LEARNING REUSABLE AGENT SUBROUTINES FROM EXPERIENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing methods induce various forms of knowledge from agent traces, from sample workflows to programmatic skills. We propose ARISE (Agent Routine Induction from Successful Experience), which identifies recurring procedures in agent traces and synthesizes them into a reusable library of agent subroutines. Across five tasks in narrative deduction, rule application, and constraint planning, it is shown to be a more effective learning strategy than prompting methods and prose and code artifact induction methods. ARISE outperforms chain-of-thought in all ten model–task settings of the main evaluation. Compared to state-of-the-art methods in agent experience induction, our method scores 6.1 percentage points over the strongest evaluated baselines, and 11.5 percentage points over code-skill induction. In our controlled experiment, ARISE also improves average accuracy by up to 15.6 percentage points compared to formatting the induced knowledge as skills and by up to 19.1 points over rewriting it as code tools.

1 INTRODUCTION

Solving complex problems with large language models (LLMs) often requires working through intermediate steps rather than generating an answer directly. How these steps are expressed and carried out is therefore an important part of reasoning-system design. Earlier methods like chain-of-thought prompting elicit intermediate reasoning in natural language, while Program-of-Thoughts expresses computations as programs executed by an external interpreter (Wei et al., 2022; Chen et al., 2023). After the emergence of interactive agents, this organization extends to the relationship between reasoning and action. For example, ReAct interleaves reasoning with actions and observations, enabling an agent to gather information and revise its approach as a task unfolds (Yao et al., 2023a), while Plan-and-Act separates high-level planning from the execution of environment-specific

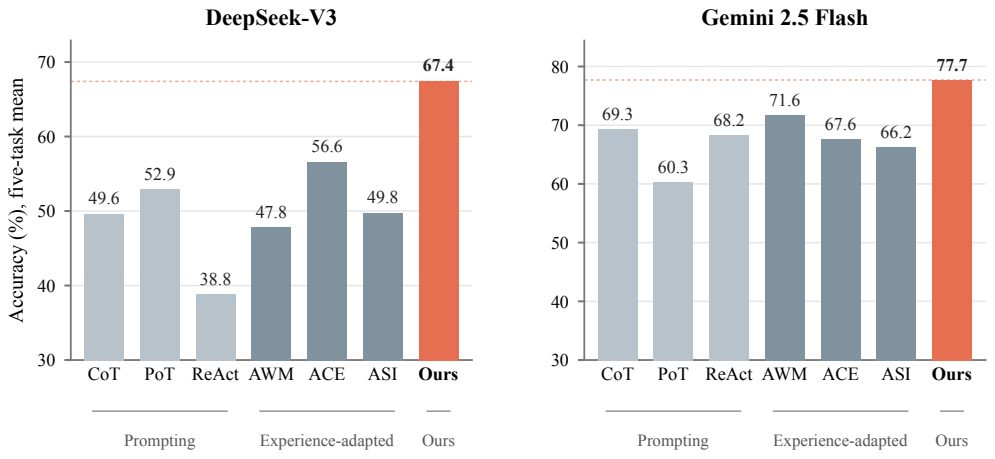


Figure 1: **Test-set accuracy by model and method.** Each panel shows the unweighted mean of the five task columns of Table 2.

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.

Library — MuSR murder mystery

`investigate_evidence(narrative: str, focus: str) -> str` `[murder-01]`
Analyzes a murder mystery narrative for specific evidence types related to a focus (suspect, weapon, location, etc.). Extracts and evaluates evidence categories: means (weapon/ability), motive (reason to kill), opportunity (presence at scene), alibi consistency (verification of whereabouts), and physical evidence (direct forensic proof). ...
Pay attention to: hard disqualifiers (ironclad alibis), conflicting evidence, reliability of witnesses, and evidence that might be misleading or require additional context. ...
Returns: a structured string analysis organized by evidence category. "Analysis for [focus]: - Means: ... - Motive: ... - Opportunity: ... - Alibi Consistency: ... Summary: ..."

`initialize_suspect_analysis(narrative: str, focus: str) -> str` `[murder-02]`
This function initiates the suspect analysis process for a murder mystery. It identifies the victim and all potential suspects mentioned in the narrative, then extracts initial clues and evidence related to each suspect. ... The agent should pay attention to: correctly identifying all named suspects and the victim; noting any immediate disqualifiers ...; being cautious of narrative red herrings or misleading initial information. ...

`+ 3 more agent subroutines: determine_if_killer, evaluate_suspect_evidence, summarize_investigation_progress (full library in Appendix A)`

At invocation the agent supplies only focus; the instance text is bound automatically, and one LLM call executes the docstring on (narrative, focus).

(partially shown)

Figure 2: **Agent subroutines induced on murder mystery solving for MuSR.** Excerpts from two of the five subroutine specifications, showing their execution signatures. The complete library is described in Appendix A.

actions (Erdogan et al., 2025). Across these approaches, reasoning guides not only the final answer, but also which actions to take and how to use their outcomes to make progress.

Such agents typically begin with predefined tools and environment actions, while task-specific routines must either be supplied by developers or worked out during execution. Experience-based adaptation offers a way to acquire these routines automatically. Agent Workflow Memory extracts reusable sample workflows from past trajectories, Agent Skill Induction constructs and verifies programmatic skills, and ACE accumulates procedural guidance in an evolving playbook in an agent's context (Wang et al., 2025a;b; Zhang et al., 2026). These approaches reduce the need to manually specify how an agent should handle recurring tasks.

Yet, what an agent should learn from experience remains a question. We propose ARISE, which learns a library of agent subroutines from successful agent trajectories. Each agent subroutine specifies a recurring operation in natural language and can be invoked by the agent similarly to a skill or a tool. This differs from prior work in that the learned subroutines are not put in an agent's context or executed by a deterministic program, but rather executed by an LLM that interprets the natural-language specification of the subroutine as shown in Figure 2. Our experiments isolate this execution choice: given the identical induced text, separate-call execution outperforms in-context skill retrieval in all ten model–task settings.

Agents repeatedly perform similar subroutines across different instances of a task. In murder-mystery problems, for example, an agent may reconstruct suspects' timelines, assess their motives, and compare evidence for competing explanations (Sprague et al., 2024). In rule-guided reasoning, it must identify applicable rules and check whether their conditions hold in the current scenario (Zhou et al., 2025). Although the evidence and intermediate conclusions change, the procedures for examining them often recur. These recurring subroutines suggest a useful level at which to reuse experience: not the answer to a previous problem or its entire solution trajectory, but a procedure that can be applied to new inputs. We extract these procedures into reusable subroutines.

We evaluate the approach on five tasks spanning narrative deduction, rule application, and constraint planning, with each model learning from its own executions and using the resulting library on held-out instances. Beyond overall accuracy, we examine how to execute learned agent subroutines, whether the learned libraries can transfer across models, and the extension to competition math problems. Our main contributions are:

- The execution interface determines the value of induced knowledge. With the induced text held fixed, agent subroutines outperform the same text delivered as retrievable skills in all ten model–task settings (+11.6 and +15.6 points on average on the two models) and the same procedures rewritten as Python tools in nine of ten (+7.9 and +19.1).
- ARISE performs up to 45.4 percentage points better than a ReAct agent, 6.1 to 10.8 points better than the strongest experience-adapted baseline on each model, and 11.5 points better than ASI (Figure 1).
- ARISE does not require expert-designed tools: libraries induced with generic tools match those induced with hand-designed ones.
- Agent subroutines induced by ARISE are effective when transferred across models. Libraries induced by one LLM remain effective when deployed unchanged on another, improving accuracy over the target model's ReAct agent on all five tasks.

2 Related Work and Motivation

Reasoning structures and tools. Prompting methods structure intermediate reasoning as chains, plans, trees, or programs (Wei et al., 2022; Wang et al., 2023b; Yao et al., 2023b; Chen et al., 2023; Gao et al., 2023), and agents interleave such reasoning with tool calls, search, and self-feedback (Yao et al., 2023a; Zhou et al., 2024b; Madaan et al., 2023; Erdogan et al., 2025). Tools can also be learned or created by the model: Toolformer learns when to call APIs, and LLMs As Tool Makers, CREATOR, CRAFT, and TroVE build reusable Python toolsets (Schick et al., 2023; Cai et al., 2024; Qian et al., 2023; Yuan et al., 2024; Wang et al., 2024b); code can also serve directly as an agent policy (Liang et al., 2023). Tool creation yields programs, whose procedures must be expressible in code; in our knowledge-format comparison, rewriting induced procedures as Python tools lowers accuracy in nine of ten model–task settings relative to agent subroutines (Section 5.2). Modular approaches delegate subtasks to separate prompts or LLM calls: Decomposed Prompting routes subtasks to specialized handlers, meta-prompting has one LM conduct expert instances of itself, and DSPy composes LLM calls behind typed signatures whose prompts are optimized automatically (Khot et al., 2023; Suzgun & Kalai, 2024; Khattab et al., 2024; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2025). In these systems, a developer defines the modules and how they compose, so the decomposition is fixed before deployment. Each agent subroutine is likewise an LLM call behind a fixed signature, but the subroutines and their specifications are induced from trajectories, and the agent composes them at run time.

Learning from agent experience. Agents store experience as verbal reflections, extracted insights, manuals, or playbooks (Shinn et al., 2023; Zhao et al., 2024; Chen et al., 2024; Zhang et al., 2026), as workflows or skill documents (Wang et al., 2025a; Ni et al., 2026), or as executable skills (Wang et al., 2024a; Zheng et al., 2025; Wang et al., 2025b); related methods reuse thought templates, cheatsheets, behaviors, or programs of thought distilled from prior problems (Yang et al., 2024; Suzgun et al., 2026; Didolkar et al., 2025; Sarch et al., 2024). In-context artifacts grow the agent's prompt as experience accumulates, and programmatic skills execute outside the context but only for procedures expressible as code. AWM also explores exposing workflows as callable actions, which replay fixed sequences of environment operations rather than invoking an LLM computation. ARISE delivers experience as procedures executed by separate LLM calls, and we compare with AWM, ACE, and ASI on the same training split.

Agent design and library learning. AFlow, ADAS, and Self-Discover search over or compose reasoning and workflow structures (Zhang et al., 2025; Hu et al., 2025; Zhou et al., 2024a); we instead fix the interaction loop and learn the operations available within it. Library learning extracts reusable abstractions from solutions to prior tasks, as program abstractions (Ellis et al., 2021; Bowers et al., 2023; Grand et al., 2024) or as natural-language reasoning abstractions trained with reinforcement learning (Qu et al., 2026). Program libraries are verified by executing them on prior solutions, and RLAD trains its abstraction generator with reinforcement learning; ARISE induces its library from a single pass of successful rollouts without updating model parameters. We apply this idea to agent trajectories, with library entries interpreted by an LLM.

