OPWEAVE: SELF-EXPANDING AGENTIC WORKFLOWS SEARCH WITH HIERARCHICAL TRANSFERABLE MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated agentic workflow design typically searches a fixed operator library, limiting adaptation when required capabilities are absent and forcing repeated search across tasks. We present OpWeave, a framework that self-expands, optimizes, and reuses a verified language of agentic workflows. From a small development subset, OpWeave profiles the task, retrieves procedural evidence and prior experience, and uses this evidence to propose typed operators and reusable workflow patterns. OpWeave then optimizes an initial workflow using development-set feedback and execution traces, where hierarchical attribution localizes failures at progressively finer granularity to keep each repair targeted. Verified elements and compact experience summaries are retained for reuse across tasks. Across six benchmarks spanning three task categories, OpWeave outperforms the strongest automated workflow design baseline on five of six tasks, with an average gain of 3.1 points. In a search-free setting, OpWeave uses memory priors to guide workflow generation and matches the strongest baseline with only 0.02% of the tokens of a full run. These results show that OpWeave enables adaptive and reusable workflow design by expanding and retaining its workflow language across tasks.

1 INTRODUCTION

Large language models (LLMs) increasingly solve complex tasks through agentic workflows composed of specialized Operators. Because effective workflows vary across tasks, recent work has focused on automating workflow design with minimal manual effort (Khattab et al., 2024; Hu et al., 2025; Zhuge et al., 2024; Zhang et al., 2025b).

Existing methods search within a workflow space whose components are supplied in advance, either by hand or from expert demonstrations. AFlow composes a predefined Operator set (Zhang et al., 2025b); A^2Flow extends this set with Operators derived from expert demonstrations (Zhao et al., 2026). Three limitations remain. First, expansion is narrow: new capabilities come from a single source and arrive only as isolated Operators, without reusable structures. Second, optimization is coarse: methods revise the whole workflow from end-to-end feedback (Hu et al., 2025; Zhang et al., 2025b) or localize failures at one fixed granularity (Ma et al., 2026), even though a failure may lie in one block, one execution path, or several interacting blocks, revisions often touch unaffected parts and raise search cost. Third, experience is short-lived: little is retained within a run (Zhao et al., 2026; Wang et al., 2025b), so similar structures and failures are rediscovered. Recent work on reusable skills and trajectory memory (Shen et al., 2026; Yan et al., 2026) does not close these gaps. Three questions therefore remain open: *what should be available for composition, where should a failing workflow be revised, and how should workflow experience be retained and reused?*

We present OpWeave, a self-expanding agentic workflow search framework; Figure 1 contrasts it with A^2Flow, the most closely related system. *What to compose:* OpWeave expands the workflow language by transforming external knowledge into verified Operators and reusable Patterns, rather than relying on a fixed set or expert demonstrations alone. *Where to revise:* OpWeave uses hierarchical attribution to identify the smallest supported revision scope, enabling targeted optimization instead of global workflow search. *What to retain and reuse:* OpWeave separates search-time experience from transferable long-term priors. These priors guide workflow synthesis and optimization

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

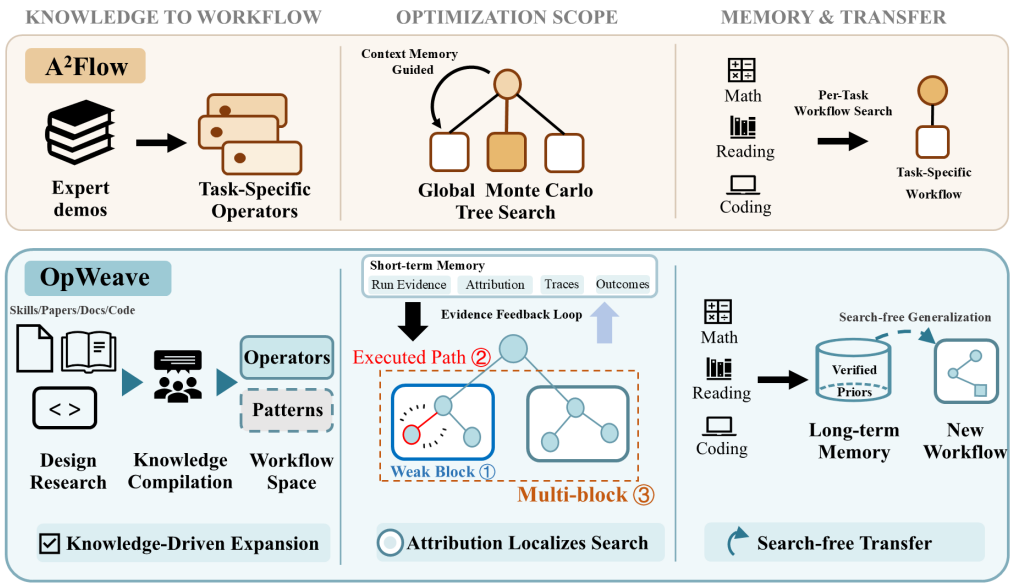


Figure 1: OpWeave versus A^{2}Flow (Zhao et al., 2026) along three axes. Knowledge: new Operators and Patterns are grounded in retrieved skills, papers, documentation, and code, rather than derived from expert demonstrations alone. Optimization: attribution-guided local repair replaces global MCTS search. Memory: long-term priors are kept separate from run evidence, search-free transfer instead of task-bound memory.

on later tasks and support search-free transfer. In this way, OpWeave expands what workflows can express, localizes where they should change, and transfers what previous searches have learned.

We evaluate OpWeave on six benchmarks from three task categories. These include reading comprehension with HotpotQA (Yang et al., 2018) and DROP (Dua et al., 2019), code generation with HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), and mathematical reasoning with GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021). Starting from scratch, OpWeave achieves the highest average score. In a search-free setting, it still achieves a higher average score than all baselines at minimal cost.

Our contributions are as follows:

- We propose **knowledge-driven capability expansion**. The Design Phase turns external knowledge into verified Operators and reusable Patterns. The workflow language therefore expands for each task rather than being fixed or derived only from expert demonstrations.
- We introduce **hierarchical attribution-guided search** and **dual memory**. Hierarchical attribution localizes failures at different workflow scopes and enables focused revisions. Dual memory uses short-term experience to guide within-run optimization and consolidates long-term priors for workflow synthesis and optimization across tasks.
- Experiments across reading, coding, and mathematical reasoning show that OpWeave achieves the highest average score. Without task search, a single agent guided by transferred memory surpasses the baselines. Its workflow language also grows across tasks and is reused in subsequent tasks.

