 class Workflow:
2     def __init__(self, name, llm_config, dataset):
3         self.name = name
4         self.dataset = dataset
5         self.llm = create_llm_instance(llm_config)
6         # Operators
7         self.custom = Custom(self.llm)
8         self.programmer = Programmer(self.llm)
9         self.sc_ensemble = ScEnsemble(self.llm)
10    async def __call__(self, problem: str):
11        # Step 1: Code-assisted computation
12        code_solution = await self.programmer(
13            problem=problem
14        )
15        # Step 2: Integrate code output with reasoning
16        formatted_solution = await self.custom(
17            input=(
18                problem + f"\nCode output: "
19                f"{code_solution['output']}"
20            ),
21            instruction=(
22                "Given the problem and code output, "
23                "provide a detailed solution with LaTeX. "
24                "Show step-by-step calculations. "
25                "Present the final answer in "
26                "\\boxed{} notation."
27            )
28        )
29        # Step 3: Generate 3 additional reasoning paths
30        solutions = [formatted_solution['response']]
31        for _ in range(3):
32            alternative = await self.custom(
33                input=problem,
34                instruction=(
35                    "Solve step by step with LaTeX. "
36                    "Outline the approach and relevant "
37                    "formulas. Provide detailed "
38                    "calculations. Present the final "
39                    "answer in \\boxed{} notation."
40                )
41            )
42            solutions.append(alternative['response'])
43
44        # Step 4: Self-consistency ensemble voting
45        final_solution = await self.sc_ensemble(
46            solutions=solutions,
47            problem=problem
48        )
49
50        # Step 5: Extract final answer in \\boxed{} format
51        refined = await self.custom(
52            input=final_solution['response'],
53            instruction=(
54                "Extract only the final answer. "
55                "Respond strictly in the format "
56                "\\boxed{answer}, without any "
57                "additional text or explanation."
58            )
59        )
60
61        return (
62            refined['response'],
63            self.llm.cost_manager.total_cost
64        )

```

Listing 7: BigCodeBench workflow.

```

1 class Workflow:
2     def __init__(self, name, llm_config, dataset):
3         self.name = name
4         self.dataset = dataset
5         self.llm = create_llm_instance(llm_config)
6
7         # Operators
8         self.custom_code_generate = CustomCodeGenerate(
9             self.llm
10        )
11        self.sc_ensemble = ScEnsemble(self.llm)
12        self.test = Test(self.llm, dataset=self.dataset)
13
14    async def __call__(self, problem: str, entry_point: str):
15        # Step 1: Generate 3 candidate implementations
16        solutions = []
17        for _ in range(3):
18            solution = await self.custom_code_generate(
19                problem=problem,
20                entry_point=entry_point,
21                instruction=""
22            )
23            solutions.append(solution['response'])
24
25        # Step 2: Ensemble voting to select best candidate
26        best_solution = await self.sc_ensemble(
27            solutions=solutions,
28            problem=problem
29        )
30
31        # Step 3: Validate with unit tests
32        test_result = await self.test(
33            problem=problem,
34            solution=best_solution['response'],
35            entry_point=entry_point
36        )
37
38        if test_result['result']:
39            return (
40                test_result['solution'],
41                self.llm.cost_manager.total_cost
42            )
43
44        # Step 4: Single retry with error context
45        improved = await self.custom_code_generate(
46            problem=problem,
47            entry_point=entry_point,
48            instruction=(
49                "The previous solution did not pass "
50                "the test cases. Analyze requirements "
51                "carefully and generate a new correct "
52                "solution."
53            )
54        )
55
56        return (
57            improved['response'],
58            self.llm.cost_manager.total_cost
59        )

```