FROM TRACES TO PROGRAMS: LEARNING PROGRAMMATIC SKILL ROUTING FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Model (LLM) agents increasingly reuse skills distilled from prior experience, but deciding which skill to apply at each step of execution remains difficult. Existing approaches either provide all skills in context, increasing the reasoning burden during execution, or select skills by semantic similarity, conflating task relevance with applicability at the current step of execution. We introduce **SCRIPT** (**S**kill **C**omposition and **R**outing via **I**nterpretable **P**rogrammed **T**ransitions), which represents experience as skill-level trajectories and separates local skill execution from global routing. Skills encode reusable procedures, while a coding agent learns an executable Python router from skill-level trajectories as execution unfolds. Minimum description length (MDL)-guided refinement favors compact routing programs, and a verifier evaluates each revision against cached routing decisions from prior and newly collected trajectories, without re-executing tasks. Separating skills from routing also enables failure attribution and repair. At deployment, the learned router selects skills directly without additional LLM calls. SCRIPT improves Success Rate over the strongest baseline by 21.1% on ScienceWorld and 11.5% on AppWorld while using up to 28.4× fewer learning tokens. With additional experience, verified router updates further improve task success while preserving prior routing decisions. Our code and data are available in the anonymous repository at `https://anonymous.4open.science/r/SCRIPT-920C/`.

1 INTRODUCTION

Large Language Model (LLM) agents increasingly solve complex tasks by invoking external tools, APIs, and reusable skills distilled from prior experience (Ni et al., 2026; Wang et al., 2024a). With reusable skills available, a key challenge is deciding which skill should govern the next step of execution (Liu et al., 2026; Su et al., 2026). Effective skill selection has therefore become a central problem in skill-augmented agents.

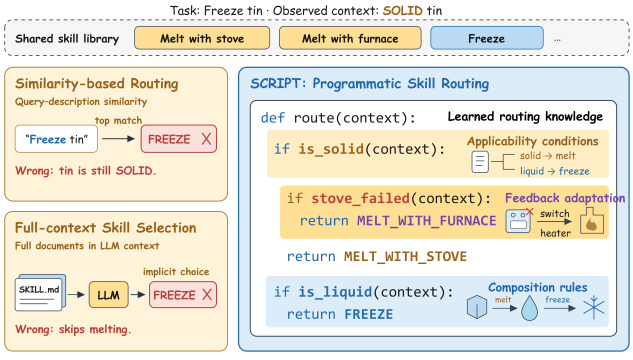


Figure 1: Comparison of similarity-based selection, providing all skills in context, and programmatic routing.

Existing approaches largely follow two paradigms: providing all skills in context (Yang et al., 2026; Agrawal et al., 2025) and similarity-based selection (Liu et al., 2026; Wang et al., 2024a). Figure 1 illustrates how both can fail on a simple example, where the task is to freeze a tin, but the tin is initially observed to be solid and must first be melted. Semantic similarity does not guarantee applicability, so similarity-based selection may retrieve a skill that matches the task instruction but is inappropriate for the current context , such as selecting FREEZE when melting is the correct next step. Providing all skills in context makes every skill available, but a skill that is useful in one situation may be irrelevant or

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.

even misleading in another. The model must determine at inference time which skill applies, and it may apply a task-relevant skill at the wrong point in execution, such as selecting FREEZE before melting the solid tin. Both paradigms thus suffer from a **relevance–applicability gap**: a skill may appear relevant to the overall task while being inapplicable to the current context. Moreover, when a task fails, task-level feedback also does not reveal whether the underlying skill was faulty or whether a correct skill was invoked at the wrong time, making **failure attribution ambiguous**.

Planning provides a distinct paradigm for learning decision policies from prior experience (Khardon, 1999). From this perspective, successful skill trajectories capture not only reusable procedures, but also the global execution flow governing how these procedures should be composed and invoked throughout a task. As illustrated in Figure 1, such a policy selects a melting skill when the tin is observed to be solid, invokes FREEZE after observing that it has become liquid, and switches to the furnace if the stove fails. These transitions can be distilled into a routing policy that maps the evolving execution context to the next applicable skill. Representing this policy as an executable program makes the routing logic explicit and directly testable, while advances in code agents make it practical to synthesize and revise such programs from experience (Wang et al., 2024b; Yang et al., 2024). A minimum description length (MDL) objective balances routing errors against program length (Rissanen, 1978), favoring compact routing logic over case-specific rules. This perspective reformulates skill selection from a **retrieval problem** into a **planning problem**: the agent learns a global policy for composing and transitioning between skills throughout execution, rather than retrieving each skill independently at individual decision points.