2 Related Work

Automated Workflow Search. Most automated workflow methods search within a supplied design space. DSPy tunes instructions and demonstrations in programmer-defined programs, GPTSwarm optimizes prompts and connections over given node functions, and AFlow searches

over a predefined Operator set (Khattab et al., 2024; Zhuge et al., 2024; Zhang et al., 2025b). Recent work expands the search space in different ways. AgentSquare evolves and recombines modules for planning, reasoning, tool use, and memory (Shang et al., 2025). MaAS samples query-dependent architectures from an agentic supernet (Zhang et al., 2025a), while AgentSwift jointly searches workflows and functional components using hierarchical search (Li et al., 2026). ADAS uses open-ended code search to generate complete agent programs (Hu et al., 2025), whereas A^{2}Flow derives new Operators from expert demonstrations (Zhao et al., 2026). SkillFoundry and OpenSkill compile external resources into tested, portable skills (Shen et al., 2026; Yan et al., 2026), but focus on skill construction rather than workflow language expansion. Together, these lines of work leave open how to build a persistent workflow language from heterogeneous external knowledge, with both Operators and reusable workflow structures.

Workflow Optimization. Workflow optimization methods differ in how they use execution feedback. AFlow and ADAS mainly evaluate complete workflow candidates using end-to-end task performance (Zhang et al., 2025b; Hu et al., 2025). ScoreFlow learns a workflow generator through score-based preference optimization in a continuous space (Wang et al., 2025a). MASS follows a three-stage process of block-level prompt optimization, workflow topology optimization, and workflow-level prompt optimization (Zhou et al., 2026). FlowSteer learns progressive graph edits from syntax validation feedback and terminal rewards (Zhang et al., 2026), while JudgeFlow attributes failures to top-level Blocks and edits the most responsible Block (Ma et al., 2026). These methods either optimize global candidates, follow a predefined optimization schedule, or operate at a fixed attribution granularity. This leaves open how to adapt repair granularity to the failure observed in each execution trace.

Experience Reuse in Workflow Design Prior work reuses experience in several forms. Agent Workflow Memory extracts recurring routines from action trajectories for later agent decisions (Wang et al., 2025b). MetaFlowLLM organizes workflows from completed tasks into a hierarchical experience tree and retrieves relevant MetaFlows for new tasks (Fan et al., 2025). SWIFT distills structural heuristics from previous AFlow searches and uses them to generate workflows for new tasks over a fixed Operator library (Du et al., 2026). These approaches reuse trajectories, workflows, or structural priors, but leave open how such experience can be coupled with a workflow language that continues to evolve across tasks.

3 METHOD

OpWeave has two phases and a dual memory (Figure 2). The Design Phase turns retrieved procedural evidence and recalled experience into verified Operators and Pattern cards, and constructs an initial workflow W_0. The Optimize Phase executes candidate workflows on development set $\mathcal{D}_{\text{dev}}$, attributes failures in their execution traces, and revises the workflow under the selection priority. Short-term memory records experience within a task; at the end of optimization, eligible experience is consolidated into long-term memory for later tasks. We first define the problem and the workflow representation shared by both phases, and then describe the two phases and the memory mechanism.

3.1 PROBLEM FORMULATION

For a task τ, let $\mathcal{D}_{\text{dev}} = \{(x_i, y_i)\}_{i=1}^{n}$ be the development set and $\mathcal{D}_{\text{test}}$ a test set. A workflow W sees only the oracle-free input x_i, without gold answers or hidden tests (code tasks keep their public tests). Executing W yields a prediction and an execution trace, $(\hat{y}_i, z_i) = \text{Exec}(W, x_i)$, and its development performance is $\text{Perf}(W) = \frac{1}{n}\sum_i \phi(\hat{y}_i, y_i)$ for the task metric ϕ. Search keeps a pool $\mathcal{A}$ of at most K candidates, ranked by the selection priority

$$q(W) = \text{Perf}(W) + U(W) - \lambda_c \overline{C}(W) - \lambda_s \overline{S}(W), \tag{1}$$

where U is a capped exploration bonus, $\overline{C}$ is the normalized execution cost estimated from per-Operator priors (Table 7), and $\overline{S}$ is the normalized structural complexity. The priority q guides only pool retention and parent sampling. OpWeave returns $W^\star = \arg\max_W \text{Perf}(W)$ over all candidates evaluated during search, and we report its test performance.

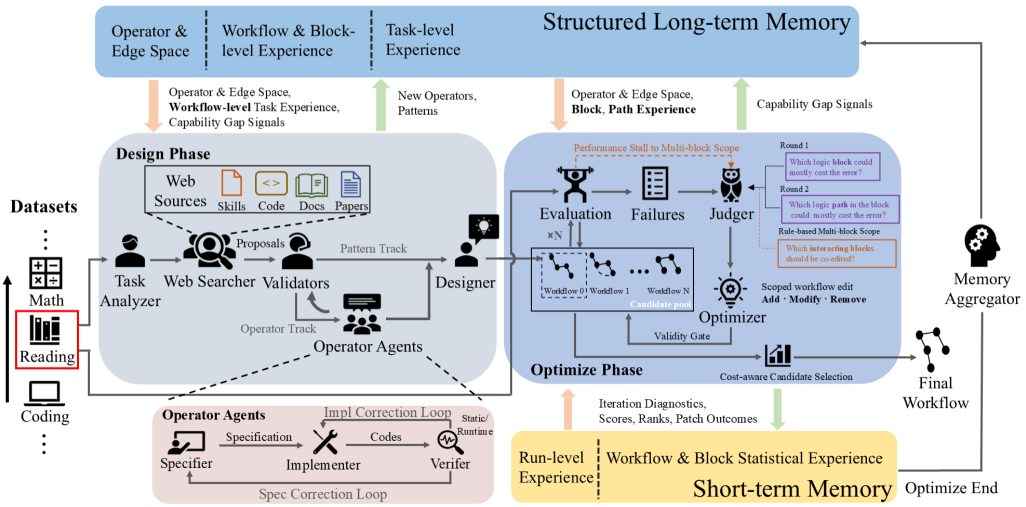


Figure 2: Overview of OpWeave. The Design Phase turns retrieved evidence and recalled priors into verified new Operators and Pattern cards, then emits an initial workflow. The Optimize Phase executes candidates on the development split, attributes failures to a Block, an executed Path, or a bounded Multi-block scope, and proposes focused revisions under a cost-aware priority. Run close promotes eligible evidence through gates and compacts advisory memory for later tasks.

