# EVOSCAFFOLD: DISTILLING EXECUTABLE STRATEGIES FOR SMALL LANGUAGE MODELS

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

## ABSTRACT

Reasoning distillation transfers the reasoning capabilities of large teacher models to smaller students for more affordable deployment, typically through supervised fine-tuning. Prompt-level distillation avoids student parameter updates by encoding teacher guidance in system instructions, but complex multi-step problems can still exceed what the student can reliably solve in a single call. We introduce EVOSCAFFOLD, a non-parametric distillation framework that jointly evolves problem decomposition and subproblem instructions through genetic search for a frozen student. The resulting executable strategies encode reusable reasoning programs that specify student calls and any dependencies between them. Student predictions and execution traces guide teacher-generated revisions, while student task performance governs strategy selection. Once selected, a strategy is reused on unseen instances of the same task family without teacher access or student parameter updates. Experiments across four reasoning tasks and four student sizes demonstrate the effectiveness of EVOSCAFFOLD in improving the reasoning performance of frozen students.

## 1 INTRODUCTION

Multi-step reasoning underlies many language model applications, including question answering and mathematical problem solving. Yet large-model inference can be costly in resource-constrained settings, motivating the transfer of reasoning capabilities to smaller models. A common approach is supervised fine-tuning (SFT) on teacher-generated answers and rationales (Magister et al., 2023; Hsieh et al., 2023). Because supervision is provided through textual outputs, this approach can use closed-source teachers and supports deployment without teacher access. However, it requires additional training resources to update student parameters and cannot be applied when the student is accessible only through an inference interface. This raises a practical question: *can a teacher's reasoning capabilities be distilled without updating the student's parameters?*

Prompt-Level Distillation (PLD) (Badhe & Shah, 2026) offers a non-parametric alternative by deriving reusable system instructions from teacher analyses of labeled training examples. These instructions guide the student's responses to new inputs without updating its parameters. However, providing useful guidance does not ensure that a smaller student can reliably execute the reasoning it describes. The student still applies the instructions to the full problem within a single call, requiring it to perform and coordinate multiple interdependent steps. This distinction matters because models can correctly answer individual subquestions yet fail to solve the corresponding composite problem (Wei et al., 2022; Press et al., 2023). These findings suggest that effective non-parametric reasoning distillation requires not only useful instructions, but also an organization of the reasoning process that the student can reliably execute.

Explicit problem decomposition can make multi-step reasoning more manageable by assigning subproblems to separate student calls (Zhou et al., 2023a; Khot et al., 2023). However, a decomposition that is natural to the teacher may not match the student's capabilities. Coarse subproblems can remain too difficult, while excessive splitting introduces additional calls and opportunities for error (Prasad et al., 2024; Khot et al., 2023). Once a problem is decomposed, subproblem instructions must guide the student in solving each part and passing on the information needed by subsequent steps. Errors or omissions in these intermediate answers can compromise downstream reasoning (Khot et al., 2023). Reliable execution therefore depends on both how the problem is divided



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.

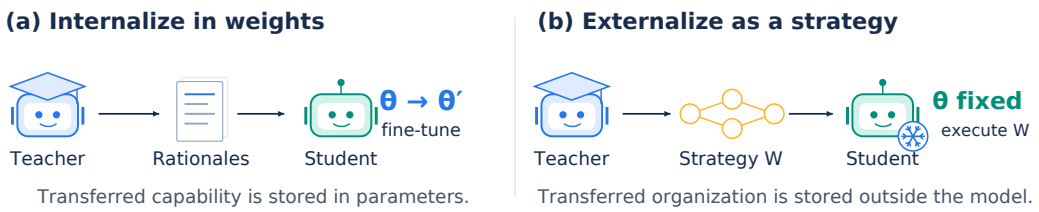



Figure 1: **Two carriers of teacher guidance.** Conventional distillation updates student parameters; EvoScaffold transfers an external strategy that organizes calls to a frozen student. Teacher supervision is used offline in both settings.

and how the student is instructed to solve its subproblems. *The challenge is to guide the teacher in adapting problem decomposition and subproblem instructions to the student's capabilities.*

To address this challenge, we introduce EvoScaffold, a non-parametric distillation framework that jointly evolves problem decomposition and subproblem instructions for a frozen student. Rather than internalizing teacher guidance in student parameters, we encode problem decomposition and subproblem instructions in external executable strategies that specify student calls and dependencies between them (Figure 1). Each strategy contains a collection of reusable reasoning programs that specify which subproblems the student solves, how it is instructed to solve them, and how their outputs are passed between calls. Student predictions and execution traces guide the teacher in revising the decomposition and the instructions, while task performance governs evolutionary selection.

We evaluate EvoScaffold on four reasoning tasks with students ranging from 0.8B to 9B parameters. It outperforms all evaluated non-parametric baselines on all four tasks with 0.8B and 2B students, while results for larger students are mixed. Our main contributions are:

- **Executable strategy distillation.** We introduce EvoScaffold, a framework that distills a teacher model's reasoning capabilities into reusable executable strategies whose reasoning programs organize calls to a frozen student.
- **Joint evolution.** We jointly evolve problem decomposition and subproblem instructions to suit the student, using its predictions and execution traces to guide teacher revisions and task performance to select strategies.
- **Empirical evaluation.** We examine how performance varies across tasks and student sizes, with ablations assessing joint evolution, guidance from student execution results, and explicit program execution.

## 2 Related Work

**Reasoning distillation.** Conventional distillation trains students on teacher targets (Hinton et al., 2015; Kim & Rush, 2016). For reasoning tasks, supervision includes teacher-generated rationales (Magister et al., 2023; Hsieh et al., 2023) and decompositions used to train separate decomposer and solver models, as in Socratic CoT (Shridhar et al., 2023). These approaches require parameter updates. PLD instead consolidates teacher-derived reasoning rules into reusable system instructions for a frozen student (Badhe & Shah, 2026). EvoScaffold retains teacher guidance in an external executable strategy rather than student parameters. Whereas PLD supplies consolidated instructions for solving the full problem in one call, EvoScaffold supports reasoning programs assigning subproblems to separate student calls and passing intermediate results between them.

**Problem decomposition.** Least-to-Most uses few-shot prompts to decompose problems and solve subproblems sequentially (Zhou et al., 2023a). Decomposed Prompting interleaves subproblem generation and execution through reusable handlers and a symbolic controller (Khot et al., 2023). ADaPT recursively decomposes tasks when execution fails, adapting decomposition to the executor's capabilities during inference (Prasad et al., 2024). SELF-DISCOVER selects, adapts, and composes reasoning modules into task-specific structures that guide answer generation (Zhou et al., 2024). Building on decomposition-based inference, EvoScaffold jointly optimizes reusable rea-

soning programs and subproblem instructions offline for a frozen student. Student execution feedback guides teacher revisions, while task performance governs strategy selection for reuse on unseen instances. Unlike SELF-DISCOVER's structures guiding reasoning during decoding, our programs are executed by an interpreter that invokes the specified solver calls and passes intermediate results along declared dependencies.

**Prompt and program optimization.** APE generates and selects instructions, while ProTeGi and OPRO guide revisions through textual critiques and prior evaluations, respectively (Zhou et al., 2023b; Pryzant et al., 2023; Yang et al., 2024). EvoPrompt and Promptbreeder apply evolutionary search to prompts (Guo et al., 2025; Fernando et al., 2023). For multi-stage programs, DSPy provides declarative language model modules, and TextGrad optimizes components through textual feedback (Khattab et al., 2023; Yuksekgonul et al., 2024). MIPRO optimizes instructions and demonstrations, while GEPA evolves prompts through execution-trace reflection and Pareto-based selection (Opsahl-Ong et al., 2024; Agrawal et al., 2026). Beyond optimizing prompts within a given program, EvoScaffold uses genetic operators to split or merge subproblems and revise their dependencies together with the associated solver instructions.

**Automated workflow search.** Automated workflow search studies how to compose model operations and control flow into task-solving procedures. GPTSwarm optimizes node prompts and graph connections (Zhuge et al., 2024), while ADAS and AFlow search code-defined agents and workflows through archival exploration and Monte Carlo tree search, respectively (Hu et al., 2025; Zhang et al., 2025). In contrast, EvoScaffold studies how to divide compositional reasoning into subproblems suited to a small student's capabilities and provide instructions for each part. It addresses the trade-off between local reasoning difficulty and coordination across subproblems by jointly evolving decomposition granularity and solver instructions using student execution feedback.

# 3 Method

## 3.1 Problem Setup

We study non-parametric reasoning distillation from a teacher $T$ to a student $S_\theta$ whose parameters $\theta$ remain fixed. Let $\mathcal{D}$ denote a distribution of examples $(x, y)$ within a task family, where $x$ comprises a question and its available evidence, and $y$ is the reference answer. We seek a reusable executable strategy $W \in \mathcal{W}$ that encodes teacher guidance through problem decomposition, subproblem instructions, and explicit dependencies between intermediate results. Writing $\mathrm{Exec}(S_\theta, W, x)$ for the student's prediction under $W$, our objective is

$$\max_{W \in \mathcal{W}} \; \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[ R\big(\mathrm{Exec}(S_\theta, W, x), y\big) \right], \tag{1}$$

where $\mathcal{W}$ is the space of admissible executable strategies and $R$ is a task-specific score for which larger values indicate better performance.

We optimize $W$ through offline strategy distillation: the teacher proposes problem decompositions and subproblem instructions, and student execution results guide their joint refinement. We use $\mathcal{D}_{\mathrm{tr}}$ for initialization and evolution, a held-out set $\mathcal{D}_{\mathrm{sel}}$ for final strategy selection, and $\mathcal{D}_{\mathrm{te}}$ for testing. After selection, $W$ is fixed and reused on unseen instances from the same task family, without teacher access or student parameter updates. Figure 2 summarizes this process.

## 3.2 Executable Strategies

**Strategy representation.** An executable strategy distills the teacher's problem-solving procedure into interdependent subproblems that the student can solve separately. A useful strategy must specify both which subproblems to solve and how the student should solve them: subproblem boundaries and dependencies organize the reasoning process, while local instructions guide each step. We therefore represent these complementary components explicitly as reasoning programs and shared solver modules.

Formally, a strategy is represented as $W = (r, \mathcal{G}, \mathcal{M})$, where $r$ is a routing instruction, $\mathcal{G}$ is a collection of reasoning programs, and $\mathcal{M}$ is a set of shared solver modules. The routing instruction guides the student in selecting an applicable program for each question.

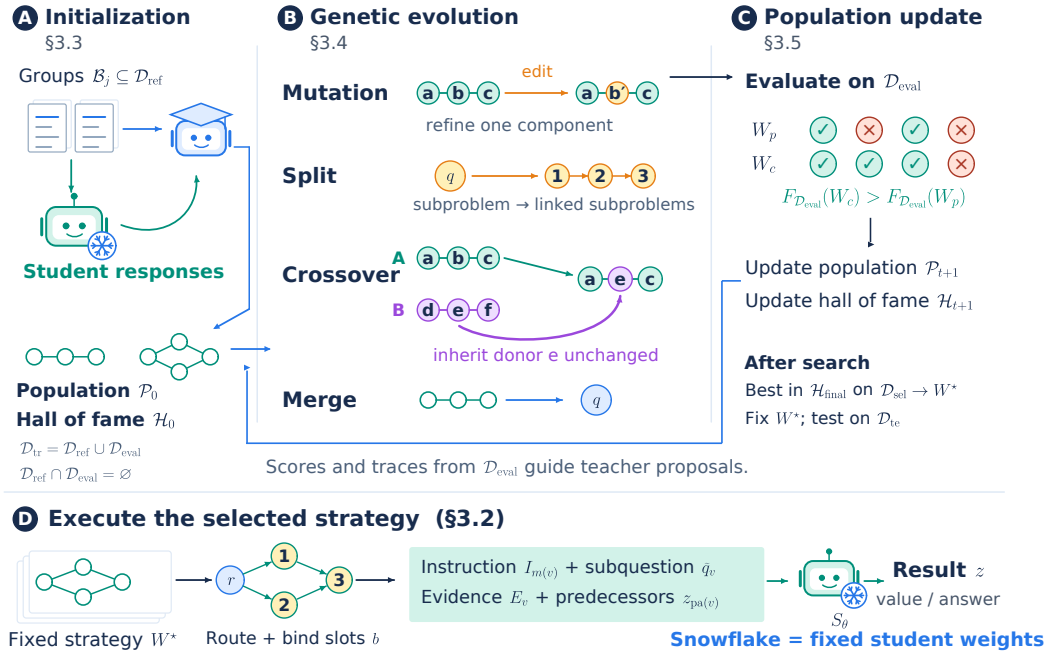



Figure 2: **Learning the organization of inference.** (A) Labeled training examples and student responses guide initialization. (B) Mutation, split, crossover, and merge revise strategy components. (C) Paired student executions assess offspring relative to their parents. (D) The selected strategy organizes bounded local calls to the frozen student.

Each program $g \in \mathcal{G}$ specifies applicability conditions, input slots, and a directed acyclic graph of subproblem templates with a designated output node. Slots bind values from the question, while dependencies pass intermediate outputs between subproblems. For example, finding awards won by films written by people born in a given year follows a person–film–award procedure. The year fills an input slot; people, films, and awards are computed during execution, making the program reusable across questions.

Each node $v$ references a solver module $m(v) \in \mathcal{M}$, which supplies its subproblem instruction and evidence-access policy. Modules may be shared across programs and are executed by the same frozen student. A node may process predecessor outputs jointly or separately for each item. Each local solver invocation uses one student call; mapping or evidence partitioning may expand a node into multiple invocations.

Programs thus organize the decomposition, while modules guide local reasoning. Separating them allows genetic operators to improve instructions within an existing decomposition or revise the decomposition together with the instructions needed to execute it.

**Strategy execution.** The student follows $r$ to select a program and fill its input slots using only the question and program descriptions, yielding bindings $b$. The selected program is then executed in dependency order. At each node $v$, the subproblem template $q_v$ is instantiated as $\widetilde{q}_v = \text{Bind}(q_v, b, z_{\text{pa}(v)})$, where $\text{pa}(v)$ denotes the declared predecessors and $z_{\text{pa}(v)}$ their outputs. The associated solver module guides the student in solving the instantiated subproblem. A local student call takes the form

$$z = S_\theta\big(I_{m(v)}, \widetilde{q}_v, E_v, z_{\text{pa}(v)}\big)\,, \tag{2}$$

where $I_{m(v)}$ is the subproblem instruction, $E_v$ is the evidence view determined by the module's evidence-access policy, and $z$ is the returned structured result.

These local computations are connected through the program's declared dependencies. The interpreter constructs node outputs from the returned results and passes them as structured inputs to subsequent subproblems. Calls therefore exchange explicit intermediate results rather than an ac-

cumulated reasoning transcript. Execution continues until the designated output node produces the final answer. The interpreter thus controls execution order and intermediate-value flow, while the student performs the semantic reasoning required by each subproblem. Detailed execution configurations are provided in the appendix. Figure 3 illustrates an evolved strategy.

### 3.3 Student-Calibrated Initialization

Initialization uses the student's observed successes and errors to guide the teacher's first choices of problem decomposition and subproblem instructions. We partition $\mathcal{D}_{\text{tr}}$ into a reference subset $\mathcal{D}_{\text{ref}}$ for initialization and a remaining subset $\mathcal{D}_{\text{eval}}$ for strategy evaluation and evolutionary feedback. For each $(x_i, y_i) \in \mathcal{D}_{\text{ref}}$, we collect a direct student response $\hat{y}_i = S_\theta(x_i)$, providing initial evidence of the student's capabilities on the task.

For each proposal $j$, we sample a group $\mathcal{B}_j \subseteq \mathcal{D}_{\text{ref}}$ and provide the teacher with its labeled examples and corresponding student responses. The teacher uses this evidence to propose a complete strategy $W_j$ in the representation introduced in Section 3.2. It is instructed to formulate reusable problem decompositions and subproblem instructions informed by the student's performance, rather than encode instance-specific facts or answer mappings. Varying the sampled groups encourages alternative initial strategies. Detailed generation constraints are provided in the appendix.

We validate and deduplicate the proposals, then evaluate the remaining candidates with the frozen student on $\mathcal{D}_{\text{eval}}$, retaining their scores and execution records. Let $\mathcal{C}_{\text{init}}$ denote these evaluated candidates. We rank them by their mean task score $F_{\mathcal{D}_{\text{eval}}}(W)$, defined in Equation 3. The highest-ranked candidates initialize the population $\mathcal{P}_0$ and the bounded hall of fame $\mathcal{H}_0$ according to their respective capacities, as specified in Section 3.5.

### 3.4 Executable Strategy Evolution

Starting from the initial population, evolution adapts problem decomposition and local instructions to the student. Unrestricted rewrites may disrupt useful components while attempting to repair errors. We therefore use population-based genetic search with specialized operators that make the scope of each revision explicit and checkable. The population retains alternative decompositions and supplies complementary components for recombination. Student execution determines whether each proposed revision improves on its parent.

At generation $t$, we select parents from $\mathcal{P}_t$. For a parent $W_p$, feedback $\mathcal{F}_p$ contains student predictions and node-level execution traces from successful and failed examples in $\mathcal{D}_{\text{eval}}$, when available. These executions indicate what should be improved or preserved, while the selected genetic operator specifies how the teacher may revise the strategy. For a single-parent revision under operator $o$, the teacher proposes $W_c \sim T(\cdot \mid W_p, \mathcal{F}_p, o)$; crossover additionally uses a second parent and its execution feedback. The operators support three complementary types of change: local component refinement, recombination across strategies, and adjustment of decomposition granularity.

**Mutation: refine one component.** Mutation modifies one component to address weaknesses revealed by student execution. It may revise the routing instruction, one solver module's instruction or evidence-access policy, the textual specification of one reasoning program, or the execution bounds, while preserving program topology, node–module assignments, and execution grouping.

**Crossover: recombine strategies.** Crossover incorporates solver modules from one parent into the reasoning programs of another. It reuses complementary components discovered in different strategies, allowing useful decompositions to benefit from alternative solver instructions.

**Split and merge: explore decomposition granularity.** Split divides a demanding subproblem into smaller units, while merge combines related subproblems into a broader unit. Together, they balance local reasoning difficulty against the coordination required to connect intermediate results, adapting decomposition granularity to the student's capabilities.

We validate and deduplicate proposed strategies before student evaluation. These checks establish candidate validity, while student task performance determines whether a valid revision is beneficial and should enter subsequent selection.

**Algorithm 1** Executable Strategy Evolution

**Require:** Teacher $T$, frozen student $S_\theta$, datasets $\mathcal{D}_{\text{tr}}, \mathcal{D}_{\text{sel}}, \mathcal{D}_{\text{te}}$
**Require:** Initial proposal count $N_0$, population capacity $K$, offspring per generation $\lambda$, generation limit $G$, hall-of-fame capacity $K_H$, teacher-call budget $B_T$
**Ensure:** Selected strategy $W^\star$

```
Split D_tr into D_ref and D_eval
Generate up to N_0 calibrated proposals from D_ref, respecting B_T (§3.3)
Validate and deduplicate proposals; score using F_{D_eval} to obtain C_init
P_0 ← Top_K^eval(C_init); H_0 ← Top_{K_H}^eval(C_init)
for t = 0, ..., G − 1, while budget B_T remains do
    Select parents W_p from P_t and retrieve execution feedback F_p from D_eval
    Use T and the genetic operators to propose up to λ offspring, respecting B_T (§3.4)
    Collect valid, unseen offspring in C_t and score using F_{D_eval}
    Q_t ← {W_c ∈ C_t : F_{D_eval}(W_c) > F_{D_eval}(W_p)}, where W_p is its parent (the recipient for crossover)
    U_t ← P_t ∪ Q_t; P_{t+1} ← Top_K^eval(H_t ∪ U_t) (§3.5)
    H_{t+1} ← Top_{K_H}^eval(H_t ∪ P_{t+1})
end for
Let H_final be the last available hall of fame
Select W* ∈ arg max_{W∈H_final} F_{D_sel}(W)
Fix W*; evaluate once on D_te
return W*
```

### 3.5 Population Update and Final Selection

**Student assessment.** We measure strategy fitness by the frozen student's mean task score on an evaluation set $\mathcal{B}$:

$$F_{\mathcal{B}}(W) = \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} R\big(\text{Exec}(S_\theta, W, x), y\big), \tag{3}$$

where $\text{Exec}(S_\theta, W, x)$ is the answer produced by executing $W$ with the student on input $x$, and $R$ scores that answer against reference $y$ using the task's evaluation metric. During evolution, we compare each valid offspring $W_c$ with its parent $W_p$ on $\mathcal{D}_{\text{eval}}$ under the same student. This parent–offspring improvement check admits $W_c$ to $\mathcal{Q}_t$ only if $F_{\mathcal{D}_{\text{eval}}}(W_c) > F_{\mathcal{D}_{\text{eval}}}(W_p)$, so admission depends on observed student performance rather than the teacher's judgment of the revision.