We propose **SCRIPT** (**S**kill **C**omposition and **R**outing via **I**nterpretable **P**rogrammed **T**ransitions), which learns reusable skills together with an executable Python router that governs skill composition throughout execution. SCRIPT abstracts each successful execution into a **skill-level trajectory**, a sequence of routing decisions that pair each execution context with the selected skill while omitting low-level actions. This abstraction separates local skill execution from global routing, with skills encoding reusable procedures for individual subgoals and the router capturing how these skills are organized throughout task execution. A coding agent synthesizes and revises the router from recorded routing decisions under an MDL objective. Each candidate router is then verified against cached routing decisions without LLM calls, skill execution, or environment interaction, enabling new routing rules to be incorporated while preserving previously validated decisions. At deployment, the learned router directly selects the next skill, while the selected skill handles local execution. The same separation also enables more precise failure attribution, since a failure can be traced to a missing or faulty skill or to an incorrect routing decision and repaired independently. Our contributions are summarized as follows:

- We reformulate skill selection from a **retrieval problem** into a **programmatic planning problem**, where routing experience is distilled into an executable policy that governs skill composition throughout task execution. This global routing policy determines the next skill before local execution, avoiding LLM-based routing inference while remaining deterministic, interpretable, and directly verifiable.
- We introduce **failure attribution and targeted repair** by separating local skill execution from global routing, allowing skill and routing errors to be identified and repaired independently. Router revisions are verified against previously validated and newly collected routing decisions without additional LLM calls or environment execution.
- We evaluate SCRIPT on ScienceWorld and AppWorld, achieving 21.1% and 11.5% relative Success Rate gains over the strongest baselines while using up to 28.4× fewer learning tokens. SCRIPT also transfers to unseen tasks and continues to improve through verified refinement with additional experience.

2 PROBLEM FORMULATION

2.1 INTERACTIVE TASKS

An LLM agent receives a task instruction $x \sim \mathcal{D}$ and interacts with a partially observable environment. At step t, it observes o_t and selects an action a_t based on the task instruction and the observable history $h_t = (o_0, a_0, \ldots, a_{t-1}, o_t)$. Executing a_t produces the next observation o_{t+1}. An episode produces a trajectory $\tau = (o_0, a_0, \ldots, a_{T-1}, o_T)$ and a normalized task score $r_x(\tau) \in [0, 1]$.

We treat a maximum-score trajectory as a successful demonstration. The agent has no access to the latent environment state.

2.2 Reusable Skills and Skill-Level Trajectories

Successful trajectories often contain recurring action sequences that accomplish the same subgoal. We abstract these sequences into reusable skills.

Definition 1 (Skill). *A skill σ_i is a parameterized procedure that executes a sequence of environment actions to accomplish a recurring subgoal. The skill library is denoted by $\mathcal{L}$.*

Because one skill invocation may contain multiple environment actions, we use t to index environment steps and k to index skill invocations. If the k-th invocation starts at step t_k and returns at t_{k+1}, its decision context is $c_k = (x, h_{t_k})$, and the invoked skill σ_k is instantiated with arguments θ_k.

Definition 2 (Skill-Level Trajectory). *Let τ be an interaction trajectory containing $K+1$ skill invocations. Its induced skill-level trajectory is*

$$\rho(\tau) = \left[(c_0, \sigma_0, \theta_0, o_{t_1}), \ldots, (c_K, \sigma_K, \theta_K, o_{t_{K+1}})\right], \tag{1}$$

where $o_{t_{k+1}}$ is the observation available after the k-th invocation. For the final invocation, $o_{t_{K+1}}$ is the terminal observation of τ.

The representation converts each successful trajectory into a sequence of routing decisions (c_k, σ_k) while leaving the internal actions of each skill to the skill implementation.

2.3 Programmatic Skill Routing

The routing problem is therefore to map each decision context c_k to the next skill σ_k.

Definition 3 (Skill Routing Program). *Given a skill library $\mathcal{L}$, a skill routing program R is executable code that selects a skill from the current decision context:*

$$\sigma_k = R(c_k; \mathcal{L}). \tag{2}$$

The pairs (c_k, σ_k) extracted from skill-level trajectories provide supervision for learning R. Because R contains no LLM calls, its decisions are reproducible and can be evaluated on recorded contexts.

2.4 Objective

The skill library and routing program jointly determine task performance. Our goal is therefore

$$(R^*, \mathcal{L}^*) = \arg\max_{R,\mathcal{L}} \mathbb{E}_{x\sim\mathcal{D},\, \tau\sim P(\tau|R,\mathcal{L},x)} \left[r_x(\tau)\right]. \tag{3}$$

SCRIPT constructs $\mathcal{L}$ from successful demonstrations and learns R from the resulting skill-level trajectories, as described in Section 3.

3 The SCRIPT Framework

In this section, we present SCRIPT, which formulates skill selection as programmatic routing over reusable skills. We first describe how trajectories are abstracted into reusable skills (Section 3.2), and then present MDL-guided routing program learning (Section 3.3). Finally, we describe program-guided execution (Section 3.4) and decoupled failure attribution and repair (Section 3.5).

3.1 Overview

Figure 2(a) shows how SCRIPT abstracts recurring action sequences into reusable skills and represents successful demonstrations as skill-level trajectories. This representation separates local skill execution from global routing, with skills determining *how* individual subgoals are executed and the router determining *which* skills should be invoked throughout task execution. As shown in Figure 2(b), a coding agent learns the routing program from recorded routing decisions under an MDL