3.2 Workflow Representation

A workflow consists of Blocks, $W = (B_1, \ldots, B_m)$. Each Block is a subworkflow of executable Operators connected by Edges. An Operator is an executable step. We use four Edge types: `seq`, `cond`, `for`, and `parallel`, denoting sequential, conditional, iterative, and parallel execution.

A Block can be represented as

$$B ::= e(g_1, \ldots, g_a), \qquad g ::= o \mid e(g_1, \ldots, g_a), \tag{2}$$

where $e \in \mathcal{E}$ is an Edge type, o is an instance of an Operator in $\mathcal{O}_\tau$, and g is either an Operator or a nested Edge. Since a Block may realize different routes at runtime, we call the route observed in the execution Path z_i. Blocks and executed Paths are the attribution units in our optimization procedure (Section 3.4). The Judger ranks Blocks, and the Optimizer revises them. The workflow language is $\mathcal{L}_\tau = (\mathcal{O}_\tau, \mathcal{E}, \mathcal{P}_\tau)$. It specifies the Operators, Edges, and reusable Patterns available for constructing workflows. The Design Phase expands $\mathcal{O}_\tau$ and $\mathcal{P}_\tau$ for each task.

3.3 Design Phase: Capability Expansion and Workflow Synthesis

The Design Phase grows the workflow language for task τ and synthesizes an initial workflow W_0 (Figure 2). It starts from long-term memory, which in a cold run is empty except for the default Operator library (Appendix Table 7) and the fixed Edge set. The Task Analyzer and Web Searcher collect task requirements and procedural evidence. The Operator Agents use this evidence to synthesize new Operators, while the Validators verify proposed Operators and Pattern cards. The Designer builds W_0 from the resulting language $\mathcal{L}_\tau$.

Task analysis and evidence retrieval. The Task Analyzer reads a small design subset of $\mathcal{D}_{\text{dev}}$ and writes a Task Brief that specifies the task requirements, required capabilities, and expected answer format. Guided by the Brief, the Web Searcher retrieves procedural evidence from skills, documentation, code, and research papers. In parallel, OpWeave recalls design priors from related tasks in long-term memory (Section 3.5). The retrieved evidence may reveal a capability missing from the current workflow language. Validators first check whether existing Operators can be composed to realize this capability. If existing Operators suffice, OpWeave stores the composition as a Pattern card; otherwise, it synthesizes a new Operator (Section 3.3).

Pattern cards. A workflow pattern composes registered Operators into a reusable local structure that provides a capability not available from any single Operator. OpWeave stores each pattern as a Pattern card. A card contains a workflow skeleton and guidance for its use, including choice rules, applicability conditions, failure signals, evidence, and verification (Appendix Table 8). Validators admit only cards whose skeletons use registered Operator types and satisfy the workflow grammar. Accepted cards enter $\mathcal{P}_\tau$ and guide the construction of W_0.

Operator synthesis. For each missing capability, three Operator Agents build an Operator. The Specifier defines its interface and acceptance tests. The Implementer turns the contract into an executable Operator. The Verifier runs static checks and the acceptance tests on the executor model, then reviews the Operator with an LLM that assigns its per-call cost prior. Failures return to the Specifier or the Implementer for revision, and only verified Operators enter $\mathcal{O}_\tau$.

Initial workflow synthesis. The Designer synthesizes the initial workflow W_0 from the Task Brief, retrieved knowledge, recalled design priors, and representative examples. It selects and instantiates appropriate Operators and Patterns, organizes them into a task-specific workflow, and generates the corresponding prompts.

3.4 Optimize Phase: Hierarchical Attribution-Guided Search

The Optimize Phase improves W_0 through local revisions guided by execution evidence (Algorithm 1). Each round samples a parent from $\mathcal{A}$. The Judger attributes the parent's failures to a revision scope, and the Optimizer proposes one revised workflow focused on that scope. The proposal is checked, evaluated on $\mathcal{D}_{\text{dev}}$, and admitted to $\mathcal{A}$ by equation 1. Short-term memory stores each round as a structured, attribution-grounded record to guide later proposals.

Hierarchical attribution. *Block.* The Block level adopts the rank-based Block Judge of JudgeFlow (Ma et al., 2026). For each stable failure i of a parent W, the Judger reads the trace and the reference answer, and ranks all m root Blocks from 1 (most responsible) to m (least responsible). The Block judgment aggregates the valid rankings:

$$B^\star = \arg\min_{B \in W} \sum_{i \in \mathcal{F}(W)} r_i(B), \tag{3}$$

where $\mathcal{F}(W)$ is the set of stable failures with a valid ranking and $r_i(B)$ is the rank of B on example i. Ties are broken in favor of the earlier Block. If W has no stable failure, the scope defaults to the first Block; if it has stable failures but no valid ranking, the round makes no proposal.

Path. When the traces show which route inside $B^\star$ was executed, the scope narrows to that executed Path, so the Optimizer edits only the part of the Block that actually ran.

Multi-block. Some failures come from interacting Blocks and cannot be repaired within a single Block. If several recent rounds evaluate new candidates but none exceeds $q(W)$, OpWeave treats the search as stalled and widens the scope to several related Blocks, which we call a Multi-block scope. This scope contains the most blamed Blocks, their neighbors in root order, and Blocks that are often blamed together on the same example. After each escalation, OpWeave pauses further escalation for a few rounds, so that the new structure produced by a Multi-block revision can be fully explored before the scope widens again.

Proposal and admission. The Optimizer receives the scope, up to five failed examples with their reference answers and grading feedback, the Judger's evidence and short-term memory. It returns one revised workflow W', which may change Operators, parameters, prompts, and Edge structure within the scope. Algorithm 1 gives the admission steps. A proposal is a duplicate if it matches an evaluated workflow, prompts included. The regression guard applies only to Multi-block proposals and rejects W' if it fails more than 10% of the development examples that its parent solved.

Short-term memory. Short-term memory records the attributed scope, edit, and outcome of each round, together with verified regressions and recent validation errors (Appendix Table 12). This memory lets the Optimizer recognize previously attempted ineffective or harmful revisions. At run close, the edit lineage is consolidated into long-term memory (Section 3.5).