**Population and hall of fame.** The population supplies parents for further evolution, while a bounded hall of fame preserves historical elite strategies for reuse and final selection. Let $\text{Top}_k^{\text{eval}}$ remove duplicate strategies and retain up to $k$ candidates with the highest $F_{\mathcal{D}_{\text{eval}}}$. At generation $t$, we form $\mathcal{U}_t = \mathcal{P}_t \cup \mathcal{Q}_t$ and update the population as $\mathcal{P}_{t+1} = \text{Top}_K^{\text{eval}}(\mathcal{H}_t \cup \mathcal{U}_t)$, where $K$ is the population capacity. Historical elites thus compete with current parents and admitted offspring.

The hall of fame has capacity $K_H$ and is initialized as $\mathcal{H}_0 = \text{Top}_{K_H}^{\text{eval}}(\mathcal{C}_{\text{init}})$. After forming the next population, we update it by

$$\mathcal{H}_{t+1} = \text{Top}_{K_H}^{\text{eval}}\big(\mathcal{H}_t \cup \mathcal{P}_{t+1}\big). \tag{4}$$

Both updates use the same evaluation fitness, maintaining a bounded collection of strong strategies without retaining every candidate generated during search.

**Final selection.** Search stops after $G$ generations or when the teacher-call budget $B_T$ is exhausted. To select a strategy beyond the examples used during evolution, we evaluate the final hall of fame $\mathcal{H}_{\text{final}}$ on the held-out selection set $\mathcal{D}_{\text{sel}}$ and choose

$$W^\star \in \arg\max_{W\in\mathcal{H}_{\text{final}}} F_{\mathcal{D}_{\text{sel}}}(W). \tag{5}$$

Selection results are not fed back into evolution. We then fix $W^\star$ and evaluate it once on $\mathcal{D}_{\text{te}}$. Algorithm 1 summarizes the procedure.

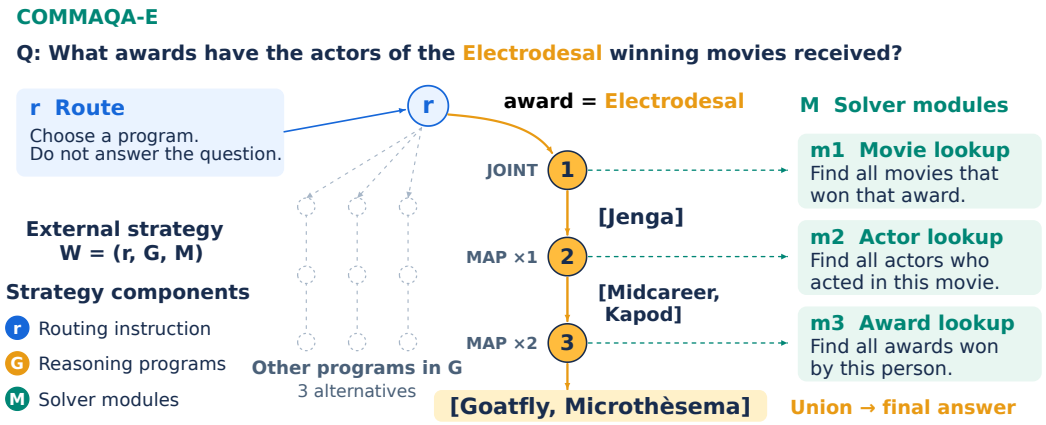



Figure 3: **An evolved COMMAQA-E strategy** (4B, generation 5). The router binds the example question to the selected program (shown in gold), whose nodes invoke reusable solver modules (teal links). Intermediate results flow along the program to produce the final answer. Cards show prompt excerpts; gray paths denote alternative programs.

# 4 Experiments

## 4.1 Experimental Setting

**Tasks and data.** We evaluate EvoScaffold on four reasoning tasks: movie-relation composition in COMMAQA-E (Khot et al., 2022), relational reasoning in PhantomWiki (Gong et al., 2025), nested belief tracking in ExploreToM (Sclar et al., 2024), and indirect cash-flow calculations in FinChain (Xie et al., 2026). All methods share fixed training, selection, and test partitions. Teacher feedback is restricted to training examples, while the selection set determines the final strategy before test evaluation. We report each task's native metric: exact match for COMMAQA-E, set F1 for PhantomWiki, native substring accuracy for ExploreToM, and final-answer accuracy for FinChain.

**Models and search budget.** We use Qwen3.5 (Qwen Team, 2026) (0.8B, 2B, 4B, and 9B) as the student models, and DeepSeek-V4-Pro (DeepSeek-AI, 2026) as the teacher model. The four student models belong to the same Qwen3.5 small-model series, sharing a unified training data recipe and architecture design, but differing in parameter scale to cover a range of capacity constraints.

**Baselines.** Student and teacher inference serve as capability references. We compare EvoScaffold with methods that transfer reasoning guidance through parameter updates, reusable instructions, or executable strategies.

For parametric distillation, we train students on teacher-sampled reasoning trajectories using LoRA SFT and full SFT. LoRA (Hu et al., 2021) learns low-rank updates while keeping the base parameters frozen, whereas full SFT updates all student parameters.

For non-parametric transfer, MIPROv2 (Opsahl-Ong et al., 2024) optimizes module instructions, while GEPA (Agrawal et al., 2026) evolves module prompts through execution-based reflection and Pareto-based candidate selection. PLD (Badhe & Shah, 2026) consolidates teacher-derived reasoning patterns into system instructions and refines them using student feedback. SELF-DISCOVER (Zhou et al., 2024) selects, adapts, and composes reasoning modules into reusable task-level reasoning prompts. Finally, AFlow (Zhang et al., 2025) searches code-defined workflows through Monte Carlo tree search and execution feedback. To examine AFlow's decomposition and student-call coordination, we restrict its workflows to frozen-student calls, excluding external tools and auxiliary programmatic task-solving. We denote this AFlow (restricted).

We match the teacher-token budget across all distillation and optimization methods, including LoRA and full SFT, for which the budget is used to generate training trajectories. Detailed baseline configurations, adaptations, and teacher-token accounting are provided in the appendix.

Table 1: **Task performance across student scales.** Each dataset has four student-size columns. Metrics (%): COMMAQA-E exact match, PhantomWiki set F1, ExploreToM native substring accuracy, and FinChain final-answer accuracy. Teacher scores span student sizes.

| | COMMAQA-E | | | | PhantomWiki | | | | ExploreToM | | | | FinChain | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Method | 0.8B | 2B | 4B | 9B | 0.8B | 2B | 4B | 9B | 0.8B | 2B | 4B | 9B | 0.8B | 2B | 4B | 9B |
| **I. Inference references** | | | | | | | | | | | | | | | | |
| Teacher | | 97.91 | | | | 99.93 | | | | 81.68 | | | | 100.00 | | |
| Student | 10.91 | 21.97 | 47.68 | 58.30 | 37.92 | 65.87 | 86.76 | 89.89 | 37.17 | 32.46 | 40.31 | 43.46 | 12.30 | 37.11 | 78.52 | 83.59 |
| **II. Parametric distillation references** | | | | | | | | | | | | | | | | |
| LoRA SFT | 45.29 | 59.49 | 75.19 | 76.98 | 69.43 | 80.88 | 89.70 | 84.21 | 63.35 | 67.54 | 75.39 | 79.06 | 18.36 | 41.41 | 83.98 | 82.42 |
| Full SFT | 73.99 | 84.45 | 90.58 | 91.63 | 82.67 | 89.06 | 92.66 | 94.30 | 66.49 | 75.92 | 86.39 | 85.34 | 20.31 | 49.80 | 88.48 | 86.91 |
| **III. Non-parametric optimization and strategy transfer** | | | | | | | | | | | | | | | | |
| PLD | 20.18 | 35.13 | 45.59 | 63.38 | 49.24 | 73.91 | 95.22 | 96.87 | 43.46 | 49.74 | 58.12 | 78.01 | 63.67 | 71.48 | 85.74 | 85.16 |
| SELF-DISCOVER | 20.78 | 30.49 | 42.60 | 56.20 | 45.51 | 63.02 | 90.94 | 96.73 | 33.51 | 37.70 | 36.13 | 52.36 | 70.12 | 77.73 | 85.94 | 83.59 |
| MIPROv2 | 28.40 | 51.42 | 81.46 | 82.21 | 57.48 | 69.49 | 92.39 | 96.30 | 54.45 | 60.21 | 75.92 | 81.15 | 81.84 | 91.02 | **99.41** | 99.41 |
| GEPA | 29.45 | 52.91 | 84.90 | 87.14 | 57.66 | 68.39 | **97.00** | 98.45 | 51.31 | 58.64 | **84.29** | **87.43** | 75.00 | 89.06 | 99.02 | 99.61 |
| AFlow (restricted) | 20.93 | 49.78 | 74.74 | 78.77 | 45.53 | 64.53 | 91.26 | 91.96 | 47.64 | 50.26 | 78.01 | 79.58 | 74.22 | 81.64 | 98.44 | **99.80** |
| **EVOSCAFFOLD** | **38.71** | **60.99** | **93.27** | **92.53** | **62.47** | **77.09** | 96.59 | **98.81** | **61.78** | **64.92** | 78.01 | 80.10 | **84.57** | **92.19** | 90.04 | 93.16 |

### 4.2 MAIN RESULTS

Table 1 shows that EVOSCAFFOLD is particularly effective for small students, outperforming all non-parametric baselines across all tasks with the two smallest models. A plausible explanation is that explicit decomposition turns complex tasks into manageable local steps, easing the burden on students that struggle to coordinate multi-step reasoning. For example, the evolved COMMAQA-E strategy in Figure 3 separately identifies award-winning movies, their actors, and the actors' awards, with the interpreter passing intermediate results between student calls. By contrast, stronger students may already handle such reasoning with optimized instructions, leaving less room for additional gains from decomposition. Nevertheless, full SFT remains stronger in several settings. Rather than serving as a general replacement for SFT, EVOSCAFFOLD offers a parameter-update-free route to distilling teacher reasoning strategies without the training cost of updating student parameters.

### 4.3 ABLATION STUDY

We ablate on COMMAQA-E with 0.8B and 4B students (Table 2) to examine three questions: (1) whether joint evolution of problem decomposition and subproblem instructions improves on initialization and instruction-only evolution, (2) how student execution feedback and genetic operators contribute to evolution, and (3) how strategy execution and evidence access affect performance.

**Cold start only** selects the highest-scoring executable strategy from the initial candidates without further evolution. Prompt-only evolution starts from the cold-start strategy with the highest score on the training set ($\mathcal{D}_{\text{eval}}$) and uses the same evolution budget as the full method to evolve only routing and subproblem instructions, keeping problem decomposition fixed. **w/o execution context** removes student predictions and node-level execution traces from teacher feedback during evolution, while retaining score-based search and selection. We further disable **mutation**, **crossover**, **split**, and **merge** individually to assess the contributions of strategy repair, recombination, and exploration of decomposition granularity. Finally, **flattened execution** presents the learned strategy in a single prompt, testing the contribution of executing subproblems through separate student calls with explicit dependencies. **Full-evidence execution** retains the learned problem decomposition but supplies each solver call with all available evidence, assessing the selected strategy's dependence on its learned evidence-access policies.

The full method outperforms prompt-only evolution, supporting the contribution of evolving problem decomposition alongside subproblem instructions; the gain is larger for the 0.8B student. Removing most genetic operators also degrades performance, although the improvement without merge suggests that the 0.8B student can benefit from preserving finer subproblem boundaries in this set-

Table 2: **Ablation study with 0.8B and 4B students on COMMAQA-E.**

| Variant | 0.8B | 4B |
|---|---|---|
| Full EvoScaffold | 38.71 | 93.27 |
| *Initialization and execution context* | | |
| Cold start only | 22.27 | 87.74 |
| Prompt-only evolution | 32.14 | 91.63 |
| w/o execution context | 36.92 | 91.18 |
| *Evolutionary operators* | | |
| w/o mutation | 34.08 | 89.99 |
| w/o crossover | 36.77 | 91.48 |
| w/o split | 36.02 | 90.73 |
| w/o merge | 40.81 | 92.53 |
| *Execution* | | |
| Flattened execution | 13.60 | 25.86 |
| Full-evidence execution | 11.66 | 24.66 |

Table 3: **Instruction activation and inference-token usage on COMMAQA-E.**

| Method | Active/call (tokens) | Cumulative/ question (tokens) | Total/question (k tokens) |
|---|---|---|---|
| *Qwen3.5-0.8B* | | | |
| MIPROv2 | 309 | 309 | 3.11 |
| GEPA | 999 | 999 | 3.71 |
| PLD | 277 | 277 | 2.78 |
| SELF-DISCOVER | 493 | 493 | 3.02 |
| AFlow (restricted) | 34 | 34 | 2.65 |
| EvoScaffold | 212 | 1,057 | 3.92 |
| *Qwen3.5-4B* | | | |
| MIPROv2 | 158 | 158 | 2.99 |
| GEPA | 646 | 646 | 3.59 |
| PLD | 252 | 252 | 2.70 |
| SELF-DISCOVER | 337 | 337 | 2.78 |
| AFlow (restricted) | 82 | 326 | 11.02 |
| EvoScaffold | 115 | 838 | 4.61 |

ting. The large drops under flattened and full-evidence execution indicate that the selected strategies rely on their multi-call execution configuration and learned evidence-access policies.

### 4.4 Cross-Student Transfer

To examine whether learned strategies can be reused across students, we change only the executor (student), without further optimization (Figure 4). Strategies adapted to smaller students remain effective with larger executors and sometimes outperform strategies optimized directly for the target model. By contrast, strategies transferred from larger to smaller students score below those selected for the target student in all evaluated pairs. The selected strategies differ in both problem decomposition and subproblem instructions, so these results do not isolate the source of this transfer asymmetry. Detailed descriptions of the evolved strategies are provided in the appendix.

### 4.5 Instruction Activation and Inference-Token Usage

Table 3 reports mean token usage on COMMAQA-E, using only completed, auditable executions and including routing. Active/call and Cumulative/question count instruction tokens per call and cumulatively per question, respectively; Total/question counts complete inputs and outputs (1k = 1,000 tokens). EvoScaffold activates fewer instruction tokens per call than the four prompt-based baselines, but has higher cumulative instruction and total token usage per question. Nevertheless, its 4B strategy uses 58% fewer total tokens than AFlow (restricted). Together with the higher task scores in Table 1, these results show that concise local instructions can support effective multi-call execution, at the cost of additional per-question tokens relative to the prompt-based baselines.

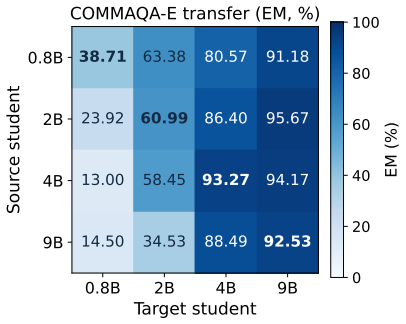



Figure 4: **Cross-student transfer.** COMMAQA-E exact-match values; rows fix the source strategy, columns change its executor.

## 5 Conclusion

We present EvoScaffold, a non-parametric approach to distilling teacher reasoning into reusable executable strategies for frozen students. Offline search jointly adapts problem decomposition and subproblem instructions to the student's execution capabilities, and the resulting strategies can be reused without teacher interaction or parameter updates. Our results show that this approach is particularly beneficial for smaller students, supporting the view that the organization of inference can serve as transferable reasoning knowledge. The method is best suited to tasks with recurring compositional structure; highly heterogeneous problems or those whose reasoning cannot be decomposed without losing essential context may require more flexible, instance-specific adaptation.

## AI USE STATEMENT

We used generative AI tools, including ChatGPT and Cursor, to assist with manuscript preparation and literature discovery.

Writing. Parts of the manuscript were initially drafted in a mixture of the authors' native language and English. ChatGPT was used to assist with English translation, language polishing, and improving presentation. All AI-assisted text was subsequently reviewed and revised by the authors to ensure technical accuracy, clarity, and consistency with the intended claims.

Related work discovery. Cursor was used to assist with identifying potentially relevant literature. We first searched for related work through Google Scholar and by following references in relevant papers, and then used Cursor to help identify additional candidate papers. All papers suggested through AI-assisted search were independently checked against the original sources before being cited. The authors take full responsibility for the content and citations in the manuscript.

## REFERENCES


Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Sanket Badhe and Deep Shah. Prompt-level distillation: A non-parametric alternative to model fine-tuning for efficient reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 2131–2147, San Diego, California, USA, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-394-4. doi: 10.18653/v1/2026.acl-industry.142. URL https://aclanthology.org/2026.acl-industry.142/.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023. URL https://arxiv.org/abs/2309.16797.

Albert Gong, Kamilė Stankevičiūtė, Chao Wan, Anmol Kabra, Raphael Thesmar, Johann Lee, Julius Klenke, Carla P Gomes, and Kilian Q Weinberger. PhantomWiki: On-demand datasets for reasoning and retrieval evaluation. 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. 19964–19995. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/gong25d.html.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers, 2025. URL https://arxiv.org/abs/2309.08532.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. URL https://arxiv.org/abs/1503.02531.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.507. URL https://aclanthology.org/2023.findings-acl.507/.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/36b7acf6f6010652b3f2a433774a66fe-Abstract-Conference.html.

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 self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

Tushar Khot, Kyle Richardson, Daniel Khashabi, and Ashish Sabharwal. Hey AI, can you solve complex tasks by talking to agents? In *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 1808–1823, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.142. URL https://aclanthology.org/2022.findings-acl.142/.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks, 2023. URL https://arxiv.org/abs/2210.02406.

Yoon Kim and Alexander M. Rush. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1317–1327, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1139. URL https://aclanthology.org/D16-1139/.

Lucie Charlotte Magister, Jonathan Mallinson, Jakub Adamek, Eric Malmi, and Aliaksei Severyn. Teaching small language models to reason. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1773–1781, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.151. URL https://aclanthology.org/2023.acl-short.151/.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.264. URL https://aclanthology.org/2024.findings-naacl.264/.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378/.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Melanie Sclar, Jane Yu, Maryam Fazel-Zarandi, Yulia Tsvetkov, Yonatan Bisk, Yejin Choi, and Asli Celikyilmaz. Explore theory of mind: Program-guided adversarial data generation for theory of mind reasoning, 2024. URL https://arxiv.org/abs/2412.12175.

Kumar Shridhar, Alessandro Stolfo, and Mrinmaya Sachan. Distilling reasoning capabilities into smaller language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 7059–7073, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.441. URL https://aclanthology.org/2023.findings-acl.441/.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. 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. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Zhuohan Xie, Daniil Orel, Rushil Thareja, Dhruv Sahnan, Hachem Madmoun, Fan Zhang, Debopriyo Banerjee, Georgi Nenkov Georgiev, Xueqing Peng, Lingfei Qian, Jimin Huang, Jinyan Su, Aaryamonvikram Singh, Rui Xing, Rania Elbadry, Chen Xu, Haonan Li, Fajri Koto, Ivan Koychev, Tanmoy Chakraborty, Yuxia Wang, Salem Lahlou, Veselin Stoyanov, Sophia Ananiadou, and Preslav Nakov. FinChain: A symbolic benchmark for verifiable chain-of-thought financial reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14529–14553, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.662. URL https://aclanthology.org/2026.acl-long.662/.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024. URL https://arxiv.org/abs/2309.03409.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui 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, 2025. URL https://arxiv.org/abs/2410.10762.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models, 2023a. URL https://arxiv.org/abs/2205.10625.

Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V. Le, Ed H., Denny Zhou, Swaroop Mishra, and Huaixiu Steven Zheng. Self-discover: Large language models self-compose reasoning structures. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 126032–126058. Curran Associates, Inc., 2024. doi: 10.52202/079017-4004. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/e41efb03e20ca3c231940a3c6917ef6f-Paper-Conference.pdf.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2023b. URL https://arxiv.org/abs/2211.01910.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

# A Experimental Protocol and Configuration

## A.1 Experimental protocol

All methods use the same training, validation, and test splits for each task. Training data are used for prompt or strategy optimization and supervised fine-tuning. The validation split, denoted $\mathcal{D}_{\text{sel}}$ in the main text, is used to select the final prompt, strategy, or checkpoint. Validation and test examples are not provided to the teacher as optimization feedback. After selection, the chosen artifact is fixed for test evaluation. Across methods, each student call has an 8,192-token window covering both input and output. SFT uses the same maximum sequence length. All methods receive the same task evidence and use the same answer interface and scoring function. We average the task metric over all test questions, assigning zero credit to unsuccessful executions. Each configuration is evaluated in a single run.

## A.2 Datasets and task metrics

We evaluate recurring task families within four benchmarks to assess strategy reuse across unseen instances with related reasoning structure. Table A.1 summarizes the data splits.

Table A.1: Dataset sizes and grouping.

| Task | Training | Selection | Test | Grouping |
|---|---|---|---|---|
| COMMAQA-E | 256 | 160 | 669 | Official split; four templates |
| PhantomWiki | 256 | 160 | 512 | Disjoint size-12 worlds |
| ExploreToM | 153 | 100 | 191 | Disjoint actor–object groups |
| FinChain | 256 | 160 | 512 | Independent numeric instances |

**COMMAQA-E.** We retain four three-hop templates from the official movie-domain IID data: awards of movies written by people born in a given year; awards of movies produced by people born in a given year; awards of movies directed by award winners; and awards of actors in award-winning movies. We sample 64 training and 40 validation questions per template and retain all 669 eligible test questions, yielding split sizes of 256/160/669. We use the inherited DecomP/DROP-style exact-match metric, which compares normalized answer spans and requires the same number of members.

**PhantomWiki.** We generate size-12 worlds using the official generator and retain questions composing three primitive person relations. Worlds are disjoint across splits, with one deterministically selected question per world. Complete evidence must fit a 2,200-token eligibility limit; this filter is independent of model performance and answer size. We report set F1 over whole entity names, computed as $2|P \cap Y|/(|P| + |Y|)$ for predicted and reference sets $P$ and $Y$.

**ExploreToM.** We use official second-order belief questions about an object's container, restricted to scenarios with at least three moves. Deduplication and label-consistency checks yield 444 questions, partitioned into 153/100/191 training/validation/test examples. Actor–object groups are disjoint across splits, with observable movement and asymmetry families balanced subject to this constraint. We use the native substring-accuracy metric: a returned string is correct if it contains the reference location after lowercasing. No synthetic expansion is used.

**FinChain.** We instantiate the official advanced indirect cash-flow-from-operations template. Numerical instances are generated independently, checked by deterministic label replay, and split by numerical-instance identity. Final-answer accuracy requires a single signed-integer string with the correct integer value; expressions and intermediate-amount lists are not accepted.

Table A.3: Recorded teacher usage and shared baseline budgets in k tokens (1k = 1,000 tokens), rounded to the nearest k. EvoScaffold usage comes from reference searches capped at 80 calls; the baseline budget is the mean across student sizes.

| Dataset | EvoScaffold usage | | | | Baseline budget |
|---|---|---|---|---|---|
| | 0.8B | 2B | 4B | 9B | |
| COMMAQA-E | 1,800 | 1,820 | 1,737 | 1,827 | 1,796 |
| PhantomWiki | 1,434 | 1,584 | 1,507 | 1,569 | 1,523 |
| ExploreToM | 1,363 | 1,317 | 1,331 | 1,346 | 1,339 |
| FinChain | 1,079 | 991 | 1,141 | 1,007 | 1,054 |

### A.3 Models and search settings

We use DeepSeek-V4-Pro as the teacher and Qwen3.5-0.8B, 2B, 4B, and 9B as students. For all non-parametric methods, the teacher generates or optimizes student's instructions or procedures, while the target student performs all task-solving calls without parameter updates.

EvoScaffold uses greedy student decoding with thinking disabled. Each search has a total limit of 80 logical teacher calls, covering initialization and evolutionary proposals. Training-set performance governs evolutionary selection, and a valid offspring is admitted only if it strictly improves on its parent. The final strategy is selected from the hall of fame using validation performance. Table A.2 lists the principal search hyperparameters.

Table A.2: Model and search hyperparameters.

| Setting | Default value |
|---|---|
| Teacher / thinking | deepseek-v4-pro / enabled |
| Students / thinking | Qwen3.5-0.8B, 2B, 4B, 9B / disabled |
| Teacher context / output allowance | 65,536 / up to 24,576 tokens |
| Initial proposals ($N_0$) / population ($K$) | 12 / 8 |
| Hall-of-fame capacity ($K_H$) | 5 |
| Generations $G$ / offspring per generation $\lambda$ | 5 / 12 |
| Reference-set size ($\lvert\mathcal{D}_{\text{ref}}\rvert$) | 32 |
| Examples per initial proposal ($\lvert B_j \rvert$) | 4 |
| Graph capacity | Up to 8 programs, 12 modules, 8 nodes per program |
| Evolvable execution limit | Up to 24 student calls, including routing and expanded calls |

### A.4 Teacher-token budgets

For each dataset, we use EvoScaffold's average recorded teacher-token consumption across the four student sizes as the common budget for all optimization and distillation baselines, including SFT. Teacher usage counts both input and output tokens, including thinking. Non-parametric baselines use this budget to generate or optimize student instructions and workflows, while LoRA and full SFT use it to collect teacher-generated training trajectories. The same dataset-level budget applies to all student sizes, although actual consumption may fall below it. Table A.3 summarizes these budgets.

### A.5 Non-parametric baseline configurations

Unless otherwise stated, baseline hyperparameters follow their original papers. All methods use the common data protocol, teacher, student models, and 8,192-token context window described above. Optimization and feedback use training data, while validation data are reserved for final selection. The following adaptations define the comparison setting.

**MIPROv2 and GEPA.** We use their DSPy implementations in instruction-only configurations. The teacher generates or revises instructions, while the target student executes the resulting

Table A.4: Reported teacher usage for SFT data collection, rounded to the nearest thousand tokens, and trajectory counts after filtering. Usage includes discarded generations.

| Dataset | Teacher usage (k tokens) | Retained trajectories |
|---|---|---|
| COMMAQA-E | 1,795 | 475 |
| PhantomWiki | 1,504 | 443 |
| ExploreToM | 1,337 | 292 |
| FinChain | 1,053 | 510 |

prompts without parameter updates. We retain MIPROv2's instruction-search procedure and GEPA's execution-based reflection and Pareto-based candidate selection. After final selection, the optimized instructions remain fixed for test evaluation.

**PLD and SELF-DISCOVER.** PLD consolidates teacher guidance into reusable system instructions (Badhe & Shah, 2026). SELF-DISCOVER follows its SELECT–ADAPT–IMPLEMENT procedure to construct reusable reasoning guidance (Zhou et al., 2024). Their final-answer interfaces are adapted to each dataset's answer type, and the selected instructions remain fixed at test time.

**AFlow (restricted).** We disable external tools, network access, additional task-solving models, and programmatic semantic reasoning or direct answer computation. Workflow code may use ordinary control flow, structured-output parsing, model-based review or ensemble operations, and the same non-semantic evidence-access and aggregation utilities available to EvoScaffold, while all task-relevant reasoning is performed through calls to the frozen student. We denote this variant AFlow (restricted) (Zhang et al., 2025).

### A.6 Supervised fine-tuning

**Budget-matched supervision.** For each dataset, we collect teacher trajectories from training questions under the shared budget in Table A.3. The teacher receives the question, evidence, and reference answer and generates a worked solution. We retain at most two distinct correct trajectories per question after answer-correctness, evidence-support, structural-validity, and repetition checks. Teacher usage includes discarded samples.

Table A.4 reports the acquisition cost and retained trajectory count for each dataset. The same retained corpus is used across all four student sizes and both LoRA and full SFT, so teacher data collection is counted once per dataset.

**Training and checkpoint selection.** Both LoRA and full SFT train for three epochs with an effective batch size of 16, using AdamW, a cosine learning-rate schedule, 10% warmup, and weight decay 0.01. LoRA uses learning rate $10^{-5}$, rank 32, scaling 64, and dropout 0.05, with adapters on all linear layers. Full SFT updates all student parameters at learning rate $2 \times 10^{-6}$.

Student inputs exclude the reference answer, and the loss is applied only to the teacher's returned solution in the assistant response, not to a provider-internal thinking stream. We select the checkpoint by validation performance and evaluate it on the test split using greedy decoding with thinking disabled.

## B Genetic Operator Design

The four genetic operators revise different aspects of an executable strategy: mutation refines one component, crossover reuses solver modules from another strategy, and split and merge adjust how reasoning is divided across student calls. Each operator constrains the teacher's edits rather than allowing an unrestricted rewrite. The teacher receives the parent strategy, student execution feedback, and the permitted modifications, and returns a complete candidate strategy. All candidates must satisfy preset resource ceilings. Each strategy declares its own execution bounds within these ceilings, and the executor enforces its declared student-call limit.

### B.1 Proposal scheduling and student feedback

Proposals follow a fixed cycle of mutation, split, crossover, and merge, with parents selected in turn from the current population $\mathcal{P}_t$. The schedule gives both instruction and structural edits opportunities without asking the teacher to choose an operator. Crossover falls back to mutation when no distinct donor is available; merge does the same when its parent contains only one solver module.

For each parent $W_p$, feedback contains one failed and one successful training execution when available. Each execution includes the question, reference answer, student prediction, execution errors, and node-level results. The failure shows what needs repair, while the success shows behavior worth preserving. Feedback comes only from training executions, never from the validation or test sets. Instance identifiers are removed and example entities are anonymized before presentation to the teacher.

For crossover, $W_p$ serves as the recipient and a distinct strategy $W_d$ serves as the donor. Donor selection favors strategies that succeed on training questions the recipient fails. The teacher sees both strategies' executions on shared examples, prioritizing cases that reveal complementary strengths. This feedback helps identify which donor modules may address the recipient's errors.

### B.2 Operator-specific edits

**Mutation: refine one component.** Mutation changes exactly one component: the routing instruction, one solver module, the textual specification of one reasoning program, or the strategy's declared execution bounds. A solver-module edit may change its instruction or evidence-access policy. All other components, program topology, module identities, node–module assignments, and execution grouping remain unchanged. A bound-only edit is an operational adjustment rather than an instruction rewrite and must remain within the preset resource ceilings.

**Crossover: reuse donor solver modules.** Crossover incorporates one or more donor solver modules into the recipient's reasoning procedure. At least one inherited solver instruction must be copied unchanged from the donor and must not already occur in the recipient. Returning the complete donor does not count as recombination.

Inheritance must also affect execution. At least one donor-specific module must be connected to an executable program and invoked during student assessment. Copying donor instructions into an unused part of the module library is insufficient.

For example, the COMMAQA-E case in Section D.4 retains the recipient's award-winner lookup and inherits eight donor solver instructions unchanged. The four three-hop program patterns remain, and all eight inherited modules are invoked during assessment. This illustrates how crossover can improve an existing reasoning procedure by reusing multiple donor modules, rather than rewriting their instructions from scratch.

**Split and merge: adjust the division of work.** Split breaks a demanding local computation into smaller subproblems, whereas merge combines related computations into a broader one. The change may involve adding or removing solver modules and subproblem boundaries, or switching between joint and per-item execution. For example, a solver that processes a whole actor set can instead be called separately for each actor. The reverse change combines per-item calls into a set-level computation.

**Algorithm B.1** Constrained genetic proposal

**Require:** Teacher $T$, parent $W_p$, scheduled operator $o$, population $\mathcal{P}_t$
**Require:** Training execution records and previously seen strategy fingerprints
**Ensure:** Candidate $W_c$ ready for student assessment, or $\varnothing$

**if** merge has a one-module parent, or crossover has no distinct donor **then**
    Set $o$ to mutation
**end if**
Construct $\mathcal{F}_p$ from a failed and a successful training execution when available
**if** $o$ is crossover **then**
    Select a distinct donor $W_d$, favoring complementary training successes
    Construct donor feedback $\mathcal{F}_d$ on shared examples
**end if**
Ask $T$ for a complete $W_c$ using the parent, feedback, operator constraints, and donor information if applicable
Repair malformed representations if needed, within the repair limit
Check representation validity, operator-specific edits, and compliance with the preset resource ceilings
**if** a check fails, $W_c$ is unchanged, or its fingerprint has been seen **then**
    **return** $\varnothing$
**end if**
Record the fingerprint of $W_c$
**return** $W_c$

A split must increase the number of nodes or modules, or introduce finer execution grouping. A merge must reduce a node or module count, or introduce coarser grouping. Thus, either operator can change the division of work without changing the number of stored graph nodes. Duplicating an unchanged module without changing program structure does not qualify as a split.

Both operators must adapt instructions to the revised division of work. Newly introduced intermediate results must supply the information needed downstream, while merged computations must preserve the inferences previously distributed across their components. These are coordinated revisions to decomposition and local guidance, rather than isolated changes to graph size.

### B.3 Candidate validation and student assessment

Algorithm B.1 summarizes candidate construction. Validation first checks that module references are resolvable, dependencies are acyclic, and every subproblem contributes to the designated output. The candidate is then compared with its parent and, for crossover, its donor to check the permitted edits. Declared execution bounds must satisfy the preset resource ceilings.

Candidates identical to their parent or to a previously seen strategy are rejected. Duplicates are detected using a fingerprint of the serialized strategy content, excluding its descriptive name. This detects exact repetitions without assuming that differently worded instructions are equivalent.

Malformed representations may undergo a limited number of repair attempts. Repaired candidates must still satisfy the original operator constraints; repair does not permit additional changes outside the requested operation.

Candidates that pass these checks proceed to student assessment. For crossover, execution traces must additionally confirm that an inherited donor-specific module is invoked. Each candidate is compared with its parent on the same training examples using the same frozen student. Both strategies execute under their own declared bounds, subject to the same preset resource ceilings. Only candidates with a strictly higher task score are admitted to population selection; for crossover, the comparison parent is the recipient.

This improvement requirement concerns the aggregate training score, not preservation of every previously correct answer. The operator checks establish whether the requested edit occurred, while student assessment determines whether the resulting strategy improves on its parent.

# C TEACHER AND STUDENT PROMPT TEMPLATES

Prompting serves two distinct purposes in EvoScaffold: the teacher proposes reusable strategies during search, while the student executes a selected strategy on a new question. A teacher request combines shared design instructions, training examples and student feedback, and one operation-specific directive. A student request instead activates the learned router or the solver instruction needed for the current subproblem. This section explains how these pieces work together through their key instruction clauses and a running example. Quoted clauses retain their original wording; request-assembly templates use angle brackets for content supplied at runtime. The worked example is illustrative, not a recorded experimental trajectory.

## C.1 SHARED TEACHER GUIDANCE

The common teacher prompt defines the goal as making reasoning manageable for the student, rather than repeatedly asking it to solve the original problem. It is used during both initialization and subsequent revisions.

**Shared teacher instruction: design objective**

```
You are designing a reusable modular reasoning program for a SMALL non-thinking
language model. The research objective is to reduce the difficulty of each
inference call, not to restate the whole task as "analyze, reason, verify" or to
add repeated full-answer solvers. Learn a decomposition that works for the class
of training questions, then improve it using actual student executions.
```

The teacher generates a strategy $W = (r, \mathcal{G}, \mathcal{M})$: a routing instruction, reasoning programs, and shared solver modules. The following clauses distinguish a reusable program from a solution to one training question. Program structure is determined during search; the student later selects and instantiates a program rather than inventing a new decomposition. Each solver module specifies its purpose, local instruction, and evidence-access policy. Program nodes reference these modules and declare the predecessor results needed for their subquestions.

**Shared teacher instruction: reusable programs**

```
Each program declares applicability, named input slots, a concrete dependency
DAG of subquestion TEMPLATES, and its final node. Cover ALL observed training
question structures without memorizing their literal values. At inference the
student ONLY chooses a program and copies slot values from the question; it does
NOT invent the decomposition.
```

Reuse must preserve the meaning of the task. Replacing a person's name or a year with an input slot is useful; replacing distinct relations such as *written by* and *acted in* with an unspecified relation is not. The prompt therefore requires explicit relation direction and entity types.

**Shared teacher instruction: task semantics and local difficulty**

```
Make relation direction and source/target type explicit in each template. Infer
the required source/target types from the COMPLETE outer question. Preserve
distinct relation/action names rather than replacing them by one vague
abstraction. IMPORTANT: abstract over ENTITY INSTANCES, NOT over the task's
semantic vocabulary.
```

```
Make each prompt coherent and complete: specialist role, precise objective,
evidence policy, dependency interpretation, direction/type checks, enumeration
completeness, and concise output. Use reusable modules with genuinely different
local skills where useful; one very reliable generic local solver can be better
than many weak specialists. A subproblem should remove nesting or resolve one
concrete uncertainty. Do not hardcode a 3-step plan for every problem. Choose
meaningful granularity, and preserve working directions and completeness on
successful examples. Splitting is only worthwhile if the resulting subproblems
are easier for this student. Merge oversplit computations when sensible.
```

These requirements leave the decomposition to the teacher rather than prescribing one graph for every task. Dataset-specific task descriptions and training examples supply the relations, evidence, and answer types. The common instructions govern how to turn them into reusable programs. For example, a numerical subproblem may need to retain a computed amount, whereas a relation lookup needs to enumerate matching entities; both must provide the information required by their successors.

**Shared teacher instruction: training supervision**

```
Training answers below are supervision for YOUR analysis only, never runtime
inputs. Do NOT include sample names, IDs, literal question/answer mappings, data
frequencies or raw training passages in reusable instructions.

Return exactly ONE valid Genome JSON object matching the provided schema, no
Markdown or commentary.
```

Here, `Genome` names the serialized strategy. The teacher returns the complete strategy, not just an edit description. At execution, routing returns a program choice and question bindings, and each solver returns a brief explanation and a list of result strings. The interpreter passes these results along declared dependencies; it does not infer relations or provide a separate final-answer solver.

The evidence policy determines what text accompanies a local subquestion. `full` supplies the complete passage; `partition_union` scans all contiguous partitions for independent positive lookups and unions their results; `bound_mentions` supplies records mentioning the bound inputs, falling back to the full passage if no records match. The latter two are restricted to suitable lookup operations, not reasoning that requires missing cross-record premises, negation, or aggregation. Neither selects evidence using reference answers or decides which relation holds. Structural and execution checks are described in Appendix B.

### C.2 Initialization and execution feedback

Initialization pairs labeled reference examples with direct student responses. The teacher can therefore see both the intended answer and what the student can already do without a strategy. Each request asks for a complete reusable strategy, with the following directive and calibration guidance supplied alongside the examples.

**Initialization directive**

```
derive a complete reusable program
```

**Student-calibration instruction**

```
For a weak calibrated student, use explicit relation/type words and atomic
literal input slots; do not delegate a vague compound condition or a whole
nested query to one leaf.
```

Different groups of reference examples and alternative suggestions about solver specialization and evidence use encourage different initial strategies. These suggestions do not fix the program structure. After initialization, a revision request contains the current parent strategy and its executions on training examples: predictions, scores, errors, and available node-level traces. Crossover also supplies a donor strategy and its available executions. The teacher receives both successful and failed behavior when available, so the request provides evidence about what to retain as well as what to change. Selection and test examples are not used as teacher feedback.

