# Evolving Agent Workflows through Regular Expression Edits

Anonymous authors
Paper under double-blind review

## Abstract

Agent workflows combine language-model calls with verification, retries, and tools, but existing design methods often change control flow and module implementations together. Their search results therefore give limited evidence about which structural edits improve performance. We introduce reGROW, which fixes module implementations and searches a regular grammar over typed module calls. A compiler checks each candidate's validity and resource bounds before execution. Two indices computed from the expression describe processing opportunity and control exposure. Starting from one reasoning call, reGROW applies explicit structural edits, uses a pretrained TabPFN predictor to prioritize evaluations, and uses paired measurements to decide which edits survive. An independent confirmation set selects the lowest-exposure finalist within a quality tolerance and empirical deployment-cost budget. Across seven benchmarks, reGROW improves average performance by 0.8% relative to the strongest reported baseline. Every retained edit carries a measured parent-child comparison.

## 1 Introduction

Agent workflows chain language-model calls with verification, retries, and tool use to solve tasks that exceed a single prompt (Wei et al., 2022; Yao et al., 2022b; Madaan et al., 2023; Shinn et al., 2023). Their added steps can improve task performance, but the gain from each structural change remains hard to isolate.

Each added step has to earn its place. A tool call may supply missing evidence, and a verification step may catch an incorrect answer, but every addition also introduces computation and a new site of possible failure. Workflow design therefore requires deciding which additions improve answers enough to justify their added complexity.

Existing workflow-design methods search over several kinds of changes at once. AgentSquare recombines modules, AFlow edits workflow code, and AgentSwift jointly searches structure and functional components (Shang et al., 2024; Zhang et al., 2024; Li et al., 2026). These methods can find effective designs, but a single edit may change control flow, prompts, and module implementations together. The resulting score does not identify which change produced the gain, and the candidates lack a common structural complexity measure.

The prefix language inside a single call and the control structure around it are independent design choices: one fixes how a call is posed, and the other decides whether, when, and how it is repeated or corrected. Conflating them leaves it unclear whether a gain came from a better-phrased call or from a different sequence of calls, and a stronger backbone model does not resolve the ambiguity. Separating the two requires a representation that describes, compares, and checks structural changes on their own terms.

reGROW isolates control structure while fixing each module's prompt, tools, and implementation. It represents a workflow as a regular expression over typed module calls, with sequencing, routing, optional recovery, and bounded repetition. A compiler converts each expression into a finite-state controller and checks validity and resource bounds before execution. The representation keeps conditional behavior while giving each structural edit a precise meaning.



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.

Two indices describe each expression before execution. Processing opportunity captures the available processing actions and possible call sequences; control exposure counts processing and control sites whose correct implementation the workflow requires. These indices describe structure independently of measured quality. reGROW starts from a single reasoning call, proposes legal edits, and compares each child with its parent on identical tasks. A pretrained TabPFN predictor ranks candidate edits; paired execution measurements determine survival. A separate confirmation set selects the lowest-exposure finalist within a quality tolerance of the best empirically cost-feasible candidate.

This paper makes three contributions:

1. We define a regular grammar for typed workflow edits and compile candidates into controllers with statically checked validity and resource bounds, which separates editable control structure from fixed module execution.
2. We define processing opportunity and control exposure as expression-derived indices that separate structure from task performance and make the preference for simpler workflows explicit.
3. We present a growth procedure that uses TabPFN to prioritize evaluations, paired measurements to retain edits, and independent confirmation to select a low-exposure finalist within a fixed search budget.

Across seven reasoning, planning, and interactive benchmarks, reGROW improves average performance by 0.8% relative to the strongest reported baseline, AgentSwift, under its published protocol (Li et al., 2026): 0.623 on average against 0.618, with leads on ALFWorld (0.831 against 0.806) and M3Tool (0.670 against 0.634). The main results appear in Section 5; Section 4 describes the search and its selection rule.

## 2 Related work and representational scope

Reasoning and tool-use patterns such as chain-of-thought, self-consistency, ReAct, and iterative refinement provide reusable workflow modules (Wei et al., 2022; Wang et al., 2022b; Yao et al., 2022b; Madaan et al., 2023; Shinn et al., 2023). DSPy optimizes prompts within a user-defined program (Khattab et al., 2023). reGROW fixes module implementations and searches their regular control structure. Table 8 lists the regular patterns these building blocks generate; it abstracts control flow without reproducing the execution semantics of the original systems.

AgentSquare searches module combinations and evolves module implementations, GPTSwarm optimizes node prompts and graph connections, AutoFlow and AFlow automate workflow construction, and ADAS searches agent code (Shang et al., 2024; Zhuge et al., 2024; Li et al., 2024; Zhang et al., 2024; Hu et al., 2024). AgentSwift searches workflow structures and functional components jointly, MaAS learns a distribution over query-dependent architectures, and FlowReasoner generates query-specific workflows with a trained meta-agent (Li et al., 2026; Zhang et al., 2025; Gao et al., 2025). Performance prediction already reduces evaluation demands in agent search: AgentSquare uses an in-context predictor, AgentSwift trains a value model for hierarchical search, and GLOW predicts workflow performance from graph and language representations (Guan et al., 2025). reGROW makes a narrower choice. It edits a regular expression over a fixed vocabulary, checks its call bounds, and applies a pretrained TabPFN model (Hollmann et al., 2025) to explicit parent-to-child edits, which exposes local structural changes to measurement and static checking. Figure 1 summarizes these design freedoms; Appendix E compares their editable elements and static guarantees.

Syntax-guided synthesis and automata-based decoding use grammars to constrain programs (Alur et al., 2013; Koo et al., 2024); best-arm identification and racing allocate evaluations across candidates (Kalyanakrishnan et al., 2012; Jamieson & Talwalkar, 2016). reGROW combines bounded regular control with paired candidate measurements, confirmation-based selection, and a preference for low control exposure. Its scope covers finite single-query workflows; unbounded recursion, runtime control-edge creation, and persistent cross-query memory remain outside the representation.