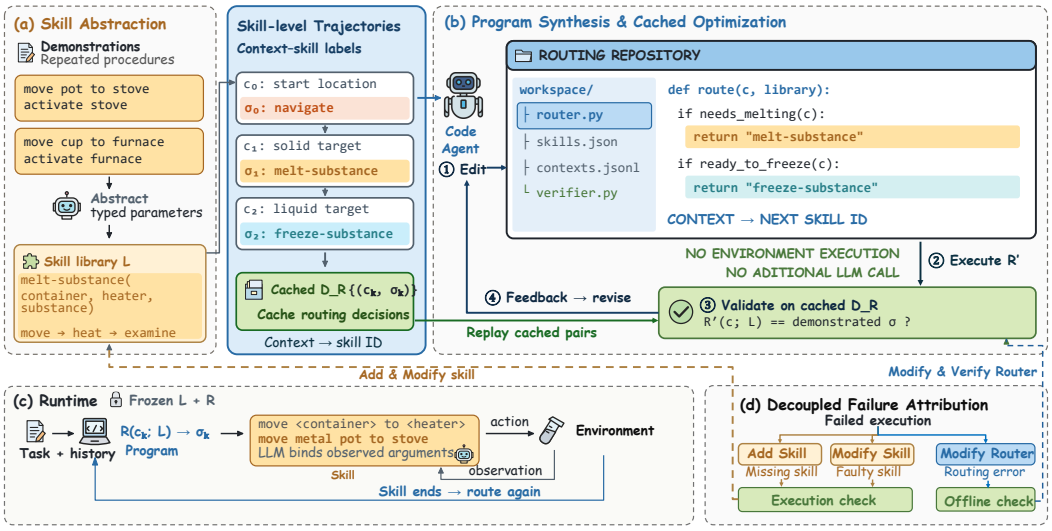


Figure 2: SCRIPT: Skill Composition and Routing via Interpretable Programmed Transitions. (a) Successful demonstrations are abstracted into reusable parameterized skills and skill-level trajectories. (b) A coding agent synthesizes and revises the routing program, while a cached verifier checks each revision against recorded routing decisions. (c) The learned router selects skills at runtime without LLM-based routing inference. (d) Execution failures are attributed to skills or routing errors for independent repair.

objective, while a verifier evaluates candidate programs without additional skill execution or environment interaction. At runtime, the learned router directly selects the next skill, while the selected skill handles local execution.

The same separation also makes failures easier to attribute, as shown in Figure 2(d). A failed execution may arise from a missing skill, a faulty skill implementation, or an incorrect routing decision. SCRIPT attributes the failure to the corresponding component and repairs only that component while preserving the successful skills and routing decisions.

3.2 Skill Abstraction from Trajectories

Prior work often represents a skill as a natural-language procedure that must be interpreted into concrete actions at each invocation (Ni et al., 2026; Wang et al., 2025c). SCRIPT instead abstracts recurring action sequences from successful trajectories into executable code skills, replacing instance-specific entities and values with parameters.

Let $\mathcal{T}_{\text{train}} = \{\tau_1, \ldots, \tau_n\}$ denote the successful training trajectories. An LLM compares multiple trajectories to identify recurring action segments that accomplish the same subgoal. Comparing different successful executions helps separate shared procedures from instance-specific entities and values. These procedures are abstracted into parameterized skills that form $\mathcal{L}$, after which each trajectory is converted into its skill-level representation $\rho_i = \rho(\tau_i)$. Details are in Appendix A.3.1.

3.3 Learning the Routing Program

SCRIPT represents skill routing as an executable policy that maps the current decision context to the next skill. The policy captures the global execution flow that governs skill selection and transitions throughout the task. Because the decision logic is executable, it can be inspected, revised, and verified independently of skill execution.

Learning data from skill sequences. After each successful interaction trajectory is represented as the skill-level trajectory ρ_i in Definition 2, every recorded skill invocation contributes a routing decision (c_k, σ_k). Aggregating these routing decisions gives the routing dataset

$$\mathcal{D}_R = \left\{ (c_k, \sigma_k) \;\middle|\; (c_k, \sigma_k, \theta_k, o_{t_{k+1}}) \in \rho_i,\ \rho_i \in \{\rho_1, \ldots, \rho_n\} \right\}. \tag{4}$$

The resulting dataset directly supervises which skill is applicable in each decision context and allows routing to be learned separately from the internal execution of each skill, following the hierarchical learning setting of Khardon (1999).

Routing program, deterministic verifier, and MDL-guided refinement. Using $\mathcal{D}_R$, an LLM-based coding agent maintains a Python codebase and revises a routing program $R(c; \mathcal{L})$ that determines the next skill from the current decision context. A deterministic verifier evaluates each candidate against the cached routing decisions and returns the decisions on which it disagrees, requiring neither skill execution nor additional environment interaction. SCRIPT selects among the proposed revisions by minimizing

$$L(R; \mathcal{D}_R) = L_{\text{code}}(R) + L_{\text{errors}}(R; \mathcal{D}_R). \tag{5}$$

The code cost discourages long or case-specific rules. The error cost groups repeated occurrences of the same routing decision so that each distinct routing decision receives equal weight. Appendix A.2 specifies the complete encoding and error weighting.

Routing agreement and program complexity. For analysis, we treat the successful demonstrations as generated by a reference routing policy with task success probability $S^\star > 0$, and let $S(R, \mathcal{L})$ denote the task success probability when routing with R over $\mathcal{L}$. Under a fixed skill library and execution mechanism, assume the demonstrations are n independent successful trajectories. Let $q_+(R)$ be the probability that R disagrees with at least one demonstrated decision along a successful trajectory.