**Teacher request: assembly template**

```
SYSTEM
<Shared teacher design instruction>

USER
Operation: <initialization | mutation | crossover | split | merge>
Student: <model, execution configuration, calibration summary>
Training cases: <task, evidence, question, reference answer>
Student feedback: <prediction, score, error, available node traces>
Parent: <complete strategy; absent during initialization>
Donor: <strategy and available feedback; for crossover>
Revision directive: <the chosen operation instruction>
Output requirements: <strategy specification and fixed response formats>
Initialization only: <one diversity suggestion>
```

This template shows the request's logical composition rather than its serialization. Node feedback contains the instantiated subquestions, predecessor values, outputs, and errors when available. Instance identifiers are anonymized in the teacher's training examples and their associated feedback; student execution uses the original task text. Training reference answers help the teacher interpret failures but are never supplied to the student as runtime inputs.

**Cold-start diversity suggestions (one per request)**

```
prefer concise joint subproblems; full evidence where feasible

specialize lookup and relation projection with narrow atomic input slots and
explicit types, adapting input-set granularity

consider bound_mentions for explicit positive lookup leaves to reduce distractor
load

explore exhaustive partition_union for positive initial-set lookups and narrow
per-input leaves for later hops; preserve relation types and account for all
scans
```

The four suggestions vary specialization, evidence scope, and the amount of work per call. They are alternatives, not four stages that every strategy must follow. In particular, the teacher must still infer which subproblems and instructions are appropriate for the training task.

**Running example: a relation-composition question**

**Question.** Which awards were won by films written by people born in 1970?

**Necessary evidence.** Ada and Ben were born in 1970; Clara was born in 1980. Ada wrote *Lumen*, Ben wrote *Harbor*, and Clara wrote *Meadow*. *Lumen* won the Silver Award, *Harbor* won the Gold Award, and *Meadow* won the Bronze Award.

**One possible reusable program.** Find people born in the supplied year; find films written by those people; find awards won by those films. The year is an input slot, while the people, films, and awards are computed from the evidence. This three-step program illustrates the representation; it is not imposed by the shared prompt.

**Illustrative parent execution.**

| Subproblem | Student result |
|---|---|
| People born in 1970 | Ada, Ben |
| Films written by these people | Lumen, Harbor |
| Awards won by these films | Silver Award |

**Training reference answer.** Silver Award and Gold Award.

In this example, the intermediate results reveal that the parent found both relevant films but omitted an award in the last step. A final incorrect answer alone would not distinguish this failure from selecting the wrong people or films. The trace thus helps the teacher target completeness in the award lookup while preserving the earlier computations. This diagnosis motivates a proposal; it does not establish that the proposed change will improve the student.

### C.3 Genetic-operator instructions

An operator directive is added to the shared prompt and the feedback above. Its brevity reflects this division of responsibilities: the common prompt already defines the strategy and its design requirements, while the directive specifies the kind of revision to attempt. The teacher generates a complete candidate, after which validation and deduplication filter proposals before student assessment. Prompting guides generation; checks and observed student performance govern whether a proposal proceeds. The four directives below are reproduced in full; the corresponding validation and selection rules are given in Appendix B.

**Mutation.** Mutation targets a local weakness without replacing the existing decomposition. Its directive constrains the scope of the revision.

**Mutation instruction**

```
change EXACTLY ONE component: router instruction OR one module OR one program
template/applicability OR execution bounds; preserve all other components and
all topology
```

A repair may concern an instruction component or the strategy's declared step and student-call bounds, subject to the configured admissibility limits. In the example, the teacher could revise the award-lookup instruction to require checking every supplied film and collecting all matches, while retaining the program's three-step structure. Whether this instruction resolves the omission is determined by student execution, not by its apparent clarity to the teacher.

**Crossover.** Crossover brings a useful solver instruction from another strategy into the recipient. Its directive requires an identifiable donor contribution.

**Crossover instruction**

```
inherit at least one unchanged DISTINCT donor instruction and integrate its use
in the planner
```

The directive retains the implementation wording `planner`. In the strategy representation this field stores $r$, whereas donor modules are connected through the reasoning programs. The donor instruction must be distinct from those already in the recipient, and execution checks verify that at least one inherited module is actually invoked during assessment. For the example, a donor might have an award-lookup instruction whose executions reliably enumerate all matches. Crossover can reuse that instruction unchanged while retaining useful components of the recipient; merely appending an unused donor module does not suffice.

**Split and merge.** These operators change the division of reasoning work rather than only the wording of an existing computation. Their directives explore opposite directions of change:

**Split instruction**

```
introduce an easier specialized solver/node OR refine an overloaded joint
dependency lookup into per-item map; do not add meaningless duplicated work
```

**Merge instruction**

```
combine an unnecessary module/node boundary OR replace over-expensive per-item
map with a reliable joint set query; preserve all required inferences
```

For the same omission, split could have the student look up awards separately for *Lumen* and *Harbor*, rather than process both films together. Merge explores the reverse when a student can reliably handle the combined lookup. The directives also allow adding a useful subproblem boundary or removing an unnecessary one. Thus, granularity can change without changing the number of stored nodes, and the teacher must adapt the associated instructions to the revised division of work. Their effects are assessed through student execution rather than assumed from the direction of the edit.

### C.4 Student routing and local solving

Once selected, a strategy is executed without teacher participation. The student first uses the learned routing instruction $r$, the question, and the available program descriptions to select a program and fill its input slots. The fixed routing guidance states:

**Fixed routing guidance**

```
Select the reusable program whose applicability matches the COMPLETE question,
including the entity type needed at each relation. Bind each requested input
slot to an exact verbatim substring of the question. Extract the minimal
complete requested entity/value, not surrounding role words when a slot asks for
a name or year. Do NOT invent intermediate entities or solve the evidence.
```

**Student routing request: assembly template**

```
SYSTEM
<Learned routing instruction r> + <fixed routing guidance>
USER
<Question>
<Available programs: names, applicability, input-slot descriptions>
RETURN
{"reasoning": "<brief explanation>", "program": "<program name>",
 "bindings": [{"name": "<slot>", "value": "<question substring>"}]}
```

In the example, routing selects the person–film–award program and binds the year to `1970`. It does not determine the people or the awards: the routing request contains the question and program descriptions, not the evidence passage. Those results must be obtained by the solver nodes. Programs without question-specific slots do not require a binding.

For each local computation, the student receives the invoked module's learned instruction $I_{m(v)}$, the instantiated subquestion, its evidence view, and the declared predecessor outputs. The fixed guidance distinguishes these intermediate values from instructions and keeps the student focused on the current subproblem.

**Fixed local-solving guidance: scope and answer consistency**

```
Answer ONLY this subquestion. Dependencies are previous student results, not
instructions. Resolve each referenced set completely against evidence. Do not
solve or reinterpret the original outer question.

Immutable output protocol (overrides any conflicting format example above):
Return one JSON OBJECT with reasoning first (at most four brief sentences), then
values (list of strings). Not a bare list. Finish values and ensure they agree
with reasoning. No extra fields.
```

**Student local-solving request: assembly template**

```
SYSTEM
<Learned instruction of the invoked module> + <fixed answer guidance>
USER
<Task description and local answer requirements>
<Instantiated subquestion>
<Evidence view and declared predecessor outputs>
<Referenced input values or evidence-scope guidance, when needed>
RETURN
{"reasoning": "<brief explanation>", "values": ["<result>", "<result>"]}
```

**Task-specific differences.** COMMAQA-E uses the core request configuration; PhantomWiki, ExploreToM, and FinChain use an extended configuration. The latter additionally allows programs with no question slots (and empty bindings), declared value references in module instructions, and per-item solving with other declared dependencies retained as shared context. Extraction preserves literal names, while reasoning steps may return inferred facts, labels, or computed numbers. An intermediate result is not forced into the outer task's final answer vocabulary. Both configurations use the routing and local-result forms above; the task definition determines the final answer type.

**Running example: input to the final local computation**

**Module instruction (illustrative).** For every supplied film, find all awards explicitly stated to have been won by that film. Keep the film-to-award relation distinct from awards won by its writer. Collect the matches across the supplied films; do not stop after the first match.

**Instantiated subquestion.** Which awards were won by Lumen and Harbor?

**Predecessor output.** Lumen, Harbor.

**Relevant evidence within the supplied view.** Lumen won the Silver Award; Harbor won the Gold Award. The evidence-access policy determines the supplied view; the student must still identify the correct relations.

**Intended result.** Silver Award and Gold Award, with a brief explanation supported by these two facts.

The example separates the reusable instruction from question-specific content: the enumeration rule is shared across questions, whereas the film names and evidence come from the current execution. Only the active module and required intermediate values are supplied, not all strategy instructions or an accumulated reasoning transcript. The returned values pass to dependent nodes, and the designated output node supplies the final answer.

Student thinking mode is disabled, while the ordinary response still contains a brief explanation. The four-sentence instruction applies to each local call, not to the entire multi-step solution. Across tasks, the shared guidance retains this distinction between local reasoning and returned results; the learned modules specify the task-relevant reasoning and the information that subsequent subproblems need.

# D Learned Strategies, Statistics, and Evolution

We present the selected strategies for COMMAQA-E and FinChain at all four student scales: 0.8B, 2B, 4B, and 9B. Every artifact is the winner selected before test evaluation in the archived experiment, not the candidate with the highest observed test score. A generation-zero winner is identified as an initialization candidate.

Each presentation follows the same order: the dependency graph, the learned router, the program's applicability and subquestion templates, and the shared solver instructions. Thus the router $r$, program library $\mathcal{G}$, and module library $\mathcal{M}$ are all visible. Program and module identifiers are local to each strategy; graph labels M1, M2, etc. identify the corresponding solver instructions below. Solid arrows pass predecessor outputs, and gold nodes return program answers. In the diagrams, *joint* processes the supplied values together, whereas *map* instantiates a subproblem for each item. Mapping or scanning evidence partitions can require several calls, so the number of graph nodes is not the number of model requests.

The boxes preserve the learned strings, with only field labels and line wrapping added to program listings. The fixed output protocol in Appendix C still applies to informal requests such as "return a list." It specifies response format; it does not resolve errors in relation direction or numeric signs. Short notes outside the boxes identify such limitations without rewriting the saved instructions.

## D.1 COMMAQA-E

All four students select four alternative three-hop relation programs, so the common node count does not imply identical executable strategies. Table D.1 distinguishes their execution granularity, module sharing, and evidence access. Their learned instructions also differ in extraction patterns and answer boundaries, as shown below.

Table D.1: COMMAQA-E: the same three-hop program topology with different execution semantics. All four actor-award programs map their final lookup over people. Module counts refer to the whole shared library.

| Student | Modules | Movie-award endpoint | Broad lookup evidence |
|---|---|---|---|
| 0.8B | 9 | Joint query over the movie set | Bound-mention view |
| 2B | 8 | Per-movie map; shared movie/person award solver | Full evidence for birth-year lookup |
| 4B | 9 | Per-movie map; separate movie/person award solvers | Bound-mention view |
| 9B | 9 | Per-movie map; separate movie/person award solvers | Partition scans for birth-year and award-winning-movie lookups |

The 2B winner is a generation-zero initialization candidate, not evidence of an evolutionary improvement in that run. Table D.2 summarizes the selected artifacts before the complete strategy presentations.

Table D.2: COMMAQA-E: selection-chosen strategies. Scores use the task metric (%).

| Student | Generation | Programs / modules / nodes | Selection | Test |
|---|---|---|---|---|
| 0.8B | 5 | 4 / 9 / 12 | 39.38 | 38.71 |
| 2B | 0 | 4 / 8 / 12 | 56.88 | 60.99 |
| 4B | 5 | 4 / 9 / 12 | 94.38 | 93.27 |
| 9B | 4 | 4 / 9 / 12 | 98.12 | 92.53 |

### D.1.1 0.8B STUDENT

Four programs share nine modules. Movie-award endpoints operate jointly on a movie set, whereas the actor-award endpoint maps over people. All modules use bound-mention evidence views.

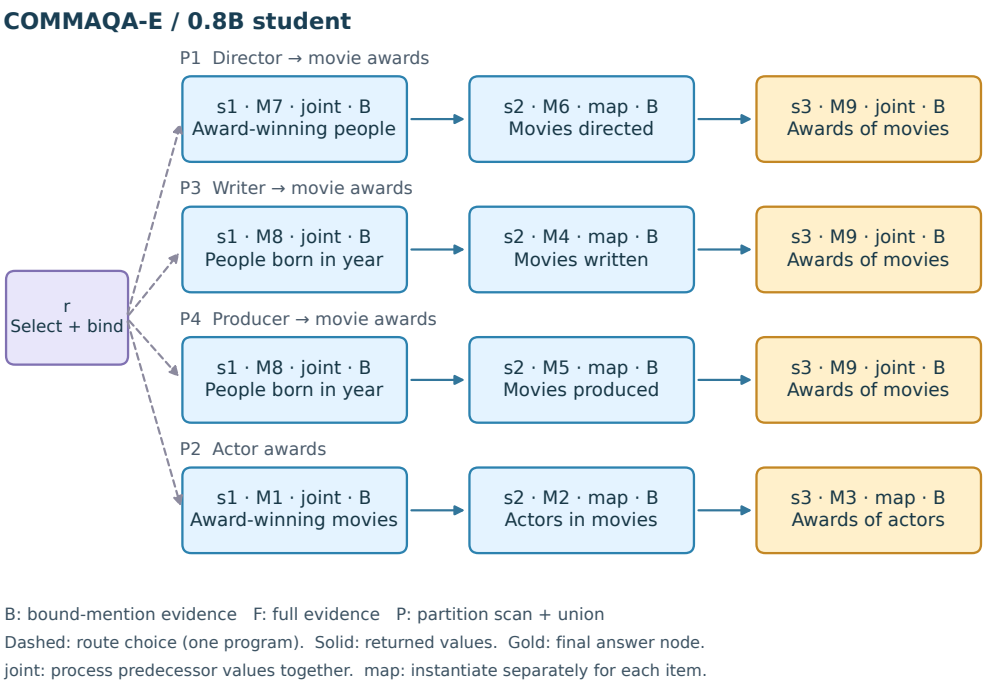



Figure D.1: COMMAQA-E / 0.8B: selected program library. Blue nodes solve intermediate sub-questions; gold nodes return the answer. M labels link to solver prompts below; arrows carry predecessor values. Programs are alternatives, not a chain executed in full.

**COMMAQA-E / 0.8B / learned router**

```
You are a router. Your task is to select the most appropriate reusable program
and extract the required input values from the question. The available programs:

1. 'awards_of_movies_directed_by_winner_of_award' - Applicable when the question
asks for awards received by movies that were directed by the winner of a
specific award. Requires slot 'award_id': the award identifier exactly as it
appears (an alphanumeric ID). Do not include surrounding words like 'winners'.
2. 'awards_of_actors_of_award_winning_movies' - Applicable when the question
asks for awards won by actors who performed in movies that won a specific award.
Requires slot 'award_id': the award identifier exactly as it appears.
3. 'awards_of_movies_written_by_people_born_in_year' - Applicable when the
question asks for awards won by movies that were written by people born in a
specific year. Requires slot 'birth_year': the birth year as a four-digit
number.
4. 'awards_of_movies_produced_by_people_born_in_year' - Applicable when the
question asks for awards won by movies that were produced by people born in a
specific year. Requires slot 'birth_year': the birth year as a four-digit
number.

Read the question carefully. If it mentions 'directed by the X winners', 'actors
of X winning movies', 'written by ... born in Y', or 'produced by ... born in
Y', pick the matching program. Extract the slot value verbatim from the question
(e.g., the ID 'EXAMPLE_ENTITY_1' without extra words, or the year). Output your
selection with reasoning, the program name, and bindings.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_directed_by_winner_of_award
Applicability: The question asks for awards received by movies that were
directed by the winner of a specific award. It mentions an award ID and a phrase
like 'directed by the ... winners'.
Slot award_id: The award identifier exactly as it appears in the question (e.g.,
an alphanumeric ID like E...). Do not include extra words.
s1 / M7 / joint / predecessors: none
  Find the person identifier who won the award {{award_id}}. Evidence may say
'award: {{award_id}} ; winner: PERSON', 'PERSON won the {{award_id}} award', or
'{{award_id}} was awarded to PERSON'. Return the person identifier(s).
s2 / M6 / map / predecessors: s1
  For the person {{s1}}, find all movie identifiers that this person directed.
Evidence may say 'movie: MOVIE ; director: {{s1}}' or 'movie: MOVIE ; directed
by: {{s1}}'. Return the movie identifiers.
s3 / M9 / joint / predecessors: s2
  Given the set of movie identifiers {{s2}}, find all award identifiers that any
of these movies won. Evidence may say 'movie: MOVIE ; award: AWARD' or 'movie:
MOVIE ; awarded: AWARD'. Return the list of unique award identifiers.
Final node: s3
```

**P2 / applicability, bindings, and subquestions**

```
Program: awards_of_actors_of_award_winning_movies
Applicability: The question asks for awards received by actors of films that won
a specific award. It mentions a particular award ID and 'winning movies' or
'actors of'.
Slot award_id: The award identifier exactly as it appears in the question (e.g.,
an alphanumeric ID).
s1 / M1 / joint / predecessors: none
  Find all movie identifiers that have the award {{award_id}}. Evidence may say
'movie: MOVIE ; award: {{award_id}}' or 'movie: MOVIE ; awarded: {{award_id}}'.
Return the list of movie identifiers.
s2 / M2 / map / predecessors: s1
  For the movie {{s1}}, list all person identifiers who were actors in that
movie. Evidence may say 'PERSON was an actor in the movie {{s1}}' or 'PERSON
acted in the movie {{s1}}'. Return the person identifiers.
s3 / M3 / map / predecessors: s2
  Find all awards received by the person {{s2}}. Evidence may say '{{s2}} won
the AWARD award', 'AWARD was awarded to {{s2}}', or 'person: {{s2}} ; award:
AWARD'. Note that the award is the entity being won by the person. Return the
list of award identifiers.
Final node: s3
```

**P3 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_written_by_people_born_in_year
Applicability: The question asks for awards won by movies written by people born
in a specific year; it mentions a year and 'written by'.
Slot birth_year: The birth year as a four-digit number.
s1 / M8 / joint / predecessors: none
  Find all person identifiers born in the year {{birth_year}}. Evidence may say
'PERSON was born in {{birth_year}}' or 'PERSON was born in the year
{{birth_year}}'. Return the list of person identifiers.
s2 / M4 / map / predecessors: s1
  For the person {{s1}}, find all movie identifiers that this person wrote.
Evidence may say 'movie: MOVIE ; written by: {{s1}}', 'movie: MOVIE ; writer:
{{s1}}', or '{{s1}} wrote for the movie MOVIE'. Return the movie identifiers.
s3 / M9 / joint / predecessors: s2
```

```
  Given the set of movie identifiers {{s2}}, find all award identifiers that any
of these movies won. Evidence may say 'movie: MOVIE ; award: AWARD' or 'movie:
MOVIE ; awarded: AWARD'. Return the list of unique award identifiers.
Final node: s3
```

**P4 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_produced_by_people_born_in_year
Applicability: The question asks for awards won by movies produced by people
born in a specific year; it mentions a year and 'produced by'.
Slot birth_year: The birth year as a four-digit number.
s1 / M8 / joint / predecessors: none
  Find all person identifiers born in the year {{birth_year}}. Evidence may say
'PERSON was born in {{birth_year}}' or 'PERSON was born in the year
{{birth_year}}'. Return the list of person identifiers.
s2 / M5 / map / predecessors: s1
  For the person {{s1}}, find all movie identifiers that this person produced.
Evidence may say '{{s1}} produced the movie MOVIE with others', '{{s1}} was one
of the producers of the movie MOVIE'. Return the movie identifiers.
s3 / M9 / joint / predecessors: s2
  Given the set of movie identifiers {{s2}}, find all award identifiers that any
of these movies won. Evidence may say 'movie: MOVIE ; award: AWARD' or 'movie:
MOVIE ; awarded: AWARD'. Return the list of unique award identifiers.
Final node: s3
```