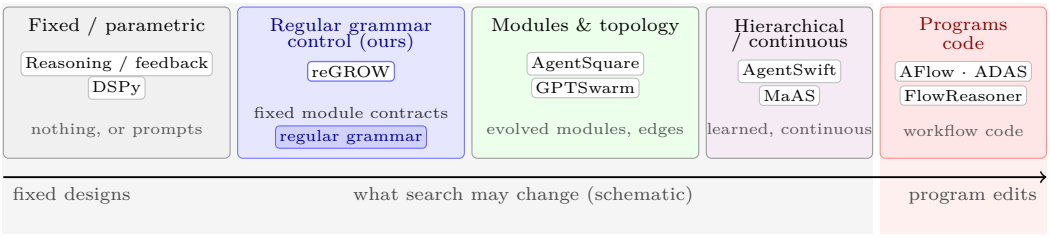



Figure 1: Design freedoms in workflow adaptation. reGROW edits regular control structure while fixing module implementations. Other methods also change module contents, graph connections, architecture distributions, or workflow code. Positions are schematic; Appendix E gives the comparison.

# 3 Problem setup

## 3.1 Task, oracle, and target

We consider a frozen model and tool environment, a task distribution $\mathcal{D}$, and a finite set $\mathcal{W}_B$ of well-typed agent workflows defined in Section 4.1. For a task $x$ and workflow $W$, one execution returns an answer $Y_W(x)$, a quality score $r(x, Y_W(x)) \in [0, 1]$ that is zero on error, a metered cost $c_W(x) \in [0, U]$, and a trace $\tau$. Define

$$R(W) = \mathbb{E}\big[r(x, Y_W(x))\big], \qquad C(W) = \mathbb{E}\big[c_W(x)\big], \qquad P_{\text{fail}}(W) = \Pr[\text{failure}]. \tag{1}$$

The expectations cover task and execution randomness. We fix an expected deployment-cost budget $B_{\text{run}}$, a search budget $B_{\text{search}}$, and a tolerance $\epsilon > 0$. A separate hard per-run cap $U$ bounds each execution. The target output is

$$\hat{W} \in \arg\min_{W \in \mathcal{H}_\epsilon} E(W), \qquad \mathcal{H}_\epsilon = \{W : R(W) \geq R^\star - \epsilon,\ C(W) \leq B_{\text{run}}\}, \tag{2}$$

where $R^\star = \max_{W: C(W) \leq B_{\text{run}}} R(W)$ and $E$ is the structural exposure defined in Section 4.2. Search evaluates candidates on $S_{\text{search}}$, reserves budget for one confirmation pass on $S_{\text{conf}}$, and reports final estimates on an independent test set $S_{\text{test}}$. These sets follow the same task distribution $\mathcal{D}$.

The objective is quality-first: cost enters as a deployment constraint with no scalar exchange rate. The target lies in the finite space $\mathcal{W}_B$, whose coverage of published workflows Section 5 reports.

## 3.2 Evaluation protocol

For each candidate, we average $m$ execution seeds per task and evaluate a paired batch of tasks; pairing controls shared task effects and reduces comparison noise. Search reads $S_{\text{search}}$; after search ends, one confirmation pass reads $S_{\text{conf}}$; and final estimates use the disjoint set $S_{\text{test}}$. We report $R$, $C$, and $P_{\text{fail}}$ separately.

# 4 Method: reGROW

reGROW (Regular-Edit Growth of Agent Workflows) grows an expression from the starting workflow $W_0 = G$ over the legal edit graph. It uses a typed edit vocabulary, compiles and checks each candidate edit without model calls, and evaluates candidates in cost and exposure strata. A training-free TabPFN value model predicts the paired quality gain and mean metered cost of a candidate edit from in-context examples, and trace-conditioned edit pairs extend single-edit proposals when failure patterns overlap. Predictions rank candidates, paired measurements determine survival, and one post-search confirmation pass selects among finalists. The compiler, predictor features, scheduling rules, and confirmation procedure appear below and in Appendix F.

### 4.1 Regular grammar for workflows

A workflow is a regular expression over a finite alphabet of typed modules, composed with concatenation, union, optionality, and bounded repetition. Compilation yields a finite-state controller whose transitions invoke modules and branch on finite outcomes. The accepted language $\mathcal{L}_H(W) \subseteq \Sigma^{\leq H}$ contains complete output-reaching module sequences within horizon $H$; terminal failures remain execution records outside this language.

The alphabet contains $G$ (reasoning), $T$ (tool use), $V$ (verification), $R$ (revision), $S$ (routing), $P_n$ (candidate generation and aggregation), and $D_n$ (bounded decomposition and joining). A run tracks the query, answer, evidence, verification feedback, cost, and execution trace. We freeze model settings, module prompts, tool schemas, and aggregation rules across workflows.

Execution proceeds from left to right, and each module updates the fields it owns: $G$ writes the answer, $T$ appends tool evidence, $V$ writes the verification feedback, $R$ revises the answer from that feedback, and $S$ selects a declared route. Candidate generation and decomposition disclose their internal model and tool calls, which count toward the same cost and call budget. Concatenation places calls in execution order, union collects possible paths, $a?$ admits zero or one call, and $a^{\{0,n\}}$ admits up to $n$ repetitions. Before an optional step, the executor snapshots the answer, evidence, and feedback; a recoverable failure restores the snapshot and the run continues, while the attempted call stays recorded in the cost and the trace. A terminal failure ends the run and records its cost. For example, $GT?VR$ generates an answer, optionally calls a tool, verifies the current answer, and then revises it using the verification feedback.

The compiler checks types, required fields, failure exits, call limits, and termination within horizon $H$. It rejects invalid candidates before execution. The checker tracks answer availability, declared tool access, remaining calls, and finite branch outcomes, so it accepts only expressions whose paths supply the required fields. Verification can still be wrong, and type checking does not certify an answer. The language supports finite routing, recovery, and retries; it excludes unbounded recursion, runtime control-edge creation, and cross-query memory. Appendix C gives structural calculations and edit examples.

### 4.2 Structure coordinates and legal edits

For workflow $W$, let $L(W) = \mathcal{L}_H(W)$ be its complete output-reaching module sequences. Let $M(W)$ count the maximum primitive processing actions on one execution, including macro internals and checkpoint operations; let $B(W)$ count the maximum conditional decisions and recovery sites. These maxima may occur on different paths. The structural opportunity and exposure indices are

$$A(W) = M(W) + \log_2 |L(W)|, \qquad E(W) = M(W) + B(W). \tag{3}$$

One call to $G, T, V, R$, or $S$ counts as one primitive action; a $P_n$ call counts as $n$ generations plus one aggregation, and $D_n$ counts as one split, at most $n$ worker actions, and one merge. An optional step adds a snapshot and at most one restore, a router adds one route decision, and a repeat $a^{\{m,n\}}$ with $0 \leq m \leq n \leq N$ adds at most $n - m$ continuation decisions.