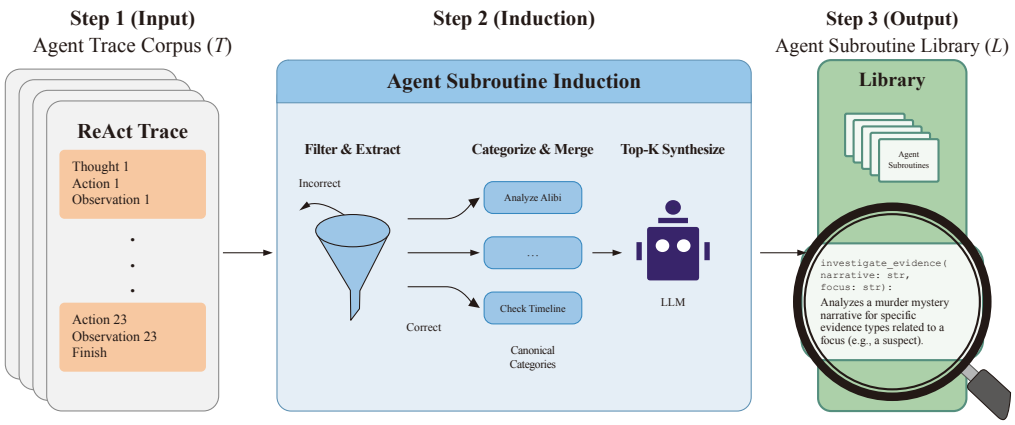


Figure 3: Core Induction Pipeline as described in Section 3.

3 Inducing Agent Subroutines

In Section 3.1, we describe the problem statement, then introduce the pipeline (Section 3.2), and describe how induced agent subroutines are deployed and executed during inference (Section 3.3).

3.1 Problem Formulation

Given a task family $\mathcal{F}$, we collect execution trajectories from a language-model agent on training instances and retain those with correct final answers, denoted by $\mathcal{T}^+$. We seek to construct a library of reusable agent subroutines from these trajectories for use on held-out instances of the same task family. The induction procedure maps

$$\mathcal{T}^+ \longmapsto \mathcal{L} = \{p_i\}_{i=1}^{M}, \qquad M \leq K,$$

where K is the maximum library size. Correctness is assessed at the trajectory level; individual intermediate steps are not independently verified.

We represent each agent subroutine with a calling interface and an execution specification:

$$p_i = (n_i, \sigma_{\mathcal{F}}, d_i),$$

where n_i is a name, $\sigma_{\mathcal{F}}$ is a predefined input–output signature shared within the task family, and d_i is a natural-language specification of the subroutine. Induction determines the subroutines included in the library and produces their names and specifications. For example, a subroutine for narrative reasoning may assess the evidence concerning a particular suspect, taking the narrative and an agent-specified focus as inputs and returning a textual assessment.

3.2 Induction Pipeline

Let $\mathcal{T} = \{t_1, \ldots, t_N\}$ be the rollouts of a ReAct agent on the training instances of a task family, with each rollout $t = ((Th_1, A_1, O_1), \ldots, (Th_L, A_L, O_L))$ representing an interleaved sequence of thoughts, actions, and observations. The agent is run once on the training split and its rollouts are analyzed once to induce a library $\mathcal{L}$ with $|\mathcal{L}| \leq K$. The procedure (Figure 3; Appendix C) has four stages.

1. *Filter.* Retain the rollouts whose final answer matches the label, $\mathcal{T}^+ = \{t \in \mathcal{T} : \textsc{Correct}(t)\}$ (using task-specific correctness checks).
2. *Extract.* Collect the per-step ReAct traces.
3. *Categorize and merge.* One LLM call labels each item with a short reasoning-move name (e.g., "verify alibi consistency"); if more than ten distinct labels result, a second call merges synonyms into 5–10 canonical categories.

4. *Top-K synthesis.* Categories are sorted by frequency. For each category with support at least m (default m=3), the first five items in corpus order are shown to an LLM, which emits an agent subroutine name and docstring (n, d); the signature σ is fixed. Synthesis stops at K agent subroutines or when no remaining category reaches m.

Induction draws only on successful training trajectories, so it applies to tasks on which the base agent solves part of the training split. On Natural Plan trip and calendar, the base agent produces 2 and 19 successful trajectories; on trip, accuracy remains near zero for the induced libraries and every evaluated method (Table 15; Appendix L).

3.3 Subroutine Execution

At inference, the induced agent subroutines are exposed as tools, replacing the base task-solving action set with $\mathcal{L} \cup \{\text{FINISH}\}$. Model parameters and the agent's underlying ReAct loop remain unchanged. The agent sees subroutine names, argument schemas, and short descriptions, and selects a subroutine and supplies its runtime arguments. The instance input is bound automatically. A separate LLM call applies the subroutine's full specification d_i to these inputs, without receiving the agent's interaction history. The result is returned as an observation, which the agent incorporates into subsequent reasoning and tool selection (Appendix C).

The learned library determines which subroutines are available and what they compute; the agent decides which subroutines to invoke and with what inputs on each instance. The library remains fixed during evaluation, while the agent selects, repeats, and composes its subroutines as needed to solve each new problem.

4 Experimental Setup

Datasets. Our main evaluation covers five tasks spanning narrative reasoning, rule application, and constraint planning. We additionally evaluate on competition mathematics to examine the method's applicability beyond these task families. We report accuracy on each task using the scoring criteria below. See Appendix L for a discussion of tasks not included in the main evaluation.

- **MuSR** (Sprague et al., 2024). We evaluate three narrative-reasoning tasks: *murder mystery*, identifying the culprit; *object placement*, inferring where a character believes an object is located; and *team allocation*, selecting an assignment of people to tasks under competing requirements. Instances are multiple choice, with two options on murder mystery, two to five on object placement, and three on team allocation; each is scored by exact match to the correct choice.
- **RuleArena NBA** (Zhou et al., 2025). This task evaluates rule-guided reasoning about proposed roster operations under the NBA Collective Bargaining Agreement. The relevant rule text is provided in the input. Each instance has a Boolean reference answer, and predicted compliance verdicts are scored by Boolean agreement.
- **Natural Plan: Meeting Planning** (Zheng et al., 2024). The agent constructs a meeting schedule subject to availability and travel-time constraints. A deterministic verifier checks schedule feasibility and counts the feasible meetings. A prediction is scored as correct when this count matches the reference schedule's.
- **AIME.** We evaluate on 43 problems from AIME 2023–2024; training trajectories are collected on 75 problems from AIME 1983–2020, so training and test problems do not overlap. Each problem requires an integer answer between 0 and 999 and is scored by exact match.

Baselines. We compare against LLM prompting techniques (CoT and PoT), a ReAct agent (ReAct), and methods that enable agents to learn from experience (AWM, ACE, and ASI), which we call experience-adapted methods. Adaptation details are in Appendix D.

- **Chain-of-Thought (CoT)** (Wei et al., 2022) uses zero-shot prompting to generate a reasoning trace and answer in one model call, without external tools.

Table 1: **Baseline methods compared in our experiments.** Settings, learned artifacts, and execution modes. Experience-adapted methods use ReAct as the base agent. Dashes indicate no learned artifact.

Method	Setting	Learned artifact	Execution
CoT	Single LLM call	—	Natural-language reasoning
PoT	Single LLM call	—	Program execution
ReAct	Agent	—	Reasoning and tool use
AWM	Experience-adapted agent	Workflows	Contextual guidance
ACE	Experience-adapted agent	Playbook	Contextual guidance
ASI	Experience-adapted agent	Programmatic skills	Code execution
Ours	Experience-adapted agent	Agent-subroutine specifications	LLM execution

- **Program-of-Thoughts (PoT)** (Chen et al., 2023) generates a Python program in one model call; sandboxed execution with standard-library support produces the answer.
- **ReAct** (Yao et al., 2023a) interleaves reasoning with calls to the task's base tools. This unadapted agent also supplies the training trajectories for our induction procedure.
- **Agent Workflow Memory (AWM)** (Wang et al., 2025a) induces natural-language workflows from the same model and source training-rollout corpus as our method, and supplies them as contextual guidance.
- **Agentic Context Engineering (ACE)** (Zhang et al., 2026) updates a natural-language playbook through generation, reflection, and curation during offline adaptation. The playbook is frozen and supplied as context during testing.
- **Agent Skill Induction (ASI)** (Wang et al., 2025b) induces executable Python skills from successful training interactions and verifies them through execution. Accepted skills become available during subsequent training interactions, and the final library is fixed during testing.

Both DeepSeek-V3 and Gemini 2.5 Flash are evaluated against all six baselines. Table 1 summarizes the experimental configurations.

Models and evaluation protocol. We use DeepSeek-V3 and Gemini 2.5 Flash for the main evaluation. We collect one ReAct rollout per training instance and induce our libraries from correct rollouts only. For the main part of our experiments, we use the same model to generate training trajectories, induce the library, and execute the resulting agent on test instances, without a stronger teacher model. The knowledge-format ablation uses both models; the base ReAct ablation uses Gemini 2.5 Flash; the rest of sensitivity sweeps use DeepSeek-V3. For AIME, we use Gemini 3.1 Flash-Lite and Gemini 2.5 Flash. To evaluate cross-model transfer, we deploy libraries induced by Gemini 2.5 Flash directly on Gemini 3.5 Flash-Lite without any modification.

5 Results

5.1 Main results

Table 2 reports held-out accuracy across five tasks. As shown, induced libraries achieve the highest average accuracy among the evaluated methods on both models: 67.4% on DeepSeek-V3 and 77.7% on Gemini 2.5 Flash. These exceed the strongest baseline by five-task average on each model—ACE and AWM, respectively—by 10.8 and 6.1 percentage points.

Relative to the ReAct agents that supplied the trajectories, average accuracy rises from 38.8% to 67.4% on DeepSeek-V3 and from 68.2% to 77.7% on Gemini 2.5 Flash, and the induced libraries score at or above zero-shot CoT. On Gemini 2.5 Flash, ARISE outperforms ASI, which learns code artifacts, on all five tasks and by 11.5 percentage points on average. ACE costs 3.6× as much in total ($13.14 vs. $3.63 per task; Figure 4) and scores 10.1 points lower; AWM is the cheapest experience-adapted method and scores 6.1 points lower. On DeepSeek-V3, the induced library has the highest average total cost in Table 12 ($5.23 per task), and NBA alone costs $21.26. ARISE is the only experience-adapted method that matches or exceeds the ReAct agent in all ten model–task

Table 2: **Main test-set results.** Test accuracy (%). Bold marks the highest value per column within each model; Avg. is the unweighted five-task mean. Ours as skill and Ours as code deliver the same induced library as retrieved skill documents or Python tools (Section 5.2). †: served through a different provider (Appendix D).