Algorithm 1 Attribution-Guided Workflow Search

Require: Initial workflow W_0, development set $\mathcal{D}_{\text{dev}}$, pool size K, budget T, temperature η

Evaluate W_0 with $U(W_0) = 0$; store $q(W_0)$; $\mathcal{A} \leftarrow \{W_0\}$
for $t = 1, \ldots, T$ **do**
  Sample parent $W \in \mathcal{A}$ with $\Pr(W) \propto \exp(q(W)/\eta)$
  Load the stable failures and Block rankings of W
  if W has stable failures but no valid ranking **then**
    Record a no-op in short-term memory; **continue**
  end if
  if search has stalled **then**
    $\sigma_t \leftarrow$ Multi-block scope
  else
    $\sigma_t \leftarrow B^\star$ (Eq. 3), narrowed to its executed Path when available
  end if
  Propose W' within σ_t; canonicalize; validate and check novelty (≤ 2 retries each)
  if W' is invalid or a duplicate **then**
    Record a no-op in short-term memory; **continue**
  end if
  Evaluate W' on $\mathcal{D}_{\text{dev}}$; store $\text{Perf}(W')$ and $q(W')$
  if σ_t is Multi-block and W' fails the regression guard **then**
    Reject W'
  else
    $\mathcal{A} \leftarrow$ top-K of $\mathcal{A} \cup \{W'\}$ by q
  end if
  Update short-term memory
end for
Consolidate short-term memory into long-term memory (Section 3.5)
return The evaluated candidate with the highest Perf

3.5 Dual Memory

OpWeave maintains short-term memory for a single run and long-term memory for reusable cross-task knowledge, including the workflow language, verified Operators, fixed Edges, Pattern cards, and priors. Long-term memory is updated during design and after each run with distilled priors, with each source task stored as one structured record (Appendix Table 13). For new tasks, relevant records are retrieved to warm-start design and guide optimization.

4 Experiments

Experimental Setup. We evaluate OpWeave on six benchmarks from three domains: HotpotQA (Yang et al., 2018) and DROP (Dua et al., 2019) for reading comprehension, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) for code generation, and GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) for mathematical reasoning. We report F1 score for HotpotQA and DROP, pass@1 for HumanEval and MBPP, and solve rate for GSM8K and MATH. We follow the AFlow protocol (Zhang et al., 2025b): a fixed $20\%/80\%$ development and test split, MATH level 5, three trials, and `gpt-4o-mini` as the workflow executor. The full OpWeave setting uses Path attribution, a three-iteration performance-stall window, and a two-turn escalation cooldown. Block-only and escalation-disabled settings are ablations. Task-local optimization runs for 20 rounds, matching the AFlow search budget, where one round is one candidate proposal followed by one full development-set evaluation, so every optimizer we compare consumes the same number of evaluations. Design and optimization use only the development split.

4.1 Main Results

Table 1 reports the comparison. The prompting baselines are taken from Table 1 of A^{2}Flow (Zhao et al., 2026), whose executor and evaluation protocol match ours. AFlow and A^{2}Flow are rerun in our environment, so all entries share the same data split and executor. OpWeave with cold memory

Table 1: Main results on six benchmarks. [a] `gpt-4o-mini` in all roles. [b] GPT-5.5 for design-time capability expansion only; workflow construction, optimization, and execution use `gpt-4o-mini`. [c] No target-task research, admission, or optimization; the target Designer and executor use `gpt-4o-mini`. Memory comes from tasks of other types (other types), from the five other tasks (other tasks), or from all tasks together with their past optimization feedback (all tasks + FB). Bold and underlining mark the best and second-best values in each workflow-design setting.

Method	**Reading**		**Coding**		**Reasoning**		**Avg.**
	HotpotQA	DROP	HumanEval	MBPP	GSM8K	MATH	
Manual Prompting							
IO[a]	68.1	68.3	87.0	71.8	92.7	48.6	72.8
CoT (Wei et al., 2022)	67.9	78.5	88.6	71.8	92.4	48.8	74.7
CoT-SC (Wang et al., 2023)	68.9	78.8	91.6	73.6	92.7	50.4	76.0
MedPrompt (Nori et al., 2023)	68.3	78.0	91.6	73.6	90.0	50.0	75.3
MultiPersona (Wang et al., 2024)	69.2	74.4	89.3	73.6	92.8	50.8	75.1
Self-Refine (Madaan et al., 2023)	60.8	70.2	87.8	69.8	89.6	46.1	70.7
Automated Workflow Design							
ADAS (Hu et al., 2025)	64.5	<u>76.6</u>	82.4	53.4	**90.8**	35.4	67.2
AFlow[a] (Zhang et al., 2025b)	**72.2**	68.4	<u>90.0</u>	<u>71.6</u>	87.5	**53.2**	73.8
A^2Flow[b] (Zhao et al., 2026)	67.7	74.7	**92.1**	70.0	88.2	51.0	<u>74.0</u>
OpWeave (cold memory)[b]	<u>71.5</u>	**79.8**	89.1	**79.8**	<u>89.8</u>	<u>52.8</u>	**77.1**
Search-free Transfer							
OpWeave (other types)[c]	71.5	78.7	87.0	<u>70.1</u>	**89.4**	39.0	72.6
OpWeave (other tasks)[c]	<u>72.0</u>	<u>79.7</u>	**90.1**	69.5	<u>88.5</u>	<u>45.9</u>	<u>74.3</u>
OpWeave (all tasks + FB)[c]	**72.7**	**82.1**	<u>87.8</u>	**77.7**	88.3	**50.2**	**76.5**

reaches the highest average score (77.1), 3.1 points above A^2Flow, the strongest automated baseline, and 1.1 points above CoT-SC (Wang et al., 2023), the strongest prompting baseline. It exceeds A^2Flow on five of six benchmarks, with the largest gains on MBPP (+9.8) and DROP (+5.1); the exception is HumanEval, where A^2Flow leads by 3.0 points. On GSM8K, every automated method falls below IO (92.7). A likely reason is that a single call suffices for this task, and search-based methods rarely recover such a minimal workflow; ADAS (Hu et al., 2025), whose seed archive contains single-call designs, degrades the least. The search-free rows show that accumulated memory can substitute for target-task search. With memory from all other tasks and no optimization, OpWeave matches the searched baselines (74.3 vs. 74.0 for A^2Flow); with memory from all tasks and past optimization feedback, it reaches 76.5, above every baseline and within 0.6 points of its searched workflow. Transfer depends on memory relevance. Memory from other task types preserves reading performance but reduces MATH to 39.0, below IO. This suggests that same-type priors are more important for mathematical reasoning.