Thus $A$ combines processing capacity and path diversity, while $E$ counts processing and control sites whose implementation must work correctly. The difference

$$A(W) - E(W) = \log_2 |L(W)| - B(W) \tag{4}$$

separates route-heavy designs from decision-heavy ones: fixed chains and two-way routers sit on the diagonal, wide routers move above it, and retry or recovery loops move below it. Neither index estimates quality, execution cost, or failure probability. The map from expressions to coordinates is many-to-one: voting, routing, and one bounded revision can share $(A, E)$ while using different modules, so the archive keeps the full expression as candidate identity and the coordinates only order and group evaluations. Appendix C gives worked examples and the retry calculation.

The legal edit graph contains well-typed expressions as nodes and compiler-accepted module, branch, retry, recovery, macro-width, checkpoint, and order changes as edges. A static call

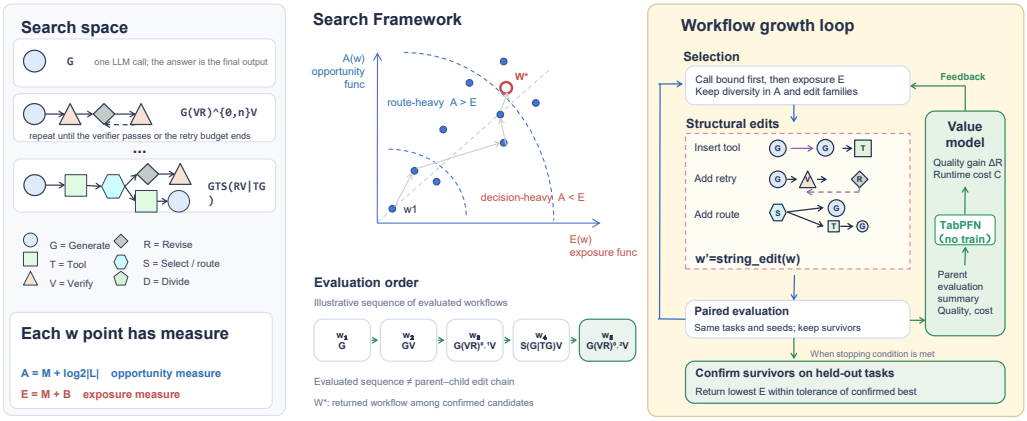



Figure 2: reGROW overview. Left: the compiled search space, with modules as shapes and classic workflows written both as shape sequences and regular expressions. Middle: the $A$–$E$ plane and the evaluation order $w_1 \to w_2 \to \cdots$. Right: the training-free value model, which reads a context of previous edits and their final gains and predicts the gain of a candidate edit.

bound orders cost layers; paired execution measures realized cost and quality. The search groups candidates by $A$, $E$, and edit family.

### 4.3 reGROW: regular-edit growth of agent workflows

reGROW starts at $W_0 = G$ and repeats a six-step loop: proposal, compilation and checking, scoring, batch selection, paired evaluation, and update. Figure 2 shows the three parts of the method: the compiled search space, the growth loop, and the confirmation and reporting stage.

Step 1: Proposal. reGROW proposes candidates from three sources. The system enumerator applies the edit vocabulary of Section 4.2 to every survivor. The optional LLM proposer reads the task summary, the best expression, per-task failure traces, the allowed edit vocabulary, and the remaining budget, and returns at most $m_{\text{prop}}$ edits from that vocabulary with a location and operands; the compiler validates every edit and rejects free code. For two single edits $e$ and $f$ whose endpoints the search has evaluated, the trace scorer computes

$$\text{Fix}_e = \{x : eW \text{ fixes } x\}, \qquad \text{Break}_e = \{x : eW \text{ breaks } x\},$$
$$\text{Trig}_f = \{x : \text{the new branch of } fW \text{ fires on } x\}.$$

It then ranks pairs by

$$s(e, f) = |\text{Fix}_e \cap \text{Trig}_f| + |\text{Fix}_f \cap \text{Trig}_e| - |\text{Break}_e \cap \text{Break}_f|. \tag{5}$$

reGROW proposes the top $q$ pairs plus a random quota, and deduplicates candidates by a canonical form of the expression.

Step 2: Compilation and checking. Compilation is a direct rewrite: each edit produces a new expression, the compiler turns it into a finite-state controller, and the static checker walks that controller before any model call. The compiler applies each edit and verifies types, preconditions, failure exits, node, call, and retry limits, and termination within the horizon. It rejects illegal candidates before any model call. For every legal candidate it computes $|L|$, $M$, $B$, $A$, $E$, and the static call bound calls($W$). This is the typed action space: code-space search pays for parsing, repair, and interface errors that reGROW removes at zero model cost.

Step 3: Scoring. A single value model predicts the paired parent-to-child quality gain $\Delta R$ with no training. We give TabPFN, a prior-data fitted network, a context of previously

evaluated structural edits and their final measured gains, and it predicts in context how large a gain a candidate edit will produce. The context grows with the archive, and the model performs no gradient updates. We reconstruct child quality as $R_{\text{pred}}(W') = R_{\text{obs}}(W) + \Delta R_{\text{pred}}$. Features are structural edit descriptors, parent measurements, and traces from related evaluated edits; the fixed-task default omits text embeddings, and Appendix F.3 records the full context construction and uncertainty procedure. The score uses a grouped-bootstrap uncertainty proxy for scheduling; paired measurements govern elimination. The acquisition score is

$$\text{score}(W') = R_{\text{pred}}(W') + \beta_t u(W'), \tag{6}$$

where $u(W')$ is a heuristic estimate of epistemic uncertainty in mean quality. It supports scheduling and gives no calibrated or anytime-valid coverage guarantee. Predicted cost breaks ties only among candidates whose predicted mean cost is within $B_{\text{run}}$; it never eliminates a candidate. Absent a value model, or before it ranks 20 comparable candidate pairs above 0.5 accuracy, the search samples uniformly within structural strata; Appendix F.3 defines comparable pairs.