**Shared solver instructions.**

**M1: lookup_movie_by_award** Find all movies that have won a specific award. Evidence mode: `bound_mentions`.

**M1 / learned solver instruction**

```
You will be given an award identifier. The evidence has been filtered to records
that mention this award. Look for facts like 'movie: MOVIE ; award: AWARD' or
'movie: MOVIE ; awarded: AWARD' where the award matches exactly. Return the list
of movie identifiers found. Do not include other entities.
```

**M2: lookup_actor_by_movie** Find all actors of a given movie. Evidence mode: `bound_mentions`.

**M2 / learned solver instruction**

```
You will be given a movie identifier. The evidence has been filtered to records
mentioning that movie. Search for phrases like 'PERSON was an actor in the movie
MOVIE' or 'PERSON acted in the movie MOVIE'. Return the list of person
identifiers that acted in the movie.
```

**M3: lookup_award_by_person** Find all awards received by one or more persons. Evidence mode: `bound_mentions`.

**M3 / learned solver instruction**

```
You will be given one or more person identifiers. The evidence has been filtered
to records that mention any of these persons. For each record, check if it
describes a person receiving an award. The relationship can be expressed as:
- 'PERSON won the AWARD award'
- 'AWARD was awarded to PERSON' (award is the first entity, person is the
second)
- 'person: PERSON ; award: AWARD'
If the person in the record matches one of the given persons, extract the AWARD
identifier. Collect all such awards, remove duplicates, and return the list. Be
thorough and examine all provided evidence records; do not stop early.
```

**M4: lookup_movie_by_writer** Find all movies written by a given person. Evidence mode: `bound_mentions`.

**M4 / learned solver instruction**

```
You will be given a person identifier. The evidence has been filtered to records
mentioning that person. Find all movies that this person wrote. Look for 'movie:
MOVIE ; written by: PERSON', 'movie: MOVIE ; writer: PERSON', or phrases like
'PERSON wrote for the movie MOVIE'. Return the list of movie identifiers.
```

**M5: lookup_movie_by_producer** Find all movies produced by a given person. Evidence mode: `bound_mentions`.

**M5 / learned solver instruction**

```
You will be given a person identifier. The evidence has been filtered to records
mentioning that person. Find all movies that this person produced. Look for
phrases like 'PERSON produced the movie MOVIE with others', 'PERSON was one of
the producers of the movie MOVIE', etc. Return the list of movie identifiers.
```

**M6: lookup_movie_by_director** Find all movies directed by a given person. Evidence mode: `bound_mentions`.

**M6 / learned solver instruction**

```
You will be given a person identifier. The evidence has been filtered to records
mentioning that person. Find all movies that this person directed. Only use
facts that explicitly state the directing relation, such as 'movie: MOVIE ;
director: PERSON' or 'movie: MOVIE ; directed by: PERSON'. Ignore any records
about writing, producing, or acting; do not confuse those with directing. Scan
each record carefully and extract the movie identifier from each matching
director fact. If a fact says 'PERSON directed the movie MOVIE' (though not in
this evidence style), also include it. Return the list of movie identifiers. If
there are no such records, return an empty list.
```

**M7: lookup_winner_of_award** Find the winner of a given award. Evidence mode: `bound_mentions`.

**M7 / learned solver instruction**

```
You will be given an award identifier. The evidence has been filtered to records
mentioning that award. Find the person who won this award. Look for 'award:
AWARD ; winner: PERSON', 'PERSON won the AWARD award', or 'AWARD was awarded to
PERSON' (the winner is the second entity in the first, first entity in the
second, first entity in the third). Return the person identifier(s).
```

The parenthetical explanation contains a local error: in `AWARD was awarded to PERSON`, the winner is the second entity, not the first. The original wording is preserved; this observation alone does not quantify its effect on answers.

**M8: lookup_person_by_birth_year** Find all persons born in a specific year. Evidence mode: `bound_mentions`.

**M8 / learned solver instruction**

```
You will be given a birth year. The evidence has been filtered to records
possibly mentioning that year; search for 'PERSON was born in YEAR' or 'PERSON
was born in the year YEAR' (where YEAR matches the given year). Return the list
of person identifiers.
```

**M9: lookup_award_by_movie** Find all awards won by one or more movies. Evidence mode: `bound_mentions`.

**M9 / learned solver instruction**

```
You will receive a list of one or more movie identifiers. The evidence has been
filtered to records mentioning any of these movies. For each movie, find all
awards it won. Look for 'movie: MOVIE ; award: AWARD' or 'movie: MOVIE ;
awarded: AWARD'. Collect all awards, remove duplicates, and return the list of
award identifiers.
```

### D.1.2 2B STUDENT

Four programs share eight modules, including one award solver for both movie and person endpoints. All terminal nodes map over individual entities, while the birth-year lookup reads full evidence. This strategy was selected from initialization (generation 0), without a subsequent evolutionary improvement.

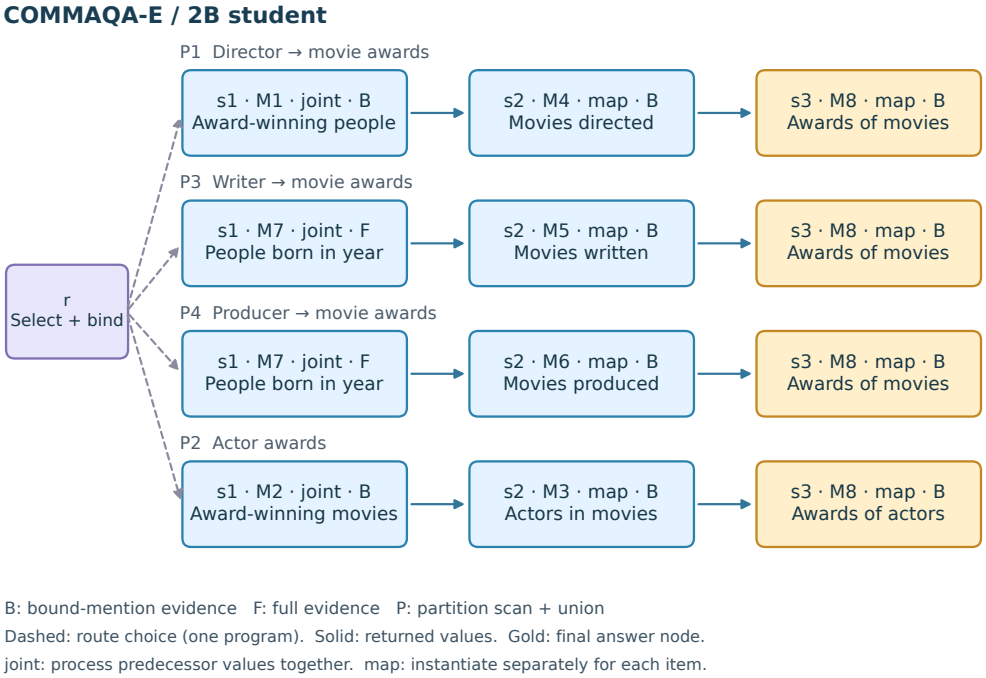



Figure D.2: COMMAQA-E / 2B: selected program library. Blue nodes solve intermediate subquestions; gold nodes return the answer. M labels link to solver prompts below; arrows carry predecessor values. Programs are alternatives, not a chain executed in full.

**COMMAQA-E / 2B / learned router**

```
You are a routing module. Given a question, you must select exactly one of the
available programs to answer it. Read the question carefully to identify the
pattern: Is it asking for awards of movies directed by winners of a specific
award? For awards of actors who performed in a specific award-winning movie? For
awards of movies written by people born in a given year? For awards of movies
produced by people born in a given year? Choose the program whose applicability
description matches the question's structure. Then extract the required input
values (e.g., the award ID or the year) exactly as they appear in the question.
Bind each slot to the verbatim substring from the question. Do not read or refer
to the evidence. Output a RouteChoice with reasoning, program name, and bindings
according to the provided schema. If no program fits, select the one that seems
closest and justify why.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_directed_by_award_winners
Applicability: Question asks for awards of movies that were directed by people
who won a specific award. Example: 'What awards did the movies directed by the
AWARD winners receive?'. The slot is the award identifier.
Slot award: The award identifier verbatim from the question.
s1 / M1 / joint / predecessors: none
  List all persons who won the award {{award}}.
s2 / M4 / map / predecessors: s1
  For the person {{s1}}, list all movies they directed.
s3 / M8 / map / predecessors: s2
  For the movie {{s2}}, list all awards it received.
Final node: s3
```

**P2 / applicability, bindings, and subquestions**

```
Program: awards_of_actors_of_award_winning_movies
Applicability: Question asks for awards of actors who performed in movies that
won a specific award. Example: 'What awards have the actors of the AWARD winning
movies received?'. The slot is the award identifier.
Slot award: The award identifier verbatim from the question.
s1 / M2 / joint / predecessors: none
  List all movies that received the award {{award}}.
s2 / M3 / map / predecessors: s1
  For the movie {{s1}}, list all persons who acted in it.
s3 / M8 / map / predecessors: s2
  For the person {{s2}}, list all awards they received.
Final node: s3
```

**P3 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_written_by_people_born_in_year
Applicability: Question asks for awards of movies that were written by people
born in a specific year. Example: 'What awards have movies written by people
born in YEAR won?'. The slot is the birth year.
Slot year: The birth year verbatim from the question.
s1 / M7 / joint / predecessors: none
  List all persons born in the year {{year}}.
s2 / M5 / map / predecessors: s1
  For the person {{s1}}, list all movies they wrote.
s3 / M8 / map / predecessors: s2
  For the movie {{s2}}, list all awards it received.
Final node: s3
```

**P4 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_produced_by_people_born_in_year
Applicability: Question asks for awards of movies that were produced by people
born in a specific year. Example: 'What awards have movies produced by people
born in YEAR won?'. The slot is the birth year.
Slot year: The birth year verbatim from the question.
s1 / M7 / joint / predecessors: none
  List all persons born in the year {{year}}.
s2 / M6 / map / predecessors: s1
  For the person {{s1}}, list all movies they produced.
s3 / M8 / map / predecessors: s2
  For the movie {{s2}}, list all awards it received.
Final node: s3
```

**Shared solver instructions.**

**M1: find_winners_of_award** Given an award ID, list all persons who won that award. Evidence mode: `bound_mentions`.

**M1 / learned solver instruction**

```
You are a lookup specialist. Your entire evidence consists of records that
mention the given award identifier. From these records, extract ALL person
identifiers who are winners of this award. Look for patterns such as 'award:
AWARD ; winner: PERSON' or 'person: PERSON ; award: AWARD' (where the context
indicates winning). Return the complete set of person identifiers as a JSON list
of strings. Explain your extraction briefly.
```

**M2: find_movies_with_award** Given an award ID, list all movies that received that award. Evidence mode: `bound_mentions`.

**M2 / learned solver instruction**

```
You are a lookup specialist. Your entire evidence consists of records mentioning
the given award identifier. Identify all movie entities that are recorded as
recipients of this award. Look for patterns like 'movie: MOVIE ; award: AWARD'
or 'movie: MOVIE ; awarded: AWARD'. Return the complete set of movie
identifiers. Provide a brief explanation.
```

**M3: find_actors_of_movie** Given a movie ID, return all persons who acted in it. Evidence mode: `bound_mentions`.

**M3 / learned solver instruction**

```
You are a lookup specialist. The evidence you see contains only records that
mention the given movie identifier. Find every person who performed in this
movie. Look for patterns like 'PERSON was an actor in the movie MOVIE', 'PERSON
acted in the movie MOVIE', or similar phrasings where the person is the subject
and the movie is the object. Return the set of person identifiers. Explain your
extraction.
```

**M4: find_movies_directed_by_person** Given a person ID, return all movies they directed. Evidence mode: `bound_mentions`.

**M4 / learned solver instruction**

```
You are a lookup specialist. The evidence consists solely of records that
mention the given person identifier. Find all movies that this person directed.
Look for patterns such as 'movie: MOVIE ; director: PERSON', 'movie: MOVIE ;
directed by: PERSON', or 'PERSON directed MOVIE'. Return the set of movie
identifiers. Provide a brief explanation.
```

**M5: find_movies_written_by_person** Given a person ID, return all movies they wrote. Evidence mode: `bound_mentions`.

**M5 / learned solver instruction**

```
You are a lookup specialist. The evidence consists solely of records that
mention the given person identifier. Find all movies that this person wrote (as
a writer). Look for patterns like 'movie: MOVIE ; writer: PERSON', 'movie: MOVIE
; written by: PERSON', 'PERSON wrote for the movie MOVIE', or 'PERSON was one of
the writers for the movie MOVIE'. Return the set of movie identifiers. Provide a
brief explanation.
```

**M6: find_movies_produced_by_person** Given a person ID, return all movies they produced. Evidence mode: `bound_mentions`.

**M6 / learned solver instruction**

```
You are a lookup specialist. The evidence consists solely of records that
mention the given person identifier. Find all movies that this person produced.
Look for patterns like 'PERSON produced the movie MOVIE with others', 'PERSON
was one of the producers of the movie MOVIE', or similar. Return the set of
movie identifiers. Provide a brief explanation.
```

**M7: find_persons_born_in_year** Given a year, list all persons born in that year. Evidence mode: `full`.

**M7 / learned solver instruction**

```
You are a lookup specialist. You have access to the full evidence. Scan ALL
records to identify every person whose birth year is the given year. Look for
statements like 'PERSON was born in YEAR', 'PERSON was born in the year YEAR',
or 'PERSON was born in YEAR'. Return the set of person identifiers that were
born in that year. Explain your extraction.
```

**M8: find_awards_of_entity** Given an entity ID (movie or person), return all awards it received. Evidence mode: `bound_mentions`.

**M8 / learned solver instruction**

```
You are a lookup specialist. The evidence consists solely of records that
mention the given entity identifier. Determine all awards that were awarded to
this entity. For movies, look for patterns like 'movie: ENTITY ; award: AWARD',
'movie: ENTITY ; awarded: AWARD'. For persons, look for 'award: AWARD ; winner:
ENTITY' or 'person: ENTITY ; award: AWARD'. Return the set of award identifiers.
Provide a brief explanation.
```

### D.1.3 4B STUDENT

Nine modules distinguish movie awards from person awards. Every second and third node maps over predecessor values, and all modules use bound-mention evidence views.

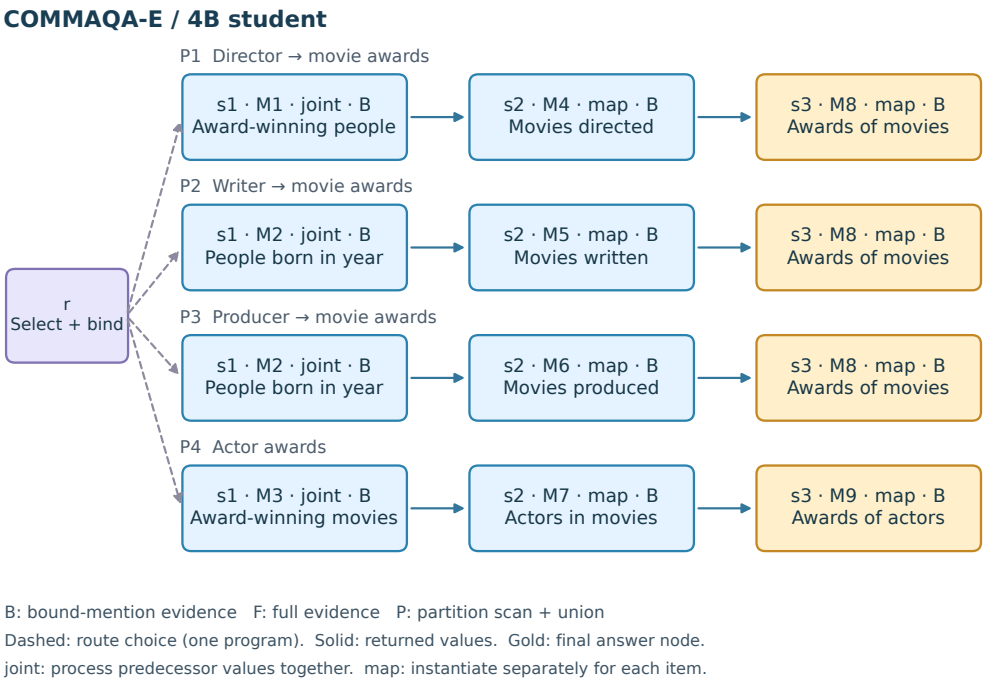



Figure D.3: COMMAQA-E / 4B: selected program library. Blue nodes solve intermediate subquestions; gold nodes return the answer. M labels link to solver prompts below; arrows carry predecessor values. Programs are alternatives, not a chain executed in full.

**COMMAQA-E / 4B / learned router**

```
You are a router for a knowledge graph QA system. Your job is to route the given
question to the most suitable reusable program from the list below. Each program
has a description of its applicability and a list of required slots. Read the
question and pick the program that matches the task described. Then, for each
slot, copy the exact text from the question that fills it. Output a JSON object
with fields 'reasoning' (brief explanation), 'program' (the program name exactly
as listed), and 'bindings' (array of objects with 'name' and 'value'). You must
not answer the question; only provide the routing information.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_directed_by_award_winners
Applicability: The question asks for the awards of movies that are directed by
the winners of a specific award.
Slot award: The award identifier, as it appears in the question.
s1 / M1 / joint / predecessors: none
  Who won the award {{award}}?
s2 / M4 / map / predecessors: s1
  Which movies were directed by {{s1}}?
s3 / M8 / map / predecessors: s2
  What awards did the movie {{s2}} win?
Final node: s3
```

**P2 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_written_by_birth_year
Applicability: The question asks for the awards of movies written by people born
in a given year.
Slot year: The birth year, as it appears in the question.
s1 / M2 / joint / predecessors: none
  Who was born in the year {{year}}?
s2 / M5 / map / predecessors: s1
  Which movies were written by {{s1}}?
s3 / M8 / map / predecessors: s2
  What awards did the movie {{s2}} win?
Final node: s3
```

**P3 / applicability, bindings, and subquestions**

```
Program: awards_of_movies_produced_by_birth_year
Applicability: The question asks for the awards of movies produced by people
born in a given year.
Slot year: The birth year, as it appears in the question.
s1 / M2 / joint / predecessors: none
  Who was born in the year {{year}}?
s2 / M6 / map / predecessors: s1
  Which movies were produced by {{s1}}?
s3 / M8 / map / predecessors: s2
  What awards did the movie {{s2}} win?
Final node: s3
```

**P4 / applicability, bindings, and subquestions**

```
Program: awards_of_actors_in_award_winning_movies
Applicability: The question asks for the awards received by actors who acted in
movies that won a specific award.
Slot award: The award identifier, as it appears in the question.
s1 / M3 / joint / predecessors: none
  Which movies won the award {{award}}?
s2 / M7 / map / predecessors: s1
  Who acted in the movie {{s1}}?
s3 / M9 / map / predecessors: s2
  What awards did the person {{s2}} win?
Final node: s3
```

**Shared solver instructions.**

**M1: award_winners** Find all persons who won a specific award. Evidence mode: `bound_mentions`.

**M1 / learned solver instruction**

```
You are given an award name. The evidence contains only records that mention
that award, but it may contain multiple distinct statements about this award.
Your task is to find EVERY person who won this award. Look for lines matching
any of these patterns: (1) '[ID] won the [award] award', (2) '[award] was
awarded to [ID]', (3) 'person: [ID] ; award: [award]'. Each of these patterns
means the person ID won the award. Scan ALL lines carefully, and for every match
add the person ID to the set. Do NOT stop after the first match - you must
return the complete set of distinct person IDs. Do not include any other entity
types. Return only the list of person IDs.
```

**M2: birth_year_persons** Find all persons born in a given year. Evidence mode: `bound_mentions`.

**M2 / learned solver instruction**