4.2 Ablations and Analysis

Table 2 isolates the contribution of each design component and consolidated experience to initial workflow generation. The average rises with every added element, from 72.7 to 76.5, and the full design already exceeds all baselines in Table 1 without search. Most of the gain comes from MATH, which rises from 33.1 to 50.2. The Task Brief adds 10.3 points, new Operators add 4.3, and consolidated experience adds 6.4. The elements are complementary rather than uniformly helpful. The Brief and new Operators lower DROP by 5.6 points; Patterns recover it (+7.1) but cost 3.9 points on MATH; consolidated experience then improves both. Thus no single element class suffices, and priors from earlier tasks select the right combination for a new task. HumanEval and GSM8K vary within 2.7 points, while MBPP gains 3.5 points, mainly from the Brief.

For the optimization phase, we use one benchmark per domain, selecting the task with the largest development headroom over W_0. Table 3 shows three results. First, optimization substantially improves performance, raising the average development score from 62.2 to 74.2, with the largest gain on MATH (23.9 to 53.8). Second, attribution-guided local revision is more effective than global rewriting under the same budget. Base OpWeave slightly exceeds MCTS on average (73.0 vs. 72.5), while Full OpWeave performs better on all three benchmarks and improves the average by 1.7 points

Table 2: Design-phase ablations. Each variant emits one initial workflow per task without optimization. B: Task Brief; O: newly verified Operators; P: verified Pattern cards; M: the consolidated registry and long-term priors from all six benchmarks, recalled at design time. Fixed language uses only the default Operators. Bold indicates the best value in each column.

Variant	Components				Reading		Coding		Reasoning		Avg.
	B	O	P	M	HotpotQA	DROP	HumanEval	MBPP	GSM8K	MATH	
Fixed language					72.5	79.0	89.8	74.2	87.5	33.1	72.7
+B	✓				71.9	74.7	89.6	**77.8**	88.1	43.4	74.2
+B+O	✓	✓			70.5	73.4	**90.3**	75.5	89.7	47.7	74.5
+B+O+P	✓	✓	✓		71.2	80.5	89.1	77.0	**90.2**	43.8	75.3
Full design	✓	✓	✓	✓	**72.7**	**82.1**	87.8	77.7	88.3	**50.2**	**76.5**

Table 3: Optimization-phase ablations. All variants start from the same W_0 for each benchmark. MCTS: AFlow's MCTS optimizer in our setting. Base: Block-level attribution only. Path: refinement restricted to the executed Path. Esc.: scope escalation to Multi-block on stall. STM: structured short-term memory. Bold indicates the best value in each column.

Variant	Components			Reading	Coding	Reasoning	Avg.
	$Path$	$Esc.$	STM	DROP	HumanEval	MATH	
Unoptimized W_0	–	–	–	79.1	83.6	23.9	62.2
MCTS	–	–	–	80.6	85.5	51.4	72.5
Base OpWeave				80.4	85.5	53.1	73.0
Full OpWeave	✓	✓	✓	**81.4**	**87.3**	**53.8**	**74.2**
w/o Path		✓	✓	80.5	87.3	52.1	73.3
w/o Stall Esc.	✓		✓	80.7	86.7	52.9	73.4
w/o STM	✓	✓		80.6	86.7	52.6	73.3

over MCTS. Third, Path attribution, stall escalation, and short-term memory are most effective when combined. Pairwise variants provide only small gains over Base, whereas enabling all three raises the average to 74.2. Their effects also vary by task. Path attribution benefits DROP and MATH but has no measurable effect on HumanEval, while stall escalation and short-term memory improve all three benchmarks. On MATH, the gain appears only when all three components are combined. Full OpWeave consistently outperforms both Base and MCTS across the three benchmarks.

Table 4 shows that OpWeave converges early. At round 5, Base and Full already reach 72.7 and 72.8, above the final score of MCTS (72.5), while MCTS and random global edit reach only 71.5 and 64.9. Local proposals are also more often valid (0.82–0.83 vs. 0.67), so fewer rounds are wasted on candidates that fail validation or execution. Total cost is higher (1.55–1.72× MCTS), because more candidates execute and trace attribution adds Judge calls. However, reaching the MCTS score costs only 0.32 of the MCTS budget for Full and 0.08 for Base.

Table 4: Sample efficiency under the 20-round budget. Random global edit: MCTS with uniform random parent selection. Avg. at round r: best development score within the first r rounds, averaged over the three benchmarks. $R^\star$: rounds needed to reach MCTS's round-20 score. Valid: fraction of proposals that pass validation and execute without runtime failure. Cost: total search tokens relative to MCTS. Cost@$R^\star$: tokens spent up to $R^\star$, relative to the same MCTS total. † Mean over HumanEval and MATH; DROP does not reach this score.

Optimizer	Avg. at round			$R^\star$	Valid	Cost	Cost@$R^\star$
	5	10	20				
MCTS	71.5	71.9	72.5	20	0.67	1.00	1.00
Random global edit	64.9	72.3	73.2	8.0	0.84	1.41	0.56
Base OpWeave	72.7	73.3	73.4	$1.0^\dagger$	0.82	1.55	$0.08^\dagger$
Full OpWeave	72.8	73.0	73.6	3.7	0.83	1.72	0.32

Table 5 shows that OpWeave's cost is dominated by workflow optimization rather than capability expansion. In the cold-memory setting, the Design Phase uses only 5.2M tokens, or 0.81% of the total budget, while optimization accounts for 640.0M tokens. Once long-term memory is available,

Table 5: Token cost of the settings in Table 1. Search-free rows exclude the cost of building the source memory give the marginal cost of a new task. Rel.: sum as a share of the cold-memory sum.