		MuSR			RuleArena	NatPlan	Avg.
Method	**Learned**	murder	object	team	NBA	meeting	
DeepSeek-V3							
CoT	–	60.0	39.6	65.0	48.5	35.0	49.6
PoT	–	67.0	60.4	**69.0**	40.9	27.0	52.9
ReAct	–	45.0	50.0	38.0	28.8	32.0	38.8
AWM	sample workflows	47.0	53.8	36.0	59.1	43.0	47.8
ACE	context playbook	**77.0**	50.0	58.0	62.1	36.0	56.6
ASI	code skill	71.0	50.0	55.0	47.0	26.0	49.8
Ours as skill†	skill document	67.0	47.2	64.0	43.9	37.0	51.8
Ours as code†	code tool	66.0	57.5	65.0	34.8	18.0	48.3
Ours (ARISE)	agent subroutine	75.0	**68.9**	68.0	**74.2**	**51.0**	**67.4**
Gemini 2.5 Flash							
CoT	–	80.0	57.5	73.0	62.1	74.0	69.3
PoT	–	72.0	72.6	60.0	56.1	41.0	60.3
ReAct	–	78.0	50.9	**76.0**	65.2	71.0	68.2
AWM	sample workflows	77.0	51.9	67.0	**74.2**	**88.0**	71.6
ACE	context playbook	80.0	72.6	70.0	65.2	50.0	67.6
ASI	code skill	75.0	62.3	59.0	69.7	65.0	66.2
Ours as skill	skill document	78.0	50.9	71.0	66.7	64.0	66.1
Ours as code	code tool	82.0	61.3	68.0	72.7	65.0	69.8
Ours (ARISE)	agent subroutine	**86.0**	**75.5**	**76.0**	71.2	80.0	**77.7**

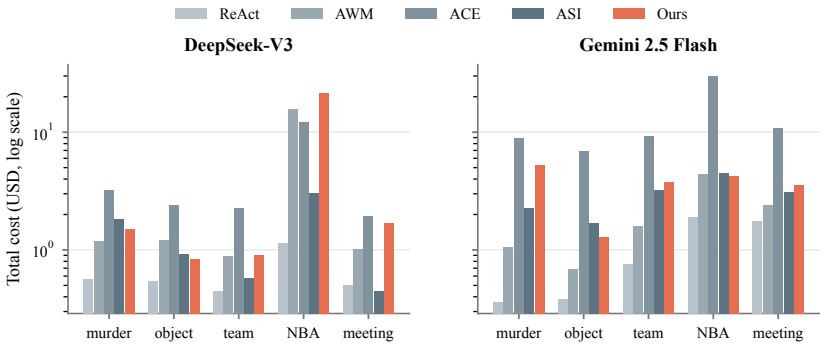


Figure 4: **Total cost per task.** US dollars on a log scale for one offline adaptation plus test-set inference; values from Table 12.

settings; on Gemini 2.5 Flash, the prose and code artifacts of ACE and ASI average below ReAct (67.6% and 66.2% vs. 68.2%), while ARISE raises it to 77.7%.

5.2 Ablation and Sensitivity Analysis

Knowledge format analysis. The *Ours as skill* and *Ours as code* rows of Table 2 hold each model's induction pipeline and library fixed and change only how the induced knowledge is delivered: as skill documents the agent retrieves by function call, or as Python tools rewritten from each specification (Appendix C.2). The skill text is identical to the subroutine specification, so the gap between the

Ours as skill row and the ARISE row isolates executing a procedure in a separate LLM call from reading the same text in the agent's own context. Agent subroutines score higher than skills in all ten model–task settings and higher than code in nine; average accuracy rises from 51.8% (skill) and 48.3% (code) to 67.4% on DeepSeek-V3, and from 66.1% and 69.8% to 77.7% on Gemini 2.5 Flash, where code is 1.5 percentage points higher on NBA. On Gemini 2.5 Flash, delivered as skills, the induced text lowers accuracy below the ReAct agent that produced the trajectories (66.1% vs. 68.2%); delivered as agent subroutines, the same text raises it to 77.7%. Whether induced knowledge helps at all depends on its execution interface.

Content ablation. Keeping the execution interface but removing the induced content has the opposite effect: replacing the specifications with a generic instruction, or additionally replacing the tool names, lowers the Gemini 2.5 Flash five-task average from 78.4% to 67.7% and 64.2%, below the 68.2% of the ReAct agent (Appendix F).

Sensitivity to base ReAct settings. We collect training trajectories under two tool settings for the base ReAct agent: (a) the generic text-search tools of the main table, which return matching sentences from the input, and (b) hand-designed tools written by experts for each task. At test time both settings replace their collection tools with the induced library. Libraries induced under both settings exceed CoT and ReAct on both tasks (Table 3), and libraries induced with generic tools match those induced with expert-designed tools (86.0% vs. 87.0% on murder; 75.5% vs. 70.8% on object): ARISE requires no expert tool design.

Table 3: **Induction under different trace-collection tool sets.** Test accuracy (%) using libraries induced from successful trajectories collected with generic or hand-designed tools.

	MuSR	
Method / setting	**Murder**	**Object**
CoT	80.0	57.5
ReAct	78.0	50.9
(a) generic	86.0	**75.5**
(b) hand-designed	**87.0**	70.8

Sensitivity to hyperparameter choice. Every maximum library size $K \in \{3, 5, 10\}$ exceeds the DeepSeek-V3 ReAct agent on all four tasks; $K{=}3$ and $K{=}5$ differ by at most 7.0 points per task, and $K{=}10$ is never best. Full discussion and detailed results are reported in Appendix G.

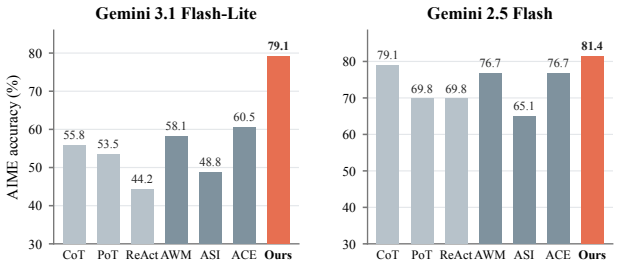
5.3 Extension to competition mathematics

We further evaluate the induction procedure on competition mathematics by learning a new library from successful AIME training trajectories. With Gemini 3.1 Flash-Lite, the induced library achieves 79.1% accuracy, compared with 60.5% for ACE, 58.1% for AWM, 55.8% for CoT, 53.5% for PoT, 48.8% for ASI, and 44.2% for ReAct (Table 4, Figure 5). AWM and our method reuse the same source training-rollout corpus. Our method exceeds AWM by 21.0 percentage points.

With Gemini 2.5 Flash, a library induced separately from that model's successful trajectories achieves 81.4% accuracy, compared with 79.1% for CoT, 76.7% for AWM and ACE, 69.8% for both PoT and ReAct, and 65.1% for ASI. The induced-library agent thus achieves the highest accuracy among the evaluated methods on both models, although the gain over the strongest baseline is smaller on Gemini 2.5 Flash. These results extend the evidence for learning reusable reasoning procedures to mathematical problem solving.

5.4 Cross-model transfer

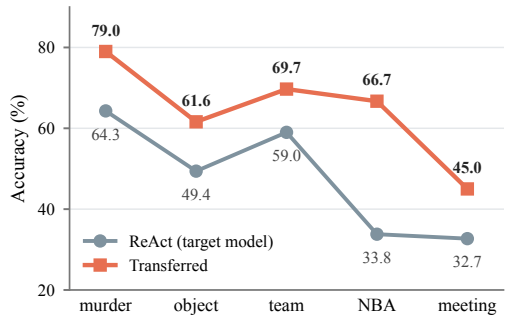
We deploy the libraries learned by Gemini 2.5 Flash on Gemini 3.5 Flash-Lite without modifying their content. Relative to Gemini 3.5 Flash-Lite's ReAct agent, the transferred libraries improve accuracy on all five tasks, with gains ranging from 10.7 percentage points on team allocation to 32.8 percentage points on NBA (Table 5, Figure 6). Cross-model transfer results show that the induced libraries are not model-specific, and can be used by other models without modification.



Method	3.1 Flash-Lite	2.5 Flash
CoT	55.8	79.1
PoT	53.5	69.8
ReAct	44.2	69.8
AWM	58.1	76.7
ASI	48.8	65.1
ACE	60.5	76.7
Ours (ARISE)	**79.1**	**81.4**

Figure 5: **AIME accuracy by model and method.** Each panel plots one column of Table 4 (43 AIME 2023–2024 problems); within each model, AWM and the induced library reuse the same source training trajectories.

Table 4: **Competition mathematics results.** Accuracy (%) of Gemini 3.1 Flash-Lite and Gemini 2.5 Flash on the same 43 AIME 2023–2024 problems. Each model induces its own library. Bold marks the highest accuracy per model.



Task	ReAct	Transferred	Δ (pp)
murder	64.3	**79.0**	+14.7
object	49.4	**61.6**	+12.3
team	59.0	**69.7**	+10.7
NBA	33.8	**66.7**	+32.8
meeting	32.7	**45.0**	+12.3

Figure 6: **Cross-model transfer by task.** Gemini 3.5 Flash-Lite accuracy with its own ReAct agent and with libraries induced by Gemini 2.5 Flash; values from Table 5.

Table 5: **Cross-model transfer of induced libraries.** Accuracy (%) of Gemini 3.5 Flash-Lite using libraries induced by Gemini 2.5 Flash and deployed without modification, compared with the target model's ReAct agent. Δ denotes the paired accuracy gain in percentage points, computed from unrounded paired accuracies.