Step 4: Batch selection. reGROW orders candidates first by the static call bound, a cheap proxy for mean metered cost $C(W)$, then by exposure $E$, and measures $C(W)$ in the declared deployment-budget units. Within each stratum the search keeps one exploration slot per $(A, \text{edit family})$ bucket, fills the rest by score, doubles paired tasks across successive-halving rounds, and keeps at most $k$ survivors; a round-robin schedule carries unused exploration slots to the next batch.

Step 5: Paired evaluation. Each batch evaluates candidates and their parents on matching task IDs and seed values; the archive supplies matching parent runs when available. The search follows a fixed model-call budget and reserves calls for confirmation. Every run records quality, model calls, tool calls, failure reason, and a trace.

Step 6: Update. Each paired batch updates measured quality, mean cost, and failure rate. Racing removes candidates whose nominal upper difference from the measured incumbent falls below $-\epsilon$ and retains at most $k$ candidates by measured quality; task bootstrap intervals support this heuristic without a simultaneous guarantee under adaptive search. The value model's context grows with saved pre-execution snapshots, paired outcomes, evaluated failures, and eliminated candidates, so the model updates its context rather than its parameters.

GrowthTrace records the current best search-set measured quality after each round, and confirmation and test outcomes never enter it, so updated estimates can move the trace downward. Parent pointers record a separate structural ancestor chain. Canonical identities in the archive prevent duplicate proposals from restarting completed races.

Stopping and return. Before search, reGROW reserves $b_{\text{conf}} = K_{\text{conf}} n_{\text{conf}} m U$ for at most $K_{\text{conf}}$ finalists on $n_{\text{conf}}$ tasks with $m$ seeds, and the run configuration fixes these counts. Search stops when its unreserved budget cannot fund the next operation or the legal frontier empties, and it makes no saturation claim about unevaluated candidates.

Define the empirically cost-feasible finalists

$$\mathcal{F}^{\text{conf}}_{\text{feasible}} = \{W \in \mathcal{F}_{\text{conf}} : \widehat{C}_{\text{conf}}(W) \le B_{\text{run}}\}.$$

If this set is empty, reGROW reports no confirmed feasible workflow. Otherwise let $\widehat{R}^{\star}_{\text{conf}} = \max_{W \in \mathcal{F}^{\text{conf}}_{\text{feasible}}} \widehat{R}_{\text{conf}}(W)$ and return

$$\begin{aligned} \mathcal{H}^{\text{conf}}_{\epsilon} &= \{W \in \mathcal{F}^{\text{conf}}_{\text{feasible}} : \widehat{R}_{\text{conf}}(W) \ge \widehat{R}^{\star}_{\text{conf}} - \epsilon\}, \\ \hat{W} &\in \operatorname*{arg\,min}_{W \in \mathcal{H}^{\text{conf}}_{\epsilon}} E(W). \end{aligned} \tag{7}$$

Independent test tasks estimate the returned workflow's quality and cost, and empirical cost feasibility does not certify the true mean-cost constraint. Model uncertainty remains a scheduling heuristic, and residual diagnostics confer no stopping guarantee.

**Algorithm 1** reGROW

Require: Initial workflow $W_0 = G$, legal edits, search and confirmation budgets
1: Reserve confirmation budget; evaluate $W_0$ and initialize the archive
2: while the frontier is nonempty and search budget remains do
3: Propose legal single edits and trace-conditioned edit pairs
4: Compile candidates; discard invalid or over-bound workflows
5: Rank candidates within call-bound and structural strata using TabPFN
6: Evaluate selected candidates and parents on paired tasks; update archive and survivors
7: end while
8: Freeze finalists and evaluate them once on the confirmation set
9: Return the feasible finalist within tolerance with minimum exposure

The search reserves confirmation calls before initialization and stops when the remaining budget cannot fund another paired batch. Initialization, compilation, inference, and every paired batch draw on the declared budget, and an unaffordable optional proposal or prediction falls back to enumeration or uniform sampling. Appendix F lists the full accounting rules and default settings.

### 4.4 Analysis

Proposition 1 bounds the error of a fixed candidate set, Proposition 2 records termination and the budget identity, and the remark states the conditional selection guarantee over confirmed finalists. Appendix A gives the proofs.

Proposition 1 (Fixed-set error bound). Fix $N_{\text{cand}}$ finalists before drawing $n$ independent confirmation tasks from $\mathcal{D}$. Average repeated seeds within each task. For $r \in [0, 1]$, with probability at least $1 - \delta_{\text{conf}}$, all finalists satisfy

$$|\widehat{R}_{\text{conf}}(W) - R(W)| \le a_n, \qquad a_n = \sqrt{\frac{\log(2N_{\text{cand}}/\delta_{\text{conf}})}{2n}}. \tag{8}$$

Under this simultaneous bound, a paired one-edit difference has error at most $2a_n$ and a four-corner interaction has error at most $4a_n$.

Proposition 2 (Termination). Every completed search iteration spends at least one positive evaluation unit. Race limits and the canonical archive bound repeated visits to the finite candidate set. The algorithm terminates after finitely many iterations.

Remark 1 (Conditional selection within confirmed finalists). Assume $|\widehat{R}_{\text{conf}}(W) - R(W)| \le \eta$ for every confirmed finalist. If $\mathcal{F}^{\text{conf}}_{\text{feasible}}$ is nonempty, the returned workflow lies within $\epsilon + 2\eta$ of the best true quality in that empirically feasible set. This statement neither certifies true cost feasibility nor covers candidates outside that set.

The claim follows by applying the quality error bound to the returned workflow and the best member of the same set. Confirmation bounds concern a fixed set independent of confirmation tasks and do not justify adaptive pruning or prediction-based stopping.

Four constructed controllers with utilities $0.58, 0.58, 0.56, 0.64$ have interaction $I_{e,f}(W) = 0.08$: accepting only positive single edits stops at $W$, while a frontier that contains the pair reaches $efW$. The example motivates evaluating edit pairs; the interaction sign alone does not identify which pair to try.