Setting	HotpotQA	DROP	HumanEval	MBPP	GSM8K	MATH	**Sum**	**Rel.**
Cold memory (M tokens)								
Design Phase	1.04	0.86	0.86	0.84	0.76	0.86	5.2	0.81%
Optimize Phase	304.3	83.7	13.9	39.2	79.1	119.8	640.0	99.2%
Total	305.3	84.6	14.7	40.0	79.9	120.7	645.3	100%
Search-free transfer (k tokens)								
Other types	32.3	31.0	30.4	30.8	13.9	32.3	170.7	0.026%
Other tasks	17.6	16.9	34.9	17.0	15.9	36.0	138.3	0.021%
All tasks + FB	19.0	18.2	37.1	56.8	16.9	19.1	167.1	0.026%

search-free transfer reduces the marginal cost to only 14k–57k tokens per task. Using memory from the other tasks requires just 138k tokens in total, 0.021% of the cold-memory cost, while retaining an average score of 74.3, compared with 74.0 for A^2Flow. The reduction is especially large on expensive tasks such as HotpotQA, where 17.6k tokens replace a 305.3M-token cold-memory run.

Table 6: Online continual accumulation with a shared long-term memory. Admitted: Operators and Pattern cards verified at each step, with registry totals in parentheses. Reused: Operators in the final workflow admitted by earlier tasks. W_0 dev: initial-workflow development score. Cost: search tokens relative to the corresponding cold-memory run in Table 1. Step 1 coincides with its cold run.

Step	**Target**	**Admitted (total)**		**Reused**	W_0 **dev**		**Test**			**Cost**
		$\mathcal{O}$	$\mathcal{P}$		Cold	Online	Cold	Online	Δ	
1	DROP	2 (2)	3 (3)	0	79.1	79.1	79.8	79.8	–	–
2	HumanEval	3 (5)	3 (6)	0	83.6	84.8	89.1	91.1	+2.0	0.89
3	MATH	0 (5)	0 (6)	0	23.9	26.2	52.8	50.1	−2.7	0.80
4	HotpotQA	1 (6)	1 (7)	1	73.3	73.2	71.5	72.4	+0.8	0.49
5	MBPP	2 (8)	1 (8)	0	74.7	71.9	79.8	77.1	−2.6	0.82
6	GSM8K	0 (8)	1 (9)	1	90.2	90.1	89.8	90.5	+0.7	1.21
Steps 2–6		+6	+6	2 of 5	69.1	69.2	76.6	76.2	−0.4	0.84

Table 6 shows that continual accumulation primarily improves reuse and search efficiency. The shared memory grows from 2 to 8 Operators and from 3 to 9 Pattern cards, with two of the five later tasks reusing an earlier Operator. Across Steps 2–6, search uses 0.84× the tokens of the corresponding cold-memory runs on average, with the largest saving on HotpotQA (0.49×). Accuracy shows no consistent trend. W_0 increases by only 0.1 points on average, while final test performance decreases by 0.4 points. Because each entry is a single run under one task order, these differences may reflect run-to-run variance or order effects. In this run, accumulation primarily affects reuse and search cost rather than accuracy.

5 Conclusion and Limitations

We present OpWeave, a framework that expands the workflow language from external knowledge, repairs workflows through hierarchical attribution, and retains experience in a dual memory across tasks. On six benchmarks spanning reading, coding, and mathematical reasoning, OpWeave achieves the highest average score among compared methods under a matched executor and budget, reaches the final MCTS score in a fraction of its rounds and cost, and transfers accumulated experience to later tasks with lower search cost and no loss of accuracy.

Several limitations remain. Experimental repetitions are limited, so we interpret differences below about one point as directional rather than conclusive. Per-round search cost is higher than MCTS because more candidates execute and attribution adds Judge calls. The cost to reach a target score is nevertheless lower. Evaluation is limited to one executor, six closed-form benchmarks, and one task order for continual accumulation. Extending OpWeave to open-ended agentic tasks, real deployment settings, and longer task sequences is left to future work.

AI USE STATEMENT

Generative AI tools were used to assist with code editing and debugging, language polishing, and literature search. All AI-assisted outputs were reviewed, verified, and revised by the authors. The research methodology, experimental design, analysis, and scientific conclusions were developed and determined by the authors, who take full responsibility for the final content of the paper.

REPRODUCIBILITY STATEMENT

We provide the workflow representation, algorithmic details, model configurations, search budgets, benchmark splits, and evaluation procedures in the main paper and appendix. The implementation code will be released publicly after the review process.

REFERENCES

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

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

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

Shiyi Du, Jiayuan Liu, Weihua Du, Yue Huang, Jiayi Li, Yingtao Luo, Xiangliang Zhang, Vincent Conitzer, and Carl Kingsford. Why search when you can transfer? amortized agentic workflow design from structural priors. *arXiv preprint arXiv:2604.25012*, 2026.

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

Shengda Fan, Xin Cong, Zhong Zhang, Yuepeng Fu, Yesai Wu, Hao Wang, Xinyu Zhang, Enrui Hu, and Yankai Lin. Generalizing experience for language agents with hierarchical metaflows. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=QsQGMijLhL.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Siddhant Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations (ICLR)*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.

Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, and Fengli Xu. Agentswift: Efficient llm agent design via value-guided hierarchical search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31843–31851, 2026.

Zihan Ma, Zhikai Zhao, Chuanbo Hua, Federico Berto, and Jinkyoo Park. JudgeFlow: Agentic workflow optimization via block judge. *arXiv preprint arXiv:2601.07477*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 46534–46594, 2023.

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

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 3841–3865, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/0ae94013da7cd459402fd77874e09ee3-Paper-Conference.pdf.

Shuaike Shen, Wenduo Cheng, Mingqian Ma, Alistair Turcan, Martin Jinye Zhang, and Jian Ma. SkillFoundry: Building self-evolving agent skill libraries from heterogeneous scientific resources. *arXiv preprint arXiv:2604.03964*, 2026.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

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

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2024.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 13–19 Jul 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

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

Zhiling Yan, Dingjie Song, Hanrong Zhang, Wei Liang, Yuxuan Zhang, Yutong Dai, Lifang He, Philip S. Yu, Ran Xu, Xiang Li, and Lichao Sun. OpenSkill: Open-world self-evolution for llm agents. *arXiv preprint arXiv:2606.06741*, 2026.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2369–2380, 2018.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 75834–75852. PMLR, 13–19 Jul 2025a. URL https://proceedings.mlr.press/v267/zhang25bi.html.

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. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations (ICLR)*, 2025b.