6 Discussion

Learning from agent experience has so far produced two kinds of artifacts: prose placed in the agent's context, such as workflows and playbooks, and code that the agent executes, such as programmatic skills. ARISE introduces a third: a natural-language procedure behind a callable interface, executed by its own LLM call. It keeps procedures in natural language, as prose methods do, and gives each one its own invocation, as code methods do. Across both models, it yields the highest five-task average among all evaluated methods and exceeds chain-of-thought in all ten model–task settings of the main evaluation. The gain comes from the execution interface. With the induced text held fixed, executing it as agent subroutines scores 11.6 and 15.6 percentage points above retrieving it as skills and 7.9 and 19.1 points above rewriting it as Python tools, on Gemini 2.5 Flash and DeepSeek-V3. The value of learned knowledge therefore depends on how it is executed as well as on what it says. The learned procedures do not depend on expert-designed tools: libraries induced with generic tools match those induced with hand-designed ones. Libraries transfer to Gemini 3.5 Flash-Lite without modification and improve it on all five tasks, and induced libraries give the highest AIME accuracy on both evaluated models. Agent subroutines thus provide a reusable form of experience that requires neither parameter updates nor expert tool design.

AI USE STATEMENT

We used generative AI tools to implement methods and experiment code and to clean and reformat datasets. We did not use generative AI tools to design or provide feedback on research methodology or experiments, develop theoretical models or conceptual frameworks, interpret results, propose hypotheses, formulate mathematical claims, generate synthetic data, or translate text; tasks involving proofs and qualitative data analysis are not applicable to this work. We also used generative AI tools to create and edit figures, edit the paper for readability, and format references. All AI-generated code was reviewed and tested by the authors, every number in the paper was computed from experiment outputs, and every reference was checked against its published version. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work evaluates on publicly available benchmarks (MUSR, RuleArena, Natural Plan, and AIME) and involves no human subjects or personally identifiable data. Induced libraries are natural-language specifications executed by the same LLMs; model parameters are not modified.

REPRODUCIBILITY STATEMENT

The induction prompts, the per-benchmark configurations, the induced libraries, and the AWM workflows used in every reported cell are included in the supplementary material; Appendix C lists the ReAct agent configuration and decoding settings. The code is included in the supplementary material.

REFERENCES

M. Bowers, T. X. Olausson, L. Wong, G. Grand, J. B. Tenenbaum, K. Ellis, and A. Solar-Lezama. Top-down synthesis for library learning. In *POPL*, 2023.

T. Cai, X. Wang, T. Ma, X. Chen, and D. Zhou. Large language models as tool makers. In *ICLR*, 2024.

W. Chen, X. Ma, X. Wang, and W. W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *TMLR*, 2023.

M. Chen, Y. Li, Y. Yang, S. Yu, B. Lin, and X. He. AutoManual: Constructing instruction manuals by LLM agents via interactive environmental learning. In *NeurIPS*, 2024.

A. Didolkar, N. Ballas, S. Arora, and A. Goyal. Metacognitive reuse: Turning recurring LLM reasoning into concise behaviors. *arXiv preprint arXiv:2509.13237*, 2025.

K. Ellis, L. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI*, 2021.

L. E. Erdogan, N. Lee, S. Kim, S. Moon, H. Furuta, G. Anumanchipalli, K. Keutzer, and A. Gholami. Plan-and-Act: Improving planning of agents for long-horizon tasks. In *ICML*, 2025.

L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. PAL: Program-aided language models. In *ICML*, 2023.

G. Grand, L. Wong, M. Bowers, T. X. Olausson, M. Liu, J. B. Tenenbaum, and J. Andreas. LILO: Learning interpretable libraries by compressing and documenting code. In *ICLR*, 2024.

S. Hu, C. Lu, and J. Clune. Automated design of agentic systems. In *ICLR*, 2025.

O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan A, S. Haq, A. Sharma, T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *ICLR*, 2024.

T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *ICLR*, 2023.

J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. In *IEEE ICRA*, 2023.

A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark. Self-Refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

J. Ni, Y. Liu, X. Liu, Y. Sun, M. Zhou, P. Cheng, D. Wang, E. Zhao, X. Jiang, and G. Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *EMNLP*, 2024.

C. Qian, C. Han, Y. R. Fung, Y. Qin, Z. Liu, and H. Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of EMNLP*, 2023.

Y. Qu, A. Singh, Y. Lee, A. Setlur, R. Salakhutdinov, C. Finn, and A. Kumar. RLAD: Training LLMs to discover abstractions for solving reasoning problems. In *ICLR*, 2026.

G. Sarch, L. Jang, M. J. Tarr, W. W. Cohen, K. Marino, and K. Fragkiadaki. VLM agents generate their own memories: Distilling experience into embodied programs of thought. In *NeurIPS*, 2024.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.

N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

C. Snell, J. Lee, K. Xu, and A. Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *ICLR*, 2025.

Z. Sprague, X. Ye, K. Bostrom, S. Chaudhuri, and G. Durrett. MuSR: Testing the limits of chain-of-thought with multistep soft reasoning. In *ICLR*, 2024.

M. Suzgun and A. T. Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

M. Suzgun, M. Yuksekgonul, F. Bianchi, D. Jurafsky, and J. Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *EACL*, 2026.

X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023.

L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *ACL*, 2023.

G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *TMLR*, 2024.

Z. Wang, G. Neubig, and D. Fried. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *ICML*, 2024.

Z. Z. Wang, J. Mao, D. Fried, and G. Neubig. Agent workflow memory. In *ICML*, 2025.

Z. Z. Wang, A. Gandhi, G. Neubig, and D. Fried. Inducing programmatic skills for agentic tasks. In *COLM*, 2025.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

L. Yang, Z. Yu, T. Zhang, S. Cao, M. Xu, W. Zhang, J. E. Gonzalez, and B. Cui. Buffer of thoughts: Thought-augmented reasoning with large language models. In *NeurIPS*, 2024.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *NeurIPS*, 2023.

L. Yuan, Y. Chen, X. Wang, Y. R. Fung, H. Peng, and H. Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *ICLR*, 2024.

M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, P. Lu, Z. Huang, C. Guestrin, and J. Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025.

J. Zhang, J. Xiang, Z. Yu, F. Teng, X. Chen, J. Chen, M. Zhuge, X. Cheng, S. Hong, J. Wang, B. Zheng, B. Liu, Y. Luo, and C. Wu. AFlow: Automating agentic workflow generation. In *ICLR*, 2025.

Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li, U. Thakker, J. Zou, and K. Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *ICLR*, 2026.

A. Zhao, D. Huang, Q. Xu, M. Lin, Y.-J. Liu, and G. Huang. ExpeL: LLM agents are experiential learners. In *AAAI*, 2024.

H. S. Zheng, S. Mishra, H. Zhang, X. Chen, M. Chen, A. Nova, L. Hou, H.-T. Cheng, Q. V. Le, E. H. Chi, and D. Zhou. NATURAL PLAN: Benchmarking LLMs on natural language planning. *arXiv preprint arXiv:2406.04520*, 2024.

B. Zheng, M. Y. Fatemi, X. Jin, Z. Z. Wang, A. Gandhi, Y. Song, Y. Gu, J. Srinivasa, G. Liu, G. Neubig, and Y. Su. SkillWeaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

P. Zhou, J. Pujara, X. Ren, X. Chen, H.-T. Cheng, Q. V. Le, E. H. Chi, D. Zhou, S. Mishra, and H. S. Zheng. Self-Discover: Large language models self-compose reasoning structures. In *NeurIPS*, 2024.

A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *ICML*, 2024.

R. Zhou, W. Hua, L. Pan, S. Cheng, X. Wu, E. Yu, and W. Y. Wang. RuleArena: A benchmark for rule-guided reasoning with LLMs in real-world scenarios. In *ACL*, 2025.

The appendices report the induced libraries, implementation and baseline details, ablations, cost, and statistical analyses.

A Induced Agent Subroutines

Below we present the induced agent subroutines evaluated in Table 2 for both models. Induction is intended to capture common reasoning moves shared across instances of a task. Independently induced libraries exhibit recurring task-relevant agent subroutines despite differences in their names and granularity. A detailed comparison is in Appendix B. These examples illustrate the level at which induction operates.

A.1 DeepSeek-V3 libraries

We summarize the DeepSeek-V3 libraries produced by the induction procedure in Section 3.2, with a maximum size of $K{=}5$. The five main tasks have five agent subroutines each; the supplementary trip library evaluated in Table 15 has two. The descriptions below summarize the agent subroutines; full specifications are retained in the library artifacts. Signatures are fixed by the synthesis prompt: `(narrative: str, focus: str)` for MuSR, `(context: str, focus: str)` for RuleArena, `(prompt: str, focus: str)` for NatPlan; all return `str`.

A.1.1 MuSR murder

- `investigate_evidence`: analyze evidence for a focused suspect, weapon, or location across categories means / motive / opportunity / alibi consistency / physical evidence; flag hard disqualifiers and conflicting witnesses.
- `initialize_suspect_analysis`: identify the victim and all suspects from the narrative; collect initial clues per suspect; output a structured JSON of victim, suspects, and per-suspect initial evidence.
- `determine_if_killer`: decide whether the focused suspect is the killer based on means, motive, opportunity, alibi consistency, and physical evidence; return `'1'` or `'0'`.
- `summarize_investigation_progress`: produce a structured per-suspect summary of case status with confidence levels, contradictions, and recommended next investigation steps.
- `evaluate_suspect_evidence`: weigh evidence for or against a focused suspect; produce a categorized list (motive / means / opportunity / physical evidence / alibi) and an overall assessment of evidence strength.

A.1.2 MuSR object placement

- `formulate_search_query`: extract entities and actions from the focus phrase to construct a precise search string targeting a specific narrative segment about object movement.
- `analyze_character_knowledge`: track when a character was present or absent during object movements; output the character's known locations, last-known location, and reasoning over knowledge gaps.
- `extract_belief_location`: determine where a target character believes a specific object is located, accounting for theory-of-mind divergence between actual location and the character's mental model.
- `infer_character_belief`: infer a character's belief about object location from observations and timing of arrivals/departures relative to movements; return believed location plus step-by-step reasoning.
- `track_object_movement_timeline`: reconstruct a chronological timeline of object movements with witnesses and visibility annotations per event.

A.1.3 MuSR team allocation

- `analyze_and_summarize_attributes`: extract per-person skills, weaknesses, and interpersonal dynamics from the narrative; bullet-point summary highlighting hard disqualifiers and tricky trade-offs.
- `seek_specific_individual_information`: extract structured per-person profiles (strengths, weaknesses, characteristics, constraints) with attention to comparative language and conflicts.
- `define_task_requirements`: extract task list, required skills per task, hard/soft constraints, and explicit/implicit objectives; flag disqualifying conditions.
- `analyze_team_allocation`: produce structured analysis of people-tasks-constraints suitable for downstream allocation reasoning, including inferred skills and relative strength comparisons.
- `make_final_allocation_decision`: select the best assignment option index given evaluated constraints; return integer index of the chosen option.