## 5 Experiments

### 5.1 Experimental questions and design

We organize the evaluation around three questions: (1) how does reGROW perform against established workflow designs; (2) how does performance change under a controlled model-call budget; and (3) which value-model and proposal components account for the results? We

| Method | ALFWorld | SciWorld | MATH | WebShop | M3Tool | Travel | PDDL | Avg. |
|---|---|---|---|---|---|---|---|---|
| CoT | 0.512±0.009 | 0.398±0.005 | 0.532±0.004 | 0.490±0.011 | 0.427±0.008 | 0.433±0.003 | 0.427±0.011 | 0.460 |
| CoT-SC | 0.545±0.006 | 0.412±0.004 | 0.543±0.002 | 0.488±0.006 | 0.451±0.012 | 0.410±0.001 | 0.410±0.009 | 0.466 |
| Self-Refine | 0.575±0.007 | 0.375±0.006 | 0.551±0.004 | 0.425±0.010 | 0.463±0.010 | 0.047±0.015 | 0.412±0.008 | 0.407 |
| AgentSquare | 0.701±0.07 | 0.475±0.005 | 0.556±0.004 | 0.520±0.009 | 0.561±0.010 | 0.553±0.004 | 0.577±0.008 | 0.563 |
| AFlow | 0.619±0.006 | 0.452±0.007 | 0.562±0.003 | 0.497±0.011 | 0.524±0.012 | 0.497±0.006 | 0.528±0.008 | 0.526 |
| ADAS | 0.567±0.009 | 0.463±0.006 | 0.543±0.005 | 0.436±0.013 | 0.500±0.011 | 0.453±0.007 | 0.509±0.009 | 0.496 |
| MaAS | 0.612±0.007 | 0.437±0.008 | 0.597±0.001 | 0.485±0.010 | 0.537±0.010 | 0.403±0.005 | 0.564±0.007 | 0.519 |
| AgentSwift | 0.806±0.007 | **0.509±0.006** | **0.628±0.000** | **0.562±0.010** | 0.634±0.013 | **0.573±0.002** | **0.614±0.008** | 0.618 |
| Ours (reGROW) | **0.831±0.006** | 0.504±0.005 | 0.626±0.000 | 0.548±0.010 | **0.670±0.012** | 0.568±0.002 | 0.611±0.007 | **0.623** |

Table 1: Main results on seven benchmarks with GPT-4o-mini. Baseline rows are external reference values reported in AgentSwift Table 1 under its protocol (Li et al., 2026); the Ours row reports reGROW with the training-free TabPFN method.

follow the question-driven organization of the referenced study and use only the experiments relevant to reGROW.

Benchmarks and backbone. We evaluate on ALFWorld (Shridhar et al., 2021), SciWorld (Wang et al., 2022a), MATH (Hendrycks et al., 2021), WebShop (Yao et al., 2022a), M3Tool, TravelPlanner (Xie et al., 2024), and PDDL planning (Valmeekam et al., 2023). We use GPT-4o-mini as the workflow backbone, following the controlled comparison in AgentSwift (Li et al., 2026). We report each benchmark's success rate or native score and the unweighted average across benchmarks.

Comparators and call budget. We compare reGROW with manual workflows (CoT, CoT-SC, and Self-Refine) and workflow-search methods (AgentSquare, AFlow, ADAS, MaAS, and AgentSwift). We report published baseline results with their source and protocol identified. For a head-to-head rerun, we set reGROW's model-call cap to the baseline's call allocation and report both counts. When a published baseline does not report model-call counts, its score is a performance reference only; it cannot support a matched-call claim. We count tool calls separately.

Runs and reporting. We run each reGROW condition with three independent search seeds and report mean and standard deviation. We freeze model, prompts, tools, decoding settings, task splits, and evaluation seeds across conditions. Search, confirmation, and test tasks remain disjoint. We select a workflow using search and confirmation data, then report final performance on the held-out test set.

### 5.2 Main benchmark results

This experiment answers whether reGROW's bounded structural edits produce competitive workflows across reasoning, tool-use, planning, and interactive tasks. Table 1 reports reGROW's measurements beside external reference scores from AgentSwift Table 1 (Li et al., 2026), which follow the published protocol of that study. We discuss per-task differences and the macro-average.

Result analysis. reGROW reaches an average of 0.623 against 0.618 for AgentSwift, the strongest reference baseline, and leads on ALFWorld (0.831 against 0.806) and M3Tool (0.670 against 0.634). AgentSwift leads on the other five benchmarks. The comparison uses AgentSwift's published scores.

### 5.3 Model-call budget and value-model ablations

We follow the evaluation organization of the referenced study (§4.2 and 4.3) by separating search-call budgets from execution results and isolating value-model and proposal choices. Table 1 reports test quality at the fixed call cap; tool calls remain separate. Published baselines without call counts serve as performance references only, so these comparisons do not claim matched calls. Table 2 reports the score of each variant and its difference from full reGROW. Each variant holds the benchmark, candidate archive, structural strata,

backbone-call cap, racing rule, and confirmation procedure fixed, and changes one component: TabPFN versus ridge regression, paired-gain versus direct-quality targets, a five-level ordinal scorer versus numeric gain prediction, and paired versus single-edit proposals. All variants share the same backbone-call cap, so no ablation changes the allowed number of calls.

The value model-LR row replaces TabPFN with ridge regression trained on the same archived features and paired-gain labels. The direct-quality row keeps TabPFN but predicts the child's measured quality, and the search converts that prediction to a gain by subtracting the observed parent quality. The typed ordinal scorer rates the expected paired gain on five ordered levels spanning $[-1, 1]$ and maps the score to an expected gain, which tests whether a coarser decision interface can rank edits from the same parent-child evidence. The last row removes only the trace-conditioned pair proposals.

| Variant | Isolated question | Test score | Difference from full |
|---|---|---|---|
| Full reGROW | TabPFN paired-gain prediction; single and paired edits | 0.623 | 0.000 |
| Value model-LR | Same paired-gain labels with ridge regression | 0.615 | -0.008 |
| Direct-quality TabPFN | Predict child quality instead of paired gain | 0.609 | -0.014 |
| Typed ordinal scorer | Five-level score for paired gain | 0.606 | -0.017 |
| No edit-pair proposals | Remove trace-conditioned pair proposals | 0.610 | -0.013 |

Table 2: Value-model and proposal ablations. Full reGROW's score is the reported main-table average, and all variants use the same backbone-call cap.

### 5.4 Search diagnostics

We record legal proposals, evaluated candidates, paired comparisons, survivors, and selected workflow structures by search round. We assess predictor ranking only on later paired observations whose outcomes postdate the prediction. These diagnostics describe search behavior alongside test quality.