Proposition 1 (Success preservation)**.** *Every deterministic router R satisfies*

$$S(R, \mathcal{L}) \geq S^\star\big(1 - q_+(R)\big). \tag{6}$$

Thus, agreement with a successful trajectory preserves its success, although disagreement need not imply failure. Let $\widehat{q}_+(R)$ be the empirical trajectory-disagreement rate, let $\mathcal{R}_b$ contain routers encoded in at most b bits, and let $C(b, \delta)$ be the complexity term defined in Appendix A.1.3. Combining Proposition 1 with an Occam argument over routing programs (Blumer et al., 1987) gives the following bound.

Corollary 1 (Routing generalization bound)**.** *With probability at least $1 - \delta$, simultaneously for all $b \geq 1$ and $R \in \mathcal{R}_b$,*

$$S(R, \mathcal{L}) \geq S^\star \left[1 - \widehat{q}_+(R) - \sqrt{\frac{C(b, \delta)}{2n}}\right]_+ . \tag{7}$$

Consequently, $\widehat{q}_+(R) + \sqrt{C(b, \delta)/(2n)} \leq \epsilon$ implies $S(R, \mathcal{L}) \geq (1 - \epsilon)S^\star$.

This bound relates agreement with cached routing decisions from successful trajectories to task success, which motivates verifying router revisions offline. It also favors shorter routers for the same empirical routing agreement, providing an Occam-style motivation for jointly controlling routing errors and program length in Equation 5. Appendix A.1 gives the assumptions and proofs.

3.4 Program-Guided Execution

At deployment, SCRIPT uses the learned router and skill library to select and execute skills. At each routing point, SCRIPT constructs the decision context $c_k = (x, h_{t_k})$. The learned router returns either the next skill $\sigma_k \in \mathcal{L}$ or $\perp$ without any LLM call or similarity search. For a selected skill σ_k, an LLM binds its parameters θ_k from the task instruction and current observations, and the environment executes the skill-generated actions until termination, after which control returns to the router. The resulting observations are appended to the history before the router is queried again. Thus, the router governs the global skill-level execution flow, while each skill handles local execution and adaptation.

If the router returns $\perp$, an LLM-based fallback policy executes a single interaction step, and the resulting interaction is appended to the observable history. The router is then queried again from the updated context, allowing fallback to handle uncovered execution one step at a time. Appendix A.3 provides the complete execution procedure.

3.5 Failure Attribution and Repair

Attribution and Proposal. Separating skill execution from routing allows failures to be attributed and repaired independently. Given a failed trajectory ρ^- collected from additional experience, an LLM examines the selected skills, their execution outcomes, and the corresponding decision contexts. It identifies an affected decision point k and proposes $\xi = (z, \eta, k)$, where $z \in \{\text{AddSkill}, \text{ModifySkill}, \text{ModifyRouter}\}$ specifies the repair type and η specifies the proposed change. These three repair types correspond to three causes of failure. A missing skill indicates that the required subgoal is not covered by $\mathcal{L}$, a faulty skill indicates that the selected skill is appropriate but its execution is inadequate, and a routing error indicates that an available skill was selected at the wrong point or omitted.

Validation and Update. SCRIPT follows a propose–validate–incorporate procedure. Each proposed repair is first validated through prefix-based re-execution on the affected task and is accepted only if it improves task performance. Accepted skill repairs are incorporated into the skill library $\mathcal{L}$, whereas accepted routing repairs are converted into new routing decisions and added to $\mathcal{D}_R$ to refine the router R. The revised router is then verified against cached routing decisions to ensure consistency with previously validated decisions without additional environment interaction. This procedure updates learned components only after execution-based validation. For routing repairs, SCRIPT updates the routing data rather than directly editing the router, allowing the coding agent to generalize task-specific corrections into reusable routing rules while preserving previously learned behavior. Appendix A.3 provides details.

4 Experiments

In this section, we conduct comprehensive experiments to evaluate SCRIPT and answer the following research questions: (RQ1) How does SCRIPT compare with existing skill learning methods? (RQ2) Can SCRIPT improve further from additional experience through explicit failure attribution, and at what computational cost? (RQ3) How does router refinement behave across optimization steps, training-data scales, and code-authoring agents? (RQ4) How do programmatic routing, routable skills, and MDL-guided selection contribute to performance and generalization?

4.1 Experimental Setup

Benchmarks. We evaluate SCRIPT on two benchmarks for interactive agents: (1) ScienceWorld (Wang et al., 2022), a text-based environment that evaluates multi-step scientific reasoning and planning, and (2) AppWorld (Trivedi et al., 2024), which evaluates cross-application reasoning and state-changing API use. Dataset construction and statistics are provided in Appendix A.4.

Baselines. We compare with four categories of baselines: learning-free execution with ReAct (Yao et al., 2023); experience-memory methods with ExpeL (Zhao et al., 2024) and ReasoningBank (Ouyang et al., 2025); skill induction and retrieval with Trace2Skill (Ni et al., 2026) and Graph-of-Skills (GoS) (Liu et al., 2026); and iterative textual evolution with ACE (Zhang et al., 2025). Appendix A.4 describes each baseline and its adaptation.