A.1.4 RuleArena NBA

- `search_cba_rule`: locate relevant CBA rule text matching a focused term or section; return excerpt with section reference (e.g., "Article XI, Section 4(b)").
- `analyze_operation_compliance`: assess whether a proposed trade, signing, or offer satisfies CBA constraints given current salary cap and team status; output compliance status, violated rules, and conditions for compliance.

- `search_cba_term`: case-insensitive search for a defined CBA term, contract type, or threshold; return all matching context lines.
- `search_cba_section`: locate a specific section identifier (e.g., "Section 8(e)(1)") and return its full text with surrounding context.
- `search_cba_exception`: locate the named exception (e.g., Traded Player Exception, MLE) and return eligibility rules, calculation method, and stacking restrictions.

A.1.5 NATPLAN MEETING

- `define_meeting_problem`: parse start location/time, friends with locations / availability windows / minimum durations, and the travel-time matrix; output structured JSON.
- `request_travel_times`: query the travel-time matrix for pairs involving a focused person or location; return a list of relevant travel durations.
- `propose_meeting_schedule`: synthesize a time-ordered schedule that maximizes friends met, respecting windows and travel; output sequence of arrival/departure events with travel segments.
- `verify_constraints`: extract and validate constraints associated with a focused person or location; report constraint violations and missing information.
- `analyze_constraints_and_requirements`: assess feasibility for the focused friend—earliest arrival, meeting window, minimum duration; output a feasibility verdict with timeline.

A.1.6 NATPLAN TRIP ($|\mathcal{L}|$=2)

- `identify_constraints`: extract and analyze required cities, stay durations, total trip length, time windows, and flight-connectivity constraints; organize the findings by constraint type and flag possible conflicts.
- `analyze_travel_network`: analyze direct flight connections and potential routes in relation to required stays and time windows; summarize connectivity options and constraints affecting travel between cities.

A.2 GEMINI 2.5 FLASH LIBRARIES

We summarize the Gemini 2.5 Flash libraries used for the five main tasks. Murder, object, and team have five agent subroutines each, NBA has four, and meeting has one. The maximum library size is K=5, not a requirement to produce exactly five agent subroutines. Signatures follow Appendix A.1.

A.2.1 MUSR MURDER

- `investigate_means_and_weapon`: identify the murder weapon and assess the focused suspect's access to or skill with it (or, for a focused weapon, the characters linked to it), noting when no link exists; return a JSON object keyed by weapon and means.
- `formulate_killer_conclusion`: name the most likely killer by weighing motive, means, opportunity, alibi consistency, physical evidence, and counter-evidence.
- `examine_suspect_profile`: extract a suspect's background, financial situation, relationships, history, and connection to the victim; return a JSON summary per category.
- `investigate_motive_and_intent`: identify and evaluate motives (financial gain, revenge, secrets, power, jealousy, fear) and the nature of intent (premeditated, impulsive, accidental) for the focused person, with supporting snippets.
- `verify_narrative_facts`: search the narrative for a specific fact or detail; return `found`, matching snippets, and a summary message as JSON.

A.2.2 MUSR OBJECT PLACEMENT

- `retrieve_entity_details`: return the sentences that explicitly mention the focused entity, without inference.

- `formulate_character_belief`: simulate the focused character's knowledge state over object movements, entrances, and exits; output "The most likely place {character} would look to find the {object} is {location}."
- `get_character_object_belief`: track every object movement and the character's presence or awareness at each; deduce the believed location, which may differ from the actual one.
- `infer_character_beliefs`: track object movements and the character's presence, together with routines and roles that shape expectation; return object, character, believed location, and step-by-step reasoning.
- `identify_character_expected_location`: return the last object location the character observed or was told of, with the focus given as "character, object"; output a one-sentence answer.

A.2.3 MuSR team allocation

- `retrieve_general_information`: summarize everything in the narrative pertinent to the focus (a person's attributes or a task's requirements), flagging hard disqualifiers and implicit cues.
- `retrieve_person_profile_details`: summarize the strengths, weaknesses, skills, experience, traits, and interpersonal dynamics of the focused people, related to task fit.
- `propose_and_finalize_allocation`: propose an allocation of people to tasks and justify it against attributes, dynamics, task requirements, and constraints, including why alternatives are worse.
- `match_skills_to_tasks`: find the textual evidence linking a person to the skill, task, or characteristic named in the focus.
- `retrieve_relationship_info`: return the exact sentences describing a specified relationship or interaction between individuals.

A.2.4 RuleArena NBA ($|\mathcal{L}|$=4)

- `search_cba_rules`: return the verbatim CBA snippet matching the focused term (definitions, exceptions, thresholds).
- `explain_nba_cba_violations`: identify each violation in the proposed operations, cite the CBA section, and give the rule and the calculation.
- `check_nba_cba_compliance`: evaluate each proposed operation against cap and roster rules by operation type, player status, and team cap status; cite sections for compliant and non-compliant operations.
- `provide_compliance_verdict`: give the final verdict on the cumulative effect of all proposed operations on cap, luxury tax, roster limits, and exceptions.

A.2.5 NatPlan meeting ($|\mathcal{L}|$=1)

- `generate_meeting_schedule`: from the start location and time, each friend's location, availability window, and minimum duration, and the travel-time matrix, produce a chronological step-by-step schedule that maximizes the number of friends met.

B Comparing Libraries Across Inducing Models

Table 6 compares libraries independently induced by DeepSeek-V3 and Gemini 2.5 Flash from their own successful trajectories. Although the models produce different names and subroutine boundaries, their specifications recover related reasoning functions. The correspondence is semantic: agent subroutines address the same recurring demands of the task, rather than reproducing an identical list of tools.

Object placement. Both models recover procedures for tracking character knowledge and inferring beliefs. DeepSeek-V3's `analyze_character_knowledge` tracks which object movements a character could observe, while `extract_belief_location` and `infer_character_belief` use that information to infer the character's believed location. Gemini's `get_character_object_belief` and `formulate_character_belief` similarly reconstruct knowledge from movements, observations, and entrances or exits. Both libraries therefore capture the distinction between an object's actual location and the location supported by a particular character's information. DeepSeek also exposes timeline reconstruction as a separate agent subroutine; Gemini incorporates this reasoning into its belief-inference specifications.

NBA: retrieving rules and assessing compliance. Both libraries contain agent subroutines for finding relevant CBA rules and evaluating proposed roster operations against them. DeepSeek separates retrieval by rule, term, section, and exception, alongside `analyze_operation_compliance`. Gemini uses a broader `search_cba_rules` agent subroutine and exposes compliance assessment, violation explanation, and a final verdict separately. The shared reasoning functions are rule retrieval and rule application; the models differ in where they place the boundaries between subroutines.

Murder: organizing and evaluating suspect evidence. Both libraries organize the investigation around suspects and evidence concerning means, motive, opportunity, and alibis. DeepSeek's `investigate_evidence` and `evaluate_suspect_evidence` combine several evidence categories within a call. Gemini exposes means and weapon analysis, motive analysis, and suspect profiling as separate agent subroutines, before forming a killer conclusion. The specifications thus recover related evidential reasoning while choosing different levels of specialization.

Shared functions, flexible granularity. These correspondences do not require a fixed decomposition. For meeting planning, Gemini induces one schedule-generation subroutine, whereas DeepSeek distributes parsing, planning, and constraint assessment across several subroutines. The commonality lies in the task-relevant reasoning functions recovered, not a one-to-one match between subroutine names or counts. This comparison complements Section 5.4: independent induction reveals related procedural content across models, while cross-model execution demonstrates that a fixed library can remain useful beyond the model that produced it. Together, the two forms of evidence support interpreting the libraries as reusable representations of task-level reasoning.

Table 6: **Libraries independently induced by two models.** Each model uses its own successful trajectories. The inventories recover related task-level reasoning functions while differing in names and granularity; the accompanying analysis explains these semantic correspondences. These are the libraries evaluated in Table 2.

Task	DeepSeek-V3	Gemini 2.5 Flash
murder	`investigate_evidence`, `initialize_suspect_analysis`, `determine_if_killer`, `summarize_investigation_progress`, `evaluate_suspect_evidence`	`investigate_means_and_weapon`, `formulate_killer_conclusion`, `examine_suspect_profile`, `investigate_motive_and_intent`, `verify_narrative_facts`
object	`formulate_search_query`, `analyze_character_knowledge`, `extract_belief_location`, `infer_character_belief`, `track_object_movement_timeline`	`retrieve_entity_details`, `formulate_character_belief`, `get_character_object_belief`, `infer_character_beliefs`, `identify_character_expected_location`
team	`analyze_and_summarize_attributes`, `seek_specific_individual_information`, `define_task_requirements`, `analyze_team_allocation`, `make_final_allocation_decision`	`retrieve_general_information`, `retrieve_person_profile_details`, `propose_and_finalize_allocation`, `match_skills_to_tasks`, `retrieve_relationship_info`
NBA	`search_cba_rule`, `search_cba_term`, `search_cba_section`, `search_cba_exception`, `analyze_operation_compliance`	`search_cba_rules`, `explain_nba_cba_violations`, `check_nba_cba_compliance`, `provide_compliance_verdict`
meeting	`define_meeting_problem`, `request_travel_times`, `propose_meeting_schedule`, `verify_constraints`, `analyze_constraints_and_requirements`	`generate_meeting_schedule`

C Implementation Details

Scoring. MUSR uses exact answer-index agreement, NBA uses Boolean agreement, and AIME uses exact agreement of the extracted integer answer. Exceptions and missing answers count as incorrect. Meeting and trip results use a deterministic verifier to evaluate plans against their constraints. For meeting, the verifier simulates the parsed itinerary and compares its number of feasible meetings with that of the reference plan; for trip, it checks city durations, flight connections, and time constraints.

C.1 Induction and execution

Induction prompts. The fixed pipeline labels extracted items with short reasoning-operation names, merges synonymous labels, and synthesizes a named agent subroutine with a short description and full natural-language specification. Category support must be at least $m=3$; the default maximum library size is $K=5$. Synthesis uses the first five items of each category in corpus order. The same stages are used across benchmarks, with task descriptions and input argument names supplied as prompt parameters. Appendix G reports the trace-filter ablation and the ablation of full specifications in the agent's prompt.

Deployment. We replace the base action set with the induced library and FINISH. The agent sees agent subroutine names, argument schemas, and short descriptions. A separate LLM call executes each full specification on the runtime arguments and bound instance input, without receiving the agent's interaction history. Its output becomes an observation in that history. The library is fixed during test evaluation; the agent chooses when to call, repeat, and combine agent subroutines.