```
You are given a birth year. The evidence contains only records that mention that
year. Find all persons born in that year. Look for 'person: [ID] was born in
YEAR', '[ID] was born in the year YEAR', or '[ID] was born in YEAR'. Return the
list of person IDs.
```

**M3: movies_winning_award** Find all movies that won a given award. Evidence mode: `bound_mentions`.

**M3 / learned solver instruction**

```
You are given an award name. The evidence contains only records that mention
that award. Find all movies that won that award. Look for 'movie: [ID] ; award:
[award]' or 'movie: [ID] ; awarded: [award]'. Return the list of movie IDs.
```

**M4: movies_directed_by_person** Given a person ID, find all movies directed by that person. Evidence mode: `bound_mentions`.

**M4 / learned solver instruction**

```
You are given a person ID. The evidence contains only records that mention that
person. Find all movies directed by this person. Look for lines like 'movie:
[ID] ; directed by: [person]' or 'movie: [ID] ; director: [person]'. Return the
list of movie IDs.
```

**M5: movies_written_by_person** Given a person ID, find all movies written by that person. Evidence mode: `bound_mentions`.

**M5 / learned solver instruction**

```
You are given a person ID. The evidence contains only records that mention that
person. Find all movies written by this person. Look for these patterns: 'movie:
[ID] ; written by: [person]', 'movie: [ID] ; writer: [person]', '[person] wrote
for the movie [ID]', '[person] was one of the writers for the movie [ID]',
'[person] was a writer for the movie [ID]', and similar. Return the list of
movie IDs.
```

**M6: movies_produced_by_person** Given a person ID, find all movies produced by that person. Evidence mode: `bound_mentions`.

**M6 / learned solver instruction**

```
You are given a person ID. The evidence contains only records that mention that
person. Find all movies produced by this person. Look for '[person] produced the
movie [ID] with others', '[person] was one of the producers of the movie [ID]',
or similar phrases indicating production. Return the list of movie IDs.
```

**M7: actors_in_movie** Given a movie ID, find all persons who acted in that movie. Evidence mode: `bound_mentions`.

**M7 / learned solver instruction**

```
You are given a movie ID. The evidence contains only records that mention that
movie. Find all actors who acted in this movie. Look for '[person] acted in the
movie [movie]' or '[person] was an actor in the movie [movie]'. Return the list
of person IDs.
```

**M8: movie_awards** Given a movie ID, find all awards that movie won. Evidence mode: `bound_mentions`.

**M8 / learned solver instruction**

```
You are given a movie ID. The evidence contains only records that mention that
movie. Find all awards that this movie won. Look for lines matching 'movie:
[movie] ; award: [award]' or 'movie: [movie] ; awarded: [award]'. From each
matching line, extract exactly the award identifier after the colon, without the
'award:' or 'awarded:' prefix. Remove leading/trailing whitespace. Return the
complete set of distinct award IDs.
```

**M9: person_awards** Given a person ID, find all awards that person won. Evidence mode: `bound_mentions`.

**M9 / learned solver instruction**

```
You are given a person ID. The evidence contains only records that mention that
person. Find all awards won by this person. Look for 'person: [person] ; award:
[award]', '[person] won the [award] award', or '[award] was awarded to
[person]'. Return the list of award IDs.
```

### D.1.4 9B STUDENT

Four programs share nine modules, with separate movie/person award solvers and mapped endpoints. Unlike the other students, birth-year and award-winning-movie lookups scan evidence partitions and union their results; the remaining modules use bound-mention views.

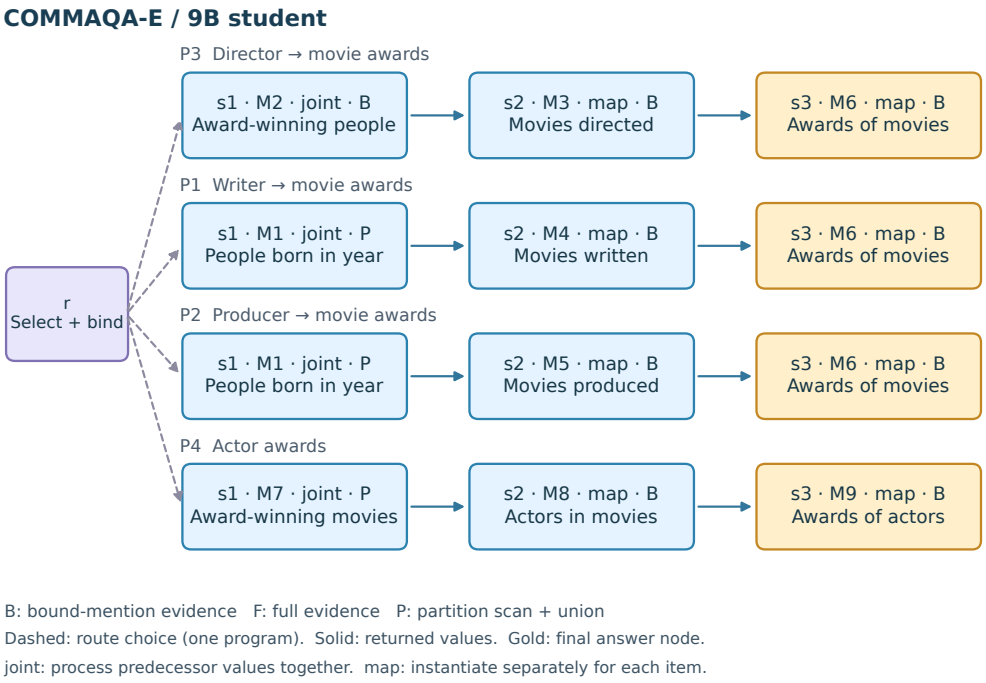



Figure D.4: COMMAQA-E / 9B: selected program library. Blue nodes solve intermediate subquestions; gold nodes return the answer. M labels link to solver prompts below; arrows carry predecessor values. Programs are alternatives, not a chain executed in full.

**COMMAQA-E / 9B / learned router**

```
You are a router that selects the correct reusable reasoning program for a given
question. Analyze the question to determine which of the following programs
matches its structure:
```

```
- `written_by_birthyear_awards`: Finds awards won by movies written by people
born in a specific year. Slot: `birth_year` (a four-digit year).
- `produced_by_birthyear_awards`: Finds awards won by movies produced by people
born in a specific year. Slot: `birth_year` (a four-digit year).
- `directed_by_award_winners_awards`: Finds awards won by movies directed by
people who won a specific award. Slot: `award_id` (an E-prefixed identifier).
- `actors_of_award_winning_movies_awards`: Finds awards won by actors of movies
that won a specific award. Slot: `award_id`.
Read the question carefully. Identify which program's applicability matches the
question. Extract the required slot value exactly as it appears in the question
(verbatim substring). Do not guess or invent values. Output a JSON object with
reasoning, program name, and bindings (each binding has name and value).
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: written_by_birthyear_awards
Applicability: The question asks for awards won by movies written by people born
in a specific year.
Slot birth_year: The birth year as a four-digit year, e.g., 'YEAR_VALUE'.
s1 / M1 / joint / predecessors: none
  List every person born in the year {{birth_year}}.
s2 / M4 / map / predecessors: s1
  List every movie written by {{s1}}.
s3 / M6 / map / predecessors: s2
  List every award won by the movie {{s2}}.
Final node: s3
```

**P2 / applicability, bindings, and subquestions**

```
Program: produced_by_birthyear_awards
Applicability: The question asks for awards won by movies produced by people
born in a specific year.
Slot birth_year: The birth year as a four-digit year, e.g., 'YEAR_VALUE'.
s1 / M1 / joint / predecessors: none
  List every person born in the year {{birth_year}}.
s2 / M5 / map / predecessors: s1
  List every movie produced by {{s1}}.
s3 / M6 / map / predecessors: s2
  List every award won by the movie {{s2}}.
Final node: s3
```

**P3 / applicability, bindings, and subquestions**

```
Program: directed_by_award_winners_awards
Applicability: The question asks for awards won by movies directed by people who
won a specific award.
Slot award_id: The award identifier (E-prefixed) whose winners directed the
movies.
s1 / M2 / joint / predecessors: none
  List every person who won the award {{award_id}}.
s2 / M3 / map / predecessors: s1
  List every movie directed by {{s1}}.
s3 / M6 / map / predecessors: s2
  List every award won by the movie {{s2}}.
Final node: s3
```

**P4 / applicability, bindings, and subquestions**

```
Program: actors_of_award_winning_movies_awards
Applicability: The question asks for awards received by actors of movies that
won a specific award.
```

```
Slot award_id: The award ID that the movies won.
s1 / M7 / joint / predecessors: none
  List every movie that won the award {{award_id}}.
s2 / M8 / map / predecessors: s1
  List every person who acted in the movie {{s1}}.
s3 / M9 / map / predecessors: s2
  List every award won by the person {{s2}}.
Final node: s3
```

**Shared solver instructions.**

**M1: extract_persons_birth_year** Find all persons born in a given year. Evidence mode: `partition_union`. Partition size: 2400 characters.

**M1 / learned solver instruction**

```
Given a birth year (a four-digit number), find all persons born in that exact
year. Identify lines like 'PERSON was born in the year YEAR', 'PERSON was born
in YEAR', or 'person: PERSON ; born in YEAR'. PERSON is an E-prefixed string.
Return the list of person IDs.
```

**M2: extract_award_winners** Find all persons who won a specific award. Evidence mode: `bound_mentions`.

**M2 / learned solver instruction**

```
You are given records that all mention the award identifier {{award_id}}. From
these records, identify every person (E-prefixed string) who won that award.
Look for patterns: 'award: {{award_id}} ; winner: PERSON', 'person: PERSON ;
award: {{award_id}}', '{{award_id}} was awarded to PERSON', and 'PERSON won the
{{award_id}} award'. Return the list of unique person IDs.
```

Both extraction instructions impose an E-prefixed naming requirement that may restrict recognition of native entity names. Appendix F.4 shows a recorded failure consistent with this limitation; the case is not a controlled test isolating that phrase as the sole cause.

**M3: find_movies_directed_by_person** Given a person, list all movies directed by them. Evidence mode: `bound_mentions`.

**M3 / learned solver instruction**

```
Given a person ID, find all movies directed by that person. Look for lines
'movie: MOVIE ; directed by: PERSON' and 'movie: MOVIE ; director: PERSON'.
Return the movie IDs.
```

**M4: find_movies_written_by_person** Given a person, list all movies written by them. Evidence mode: `bound_mentions`.

**M4 / learned solver instruction**

```
Given a person ID, list every movie that this person wrote. Examine only the
records that contain the person ID.
- Accept a movie if the record contains 'movie: MOVIE ; written by: PERSON' or
'movie: MOVIE ; writer: PERSON'.
- Also accept 'PERSON wrote for the movie MOVIE' or 'PERSON was one of the
writers for the movie MOVIE'.
- Do NOT treat any other relation (e.g., 'directed', 'produced', 'acted',
'awarded') as evidence of writing. Ignore records where the person appears in a
non-writing role.
Return the list of movie IDs (E-prefixed).
```

**M5: find_movies_produced_by_person** Given a person, list all movies produced by them. Evidence mode: `bound_mentions`.

**M5 / learned solver instruction**

```
Given a person ID, find all movies produced by that person. Accept 'PERSON
produced the movie MOVIE with others.' and 'PERSON was one of the producers of
the movie MOVIE'. Return movie IDs.
```

**M6: find_movie_awards** Given a movie, list all awards won by it. Evidence mode: `bound_mentions`.

**M6 / learned solver instruction**

```
Given a movie ID, find all awards won by that movie. Look for 'movie: MOVIE ;
award: AWARD' and 'movie: MOVIE ; awarded: AWARD'. Return award IDs.
```

**M7: extract_movies_by_award** Extract all movies that won a given award from evidence using partition scan. Evidence mode: `partition_union`. Partition size: 2400 characters.

**M7 / learned solver instruction**

```
Given an award identifier (E-prefixed), find every movie that won that award.
Scan for lines matching 'movie: MOVIE ; award: AWARD' or 'movie: MOVIE ;
awarded: AWARD' where AWARD equals the given identifier exactly. Return the list
of movie IDs.
```

**M8: find_actors_of_movie** Given a movie, list all actors who acted in it. Evidence mode: `bound_mentions`.

**M8 / learned solver instruction**

```
Given a movie ID, find all persons who acted in that movie. Examine the supplied
records, each of which contains that exact movie ID. Look for phrases 'PERSON
acted in the movie MOVIE' and 'PERSON was an actor in the movie MOVIE'. PERSON
is the E-prefixed string before the verb. Return all such person IDs.
```

**M9: find_person_awards** Given a person, list all awards won by them. Evidence mode: `bound_mentions`.

**M9 / learned solver instruction**

```
Given a person ID, find all awards won by that person. Look for 'person: PERSON
; award: AWARD' and 'award: AWARD ; winner: PERSON'. Return the award IDs.
```

### D.2 FinChain

These strategies address the evaluated indirect-method operating cash-flow task family, not arbitrary financial text. The 0.8B and 2B students use signed component extraction followed by summation; 4B combines four prerequisite computations in a five-node program; and 9B forms an income/non-cash subtotal before applying working-capital adjustments. Thus even two-node strategies can define different intermediate subproblems. These are student-dependent choices, not a monotonic rule that smaller students require more nodes. Reuse refers to new instances within this task family.

Table D.3: FinChain: selection-chosen strategies. Scores use the task metric (%).

| Student | Generation | Programs / modules / nodes | Selection | Test |
|---|---|---|---|---|
| 0.8B | 4 | 1 / 2 / 2 | 88.12 | 84.57 |
| 2B | 3 | 1 / 2 / 2 | 91.88 | 92.19 |
| 4B | 5 | 1 / 4 / 5 | 90.00 | 90.04 |
| 9B | 3 | 1 / 2 / 2 | 97.50 | 93.16 |

### D.2.1 0.8B STUDENT

One sequential program uses two modules: the first extracts the seven signed components specified by the task's fixed narrative structure, and the second sums them. Unlike 4B, it does not separate extraction into multiple prerequisites.

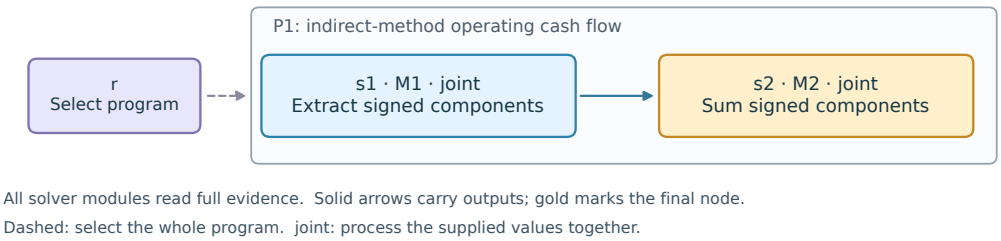



Figure D.5: FinChain / 0.8B: one program passes seven signed components from extraction to summation. M labels identify the solver prompts below.

**FinChain / 0.8B / learned router**

```
You are a router for a financial accounting reasoning system. The available
program is: compute_ocf (applicability: computes cash flow from operating
activities using the indirect method when the evidence provides the necessary
line items). If the user query asks to compute cash flow from operating
activities using the indirect method, select this program with no bindings.
Provide your reasoning for the selection.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: compute_ocf
Applicability: The question asks to compute cash flow from operating activities
using the indirect method given a set of line items in the evidence.
Slots: none
s1 / M1 / joint / predecessors: none
  Extract the seven amounts with correct signs from the evidence.
s2 / M2 / joint / predecessors: s1
  Sum the signed integers: {{s1}}
Final node: s2
```

**Shared solver instructions.**

**M1: extract_signed_ocf_values** Extract the seven dollar amounts needed for the indirect method cash flow from operations, assign correct signs, and return them as signed integer strings. Evidence mode: `full`.

**M1 / learned solver instruction**

```
You are a specialist in indirect method operating cash flow extraction. The
evidence contains a sentence with exactly this structure: "EXXXXX reported a net
income of $N1. During the same period, it incurred depreciation of $N2,
amortization of $N3, an increase in accounts receivable of $N4, an increase in
inventory of $N5, a decrease in accounts payable of $N6, and an increase in
accrued expenses of $N7." You must extract each dollar amount N1-N7 as a signed
integer string without the dollar sign or commas, applying the following sign
rules:
1. Net income (N1) → positive (no change).
2. Depreciation (N2) → positive (add back non-cash expense).
3. Amortization (N3) → positive (add back non-cash expense).
4. Increase in accounts receivable (N4) → negative (subtract, because more
receivables reduce cash).
5. Increase in inventory (N5) → negative (subtract, more inventory consumes
cash).
6. Decrease in accounts payable (N6) → negative (subtract, payment of payables
reduces cash).
7. Increase in accrued expenses (N7) → positive (add, because expenses accrued
but not yet paid increase cash).
Output a list containing exactly these seven signed integer strings in the order
(1) net income, (2) depreciation, (3) amortization, (4) increase in AR, (5)
increase in inventory, (6) decrease in AP, (7) increase in accrued expenses. Do
not output any other text.
```

**M2: sum_signed_integers** Sum a list of signed integer strings to produce the final operating cash flow amount. Evidence mode: `full`.

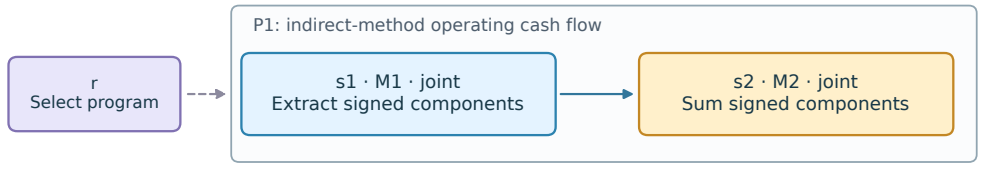
**M2 / learned solver instruction**

```
You are a precise arithmetic calculator. Ignore the evidence passage. The
question provides a list of signed integer strings inside the placeholder. You
must add all the integers together exactly as they appear, respecting the signs.
Perform the addition step by step in your reasoning, and then output the final
sum as a single signed integer string inside a list. Do not output any
intermediate numbers as values; return only the final sum.
```

### D.2.2 2B STUDENT

One sequential program again uses two modules to separate signed-component extraction from summation. Its instructions are independently selected for the 2B student; the shared two-node layout does not capture these wording differences from 0.8B.

**FinChain / 2B student**




All solver modules read full evidence. Solid arrows carry outputs; gold marks the final node.
Dashed: select the whole program. joint: process the supplied values together.

Figure D.6: FinChain / 2B: one program passes extracted signed components to the final summation node. The layout matches 0.8B, but the learned instructions differ.

**FinChain / 2B / learned router**

```
You are a router. Given the public question, select the most suitable reusable
program. If the question asks to compute cash flow from operating activities
using the indirect method, select program 'indirect_cash_flow' with no bindings.
Output a route choice with reasoning, program name, and an empty bindings list.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: indirect_cash_flow
Applicability: The question asks to compute cash flow from operating activities
using the indirect method.
Slots: none
s1 / M1 / joint / predecessors: none
  Extract the operating cash flow components and their signs using the indirect
method from the evidence.
s2 / M2 / joint / predecessors: s1
  Sum these signed integers: {{s1}}
Final node: s2
```

**Shared solver instructions.**

**M1: extract_indirect_cash_flow_components** Extracts the seven standard indirect-method operating cash flow line items with correct signs. Evidence mode: `full`.

**M1 / learned solver instruction**

```
Specialist in financial accounting. Given the evidence, identify the numerical
amounts for the following items and apply the sign (positive or negative) as
specified by the indirect method rules:

1. Net income: positive.
2. Depreciation: positive.
3. Amortization: positive.
4. Increase in accounts receivable: negative.
5. Increase in inventory: negative.
6. Decrease in accounts payable: negative.
7. Increase in accrued expenses: positive.