Implementation. We use Gemini 3 Flash as the common LLM backbone for all methods. For RQ3, we additionally evaluate SCRIPT with Claude Code using Opus 5 as the coding agent. We study two settings: train-only evaluation, where learned skills and routing programs are evaluated without further updates, and evolution with additional experience, where development experience is used for failure attribution and repair before evaluation. We report normalized Score, Success Rate, Scenario Success Rate, token consumption, and routing accuracy where applicable. Normalized Score is reported on a 0–100 scale. Full implementation details are provided in Appendix A.4.

4.2 Overall Performance and Efficiency (RQ1)

SCRIPT achieves the highest Success Rate on both benchmarks, with larger gains on unseen tasks. As shown in Table 1, SCRIPT improves unseen Success Rate over the strongest baseline by 21.1% on ScienceWorld and 11.5% on AppWorld. This pattern is consistent with SCRIPT's separation of local skill execution from global routing. Skills encode parameterized execution procedures

Table 1: Train-only performance on ScienceWorld and AppWorld. **Bold** indicates the best result, and underline indicates the second best. Subscripts indicate absolute changes relative to ReAct. Token consumption is reported in thousands per sample.

		Seen			Unseen			Tokens (K)	
Dataset	Method	Score	Success (%)	Scenario SR (%)	Score	Success (%)	Scenario SR (%)	Learn	Execute
ScienceWorld	ReAct	60.77	47.44	0.00	64.51	48.17	5.56	–	364.71
	ReasoningBank	69.03$_{\uparrow 8.26}$	56.41$_{\uparrow 8.97}$	5.56$_{\uparrow 5.56}$	68.93$_{\uparrow 4.42}$	52.44$_{\uparrow 4.27}$	5.56	368.28	399.72
	ExpeL	70.85$_{\uparrow 10.08}$	60.26$_{\uparrow 12.82}$	22.22$_{\uparrow 22.22}$	72.13$_{\uparrow 7.62}$	63.41$_{\uparrow 15.24}$	5.56	896.49	410.75
	Trace2Skill	69.83$_{\uparrow 9.06}$	54.49$_{\uparrow 7.05}$	16.67$_{\uparrow 16.67}$	72.99$_{\uparrow 8.48}$	59.15$_{\uparrow 10.98}$	11.11$_{\uparrow 5.55}$	380.99	505.30
	GoS	73.96$_{\uparrow 13.19}$	55.77$_{\uparrow 8.33}$	0.00	73.68$_{\uparrow 9.17}$	54.88$_{\uparrow 6.71}$	11.11$_{\uparrow 5.55}$	377.54	602.16
	ACE	79.00$_{\uparrow 18.23}$	70.51$_{\uparrow 23.07}$	33.33$_{\uparrow 33.33}$	79.94$_{\uparrow 15.43}$	66.46$_{\uparrow 18.29}$	27.78$_{\uparrow 22.22}$	1541.24	585.08
	SCRIPT	**85.22**$_{\uparrow 24.45}$	**75.64**$_{\uparrow 28.20}$	**55.56**$_{\uparrow 55.56}$	**87.68**$_{\uparrow 23.17}$	**80.49**$_{\uparrow 32.32}$	**50.00**$_{\uparrow 44.44}$	99.25	112.88
AppWorld	ReAct	82.87	54.39	31.58	79.82	54.17	30.36	–	46.43
	ReasoningBank	92.66$_{\uparrow 9.79}$	80.70$_{\uparrow 26.31}$	57.89$_{\uparrow 26.31}$	89.98$_{\uparrow 10.16}$	75.00$_{\uparrow 20.83}$	58.93$_{\uparrow 28.57}$	35.50	76.38
	ExpeL	92.29$_{\uparrow 9.42}$	77.19$_{\uparrow 22.80}$	57.89$_{\uparrow 26.31}$	84.06$_{\uparrow 4.24}$	61.90$_{\uparrow 7.73}$	41.07$_{\uparrow 10.71}$	100.36	72.37
	Trace2Skill	80.10$_{\downarrow 2.77}$	61.40$_{\uparrow 7.01}$	31.58	83.00$_{\uparrow 3.18}$	62.50$_{\uparrow 8.33}$	42.86$_{\uparrow 12.50}$	42.77	69.70
	GoS	88.20$_{\uparrow 5.33}$	73.68$_{\uparrow 19.29}$	52.63$_{\uparrow 21.05}$	85.18$_{\uparrow 5.36}$	62.50$_{\uparrow 8.33}$	35.71$_{\uparrow 5.35}$	43.85	66.50
	ACE	93.92$_{\uparrow 11.05}$	84.21$_{\uparrow 29.82}$	**68.42**$_{\uparrow 36.84}$	91.12$_{\uparrow 11.30}$	77.38$_{\uparrow 23.21}$	66.07$_{\uparrow 35.71}$	252.92	206.66
	SCRIPT	**95.00**$_{\uparrow 12.13}$	**85.96**$_{\uparrow 31.57}$	**68.42**$_{\uparrow 36.84}$	**95.42**$_{\uparrow 15.60}$	**86.31**$_{\uparrow 32.14}$	**76.79**$_{\uparrow 46.43}$	8.90	46.05

Table 2: Performance on ScienceWorld after evolution. Subscripts indicate changes from train-only results; "w/o classification" removes explicit failure attribution.