Decoding and request budgets. The Gemini main-grid repeat evaluations, the ACE and ASI baselines, knowledge-format ablations, and AIME evaluations use $T=0$. The default ReAct limit is 50 model requests, with 200 for the Gemini 2.5 Flash NBA agent configurations, including ReAct, AWM, ASI, ACE, and the induced-library agent. These are limits on the outer agent's requests, not on tool invocations or all nested LLM calls. AIME's direct and agent subroutine execution calls allow

up to 65 536 output tokens. PoT generates a program in one model call; the Gemini 2.5 Flash NBA PoT evaluation uses a 600-second call timeout.

C.2 KNOWLEDGE FORMAT COMPARISONS

In this experiment, the base ReAct agent is the same across three settings (skill, code and agent subroutines). The prompt and the harness are held fixed. Only the format of the induced knowledge differs. The agent sees the same short names and argument schemas in all three settings, but the full specifications are delivered in different formats.

Skill. The agent retrieves the induced skills through function calls. Each skill's text is identical to the corresponding agent subroutine's specification.

Code. An agent rewrote each subroutine specification as a standard-library Python function that performs regex and keyword matching over the task input; calling the tool runs this function. A function was accepted only if it ran without errors on three training instances and produced varied outputs; brittle functions and functions that failed to generalize were rejected.

D BASELINE ADAPTATION

All three experience-adapted baselines share the setting of ARISE: the same base ReAct agent and base tools, the same training split, the same model for adaptation and testing, and the same test split and request limits. Each learned artifact is frozen after adaptation. All three use gold labels during adaptation, as ARISE does through its correctness filter; ACE and ASI receive them as feedback on every update.

AWM. We induce one workflow per task with a single LLM call over up to 20 successful training rollouts of the base ReAct agent, with each step truncated to its thought and action. The prompt asks for a numbered workflow of 5–10 steps. At test time the workflow is prepended to the agent's prompt as a learned guide.

ACE. ACE adapts a playbook through generator, reflector, and curator roles (Zhang et al., 2026). The generator is the base ReAct agent with the current playbook; the reflector analyzes each trajectory together with the gold answer; the curator adds up to three bullets per problem, merged deterministically with helpful/harmful counters, near-duplicate removal, and a 10K-token budget. Adaptation runs five epochs over the training split (375 updates; 350 for meeting), five passes compared with the single pass of rollouts that ARISE analyzes, and yields playbooks of 88–266 bullets. The final playbook is supplied in the agent's prompt at test time.

ASI. ASI makes one online pass over the training split (Wang et al., 2025b). After each correct episode, one LLM call induces up to three Python skills and rewrites the trajectory with them; a skill is admitted only if re-solving the task with it reaches the correct answer, calls a newly induced skill, and raises no skill error. Episode correctness is judged against the gold label. Skills run in a sandbox and are available as tools in later episodes; final libraries contain 1–27 skills. At test time the skills are added to the agent's base tools.

DeepSeek-V3 serving. The DeepSeek-V3 ACE, ASI, skill-delivery, and code-delivery runs use DeepSeek-V3 served through OpenRouter, as our original provider no longer serves the model; all other DeepSeek-V3 rows use the original provider.

E INDUCTION UNDER DIFFERENT BASE REACT SETTINGS

Table 3 provides results for induction under different base ReAct settings. Here we provide more details on the hand-designed tools and fine-grained results.

E.1 HAND-DESIGNED COLLECTION TOOLS

The hand-designed condition uses the following three agent subroutines per task. They were authored for this study and executed through the same LLM execution interface as the induced agent subroutines. They are available only during trace collection in this ablation and are not added to its test-time libraries.

MUSR murder.

- `extract_suspects_and_evidence`: extract victim, crime details, and per-suspect evidence (motive, means, opportunity, alibi claim, alibi witnesses, suspicious behavior, physical evidence).
- `verify_alibis`: cross-reference each suspect's alibi against the narrative; identify alibi gaps, contradictions, and corroborating evidence.
- `deduce_murderer`: synthesize evidence and verified alibis to identify the perpetrator; weight physical evidence and alibi contradictions over motive alone.

MUSR object placement.

- `extract_movements`: enumerate every object-movement event chronologically with actor, source, destination, and lists of present/absent witnesses.
- `extract_discoveries`: list incidental observations where a character learns an object's location without witnessing the move (saw directly, told by, inferred from event).
- `infer_belief`: combine movements, discoveries, and a re-read of the narrative to determine where the target person believes the object is located.

E.2 EXTRACTED ITEMS ACROSS COLLECTION SETTINGS

Table 7 reports the items extracted from successful trajectories in the two collection settings above. A thought item is an extracted reasoning step; an action item is a tool-call record. These are item counts, not trajectory counts. The hand-designed murder condition supplies action items only.

Table 7: **Items extracted from successful collection trajectories.** Each condition uses the same 75 training instances per task. Counts distinguish thought items from action items supplied to induction; corresponding test accuracies appear in Table 3.

	MUSR murder		MUSR object	
Collection setting	Thought items	Action items	Thought items	Action items
(a) generic	23	91	31	72
(b) hand-designed	0	154	40	101

F CONTENT ABLATION

This ablation asks whether the gains of ARISE come from the content of the induced procedures or from executing each tool call in a separate LLM call. Table 8 compares three configurations on Gemini 2.5 Flash. All three use the same base ReAct agent, the same number of tools, and the same execution interface: every tool call is executed by a separate LLM call on the instance input and the agent-supplied `focus`. Only the information attached to the tools differs.

ARISE. The induced library as deployed in the main experiments: induced tool names and short descriptions in the agent's tool schema, and full natural-language specifications given to the executing LLM call.

ARISE with generic content. Tool names, signatures, and argument descriptions are kept; every short description and full specification is replaced by the same generic instruction ("Helper reasoning tool. Provide a brief reasoning chain (2–4 sentences) identifying how you applied this approach, then give your final answer.").

Generic tools. Tools are renamed `helper_1`, ..., `helper_`k and carry no descriptions or specifications: the agent sees only the tool names and the `focus` argument, and the executing LLM call receives only the function signature.

Results. The induced content accounts for most of the gain. Replacing the specifications with a generic instruction lowers the five-task average from 78.4% to 67.7%, and removing the induced names as well lowers it to 64.2%. The execution interface alone does not reproduce the gains of ARISE: with generic tools, the five-task average falls below the 68.2% of the ReAct agent in Table 2, and accuracy on meeting drops from 83.0% to 44.0%. The effect of content is concentrated on meeting and object, where ARISE exceeds generic tools by 39.0 and 21.7 percentage points, and it is also visible on team (+4.0 over generic tools, +9.0 over generic content). On murder and NBA, the three configurations lie within 4.5 points of one another; on these tasks, generic tools still exceed the ReAct agent of Table 2 (84.0% vs. 78.0% on murder, 68.2% vs. 65.2% on NBA). Induced names carry part of the content: keeping them without specifications adds 3.5 points on average over generic tools, most of it on meeting (+15.0), whose library consists of a single tool.

Table 8: **Content ablation.** Test accuracy (%) with Gemini 2.5 Flash. Subscripts give the difference from *generic tools* in percentage points. ARISE values are from the evaluation runs paired with this ablation.

Task	ARISE	ARISE with generic content	Generic tools
MuSR murder	87.0$_{+3.0}$	83.0$_{-1.0}$	84.0
MuSR object	73.6$_{+21.7}$	55.7$_{+3.8}$	51.9
MuSR team	77.0$_{+4.0}$	68.0$_{-5.0}$	73.0
RuleArena NBA	71.2$_{+3.0}$	72.7$_{+4.5}$	68.2
NatPlan meeting	83.0$_{+39.0}$	59.0$_{+15.0}$	44.0
Avg.	**78.4**$_{+14.2}$	67.7$_{+3.5}$	64.2

G Sensitivity Sweep

A useful library must provide enough specialized agent subroutines without making every possible instruction part of the agent's decision process. We examine this trade-off through library size and specification visibility, using DeepSeek-V3.

Library size. Categories are processed in decreasing support until K agent subroutines have been synthesized or no eligible categories remain; categories below m=3 are excluded and unsuccessful synthesis attempts are skipped. Table 9 varies $K \in \{3, 5, 10\}$. A larger inventory is not consistently better: K=3 performs best on murder and object, while K=5 performs best on team and meeting. The K=10 library is never best in this sweep. These results support a compact library; we use K=5 as a common setting across tasks.

Table 9: **Sensitivity to maximum library size.** Test accuracy (%) with DeepSeek-V3 for $K \in \{3, 5, 10\}$. Entries are individual evaluations; bold marks the highest observed accuracy per task.

Task	K=3	K=5 (default)	K=10
MuSR murder	**80.0**	75.0	77.0
MuSR object	**71.7**	68.9	51.9
MuSR team	61.0	**68.0**	62.0
NatPlan meeting	49.0	**51.0**	45.0

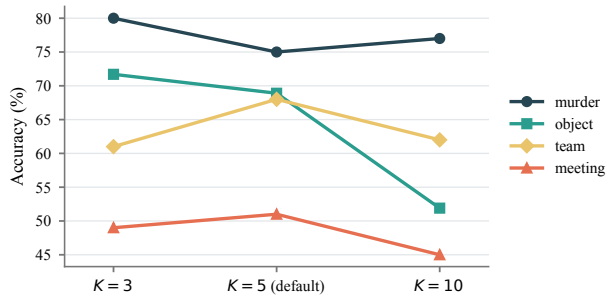


Figure 7: **Accuracy by maximum library size.** DeepSeek-V3 test accuracy for $K \in \{3, 5, 10\}$; values from Table 9.

Trace filtering and specification visibility. Table 10 crosses successful-only versus unfiltered training trajectories with whether the agent sees the full agent subroutine specifications. In every condition, the agent sees names, argument schemas, and short descriptions, and a separate LLM call receives the full specification to execute an agent subroutine. *Hidden* therefore means hidden from the agent's prompt, not absent from execution. Correctness filtering improves the hidden-specification condition on all three tasks. Exposing full specifications to the agent does not add a consistent benefit and lowers accuracy on murder and team for the successful-only libraries. This supports the default separation between a short interface for selecting agent subroutines and a full specification for executing them.

Table 10: **Trace filtering and specification visibility.** Run-level test accuracy (%) with DeepSeek-V3 on MuSR. Visibility refers to the full specification in the agent's prompt; agent subroutine execution calls receive it in all conditions. Variant A is the default. Bold marks the highest accuracy per task.