Mingda Zhang, Wenjin Liu, Tiesunlong Shen, Qika Lin, Rui Mao, Erik Cambria, Xiaoying Tang, and Haoran Luo. FlowSteer: Interactive agentic workflow orchestration via end-to-end reinforcement learning. *arXiv preprint arXiv:2602.01664*, 2026.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A^2Flow: Automating agentic workflow generation via self-adaptive abstraction operators. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 40, pp. 29930–29938, 2026. doi: 10.1609/aaai.v40i35.40240.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan O Arik. Multi-agent design: Optimizing agents with better prompts and topologies. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=I05H9RUzHB.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A Appendix

A.1 The default Operator library

Table 7 summarizes the default Operator library, including each Operator's role, function, and cost prior.

A.2 Pattern Cards

Table 8 shows two verified Pattern cards, admitted on DROP and HumanEval. Each card records a reusable usage template over registered Operators as a skeleton in the Edge grammar equation 3.2, together with its choice rules, applicability conditions, failure signals, supporting evidence, and verification reason.

Table 7: The default Operator library. Cost is the per-call prior used by the admission priority equation 1, rated by an LLM with `generate` as the unit (1.0). An admitted `implemented_skill` is rated by an LLM on the same scale after it is generated. † denotes parameterized defaults: `multi_generate_ensemble` costs $n+0.5$ for n solutions, while `programmer` costs $1+0.75r$ for r repairs.

Role	Operator	Function	Cost
Planning	`decompose_plan`	Produce subgoals, constraints, and risk notes for later steps.	0.8
Retrieval	`grounded_context_extractor`	Extract evidence for one query from the visible task context.	1.2
Generation	`generate`	Produce a direct candidate answer.	1.0
Generation	`generate_cot`	Produce a candidate with explicit intermediate reasoning.	1.3
Ensemble	`multi_generate_ensemble`	Generate several candidates, record them as a candidate set, and select one.	$3.5^{\dagger}$
Selection	`compare_candidates`	Select from the nearest visible candidate set; the only legal parallel aggregator.	0.8
Repair	`self_refine`	Revise the candidate under matching negative verification feedback.	2.0
Verification	`test`	Execute public tests or audit visible evidence; authoritative for code only.	0.1
Verification	`critic_review`	Review the candidate against visible evidence; always advisory.	0.8
Tool use	`programmer`	Synthesize, execute, and repair sandboxed Python code.	$2.5^{\dagger}$
Tool use	`react_tool_call`	Call configured tools such as a calculator or restricted Python.	1.2
Verified capability	`implemented_skill`	Run a registry-backed prompt implementation admitted in the Design Phase.	*

Table 8: Two verified Pattern cards, one per task type, from the registries admitted by the cold-memory runs of Table 1. Fields are abridged from the stored cards, and prompt hints and risk notes are omitted. Skeleton notation: `role: type` binds a role to an Operator type; `a | b` lists alternatives, the first being the default; `?` marks an optional slot; `for<=`k bounds iteration; `cond(x.f: v -> g)` runs slot `x` and executes g when its field `f` returns `v`.

(a) Operation calibrated discrete reasoning, admitted on DROP (reading)	
Intent	Choose the discrete operation that the question requires, bind every operand to its entity and event, and return a short answer.
Skeleton	`seq(plan: decompose_plan, gather: grounded_context_extractor, solve: generate_cot, arith_check?: test)`
Choice rule	`arith_check`: enable only when the candidate shows explicit arithmetic over visible operands; its verdict is diagnostic and never routes control flow on a reading task.
Use when	The question asks for a count, sum, difference, comparison, temporal order, or threshold filter, and the passage contains several competing quantities, dates, entities, or events.
Avoid when	The question is a single span lookup with no competing facts, or the operation cannot be inferred from the visible question.
Failure signals	The operation type conflicts with the question wording; arithmetic is applied where direct extraction was required; numbers from different subjects or dates are compared.
Evidence	Three papers on numerical reasoning over text, temporal question answering, and reasoning with generated programs.
Verification	Maps entirely to default Operators, reads only visible passage evidence, and never routes on a reading verdict.
(b) Public-test-gated code repair loop, admitted on HumanEval (coding)	
Intent	Draft code, run the public tests, and repair only when a fresh test verdict fails, for at most two rounds.
Skeleton	`seq(draft: generate | generate_cot, check: test, repair?: for<=2(cond(check.result: fail -> seq(fix: self_refine, recheck: test))))`
Choice rules	`draft`: use `generate` by default and `generate_cot` when edge cases are tricky or the specification is ambiguous. `repair`: enable only on a code task with a fresh failing public test verdict and actionable diagnostics.
Use when	The task has visible public tests, failures carry actionable evidence, and the defect is likely local, such as boundary handling, indexing, comparisons, or output format.
Avoid when	No public tests exist or the verdict is inconclusive; the candidate already passes; repair would rely on hidden tests or on advisory signals such as critic text.
Failure signals	Tests still fail after the repair budget; a repair changes the function signature; the output contains prose instead of complete code.
Evidence	Three papers on self-debugging with execution feedback, the limits of self-repair, and multi-turn code generation with feedback.
Verification	Maps entirely to default Operators and routes repair only on the public test verdict, which is authoritative for code.

A.3 WORKFLOWS

Table 9 shows the workflows returned on DROP, MBPP, and GSM8K. Each is the candidate with the best development score in its run and is the workflow scored in Table 1. Blocks are written as Edge programs over Operator types, and Operators admitted in the Design Phase of the same task are marked with an asterisk.

Table 9: Workflows returned by the cold-memory runs of Table 1, one per task type. Each is the candidate with the best development score in its run. Test denotes its test score reported in Table 1, while dev reports the development score of the initial workflow W_0. Blocks execute in the listed order, and * marks an Operator admitted during the Design Phase of the same task. `cond(test.result: fail -> g)` runs `test` and executes g only after a fresh failing verdict.

Block	Edge program	Function
DROP: test 79.8; dev 79.1 at W_0		
1	`seq(grounded_context_extractor, numeric_evidence_worksheet*, generate_cot, multi_generate_ensemble)`	Gather evidence, classify the operation and bind its operands, reason to a candidate, and sample further candidates.
2	`seq(compare_candidates, concise_answer_contract_normalizer*)`	Select one candidate and reduce it to the requested short answer.
MBPP: test 79.8; dev 74.7 at W_0		
1	`seq(decompose_plan, multi_generate_ensemble, cond(test.result: fail -> seq(self_refine, test)), python_answer_contract_normalizer*)`	Plan, generate and select an implementation, repair it once on a failing public test, and keep exactly one Python solution.
2	`seq(test)`	Retest the normalized candidate.
3	`cond(test.result: fail -> seq(self_refine, test))`	Repair a second time on a fresh failure.
4	`seq (python_answer_contract_normalizer*)`	Normalize the final candidate.
GSM8K: test 89.8; dev 90.2 at W_0		
1	`seq(arithmetic_quantity_extractor*, generate_cot)`	Record quantities, roles, and the unknown, then set up the computation.
2	`seq(programmer, test)`	Execute the computation in the sandbox and audit it without routing.
3	`seq(generate)`	Return the answer as a bare number.