	Train-only		+Evolution		
Method	Score	Success (%)	Score	Success (%)	Tokens (K)
ReasoningBank	68.93	52.44	66.52$_{\downarrow 2.41}$	49.39$_{\downarrow 3.05}$	405.59
Trace2Skill	72.99	59.15	76.76$_{\uparrow 3.77}$	65.24$_{\uparrow 6.09}$	484.62
ACE	79.94	66.46	81.16$_{\uparrow 1.22}$	68.90$_{\uparrow 2.44}$	1878.51
SCRIPT	**87.68**	**80.49**	**89.49**$_{\uparrow 1.81}$	**82.93**$_{\uparrow 2.44}$	403.96
w/o classification	87.68	80.49	88.64$_{\uparrow 0.96}$	81.10$_{\uparrow 0.61}$	–

Table 3: Repair distribution and effectiveness on the five ScienceWorld task types affected by evolution. Gains are averaged per affected task type.

Repair type	Proposed	Accepted	Accept.	Gain/ type	Gain/ repair
Modify router	31	21	67.7%	**+4.39**	**+1.05**
Modify skill	64	45	70.3%	+2.14	+0.24
Add skill	1	1	1/1	+0.01	+0.05
Total	96	67	69.8%	+6.54	+0.49

that can be reused across instances, while the router captures the global skill-level execution flow throughout a task. This separation allows local procedures to transfer across instances while the router composes them according to the evolving execution context, rather than relying on instance-specific routing inference at each step.

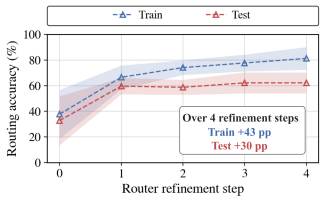
SCRIPT requires the lowest learning and execution cost, as skill-level trajectories allow the router to be refined and verified against cached routing decisions without re-executing tasks. As shown in Table 1, SCRIPT uses 15.5× fewer learning tokens than ACE on ScienceWorld and 28.4× fewer on AppWorld. At deployment, the executable router directly selects the next skill without additional LLM inference for routing or placing skill documents in context, substantially reducing execution cost while preserving explicit skill applicability and composition logic.

4.3 Evolution with Additional Experience (RQ2)

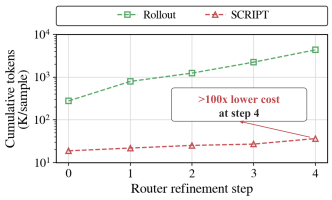
SCRIPT continues to improve from additional experience while preserving previously validated routing decisions at low cost. As shown in Table 2, evolution raises SCRIPT's Success Rate to 82.93%, 20.4% higher than the strongest baseline, while consuming the fewest evolution tokens and 4.7× fewer than ACE. Every router revision is verified against both previously validated decisions and newly collected routing decisions, allowing the router to incorporate new rules without losing earlier behavior. In contrast, ReasoningBank's Success Rate drops by 5.8% after incorporating additional experience.

Explicit failure attribution makes refinement more effective by directing repairs to the affected component. In Table 2, removing failure classification reduces the relative Success Rate gain from 2.44 to 0.61 percentage points. Skills encode local execution strategies, while the router encodes global composition knowledge, so the two components address different failure sources. Explicit attribution directs each repair to the affected component rather than applying a generic update across skill and routing behavior.

Router repairs yield broader gains than skill repairs. Table 3 shows that each accepted router repair delivers 4.4× the average gain of an accepted skill repair. This larger gain is consistent with routing rules affecting decisions across multiple contexts rather than a single local execution.



(a) Routing accuracy across refinement steps.



(b) Cumulative token cost across refinement steps.

Figure 3: Analysis of router refinement on ScienceWorld.

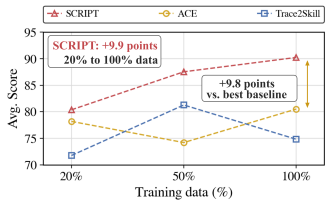


Figure 4: Average Score on five tasks in ScienceWorld with different fractions of training data.

Table 5: Ablation on AppWorld. Highlighted rows denote SCRIPT. Tokens are reported in thousands per episode.

Component	Setting	Score	Success (%)	Scenario SR (%)	Tokens
Routing strategy	programmatic routing	**95.42**	**86.31**	**76.79**	46.05K
	similarity routing	86.40	71.43	57.14	80.83K
	all skills in context	93.62	82.14	67.86	119.85K
Skill representation	parameterized skills	**95.42**	**86.31**	**76.79**	46.05K
	text-only skills	93.68	82.14	69.64	55.87K
	raw gold code	94.19	85.12	71.43	47.58K

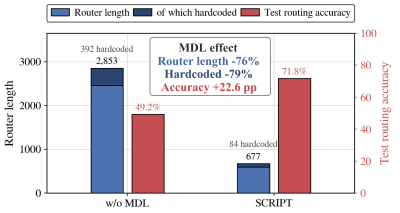


Figure 5: MDL-guided router refinement.

Representing routing as a deterministic program further makes router revisions explicit and directly verifiable against cached routing decisions.