Variant	Trace filter	Full specification	MuSR murder	MuSR object	MuSR team
A	correct	hidden	**75.0**	**68.9**	68.0
B	correct	visible	69.0	**68.9**	63.0
C	any	hidden	74.0	66.0	60.0
D	any	visible	70.0	67.0	**73.0**

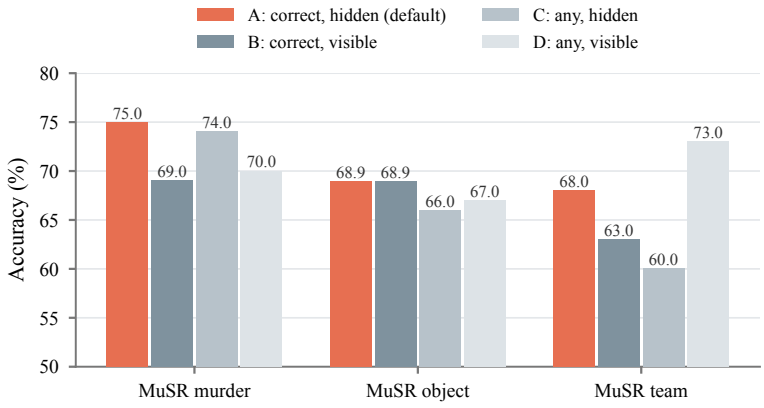


Figure 8: **Trace filtering and specification visibility.** DeepSeek-V3 test accuracy for variants A–D; values from Table 10.

H REPEATED EVALUATIONS ON MUSR

Table 11 reports DeepSeek-V3 means and standard deviations over three runs of each listed method on the same MUSR test partitions.

Table 11: **Repeated DeepSeek-V3 evaluations.** Test accuracy (%), reported as mean ± sample standard deviation across three runs on each fixed MUSR test partition. Bold marks the highest mean per task.

Method	murder	object	team
CoT	62.3 ± 2.5	41.5 ± 2.5	63.3 ± 2.1
PoT	63.3 ± 9.1	62.6 ± 4.6	63.7 ± 4.6
ReAct	54.3 ± 8.1	46.9 ± 3.3	35.7 ± 2.5
AWM	38.3 ± 7.6	49.7 ± 14.1	35.7 ± 12.5
Ours (ARISE)	$\mathbf{71.0 \pm 4.0}$	$\mathbf{68.9 \pm 3.8}$	$\mathbf{68.7 \pm 3.1}$

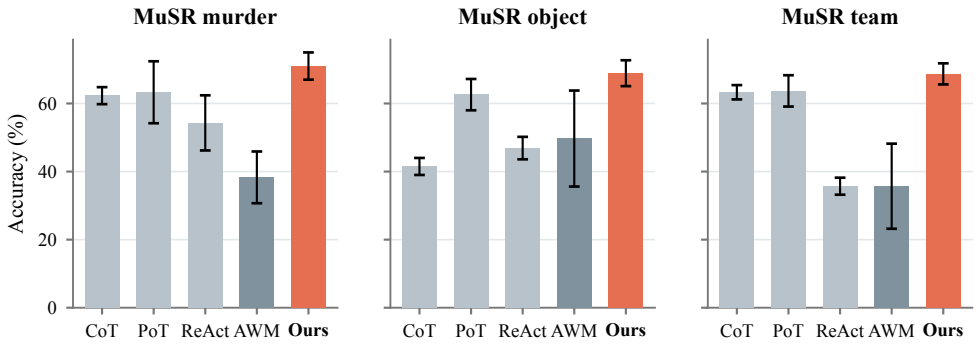


Figure 9: **Repeated DeepSeek-V3 evaluations.** Mean test accuracy with sample standard deviation over three runs; values from Table 11.

ARISE has the highest mean on all three tasks, exceeding PoT by 7.7, 6.3, and 5.0 percentage points on murder, object, and team. Thus the advantage extends beyond the single-run evaluations, including comparisons where the competing methods vary substantially across runs.

I COST ANALYSIS

Induced agent subroutines change how the agent allocates inference: a step in its trajectory can invoke a separate, task-specific LLM computation. Table 12 reports total cost, combining offline adaptation and test-time inference. These measurements complement accuracy by describing the computation used to execute the learned procedures.

Table 12: **Total cost per task.** US dollars for one offline adaptation plus inference on the full test set. Avg. is the unweighted five-task mean. †: served through a different provider (Appendix D).

	MUSR			RuleArena	NatPlan	Avg.
Method	murder	object	team	NBA	meeting	
DEEPSEEK-V3						
ReAct	0.56	0.54	0.45	1.13	0.50	0.64
AWM	1.18	1.21	0.88	15.74	1.02	4.01
ACE	3.20†	2.41†	2.28†	12.19†	1.93†	4.40†
ASI	1.81†	0.92†	0.57†	3.02†	0.45†	1.35†
Ours (ARISE)	1.49	0.84	0.90	21.26	1.67	5.23
GEMINI 2.5 FLASH						
ReAct	0.36	0.38	0.75	1.91	1.76	1.03
AWM	1.06	0.69	1.59	4.41	2.42	2.03
ACE	8.84	6.83	9.23	30.00	10.78	13.14
ASI	2.28	1.68	3.22	4.49	3.10	2.95
Ours (ARISE)	5.28	1.28	3.78	4.25	3.55	3.63

J PAIRED COMPARISONS AND RESPONSE COMPLETION

Paired comparisons help distinguish broad average gains from task-specific behavior.

Statistical procedure. We use 95% percentile-bootstrap intervals with 10 000 case-level resamples. For repeated evaluations, each case is first averaged within method, then paired differences are resampled. Missing and invalid answers count as incorrect in primary accuracy. The conditional comparison in Table 14 averages valid-answer repeats only and retains cases with at least one valid answer from each method.

The single-run intervals favor the induced library over ReAct on all tasks except trip, and over PoT on NBA and meeting. In the Gemini sensitivity analysis, gains remain positive on murder, object, NBA, and meeting after restricting to valid-answer repeats, although their magnitude decreases on murder and NBA. The gains therefore persist in comparisons that separate answer completion from accuracy among returned answers. Table 13 reports the corresponding repeated comparison with CoT: the intervals exclude zero on object and meeting, include zero on murder and NBA, and show no gain on team.

Table 13: **Repeated comparison with CoT on Gemini 2.5 Flash.** Mean test accuracy (%) over three evaluations of the fixed induced library and three CoT runs on the same test cases; Δ is the paired induced-minus-CoT difference (percentage points) with a 95% paired case-bootstrap interval. The induced runs are those of Table 14.

Task	**Ours (ARISE)**	**CoT**	Δ (pp)	**95% CI**
MUSR murder	87.3	81.3	$+6.0$	$(-0.7, 13.0)$
MUSR object	74.2	59.1	$+15.1$	$(6.0, 23.9)$
MUSR team	74.3	75.0	-0.7	$(-6.0, 4.3)$
RuleArena NBA	71.2	66.2	$+5.1$	$(-3.5, 13.6)$
Natural Plan meeting	81.3	69.7	$+11.7$	$(2.3, 21.0)$

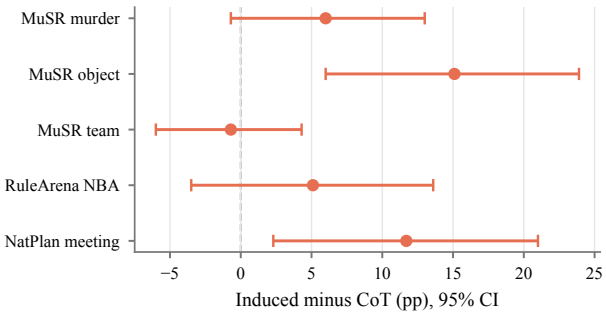


Figure 10: **Repeated comparison with CoT.** Paired induced-minus-CoT difference on Gemini 2.5 Flash with 95% case-bootstrap intervals; values from Table 13.

Table 14: **Sensitivity to invalid or absent answers.** Paired induced-library minus ReAct accuracy differences (percentage points), Gemini 2.5 Flash. All-instance Δ counts every failure as incorrect; conditional Δ uses valid-answer repeats only, with n retained cases. Invalid/absent counts are summed over repeats (ReAct / Induced), including out-of-range answer indices. These separate repeat evaluations are not the main-table aggregates.

Model	**Task**	All-instance Δ	Conditional Δ	n	Invalid/absent R/I
Flash	murder	+10.7	+7.3	100	12/0
Flash	object	+19.5	+19.7	106	0/1
Flash	team	−1.7	−4.0	100	10/1
Flash	NBA	+5.1	+1.5	65	10/0
Flash	meeting	+9.0	+9.0	100	1/0

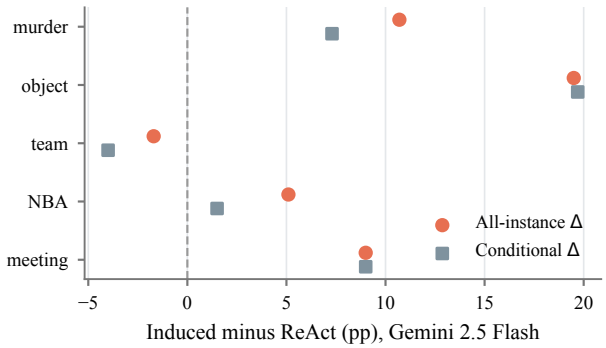


Figure 11: **Sensitivity to invalid or absent answers.** All-instance and conditional induced-minus-ReAct differences on Gemini 2.5 Flash; values from Table 14.

Table 15: **DeepSeek-V3 accuracy and uncertainty.** Test accuracy (%) with 95% case-bootstrap intervals in brackets; bold marks the highest point estimate per task. These individual evaluations differ from the repeated-run summaries. Meeting and trip use the deterministic verifier. Intervals condition on the observed test outcomes.