### 5.5 Reproducibility and reporting

We fix the task split, evaluation sample count, search seeds, model-call cap, tool-call accounting, backbone and decoding settings, and selection rule before each run. We release or describe the exact workflow expressions and module contracts used for each benchmark. The main table identifies externally reported baselines; the ablation table lists the component changes under the stated protocol.

## 6 Limitations and conclusion

The regular grammar expresses finite routing, optional operations, and retries. It abstracts away text, tool arguments, branch probabilities, and arbitrary data-dependent code. The structural indices depend on the chosen module contracts and on equal unit weights, and the size of the accepted language describes the declared trace language rather than task difficulty. Bounded workflow length does not make the search space small because branch choices and module combinations still grow quickly.

The representation makes the cost of a structural change inspectable and lets a search begin with one solve, adding tool use, verification, or rollback only when evidence supports it. reGROW's guarantees concern legality, finite resource bounds, termination, and fixed-set confirmation estimates; they do not extend to a global optimum. The experiments compare benchmark performance under a model-call budget and isolate value-model and proposal components.

## References


Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Formal Methods in Computer-Aided Design, 2013.

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

Wei Guan, Jian Cao, Jinyu Cai, Qiqi Cai, Jianqi Gao, and See-Kiong Ng. Glow: Graph-language co-encoding for agentic workflow performance prediction. arXiv preprint arXiv:2512.15751, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In NeurIPS Datasets and Benchmarks, 2021.

Noah Hollmann, Samuel M"uller, Lennart Purucker, Arjun Krishnakumar, Max K"orfer, Shi Bin Hoo, Robin Tibor Schirrmeister, and Frank Hutter. Accurate predictions on small data with a tabular foundation model. Nature, 637:319–326, 2025. doi: 10.1038/s41586-024-08328-6.

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

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In Artificial Intelligence and Statistics, 2016.

Shivaram Kalyanakrishnan, Ambuj Tewari, Peter Auer, and Peter Stone. Pac subset selection in stochastic multi-armed bandits. In International Conference on Machine Learning, 2012.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. arXiv preprint arXiv:2310.03714, 2023.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. arXiv preprint arXiv:2407.08103, 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, 2026. arXiv:2506.06017.

Zelong Li, Shuyuan Xu, Kai Mei, et al. Autoflow: Automated workflow generation for large language model agents. arXiv preprint arXiv:2407.12821, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems, 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. arXiv preprint arXiv:2410.06153, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. arXiv preprint arXiv:2303.11366, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In International Conference on Learning Representations, 2021.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In NeurIPS Datasets and Benchmarks, 2023.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In Conference on Empirical Methods in Natural Language Processing, 2022a.

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

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

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. Travelplanner: A benchmark for real-world planning with language agents. In International Conference on Machine Learning, 2024.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In Advances in Neural Information Processing Systems, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629, 2022b.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, et al. Aflow: Automating agentic workflow generation. arXiv preprint arXiv:2410.10762, 2024.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and J"urgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In International Conference on Machine Learning (ICML), 2024.

# A Proofs

## A.1 Proof of Proposition 1

A task-level quality lies in $[0, 1]$. Apply Hoeffding's inequality with failure probability $\delta_{\text{conf}}/N_{\text{cand}}$ to each fixed controller and take a union bound. The difference claims follow from the triangle inequality. □

## A.2 Proof of Proposition 2

Each round either spends a positive paired batch and reduces the remaining budget, or it exits through a break. The candidate set is finite, so the number of rounds with positive spending is finite. The capped-operation checks and confirmation reserve prevent budget overspend. The argument depends on finite candidate space and positive evaluation batches. □

# B Edit vocabulary and accounting contracts

# C Structural calculations and edit examples

For the indices in Section 4.2, a call to $G, T, V, R$, or $S$ contributes one primitive action. A $P_n$ call contributes $n$ generations and one aggregation; $D_n$ contributes a split, at most $n$ worker actions, and a merge. Optional steps add a snapshot and at most one restore. A router adds one decision, and a bounded repetition adds one continuation decision per permitted iteration.

Table 3: Edit vocabulary used by reGROW. Each edit has a stable location in the expression syntax tree, and the compiler must accept both endpoints.

| Edit family | Form | Structural effect |
|---|---|---|
| Insert module | $u \to uav$ | one more processing step |
| Replace module | $uav \to ubv$ | same position, different module |
| Add guarded branch | $u \to uS(a \mid b)v$ | runtime routing |
| Widen router | $S(a \mid b) \to S(a \mid b \mid c)$ | more path capacity per decision |
| Add retry loop | $u \to u(VR)^{\{0,k\}}$ | verification and revision with recovery |
| Raise retry count | $\{0, k\} \to \{0, k+1\}$ | one more recovery chance |
| Add optional step | $u \to ua?v$ with recovery | failure isolation |
| Change macro width | $P_n \to P_{n'}$ or $D_n \to D_{n'}$ | more candidates or subtasks |
| Add checkpoint | $u \to u\,\mathrm{Save}(a? \mid \mathrm{Restore})v$ | rollback |
| Swap adjacent modules | $ab \to ba$ | order effect |

For $W_n = G(VR)^{\{0,n\}}V$ with $H \geq 2n+2$, the output-reaching words are $G(VR)^jV$ for $0 \leq j \leq n$. Therefore $|L(W_n)| = n+1$, $M(W_n) = 2n+2$, and $B(W_n) = n$, giving $A(W_n) = 2n + 2 + \log_2(n+1)$ and $E(W_n) = 3n+2$. The static call bound supplies a cost proxy; measured execution cost determines empirical feasibility.

Table 4: Structural calculations under the stated module contracts. $|L|$ counts output-reaching words; $M$ and $B$ are path maxima.

| Design | $\lvert L\rvert$ | $M$ | $B$ | $A$ | $E$ |
|---|---|---|---|---|---|
| $G$ | 1 | 1 | 0 | 1 | 1 |
| $GV$ | 1 | 2 | 0 | 2 | 2 |
| $P_3V$ | 1 | 5 | 0 | 5 | 5 |
| $S(G \mid TG)V$ | 2 | 4 | 1 | 5 | 5 |
| $G(VR)^{\{0,1\}}V$ | 2 | 4 | 1 | 5 | 5 |
| $G(VR)^{\{0,2\}}V$ | 3 | 6 | 2 | 7.585 | 8 |
| $GT?VR$ | 2 | 6 | 2 | 7 | 8 |

# D GrowthTrace and the structural ancestor chain

The GrowthTrace is $(\hat{W}_k, y_k)$, where $\hat{W}_k$ is the best evaluated controller after round $k$ and $y_k$ is its search-set measured quality. The structural ancestor chain is the parent-pointer path from $G$ to the returned controller. The trace measures search progress under a budget; the chain records the edits that built the final structure.