4.4 Router Refinement and Scaling (RQ3)

Iterative router refinement improves both training and held-out routing accuracy, suggesting that refinement captures routing regularities shared across decisions rather than memorizing individual cases. Across four refinement steps, Figure 3(a) shows gains of 43 percentage points on training decisions and 30 points on held-out routing decisions. Learning from skill-level trajectories allows the coding agent to learn routing rules that apply beyond individual training decisions.

For router candidate evaluation, cached routing decisions replace environment rollouts with direct program execution, making refinement over 100× more token-efficient. Figure 3(b) compares cumulative token consumption with rollout-based optimization, which evaluates revisions by executing tasks on a validation set (Yang et al., 2026). SCRIPT consumes tokens only when the coding agent revises the router based on feedback from cached routing decisions, after which each candidate can be tested repeatedly without task-level LLM execution or environment rollouts.

Routing decisions extracted from skill-level trajectories provide both learning supervision and verification cases, allowing SCRIPT to improve as more trajectories become available. As shown in Figure 4, increasing the training data from 20% to 100% raises SCRIPT's average Score by 9.9 points. Additional trajectories expose more routing regularities, allowing SCRIPT to refine case-specific experience into increasingly shared routing logic while remaining consistent with recorded decisions, rather than accumulating overlapping or conflicting experience entries.

Table 4: Performance on ScienceWorld across different code agents.

Code author	Score	Success (%)
Gemini 3 Flash	87.68	80.49
CC (Opus 5)	**89.48**	**82.32**

A stronger coding agent improves task success without adding deployment-time routing cost. For this comparison, we replace Gemini 3 Flash with Claude Code running Opus 5 and its native tools. As shown in Table 4, Success Rate increases by 1.83 points. Because routing is expressed as a program, SCRIPT can directly benefit from advances in coding agents while retaining deterministic, low-cost deployment.

4.5 Component and MDL Ablations (RQ4)

Programmatic routing learns explicit routing rules from prior experience and addresses the relevance–applicability gap in skill selection. As shown in Table 5, replacing programmatic routing with similarity-based selection lowers Success Rate by 14.88 points and Scenario Success Rate by 19.65 points. This substantial degradation provides empirical evidence that semantic relevance alone is insufficient for determining which skill is applicable during execution, consistent with the relevance–applicability gap illustrated in Figure 1. Providing all skills in context alleviates the limitation of similarity-based retrieval, but still requires the LLM to resolve skill applicability during execution, trailing programmatic routing by 4.17 points in Success Rate while consuming 2.6× more tokens. In contrast, SCRIPT distills prior routing experience into explicit executable rules that directly determine the applicable skill from the evolving execution context.

Parameterized skills preserve concrete action procedures while abstracting away instance-specific details, enabling reuse across tasks. Table 5 compares them with text-only skills, which describe procedures in natural language, and raw gold code, which retains action sequences and instance-specific values from successful trajectories. Parameterized skills improve Scenario Success Rate by 7.15 points over text-only skills and by 5.36 points over raw gold code. Text-only skills require action translation at each invocation, whereas raw code remains tied to its source trajectory. Combining executability with abstraction therefore provides the largest scenario-level benefit.

The MDL objective steers the code agent toward shorter routers that transfer better to held-out routing decisions. Figure 5 compares routers refined with and without the MDL objective. MDL reduces router length from 2,853 to 677 and hardcoded content from 392 to 84, while increasing held-out routing accuracy from 49.2% to 71.8%. The reduction in hardcoded content suggests that, without MDL, the code agent more readily introduces case-specific conditions, whereas the length penalty favors compact routing rules that also achieve higher held-out accuracy.

5 Related Work

Planning and Programmatic Policies. Planning has long studied policies that transfer across problem instances. Khardon (1999) learns action-selection strategies from demonstrated executions, and generalized planning computes reusable policies such as finite-state controllers and executable planning programs (Segovia-Aguas et al., 2018; 2019; Francès et al., 2021), which LLMs can now synthesize from a few training tasks (Silver et al., 2024). These approaches rely on symbolic state and action models. SCRIPT brings this perspective to skill-based LLM agents without such models, learning an executable routing policy over skills from successful trajectories and verifying it.

Skill Learning for LLM Agents. Prior work distills interaction trajectories into reusable skills, represented either as textual procedures (Wang et al., 2025c; Ni et al., 2026) or as executable programs that compose environment actions (Wang et al., 2024a; 2025b; Guo et al., 2026). SkillOpt further optimizes skills through execution feedback, but requires additional data for validation (Yang et al., 2026). These methods leave the choice of which skill to invoke to an LLM that reads skill descriptions placed in its context (Wang et al., 2025b;a), or to retrieval by semantic similarity (Su et al., 2026), or dependency structure (Liu et al., 2026). They thus improve how individual skills are represented, executed, or retrieved. SCRIPT separates skill execution from skill-level decision making and learns an executable routing policy that determines which skill should be invoked as the execution context evolves.

6 Conclusion

We introduced SCRIPT, a programmatic skill-routing framework that separates reusable local execution from global skill-level control. Rather than selecting skills through repeated LLM inference or semantic retrieval, SCRIPT distills successful experience into an executable routing policy. Skill-level trajectories provide routing supervision, while deterministic execution and cached verification enable efficient router refinement; the same separation supports targeted failure attribution and repair. Experiments on ScienceWorld and AppWorld demonstrate improved task success, transfer to unseen tasks, and lower learning and execution costs. These results show that explicit, verifiable routing programs can turn successful agent experience into reusable execution policies.