Task	CoT	PoT	ReAct	AWM	Ours (ARISE)
MuSR murder	60.0 [50.0, 69.0]	67.0 [58.0, 76.0]	45.0 [35.0, 55.0]	47.0 [38.0, 57.0]	**75.0** [66.0, 83.0]
MuSR object	39.6 [30.2, 49.1]	60.4 [50.9, 69.8]	50.0 [40.6, 59.4]	53.8 [44.3, 63.2]	**68.9** [60.4, 77.4]
MuSR team	65.0 [55.0, 74.0]	**69.0** [60.0, 78.0]	38.0 [29.0, 48.0]	36.0 [27.0, 45.0]	68.0 [59.0, 77.0]
RuleArena NBA	48.5 [36.4, 60.6]	40.9 [28.8, 53.0]	28.8 [18.2, 39.4]	59.1 [47.0, 71.2]	**74.2** [63.6, 84.8]
NatPlan meeting	35.0 [26.0, 45.0]	27.0 [18.0, 36.0]	32.0 [23.0, 41.0]	43.0 [33.0, 53.0]	**51.0** [41.0, 61.0]
NatPlan trip	0.0 [0.0, 0.0]	0.0 [0.0, 0.0]	1.0 [0.0, 3.1]	**2.1** [0.0, 5.2]	0.0 [0.0, 0.0]

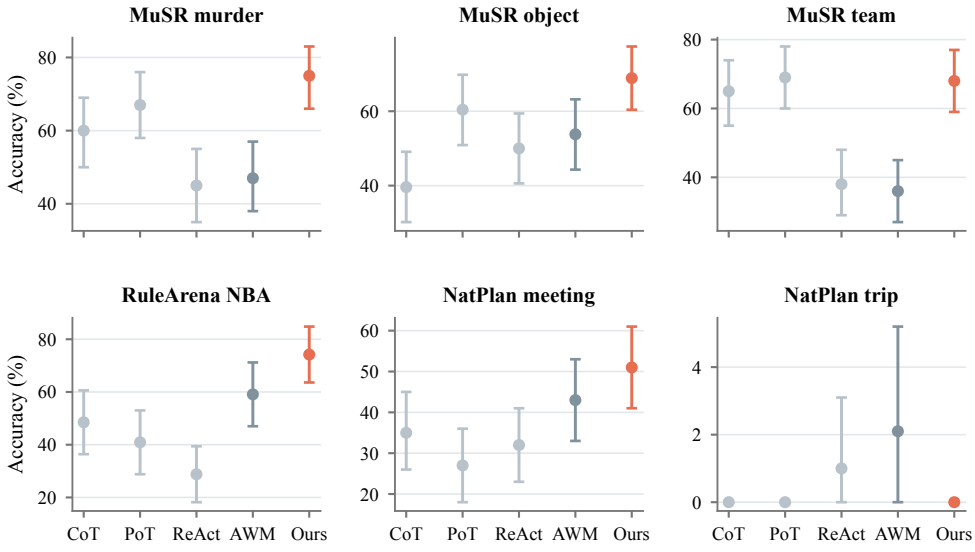


Figure 12: **DeepSeek-V3 accuracy and uncertainty.** Test accuracy with 95% case-bootstrap intervals; values from Table 15.

Table 16: **Paired single-run differences with DeepSeek-V3.** Induced-library minus baseline accuracy (percentage points), with 95% paired case-bootstrap intervals for the runs in Table 15. A star marks an interval excluding zero. Intervals are unadjusted for multiple comparisons.

Task	CoT	PoT	ReAct	AWM
MuSR murder	$+15.0^{\star}$ $[+4.0, +26.0]$	$+8.0$ $[-1.0, +18.0]$	$+30.0^{\star}$ $[+18.0, +41.0]$	$+28.0^{\star}$ $[+16.0, +40.0]$
MuSR object	$+29.2^{\star}$ $[+16.0, +41.5]$	$+8.5$ $[-0.9, +18.9]$	$+18.9^{\star}$ $[+6.6, +31.1]$	$+15.1^{\star}$ $[+2.8, +26.4]$
MuSR team	$+3.0$ $[-7.0, +13.0]$	-1.0 $[-12.0, +10.0]$	$+30.0^{\star}$ $[+17.0, +43.0]$	$+32.0^{\star}$ $[+19.0, +44.0]$
RuleArena NBA	$+25.8^{\star}$ $[+12.1, +39.4]$	$+33.3^{\star}$ $[+19.7, +47.0]$	$+45.5^{\star}$ $[+28.8, +62.1]$	$+15.2$ $[0.0, +31.8]$
NatPlan meeting	$+16.0^{\star}$ $[+7.0, +25.0]$	$+24.0^{\star}$ $[+14.0, +34.0]$	$+19.0^{\star}$ $[+9.0, +29.0]$	$+8.0$ $[-1.0, +17.0]$
NatPlan trip	0.0 $[0.0, 0.0]$	0.0 $[0.0, 0.0]$	-1.0 $[-3.1, 0.0]$	-2.1 $[-5.2, 0.0]$

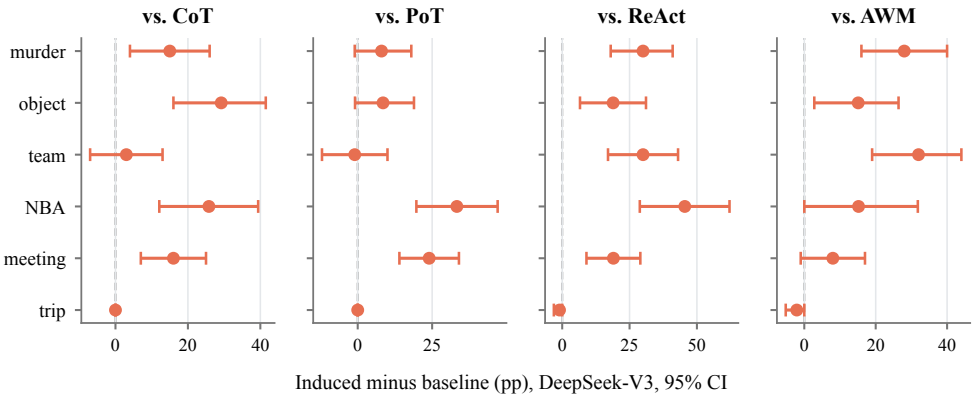


Figure 13: **Paired single-run differences with DeepSeek-V3.** Induced-library minus baseline accuracy with 95% paired case-bootstrap intervals; values from Table 16.

J.1 CROSS-MODEL TRANSFER STATISTICS

Table 17 reports the paired accuracy gains and confidence intervals for Table 5. Libraries induced by Gemini 2.5 Flash are deployed unchanged on Gemini 3.5 Flash-Lite and compared with that target model's ReAct agent on the same test instances. Each difference is formed from repeated measurements of the same cases, before rounding the accuracies. All five intervals exclude zero, supporting reuse of the learned procedures across executing models.

Table 17: **Paired cross-model transfer gains.** Δ is the transferred-library accuracy minus target-model ReAct accuracy, in percentage points, with the corresponding 95% confidence interval. The source model is Gemini 2.5 Flash and the target model is Gemini 3.5 Flash-Lite.

Task	Δ (pp)	**95% CI**
MUSR murder	+14.7	(8.3, 21.3)
MUSR object	+12.3	(3.5, 21.4)
MUSR team	+10.7	(3.2, 18.5)
RuleArena NBA	+32.8	(22.7, 42.9)
Natural Plan meeting	+12.3	(6.3, 18.3)

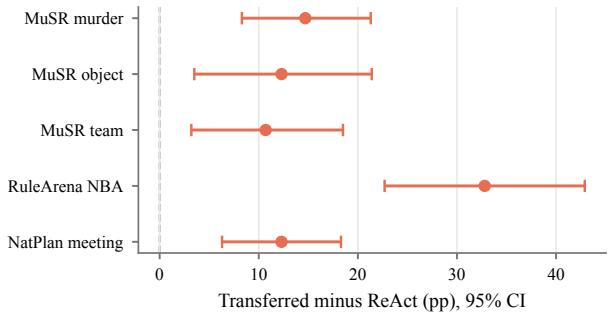


Figure 14: **Paired cross-model transfer gains.** Transferred-library minus target-model ReAct accuracy with 95% confidence intervals; values from Table 17.

K ADDITIONAL-SAMPLING CONTROL

We evaluate self-consistency (Wang et al., 2023a) with 20 CoT samples per instance on DeepSeek-V3. Samples use T=0.7 with response caching disabled. The final answer is the majority of successfully parsed answer indices, with ties resolved by the earliest vote. If no sample yields a valid answer index, the instance is scored as incorrect. This tests whether additional sampling without a learned library, a common way to scale test-time compute (Snell et al., 2025), offers a competitive alternative.

Table 18: **Self-consistency with 20 samples.** DeepSeek-V3 test accuracy (%) on the same fixed test cases as the induced-library runs in Table 15. Δ is SC@20 minus induced-library accuracy, in percentage points.

Task	**SC@N=20**	**Ours (ARISE)**	Δ
MUSR murder	66.0	75.0	−9.0
MUSR object	45.3	68.9	−23.6

SC@20 remains below the induced-library agent by 9.0 points on murder and 23.6 points on object. The comparison supports learning reusable agent subroutines as an alternative to repeatedly sampling complete solutions.

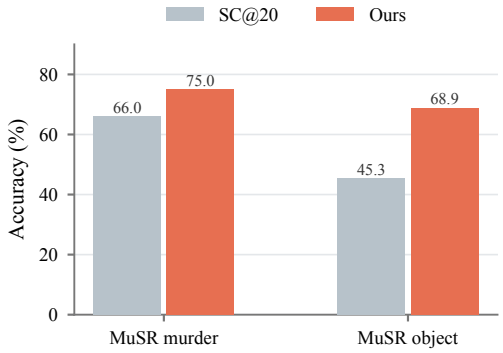


Figure 15: **Self-consistency with 20 samples.** DeepSeek-V3 test accuracy of SC@20 and the induced library; values from Table 18.

L Tasks Outside the Main Evaluation

Our main evaluation studies whether recurring reasoning procedures can be learned from successful agent experience and reused on new instances. Supplementary evaluations on other RuleArena and Natural Plan subtasks highlight two considerations for extending this approach: obtaining sufficient successful experience for induction and choosing suitable implementations for the learned procedures.

Insufficient successful experience. On Natural Plan trip and calendar, the collection agent produces 2 and 19 successful trajectories, too few to cover the tasks' recurring planning operations. On trip, accuracy remains near zero for the induced libraries and for every evaluated method (Table 15). Better experience collection would extend induction to such tasks.

Calculation-heavy tasks. RuleArena airline (baggage-fee calculation, e.g., applying route- and cabin-specific allowances and overweight and oversize surcharges to several bags on a Mumbai–Minneapolis flight) and tax (tax calculation, e.g., determining the balance owed by a married couple filing jointly after combining wage and retirement income, applying deductions and credits, and subtracting withheld tax) require long numerical calculations under strict rules, which suit code execution: a Python program induced for these subtasks extracts over 20 parameters from the problem statement and computes the answer directly. Their rule texts are also long, which makes agent training costly (about $800 per training run). We therefore evaluate arithmetic-heavy reasoning on AIME instead.