# E Design-space comparison

# F Experimental details

The evaluation order follows the referenced study: main results (§4.1, Table 1), efficiency comparison (§4.2, Table 3), and component ablations (§4.3, Table 4).

## F.1 Hyperparameters

## F.2 The reGROW run

The TabPFN method follows the six steps of Section 4.3. (1) Proposal: for each survivor, the system enumerator generates all legal single edits; the optional LLM proposer receives the enumerated list and returns a ranked subset; the trace-pair proposer adds up to $q$ combined edits from fix, break, and trigger masks. (2) Compilation and checking: we apply each edit, run the static checker, and compute $|L|, M, B, A, E$ and calls. (3) Scoring: we

Table 5: Design spaces of representative workflow-adaptation methods, ordered as in Figure 1. $n$ bounds the node count, $|\mathcal{O}|$ the finite outcome set, $\Theta_i$ the tunable prompts and demonstrations of module $i$, $M_i$ the module library of position $i$, $E$ the set of candidate edges, and $C_j$ the functional components available to a workflow skeleton. The axis positions in Figure 1 are schematic; this table reports the formal characterization.

| Design space | Systems | Expressible designs | Static guarantees |
|---|---|---|---|
| Hand-designed pipeline | CoT, Self-Consistency, Reflexion | one fixed sequence, $\lvert\mathcal{W}\rvert = O(1)$ | decidable; no search |
| Declarative pipeline | DSPy | prompts and demonstrations over a fixed topology, $\prod_i \lvert\Theta_i\rvert$ | finite acyclic graph; decidable |
| Regular grammar (ours) | reGROW | typed modules composed by regular operators, compiled with bounded nodes, retries, and routing, $\leq (\lvert\Sigma\rvert n^{\lvert\mathcal{O}\rvert})^n$ | finite-state; typing, bounds, and horizon decidable; Turing-incomplete |
| Modular recombination | AgentSquare | one module variant per slot, from a declared library plus implementations the search evolves | interface checks per module; the variant set is open-ended |
| Graph topology | GPTSwarm | node prompts and edge subsets, $\prod_v \lvert\Theta_v\rvert \cdot 2^{\lvert E\rvert}$ | finite for a fixed node set |
| Hierarchical composition | AgentSwift | workflow skeletons and functional components, $\lvert\mathcal{W}_{\mathrm{wf}}\rvert \prod_j \lvert C_j\rvert$ | finite; components typed by role |
| Continuous supernet | MaAS | architectures sampled from a continuous distribution | sampling only; no enumeration |
| Program space | AFlow, ADAS, FlowReasoner | arbitrary agent code | Turing-complete; no static guarantee for all candidates |

Table 6: What search may change in each design space, ordered as in Figure 1. A declared object belongs to a vocabulary that a compiler enumerates; the search creates a generated object, which has no fixed enumeration. reGROW edits only the control structure of a declared vocabulary, which keeps every candidate finite and statically checkable.

| Design space | What search may change | Objects searched |
|---|---|---|
| Hand-designed pipeline | nothing | declared |
| Declarative pipeline (DSPy) | prompts and demonstrations of declared modules | declared |
| Regular grammar (ours) | regular expression structure only: module insertion, routing, retry, checkpointed rollback; modules, prompts, and tools frozen | declared, grammar-checked |
| Modular recombination (AgentSquare) | which module variant fills each of four slots, including implementations the LLM evolves | declared plus generated |
| Graph topology (GPTSwarm) | node prompts and connectivity within a declared node set | declared, generated text |
| Hierarchical composition (AgentSwift) | workflow skeleton and functional components | declared |
| Continuous supernet (MaAS) | a continuous distribution over architectures, sampled per query | continuous |
| Program space (AFlow, ADAS, FlowReasoner) | arbitrary agent code | generated, unbounded |

build structural, trace, and parent-history features; one training-free TabPFN value model reads a context of previous edits and their final gains and predicts the candidate's quality gain and mean child cost in context. We omit text embeddings for a fixed task set. Joint grouped bootstrap supplies the heuristic uncertainty proxy in Eq. (6); Appendix F.3 defines its diagnostics. (4) Batch selection: we open the cheapest non-empty cost layer first, then

Table 7: Default reGROW hyperparameters. Each benchmark assigns the same total search budget to all compared methods.

| Parameter | Value |
|---|---|
| Horizon $H$ | 16 |
| Max traces per expression | 4096 |
| Initial batch $n_0$ | 2–4 |
| Seeds per task | 1–2 |
| Tolerance $\epsilon$ | 0.05 |
| Confirmation failure probability $\delta_{\text{conf}}$ | 0.05 |
| Keep fraction | 1/3 |
| Survivors per stratum $k$ | 3 |
| Pair quota $q$ | 2–4 |
| Cost layers | $\{1, 2, 3, 5, 8, 13, \dots\}$ |
| Value model | Training-free TabPFN: in-context paired gain and child cost |

the lowest exposure stratum, then keep one exploration slot per $(A, \text{edit family})$ bucket, with uniform within-stratum sampling until ranking checks pass. (5) Paired evaluation: the batch runs on the same task IDs and matched task IDs and seed values. (6) Update: we update paired estimates and nominal racing intervals, retire candidates, keep $k$ survivors per stratum, record search-set GrowthTrace and parent pointers, and expand the frontier. The search starts at $W_0 = G$. One post-search confirmation pass selects among tested finalists; an independent test set supplies final reported estimates. The confirmation guarantee covers that tested set alone.

### F.3 TabPFN records and validation

Features precede execution. Each record describes a legal parent–child edit before its evaluation. Structural features cover both endpoints: $A$, $E$, call bounds, module counts, branch and retry limits, edit family, parameters, syntax-tree location, and adjacent module types. The record includes changes in structural quantities and measured parent quality, cost, sample count, failure modes, and trigger rates. Related single-edit and combination traces contribute fix/break/trigger proportions only when those executions already exist. A new candidate supplies no outcomes before execution. Paid pilot data require a new snapshot that distinguishes the pilot from later target tasks. The fixed-task default omits text embeddings.