A.4 SELF-GENERATED OPERATORS

Tables 10 and 11 show two Operators synthesized and verified during the Design Phase.

Table 10: The `numeric_evidence_worksheet` Operator synthesized on DROP.

Field	Content
Admitted on	DROP (reading).
Intent	Extract grounded quantities, entity bindings, the required operation, and the expected answer shape from the visible passage. It produces no final answer.
Capability gap	Existing retrieval Operators provide general evidence but do not identify the required arithmetic operation or answer format.
Authority	Read only. It publishes a `numeric_worksheet` record for later Operators.
Output	A structured worksheet containing the operation type, grounded operands, operation plan, answer shape, and caveats.
Acceptance fixture	Given a passage asking how many cities were visited in 2019, the Operator must identify the two relevant cities and classify the operation as `count`, without producing the answer.
Constraints	Uses only visible task information. It cannot access the web, execute code, read a candidate answer, or compute the final result.
Verification	Static validation and smoke execution on `gpt-4o-mini` passed. The artifact was verified and assigned a cost prior of 1.3.
Evidence	The DROP paper and two references on program-aided reasoning.
Used in	DROP workflow, Block 1 (Table 9).

Table 11: The `python_answer_contract_normalizer` Operator synthesized on MBPP.

Field	Content
Admitted on	MBPP (coding).
Intent	Reduce the current candidate to one Python solution by removing prose, markdown fences, duplicate alternatives, and trailing explanations while preserving the requested function.
Capability gap	Existing generation and repair Operators may alter the algorithm, while verification Operators can detect formatting problems but cannot normalize the candidate.
Authority	Write. It replaces the candidate and publishes a `normalization_summary`.
Output	One Python solution together with a summary of the normalization changes.
Acceptance fixture	A candidate containing prose and a fenced Python function must be reduced to the function alone while preserving its interface.
Constraints	No web access, shell, or code execution. The Operator may remove formatting artifacts but must preserve the apparent algorithm and requested interface.
Verification	Static validation and smoke execution on `gpt-4o-mini` passed. The artifact was verified and assigned a cost prior of 2.6.
Evidence	Three references on code evaluation, execution-based selection, and self-refinement.
Used in	MBPP workflow, Blocks 1 and 4 (Table 9).

A.5 SHORT-TERM MEMORY

Table 12 shows that each optimization round appends one attempt record. The proposer sees the most recent records together with the verified regressions and validity failures, and run close reduces the per-round records to edit outcomes for the long-term prior.

Table 12: Categories of the structured short-term memory of one optimization run.

Category	**Content**	**Reader and use**
Attempt records	One record per round: the candidate and its parent, the admission outcome, a diff against the parent, the change in performance, priority, cost, and complexity, and the Block or Path that attribution selected.	Proposer sees the last three rounds and avoids edits that failed or did not help.
Verified regressions	Operator types added by a proposal that scored more than 0.1 below its parent, with the size of the drop.	Proposer is told not to reintroduce them; a single such type also loses the exploration reward.
Validity failures	Up to eight recent validation errors of rejected proposals, with their counts.	Proposer is told not to repeat them.
Edit lineage	Parent link, workflow, and scores of every round that produced a new proposal.	Run close turns each change into an edit outcome labeled positive or negative.

A.6 LONG-TERM MEMORY

Table 13 shows that Long-term memory has two parts. The verified registry $\mathcal{L} = (\mathcal{O}, \mathcal{E}, \mathcal{P})$ holds the executable language and grows only through the typed admission of the Design Phase. The advisory prior is one compacted record per source task, written at run close; recall ranks the priors against the current task signature, keeps the top five, and shows each reader only its own categories.

Table 13: Categories of long-term memory. It contains a verified registry for the workflow language and an advisory prior.

Category	Content	Reader and use
Verified registry for the workflow language $\mathcal{L}_\tau$		
Operator space $\mathcal{O}$	The default Operator types of Table 7 and every Operator admitted in the Design Phase: its specification with intent, applicability, input and output schemas, acceptance fixtures, and sources; the prompt-backed implementation bound to its artifact hash; and the Verifier record. Proposals that failed verification are kept as non-executable evidence.	Designer and Optimizer may reference only registered types, and validation rejects any other reference; the runtime executes an admitted Operator only through its bound artifact.
Edge space $\mathcal{E}$	The four control-flow Edge types `seq`, `cond`, `for`, and `parallel`, their accepted aliases, and the grammar bounds: nesting depth at most five, one to five iterations per `for`, and two to four branches per `parallel`. Fixed in code; no run writes to it.	Designer and Optimizer compose Blocks over it; validation canonicalizes aliases and checks the bounds.
Pattern library $\mathcal{P}$	One card per verified Pattern (Table 8): a skeleton in the Edge grammar over registered Operator types, choice rules, use and avoid conditions, failure signals, prompt hints, risks, research evidence and sources, the task families it applies to, and the verification reason.	Designer receives the cards whose task families match the current task as templates for W_0; Optimizer sees the cards as advisory context; a card never executes on its own.
Advisory prior, one record per source task		
Applicability and provenance	Task family, answer and reasoning traits, failure risks, and tool affordances of the source run; the number of supporting runs and a confidence label. The dataset name is redacted.	Recall ranks priors by overlap with the current task; both Designer and Optimizer see it.
Design priors	The best workflows of the source run as skeletons of Edge and Operator types, without prompts, and per-Operator outcome summaries.	Designer uses them as warm-start material for W_0.
Edit priors	Outcome statistics of past edits by edit kind and target Block role, of Operator kinds by Block role, and of Block-role sequences, with cost guardrails.	Optimizer uses them as advisory context of the lowest priority.
Failure summary	Dominant failure modes and capability gaps of the source run, with suggested remedies.	Design-time research steers retrieval toward missing capabilities.
Path priors	Structural signature of the last attributed Path, and how often each Path signature was executed and blamed.	Optimizer sees the signature; Path refinement uses the counts as a prior.