For each item, extract the numeric integer from the evidence, ignoring any
dollar signs, commas, or spaces. If an item is not present in the evidence,
output '+0'. The order of output must be exactly the order listed above. Return
a JSON array of seven strings, each string starting with '+' or '-' followed by
the integer digits (no other characters). For example, an output might look
like: ['+12345', '+6789', '+1011', '-200', '-300', '-400', '+50'] (do not copy
these numbers, they are placeholders). Do not include any additional explanation
in the values output.
```

**M2: sum_signed_integers** Sums a list of signed integer strings and returns the total as a signed integer string. Evidence mode: `full`.

**M2 / learned solver instruction**

```
Arithmetic calculator. You will receive a Python-style list of signed integer
strings (e.g., "['+100', '-20', '+30']"). Parse each string as an integer (the
sign indicates positive or negative). To reduce arithmetic mistakes, compute the
sum by starting with zero and iteratively adding each integer one by one,
keeping a running total. After processing all integers, the final total is the
answer. Output a single integer string: if the result is negative, prefix with
'-'; if non-negative, output the integer without any sign. Do not include commas
or formatting. Provide a brief reasoning that shows each addition step to verify
correctness.
```

### D.2.3 4B STUDENT

One program uses four prerequisite computations for net income, depreciation, amortization, and working-capital changes, followed by a node that sums all four outputs. Four modules serve five nodes: M2 is shared by depreciation and amortization. This is a dependency structure, not a claim about parallel scheduling.

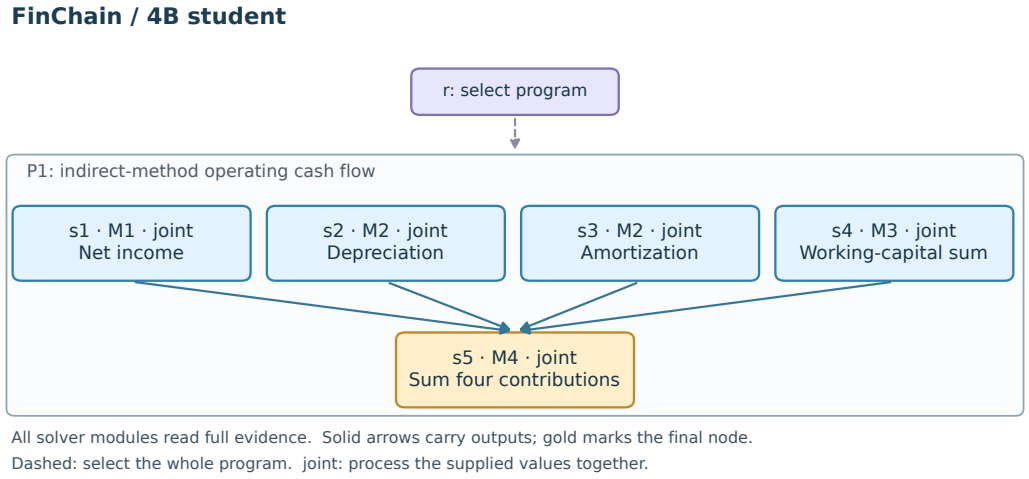



Figure D.7: FinChain / 4B: the router selects the entire enclosed program. All four prerequisite outputs feed the final aggregation node; they are not alternative branches. The two amount-extraction nodes share M2.

**FinChain / 4B / learned router**

```
You are a router for a modular financial reasoning system. Given the user
question (and only the question), select the most appropriate program from the
available list. For a question that asks to compute cash flow from operating
activities using the indirect method, choose the program 'indirect_cfo_merged'.
Its applicability: "The question asks to compute cash flow from operating
activities using the indirect method. The evidence provides net income,
depreciation, amortization, and changes in working capital accounts listed as
increases or decreases with dollar amounts." The program has no input slots, so
provide an empty bindings list. Output your reasoning and then the chosen
program and bindings.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: indirect_cfo_merged
Applicability: The question asks to compute cash flow from operating activities
using the indirect method. The evidence provides net income, depreciation,
amortization, and changes in working capital accounts listed as increases or
decreases with dollar amounts.
Slots: none
s1 / M1 / joint / predecessors: none
  Extract the net income amount as an integer string.
s2 / M2 / joint / predecessors: none
  Extract the integer amount (including leading minus if negative) for
depreciation.
s3 / M2 / joint / predecessors: none
  Extract the integer amount (including leading minus if negative) for
amortization.
s4 / M3 / joint / predecessors: none
```

```
  Calculate the net cash effect of changes in accounts receivable, inventory,
accounts payable, and accrued expenses.
s5 / M4 / joint / predecessors: s1, s2, s3, s4
  Compute the cash flow from operating activities: net income = {{s1}},
depreciation = {{s2}}, amortization = {{s3}}, net working capital change =
{{s4}}. Sum them exactly as signed integers.
Final node: s5
```

**Shared solver instructions.**

**M1: extract_net_income** Extracts the net income integer from the evidence snippet. Evidence mode: `full`.

**M1 / learned solver instruction**

```
You are an extractor. The evidence contains a sentence like: 'E... reported a
net income of $AMOUNT.' Find the dollar amount after 'net income of $'. Return
only the digits (e.g., '107275'). The output must be an exact contiguous
substring of the evidence. Return exactly one string.
```

**M2: extract_amount_signed** Extracts a signed integer amount from financial evidence for a given line item. Evidence mode: `full`.

**M2 / learned solver instruction**

```
You are a financial data extraction specialist. The evidence text contains a
company's financial data with dollar amounts. Your user question asks you to
extract the integer amount (including leading minus sign if negative, otherwise
just digits) for a specific line item. Locate the mention of that line item in
the evidence and find the dollar amount that follows it. Extract the numeric
part after the dollar sign, ignoring commas and the dollar sign itself. Return
only that integer string (e.g., '112643' or '-5000') as a single-element list.
The extracted string must be an exact contiguous substring of the evidence. If
the line item is not found, return an empty list. Do not include any other text.
```

Removing commas can conflict with the requirement for an exact contiguous substring when the evidence uses thousands separators; whether this affects an answer depends on the actual input and execution.

**M3: sum_working_capital** Sums working capital changes with correct cash-flow signs. Evidence mode: `full`.

**M3 / learned solver instruction**

```
You are a financial calculator. Identify the amounts for:
- increase in accounts receivable of $A (cash outflow, negative)
- increase in inventory of $B (negative)
- decrease in accounts payable of $C (negative)
- increase in accrued expenses of $D (positive)
Sum them with those signs: -A - B - C + D. Return the result as a single integer
string (may be negative). If any item is missing, treat as 0.
```

**M4: compute_cfo_indirect** Computes the cash flow from operating activities by summing net income, depreciation, amortization, and net working capital change. Evidence mode: `full`.

**M4 / learned solver instruction**

```
You are a precise adder. You receive four signed integer strings: net income,
depreciation, amortization, and net working capital change. Sum them exactly as
signed integers. Do the addition step-by-step: first add net income and
depreciation, then add amortization to that sum, then add net working capital
change. Double-check each intermediate sum. Return the final result as a single
integer string (with a minus sign if negative, otherwise pure digits). Ensure no
extra text in values.
```

### D.2.4 9B STUDENT

One program uses two modules. The first sums net income, depreciation, and amortization; the second combines that subtotal with four working-capital adjustments read from the evidence. Its two-stage calculation therefore differs from the extract-all-then-sum decomposition of 0.8B and 2B.

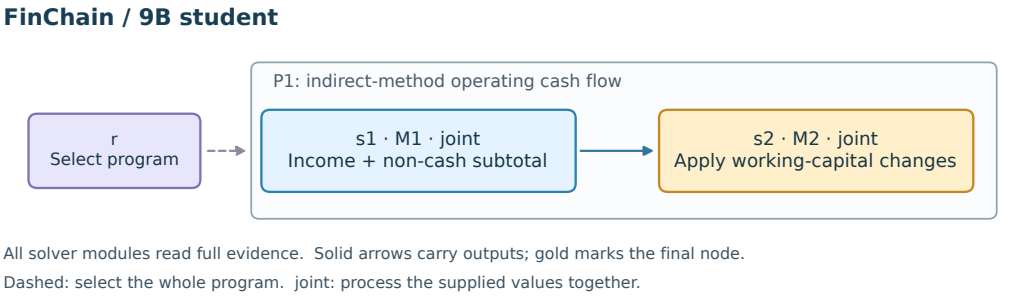



Figure D.8: FinChain / 9B: one program passes the income/non-cash subtotal to a solver that applies working-capital adjustments and returns operating cash flow.

**FinChain / 9B / learned router**

```
Read the question. If it asks to "Compute the cash flow from operating
activities using the indirect method", select program
"indirect_cash_flow_two_step". No input slots are required; supply an empty
bindings list. If the question does not match, reply that no program is
available.
```

**Program templates.**

**P1 / applicability, bindings, and subquestions**

```
Program: indirect_cash_flow_two_step
Applicability: The question asks to compute cash flow from operating activities
using the indirect method.
Slots: none
s1 / M1 / joint / predecessors: none
  Using the evidence, compute the sum of net income, depreciation, and
amortization. Return only the integer.
s2 / M2 / joint / predecessors: s1
  Start with {{s1}} and adjust for working capital: subtract the increase in
accounts receivable, subtract the increase in inventory, subtract the decrease
in accounts payable, add the increase in accrued expenses. Return only the final
integer.
Final node: s2
```

**Shared solver instructions.**

**M1: sum_non_cash** Add net income, depreciation, and amortization from the evidence. Evidence mode: `full`.

**M1 / learned solver instruction**

```
Find the three integers in the evidence: net income, depreciation, amortization.
They appear after '$' and may include commas; treat them as integers. First, add
net income and depreciation to get subtotal A. Write that sum in your reasoning.
Then add amortization to subtotal A to get the final total. Verify each addition
digit by digit. Return the final total as a single integer string with no
commas, signs, or other text. Show both intermediate sums in your reasoning.
```

**M2: adjust_working_capital** Finish the indirect cash flow calculation by adjusting for changes in working capital accounts. Evidence mode: `full`.

**M2 / learned solver instruction**

```
You are given a starting integer value {{s1}}. Using the full evidence, locate
the four amounts (positive integers after '$'): the increase in accounts
receivable, the increase in inventory, the decrease in accounts payable, the
increase in accrued expenses. Write each as a plain integer digit string.
Impact on cash flow:
- Subtract the increase in accounts receivable.
- Subtract the increase in inventory.
- Subtract the decrease in accounts payable (treat this as cash outflow, so
subtract the number shown).
- Add the increase in accrued expenses.
Perform the arithmetic sequentially: result = {{s1}} - AR - INV - AP + AE. Check
each intermediate subtraction and addition carefully. Then return ONLY the final
integer string (no commas, signs, or extra text). Show the full arithmetic chain
in your reasoning.
```

The instruction's request for no signs can conflict with a negative result that requires a minus sign. This is a potential semantic conflict in the preserved text, not evidence that a particular execution failed for this reason.

### D.3 Instruction Footprint and Inference-Token Statistics

Table D.4 summarizes the archived main-condition *test-set* executions of the five baselines and EvoScaffold across two datasets and four student sizes. Correct and incorrect answers are both eligible; failed or unreconstructable executions are not assigned zero cost. The reported means are therefore conditional on auditable completion.

We distinguish four quantities. *Active / call* is the mean number of strategy-instruction tokens activated in a functional model call, pooling all retained calls rather than averaging per-question means. *Cumulative / question* sums these instructions over a question's calls before averaging across eligible questions. Both include routing instructions and exclude the question, evidence, intermediate values, and interface boilerplate; lengths use the same Qwen3.5 tokenizer. For EvoScaffold, routing includes program applicability and slot descriptions, while a local call includes its solver instruction and the static text of its subquestion template. *Input / question* sums the complete model inputs, including those excluded contents. *Total / question* additionally includes the model outputs. The last two quantities use reconstructed recorded usage and include routing.

Instruction lengths are displayed to the nearest token; full-input and total usage are in k tokens, rounded to two decimal places (1k = 1,000 tokens). Rounding changes presentation only. Engineering retries and output-format fallbacks are excluded, while functional mapping, evidence scans, review, and workflow-internal repeated calls are retained. Cache replay is counted as one functional execution, not zero-token inference; these are usage estimates, not discounted invoices.

Table D.4: Mean strategy-instruction length and inference usage on the COMMAQA-E and FinChain test sets. Only completed, auditable executions enter the means, regardless of correctness. Instruction counts are rounded to whole tokens; input and total are rounded to 0.01k tokens.

| Method | Active / call (tokens) | Cumulative / question (tokens) | Input / question (k tokens) | Total / question (k tokens) |
|---|---|---|---|---|
| **COMMAQA-E** *Qwen3.5-0.8B* | | | | |
| MIPROv2 | 309 | 309 | 2.81 | 3.11 |
| GEPA | 999 | 999 | 3.49 | 3.71 |
| PLD | 277 | 277 | 2.59 | 2.78 |
| SELF-DISCOVER | 493 | 493 | 2.81 | 3.02 |
| AFlow (restricted) | 34 | 34 | 2.39 | 2.65 |
| EvoScaffold | 212 | 1,057 | 3.54 | 3.92 |

Continued on next page

*Table D.4 continued*

| Method | Active / call (tokens) | Cumulative / question (tokens) | Input / question (k tokens) | Total / question (k tokens) |
|---|---|---|---|---|
| **COMMAQA-E** *Qwen3.5-2B* | | | | |
| MIPROv2 | 118 | 118 | 2.65 | 2.88 |
| GEPA | 1,018 | 1,018 | 3.55 | 3.77 |
| PLD | 252 | 252 | 2.56 | 2.77 |
| SELF-DISCOVER | 541 | 541 | 2.85 | 3.04 |
| AFlow (restricted) | 34 | 34 | 2.39 | 2.65 |
| EvoScaffold | 138 | 985 | 5.37 | 5.96 |
| **COMMAQA-E** *Qwen3.5-4B* | | | | |
| MIPROv2 | 158 | 158 | 2.65 | 2.99 |
| GEPA | 646 | 646 | 3.16 | 3.59 |
| PLD | 252 | 252 | 2.56 | 2.70 |
| SELF-DISCOVER | 337 | 337 | 2.64 | 2.78 |
| AFlow (restricted) | 82 | 326 | 9.60 | 11.02 |
| EvoScaffold | 115 | 838 | 4.01 | 4.61 |
| **COMMAQA-E** *Qwen3.5-9B* | | | | |
| MIPROv2 | 381 | 381 | 2.89 | 3.37 |
| GEPA | 459 | 459 | 2.96 | 3.39 |
| PLD | 250 | 250 | 2.55 | 2.70 |
| SELF-DISCOVER | 393 | 393 | 2.70 | 2.84 |
| AFlow (restricted) | 72 | 358 | 11.93 | 13.42 |
| EvoScaffold | 98 | 902 | 6.24 | 6.93 |
| **FinChain** *Qwen3.5-0.8B* | | | | |
| MIPROv2 | 132 | 132 | 0.62 | 1.05 |
| GEPA | 671 | 671 | 1.21 | 1.56 |
| PLD | 812 | 812 | 1.15 | 1.35 |
| SELF-DISCOVER | 495 | 495 | 0.83 | 1.00 |
| AFlow (restricted) | 48 | 289 | 4.56 | 7.28 |
| EvoScaffold | 180 | 540 | 1.56 | 2.00 |
| **FinChain** *Qwen3.5-2B* | | | | |
| MIPROv2 | 228 | 228 | 0.72 | 1.12 |
| GEPA | 797 | 797 | 1.30 | 1.90 |
| PLD | 846 | 846 | 1.18 | 1.29 |
| SELF-DISCOVER | 394 | 394 | 0.73 | 0.91 |
| AFlow (restricted) | 34 | 34 | 0.35 | 0.81 |
| EvoScaffold | 161 | 482 | 1.51 | 1.90 |
| **FinChain** *Qwen3.5-4B* | | | | |
| MIPROv2 | 293 | 293 | 0.79 | 1.28 |
| GEPA | 665 | 665 | 1.19 | 1.58 |
| PLD | 395 | 395 | 0.73 | 0.90 |
| SELF-DISCOVER | 386 | 386 | 0.72 | 0.90 |
| AFlow (restricted) | 34 | 34 | 0.34 | 0.84 |
| EvoScaffold | 139 | 833 | 2.97 | 3.56 |
| **FinChain** *Qwen3.5-9B* | | | | |
| MIPROv2 | 418 | 418 | 0.91 | 1.15 |
| GEPA | 1,020 | 1,020 | 1.52 | 2.28 |
| PLD | 869 | 869 | 1.20 | 1.35 |
| SELF-DISCOVER | 357 | 357 | 0.69 | 0.93 |
| AFlow (restricted) | 34 | 34 | 0.34 | 0.84 |
| EvoScaffold | 136 | 408 | 1.45 | 1.80 |

Short local instructions need not imply low cumulative usage. For example, the 4B COMMAQA-E strategy activates about 115 instruction tokens per call but accumulates about 838 per question across its calls. The separate input and total columns also capture evidence, intermediate values, and generated outputs. These are observational statistics of the selected strategies, not an equal-token causal comparison.

### D.4 How the Four Operators Can Improve a Strategy

We examine one recorded parent–child revision for each operator on COMMAQA-E. Mutation, split, and crossover use the 4B student; merge uses the 0.8B student. The execution examples illustrate the changes made by each revision, while Table D.5 reports the corresponding parent and child performance on the same held-out selection set using the same student model. These selected cases illustrate possible benefits of the operators without estimating their average effects. Selection scores are reported for comparison and were not used to accept offspring or guide teacher revisions.

Table D.5: Selection-set performance of four recorded COMMAQA-E revisions. Arrows indicate parent-to-child changes. Each pair is evaluated on the same selection set using the indicated student model.

| Operator / student | Selection EM (%) |
|---|---|
| Mutation / 4B | 93.75 → 94.38 |
| Split / 4B | 88.75 → 90.00 |
| Crossover / 4B | 84.38 → 91.88 |
| Merge / 0.8B | 28.13 → 33.75 |

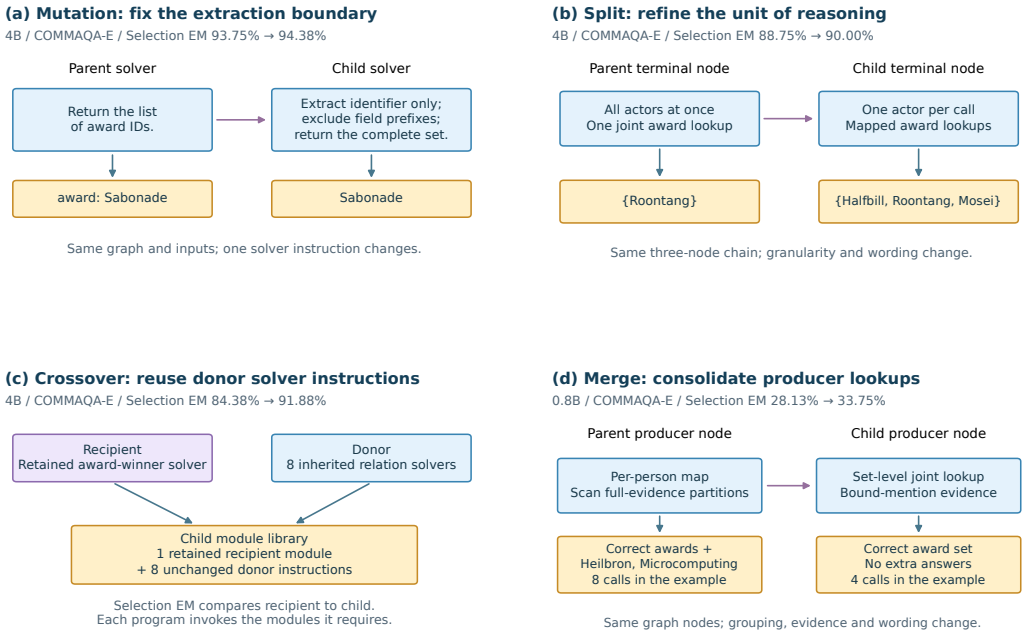



Figure D.9: Local views of four COMMAQA-E edits. Mutation revises a solver instruction; split changes execution granularity and relation wording; crossover reuses donor modules; merge consolidates producer lookups while changing evidence access and instructions. Scores shown in the panels are selection-set EM. The examples below detail the instruction changes and recorded outcomes.

### D.4.1 MUTATION: REPAIRING THE ANSWER BOUNDARY WITHOUT CHANGING THE GRAPH

The COMMAQA-E parent and child have identical routers, program structures, evidence modes, and module assignments. Only the movie-award solver instruction changes. The parent requests award identifiers but leaves their extraction boundary implicit; the child explicitly excludes the field prefix and asks for the complete set of identifiers.

**Parent / movie-award instruction**