Paired labels use a shared task panel. Let $\bar{r}_W(x) = m^{-1} \sum_{s=1}^{m} r(W, x, s)$ average seed values for task $x$. For a paired panel $X$, one table row has targets

$$d_X(W, e) = |X|^{-1} \sum_{x \in X} [\bar{r}_{e(W)}(x) - \bar{r}_W(x)], \qquad c_X(e(W)) = |X|^{-1} \sum_{x \in X} \bar{c}_{e(W)}(x).$$

The value model receives these targets as in-context examples rather than fitting parameters. We retain task-level records for resampling; repeated seeds do not become independent workflow rows. Evaluated failures and eliminated candidates remain in the archive. Static rejections receive no outcome label. Partial and extended panels carry their task counts; an extended panel replaces its earlier aggregate in the fitting table, avoiding duplicate labels.

Validation separates time and related records. Canonical expression IDs identify parents and children. A task-family ID groups each original task with its paraphrases and derivatives. All seeds of a task stay together. Before search, task families split into disjoint search, confirmation, and test sets. Confirmation and test outcomes never enter the value model's context or tune its predictions.

Online diagnostics save predictions before outcomes arrive and compare candidates on the same subsequent paired panel within one cost/exposure stratum. Only pairs with unequal measured quality count; ties do not supply ranking evidence. A correct ordering scores one,

an incorrect ordering zero, and a predicted tie one half. The default enables value-model ranking after at least 20 such pairs and accuracy above 0.5, and reverts to uniform sampling if the running check fails. This is an activation heuristic, not a significance test.

Joint resampling includes parent error. TabPFN supplies training-free tabular regression predictions through in-context learning (Hollmann et al., 2025). We give it a context of previously evaluated edits and their final gains, and it performs no gradient updates. For exploration we form $J = 8$ bootstrap contexts; context construction and inference cost enter $B_{\text{pred}}$. We sample recorded task families as blocks across all workflows, preserving paired outcomes and all seeds. We also resample available edit-lineage groups. Each replicate recomputes parent summaries and aggregate labels with each feature restricted to observations available at its original snapshot, refreshes the regression context, and predicts

$$R^{(j)}_{\text{pred}}(W') = R^{(j)}_{\text{obs}}(W) + \Delta R^{(j)}_{\text{pred}}(W'), \quad u(W') = \text{sd}^{J}_{j=1}\left[R^{(j)}_{\text{pred}}(W')\right].$$

Joint resampling preserves dependence between the parent estimate and predicted gain. $u$ is a proxy for uncertainty about mean quality, not total single-run outcome variance. The archive reports within-task execution variance separately. Sparse lineages can underestimate uncertainty; exploration quotas and uniform fallback remain active. Residual coverage and ranking on subsequent candidates diagnose this proxy without establishing a calibrated confidence bound. The configuration records the TabPFN checkpoint, library version, resampling seed, group IDs, replicate count, and activation threshold before each run.

### F.4 Coverage audit

The regular grammar covers finite outcome routing, optional steps with recovery, retry, parallel voting, and decomposition over a declared alphabet. It does not cover runtime creation of control edges, unbounded recursion, or cross-query memory. The evaluation protocol includes a coverage audit over a frozen catalog of published designs, which records the published structures the grammar can express. The 65 grammar expressions in Figure 3 are arithmetic demonstrations, not a coverage percentage.

### F.5 Reproducibility

The implementation is in the project's main/ directory. The CLI provides python -m regrow run --config configs/mock.json for an offline run and python -m regrow run --config configs/llm.json for an OpenAI-compatible endpoint. The package includes a deterministic mock oracle for tests, an optional LLM oracle with round-robin endpoints and enable_thinking=false, and 29 unit tests for the earlier $k$-nearest-neighbour prototype. AgentSwift Table 1 supplies the baseline reference rows.

## G Additional structural material

Table 8: Common workflow structures and their regular patterns. Each pattern lists possible module calls from left to right.

| Meaning | Regular pattern |
|---|---|
| One reasoning call | G |
| Sequential pipeline | abc |
| Exactly $n$ repetitions of one module | a{n} |
| Optional step; restore the prior working state if it fails | ab?c |
| Route by the query or current state to one of two declared modules | S(a\|b)c |
| Generate multiple candidates and vote | P_n |
| Process subtasks and join their results | D_n |
| Verify and revise as needed, with at most $n$ revisions | G(VR){0,n}V |
| Alternate reasoning and tool calls, with at most $n$ tool calls | G(TG){0,n} |
| Select among two declared workers at most $n$ times, then aggregate | (S(a\|b)){1,n}G |

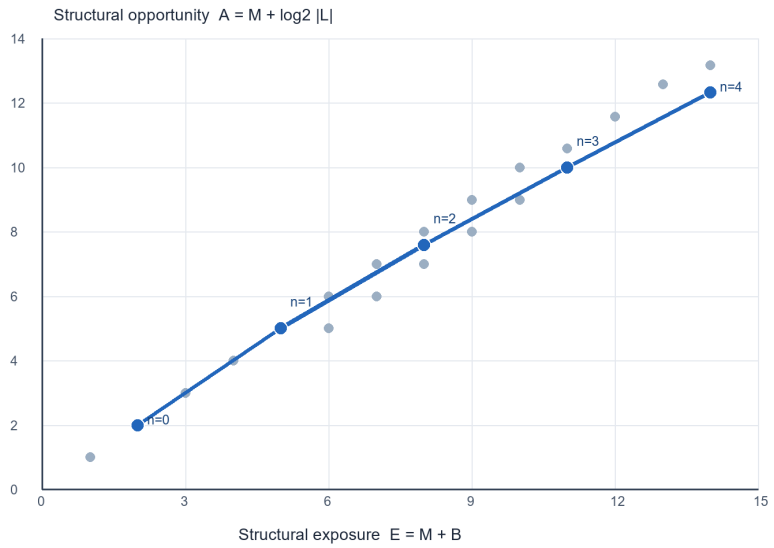



Figure 3: Coordinates computed from 65 grammar expressions under one module contract. Gray marks can overlap; blue marks show $W_n = G(VR)^{\{0,n\}}V$ for $n = 0, \ldots, 4$. The catalog combines grammar instances of published workflow patterns with our own compositions; it is an arithmetic demonstration, not a coverage estimate or a quality ranking.