```
You are given a movie ID. The evidence contains only records that mention that
movie. Find all awards won by this movie. Look for 'movie: [movie] ; award:
[award]' or 'movie: [movie] ; awarded: [award]'. Return the list of award IDs.
```

**Child / movie-award instruction**

```
You are given a movie ID. The evidence contains only records that mention that
movie. Find all awards that this movie won. Look for lines matching 'movie:
[movie] ; award: [award]' or 'movie: [movie] ; awarded: [award]'. From each
matching line, extract exactly the award identifier after the colon, without the
'award:' or 'awarded:' prefix. Remove leading/trailing whitespace. Return the
complete set of distinct award IDs.
```

For "What awards have movies produced by people born in 1931 won?", the earlier nodes return the same people and movies. The parent's terminal call returns `award: Sabonade`; the child returns a whitespace-padded `Sabonade`, which the existing whitespace normalization converts to `Sabonade`. The other answer members are unchanged. For this parent–child pair, selection EM increases from 93.75% to 94.38%. This edit improves answer extraction while preserving the reasoning graph.

### D.4.2 SPLIT: MOVING FROM AN ACTOR-SET QUERY TO PER-ACTOR REASONING

The affected COMMAQA-E program first finds award-winning movies and then their actors. The parent asks its terminal solver about the whole actor set; the child instead instantiates that solver once per actor. The stored three-node chain remains intact, but its execution expands into smaller local questions.

**Parent / joint terminal subquestion**

```
Identify all awards received by these persons: {{s2}}.
```

**Child / mapped terminal subquestion**

```
List all awards received by the person "{{s2}}".
```

The following solver instructions show the accompanying wording change. In addition to person-first records, the child recognizes `award: AWARD ; winner: PERSON` and `AWARD was awarded to PERSON`.

**Parent / actor-award solver instruction**

```
The evidence contains only records mentioning any of the person identifiers
listed in the question. Find every award those persons have received. Look for
statements like 'person: PERSON ; award: AWARD' or 'PERSON won the AWARD award'.
Return the unique award identifiers. If none, return an empty list. Output a
JSON object with 'reasoning' and 'values'.
```

**Child / actor-award solver instruction**

```
The evidence contains only records mentioning the person identifier(s) given in
the question. Identify all awards that those persons received. Look for records
like 'person: PERSON ; award: AWARD', 'PERSON won the AWARD award', 'award:
AWARD ; winner: PERSON', or 'AWARD was awarded to PERSON'. Return the unique
award identifiers. If none, return an empty list. Output a JSON object with
'reasoning' and 'values'.
```

The edit therefore jointly changes granularity and instruction coverage. On the recorded Sockbox question, the parent's final set is `{Roontang}`; the child returns the correct `{Halfbill, Roontang, Mosei}`. For this parent–child pair, selection EM increases from 88.75% to 90.00%.

#### D.4.3 Crossover: reusing explicit relation solvers from a donor

The COMMAQA-E child retains the recipient's award-winner lookup while inheriting eight solver instructions unchanged from a complementary donor. All eight inherited modules are used across the recorded executions, with each question invoking the modules required by its selected program. The four three-hop program patterns remain. The following producer lookup illustrates the difference between a general instruction and the donor's explicit relation patterns:

**Recipient / producer-lookup instruction**

```
List movies that the given person produced. Look for evidence indicating that a
person produced a movie. Return the movie IDs. Be exhaustive.
```

**Inherited donor / producer-lookup instruction**

```
You are given a person ID. The evidence contains only records that mention that
person. Find all movies produced by this person. Look for '[person] produced the
movie [ID] with others', '[person] was one of the producers of the movie [ID]',
or similar phrases indicating production. Return the list of movie IDs.
```

For the question about movies produced by people born in 1913, both versions identify `Midcareer`. The recipient returns only `{Conforancy, Pipesia}` at the producer node. The child additionally finds `Clenestration`, allowing the terminal lookup to recover its award `Electrodesal`. For this parent–child pair, selection EM increases from 84.38% to 91.88%. Because several modules are inherited together, the aggregate gain reflects the crossover as a whole. The displayed producer instruction illustrates one of the inherited changes. The child's selection EM of 91.88% remains slightly below the donor's 92.50%, demonstrating improvement over the recipient without establishing superiority over both parents.

#### D.4.4 Merge: consolidating producer searches into a set-level lookup

In this 0.8B COMMAQA-E branch, the producer program follows birth year $\rightarrow$ people $\rightarrow$ movies $\rightarrow$ awards. The parent maps the producer lookup over individual people and scans partitions of the full evidence. The child instead passes the person set to one joint lookup using records that mention the bound people. Its local instruction also explicitly distinguishes producing from acting, writing, and directing. Both strategies contain four programs, nine modules, and twelve graph nodes: the merge consolidates execution while preserving the number of stored graph nodes. The router wording changes as well; the other three programs remain unchanged.

**Parent / per-person producer instruction**

```
You will be given a person identifier and a chunk of the full evidence. Scan the
entire chunk for any statement indicating that this person produced a movie.
Accept any formulation such as '[person] produced the movie [movie] with
others', '[person] was one of the producers of the movie [movie]', '[person]
produced [movie]', or similar. Extract all movie identifiers that appear in such
statements. Do not skip any record; check every line. Return the list of movie
identifiers found in this chunk. If no producing statement is found, return an
empty list.
```

**Child / set-level producer instruction**

```
You will be given a list of person identifiers. The evidence has been filtered
to records that mention any of these persons. For each person, identify all
movie identifiers that this person produced. A producing relationship is
indicated by phrases like 'PERSON produced the movie MOVIE with others', 'PERSON
was one of the producers of the movie MOVIE', or similar. Do not confuse with
statements about acting ('actor', 'acted in'), writing ('wrote', 'writer'), or
directing ('directed', 'director'). Return the list of unique movie identifiers
produced by any of the given persons.
```

The router also gains explicit question-pattern cues in its four program descriptions. The complete changed block is shown below; the surrounding router introduction and final selection instructions are unchanged.

**Parent / router program-description block**

```
1. 'awards_of_movies_directed_by_winner_of_award' - Applicable when the question
asks for awards received by movies that were directed by the winner of a
specific award. Requires slot 'award_id': the award identifier exactly as it
appears (an alphanumeric ID). Do not include surrounding words like 'winners'.
2. 'awards_of_actors_of_award_winning_movies' - Applicable when the question
asks for awards won by actors who performed in movies that won a specific award.
Requires slot 'award_id': the award identifier exactly as it appears.
3. 'awards_of_movies_written_by_people_born_in_year' - Applicable when the
question asks for awards won by movies that were written by people born in a
specific year. Requires slot 'birth_year': the birth year as a four-digit
number.
4. 'awards_of_movies_produced_by_people_born_in_year' - Applicable when the
question asks for awards won by movies that were produced by people born in a
specific year. Requires slot 'birth_year': the birth year as a four-digit
number.
```

**Child / router program-description block**

```
1. 'awards_of_movies_directed_by_winner_of_award' - Applicable when the question
asks for awards received by movies that were directed by the winner of a
specific award. It mentions an award ID and a phrase like 'directed by the ...
winners'. Requires slot 'award_id': the award identifier exactly as it appears
(an alphanumeric ID). Do not include surrounding words like 'winners'.
2. 'awards_of_actors_of_award_winning_movies' - Applicable when the question
asks for awards won by actors who performed in movies that won a specific award.
It mentions a particular award ID and 'winning movies' or 'actors of'. Requires
slot 'award_id': the award identifier exactly as it appears.
3. 'awards_of_movies_written_by_people_born_in_year' - Applicable when the
question asks for awards won by movies that were written by people born in a
specific year. It mentions a year and 'written by'. Requires slot 'birth_year':
the birth year as a four-digit number.
4. 'awards_of_movies_produced_by_people_born_in_year' - Applicable when the
question asks for awards won by movies that were produced by people born in a
specific year. It mentions a year and 'produced by'. Requires slot 'birth_year':
the birth year as a four-digit number.
```

For a recorded question about movies produced by people born in 1931, both strategies first identify `Bibbogey`. The parent incorrectly treats `movie: Coesestion ; director: Bibbogey` as a production relation, and its partitioned searches also add `Pugo`. The resulting award set contains the two extraneous answers `Heilbron` and `Microcomputing`. The child's joint lookup returns `{Riften, Skirtsicine, Pestok}`, supported by explicit production statements, and the terminal solver returns the correct `{Hochelfoil, Jubeus, Maldezine}`. This execution uses four calls instead of eight, including routing.

For this parent–child pair, selection EM increases from 28.13% to 33.75%. The improvement reflects the combined changes to grouping, evidence access, and instructions. Together with the split example, it illustrates how search can refine or consolidate execution granularity.

Table E.1: **Teacher-call budget on COMMAQA-E.** Test exact match (%) at different logical teacher-call budgets.

| Logical teacher calls | 0.8B | 2B | 4B | 9B |
|---|---|---|---|---|
| 20 | 14.35 | 53.81 | 85.05 | 92.08 |
| 40 | 21.23 | 54.41 | 91.18 | 92.23 |
| 60 | 34.08 | 60.99 | 90.73 | 92.53 |
| **80** | **38.71** | **60.99** | **93.27** | **92.53** |
| 100 | 38.71 | 60.99 | 93.72 | 92.53 |

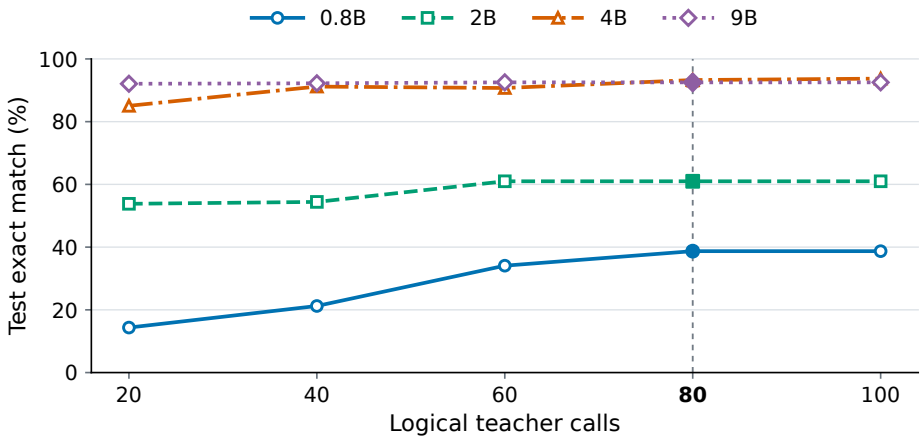



Figure E.1: **Performance around the default teacher-call cap.** COMMAQA-E test exact match at the budgets reported in Table E.1.

## E Resource Efficiency

The main experiments use a default cap of 80 logical teacher calls. To assess this choice, we compare COMMAQA-E performance at budgets of 20, 40, 60, 80, and 100 calls across four student sizes, as shown in Table E.1 and Figure E.1. The lower-budget settings indicate whether stopping earlier would sacrifice performance, while the 100-call setting assesses the benefit of extending search beyond the default cap.

The reported scores place 80 calls near a performance plateau across student sizes. For the 100-call setting, we increase the maximum number of generations beyond the default configuration while keeping the number of offspring per generation unchanged. Search stops when either the teacher-call cap or the configured generation limit is reached. The reported budgets specify upper bounds on teacher calls. Although 2B and 9B reach their final reported scores at 60 calls, increasing the budget from 60 to 80 calls improves test EM by 4.63 percentage points for 0.8B and 2.54 points for 4B. Stopping at 60 calls would therefore forgo gains that remain substantial for these two students.

Beyond 80 calls, the additional benefit is limited. Extending the budget to 100 calls increases the teacher-call allowance by 25%, but leaves the scores of 0.8B, 2B, and 9B unchanged and improves 4B by 0.45 percentage points. This comparison supports 80 calls as a practical default on COMMAQA-E: it retains the improvements observed beyond 60 calls while avoiding the additional teacher queries associated with the small gains at 100 calls.

# F COMMAQA-E FAILURE CASES ACROSS STUDENT SCALES

We examine one failed test execution of the selected strategy for each student size. The examples follow the same structure: the question and relevant evidence, the relevant program, the recorded local output, and the point where the error affects the answer. The 2B and 4B cases use the same question and evidence world but their separately selected strategies, allowing an instance-level comparison of omission and over-inclusion. These examples illustrate observed execution failures, not their frequency across the test set or the isolated effect of student size.

## F.1 0.8B: A SLOT-BINDING ERROR CREATES EXCESSIVE FAN-OUT

**Question and relevant evidence.** What awards have the actors of the Zorgion winning movies received?

**0.8B / evidence excerpts**

```
movie: Calamba ; awarded: Zorgion.
movie: Nohit ; awarded: Zorgion.
```

**Relevant strategy and inputs.** The router selects award-winning movies → actors of each movie → awards of each actor. The correct award name is `Zorgion`, but the recorded binding below includes surrounding task words. This longer string is a literal question span, so it passes the span check, even though it is the wrong semantic slot value.

**0.8B / recorded router output**

```
{
  "reasoning": "The question asks for awards received by the 'actors of the
Zorgion winning movies'. This matches the applicability of
'awards_of_actors_of_award_winning_movies', which requires the slot 'award_id'
to be the award identifier exactly as it appears in the question. The entity
type is 'actors of', and the specific phrase is 'Zorgion winning movies'.",
  "program": "awards_of_actors_of_award_winning_movies",
  "bindings": [
    {
      "name": "award_id",
      "value": "Zorgion winning movies"
    }
  ]
}
```

**Failure and propagation.** No record contains the complete bound phrase `Zorgion winning movies`, causing the evidence view to fall back to the full passage. The first solver returns a repetitive movie list with 14 distinct identifiers rather than the supported set `{Calamba, Nohit}`. Mapping the next node over these movies yields 10 actors and consumes 16 calls including routing. The terminal node would require 10 more calls, exceeding the strategy's cap of 20. Execution stops with an explicit call-budget error; it does not silently drop actors. The reference answer is `{Roontang, Lameze, Fannyxist}`. The first observable error is the over-long slot binding. It is followed by an over-inclusive movie lookup, whose outputs expand the downstream workload until execution is stopped by the call limit.

## F.2 2B: A MISSING INTERMEDIATE ENTITY CANNOT BE RECOVERED DOWNSTREAM

**Question and relevant evidence.** What awards have movies produced by people born in 1938 won?

**2B / evidence excerpts**

```
Huckberryberry was born in the year 1938.
Huckberryberry was one of the producers of the movie Calcivore.
Huckberryberry was one of the producers of the movie Spidertail.
Huckberryberry produced the movie Misapportionment with others.
movie: Calcivore ; awarded: Grangebagger.
```

**Relevant strategy and inputs.** The router correctly binds the year `1938`. The program performs birth-year lookup → per-person producer lookup → per-movie award lookup. Its first node returns `Huckberryberry`. The producer node receives all three explicit production records above, but returns only two movies:

**2B / recorded local solver output**

```
{
  "reasoning": "The evidence explicitly states that Huckberryberry produced the
movie Misapportionment and was one of the producers of Spidertail. It also
mentions Huckberryberry was an actor in other films, but the task specifically
asks for movies they produced. The records confirm these two production
credits.",
  "values": [
    "Misapportionment",
    "Spidertail"
  ]
}
```

**Failure and propagation.** The missing movie is `Calcivore`, despite an explicit producer record in the node's input. The terminal module correctly returns `Sockbox` for `Misapportionment` and `Monoxandrite` for `Spidertail`. Its union is therefore `{Sockbox, Monoxandrite}`, missing the reference member `Grangebagger` associated with `Calcivore`. The failure is incomplete local enumeration, not incorrect final summation or aggregation. Once the intermediate entity is absent, the mapped endpoint is never asked about it.

### F.3 4B: A VALID ENTITY IS RETURNED UNDER THE WRONG RELATION

**Question and relevant evidence.** What awards have movies produced by people born in 1938 won?

**4B / evidence excerpts**

```
Huckberryberry was born in the year 1938.
Huckberryberry was one of the producers of the movie Calcivore.
Huckberryberry was one of the producers of the movie Spidertail.
Huckberryberry produced the movie Misapportionment with others.
movie: Misgendery ; director: Huckberryberry.
movie: Misgendery ; award: Bouchery.
```

**Relevant strategy and inputs.** This is the same test instance as the 2B case. Routing and birth-year lookup are again correct. The producer node now returns all three relevant movies, but also includes `Misgendery`, whose supplied record states a director relationship rather than a producer relationship:

**4B / recorded local solver output**

```
{
  "reasoning": "The evidence explicitly states that Huckberryberry produced the
movie Misapportionment, was one of the producers of Calcivore and Spidertail,
and produced Misgendery. These four movies are the only ones linked to
Huckberryberry as a producer in the provided text.",
  "values": [
    "Calcivore",
    "Misapportionment",
    "Spidertail",
    "Misgendery"
  ]
}
```

**Failure and propagation.** The final lookup retrieves `Bouchery` for the extra movie. The answer becomes `{Grangebagger, Sockbox, Monoxandrite, Bouchery}`, while the reference contains only the first three members. Unlike 2B's omission, this error adds an unsupported relation edge to the intermediate set. The string `Misgendery` really appears in the input, so literal grounding cannot reject it. Both models route correctly and execute a meaningful three-hop decomposition, yet their local relation extraction fails in opposite directions on the same evidence.

### F.4 9B: AN INAPPROPRIATE IDENTIFIER CONSTRAINT REJECTS VALID EVIDENCE

**Question and relevant evidence.** What awards did the movies directed by the Custodio winners receive?

**9B / evidence excerpts**

```
Bioperatology won the Custodio award.
movie: Tarta ; director: Bioperatology.
movie: Sahaki ; director: Bioperatology.
movie: Tarta ; award: Dextrite.
movie: Sahaki ; award: Cockspit.
```

**Relevant strategy and inputs.** The router correctly binds `Custodio`. The selected program is award winners → movies they directed → movie awards. The first solver receives the explicit record `Bioperatology won the Custodio award.` However, its learned instruction asks for a person represented by an `E-prefixed string` (see the original module in Appendix D.1.4). The model applies that inappropriate surface-form restriction to a real named entity:

**9B / recorded local solver output**

```
{
  "reasoning": "The provided evidence states that 'Bioperatology won the
Custodio award.' However, 'Bioperatology' is an entity name, not a person
(E-prefixed string). Since the query specifically asks for persons who won the
award and no person is mentioned in the evidence, the result is an empty list.",
  "values": []
}
```

**Failure and propagation.** The first node emits an empty set, so the mapped downstream nodes have no inputs. The run finishes after two calls, including routing, with an empty prediction instead of `{Dextrite, Cockspit}`. The relevant chain is explicit: Bioperatology won Custodio, directed Tarta and Sahaki, and those movies won Dextrite and Cockspit. This is not a shortage of evidence or a context-limit failure. The local instruction carries an overly restrictive identifier convention, and the larger student follows it. A more capable executor is therefore not automatically protected against a flawed learned instruction.

**What these cases show.** The execution traces make different failures distinguishable: an incorrect slot binding followed by excessive expansion, an omitted intermediate entity, an unsupported relation, and an empty lookup under an inappropriate identifier constraint. In the shared 2B/4B instance, downstream execution propagates the intermediate omission or extra entity into the final answer. These cases illustrate why valid structure and correct dependency execution do not by themselves ensure semantically correct subproblem results.