AI USE STATEMENT

Generative AI systems are central components of the method and experiments in this work: language models execute skills, and coding agents synthesize and revise routing programs, as described in the paper. During manuscript preparation, generative AI tools also assisted with revising prose. The authors take responsibility for the final text, code, figures, and scientific claims.

ETHICS STATEMENT

This work uses public benchmarks with simulated environments and does not involve human participants, personal data, or deployment on real user accounts. Nevertheless, agents that select and execute reusable procedures may propagate errors from the underlying language model, learned skills, or routing program. Such errors could have larger consequences in real systems that expose external tools or irreversible actions. SCRIPT's executable router and cached verification improve auditability but do not constitute a safety guarantee outside the evaluated contexts. Deployments in consequential settings should therefore use sandboxing, least-privilege tool access, explicit approval for irreversible actions, and additional evaluation under the intended distribution. The method also inherits potential biases and limitations of its language-model backbone and training experience.

REPRODUCIBILITY STATEMENT

We provide the implementation, data, and experimental artifacts in the anonymous repository at `https://anonymous.4open.science/r/SCRIPT-920C/`. Section 4.1 specifies the evaluated benchmarks, model backbone, comparison methods, and evaluation settings. Appendix A.4 documents the data splits, baseline adaptations, learning and evolution protocols, metrics, and token accounting. The appendix additionally provides the implemented MDL cost (Appendix A.2), complete learning, execution, and repair procedures (Appendix A.3), the prompts used for skill induction and router authoring (Appendix A.6), task-level results, and a case study grounded in recorded trajectories. Together, these materials specify how to reconstruct the learned artifacts and reproduce the reported evaluations.

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, Daniel Klein, Matei Zaharia, and Omar Khattab. GEPA: reflective prompt evolution can outperform reinforcement learning. *CoRR*, abs/2507.19457, 2025. doi: 10.48550/ARXIV.2507.19457. URL `https://doi.org/10.48550/arXiv.2507.19457`.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24(6):377–380, 1987. doi: 10.1016/0020-0190(87)90114-1. URL `https://doi.org/10.1016/0020-0190(87)90114-1`.

Guillem Francès, Blai Bonet, and Hector Geffner. Learning general planning policies from small examples without supervision. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 11801–11808. AAAI Press, 2021. doi: 10.1609/AAAI.V35I13.17402. URL `https://doi.org/10.1609/aaai.v35i13.17402`.

Zhongxin Guo, Danrui Qi, Hanwen Gu, Peng Cheng, and Yongqiang Xiong. SKILL-DISCO: distilling and compiling agent traces into reusable procedural skills. *CoRR*, abs/2606.26669, 2026. doi: 10.48550/ARXIV.2606.26669. URL `https://doi.org/10.48550/arXiv.2606.26669`.

Roni Khardon. Learning to take actions. *Machine Learning*, 35(1):57–90, 1999.

John Langford. Tutorial on practical prediction theory for classification. *J. Mach. Learn. Res.*, 6: 273–306, 2005. URL `https://jmlr.org/papers/v6/langford05a.html`.

Dawei Liu, Zongxia Li, Hongyang Du, Xiyang Wu, Shihang Gui, Yongbei Kuang, and Lichao Sun. Graph of skills: Dependency-aware structural retrieval for massive agent skills. *CoRR*, abs/2604.05333, 2026. doi: 10.48550/ARXIV.2604.05333. URL https://doi.org/10.48550/arXiv.2604.05333.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *CoRR*, abs/2603.25158, 2026. doi: 10.48550/ARXIV.2603.25158. URL https://doi.org/10.48550/arXiv.2603.25158.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. *CoRR*, abs/2509.25140, 2025. doi: 10.48550/ARXIV.2509.25140. URL https://doi.org/10.48550/arXiv.2509.25140.

Jorma Rissanen. Modeling by shortest data description. *Autom.*, 14(5):465–471, 1978. doi: 10.1016/0005-1098(78)90005-5. URL https://doi.org/10.1016/0005-1098(78)90005-5.

Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Computing hierarchical finite state controllers with classical planning. *J. Artif. Intell. Res.*, 62:755–797, 2018. doi: 10.1613/JAIR.1.11227. URL https://doi.org/10.1613/jair.1.11227.

Javier Segovia-Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. Computing programs for generalized planning using a classical planner. *Artif. Intell.*, 272:52–85, 2019. doi: 10.1016/J.ARTINT.2018.10.006. URL https://doi.org/10.1016/j.artint.2018.10.006.

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized planning in PDDL domains with pretrained large language models. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, pp. 20256–20264. AAAI Press, 2024. doi: 10.1609/AAAI.V38I18.30006. URL https://doi.org/10.1609/aaai.v38i18.30006.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization for LLM agents. *CoRR*, abs/2403.02502, 2024. doi: 10.48550/ARXIV.2403.02502. URL https://doi.org/10.48550/arXiv.2403.02502.

Weihang Su, Jianming Long, Qingyao Ai, Yichen Tang, Changyue Wang, Yiteng Tu, and Yiqun Liu. Skill retrieval augmentation for agentic AI. *CoRR*, abs/2604.24594, 2026. doi: 10.48550/ARXIV.2604.24594. URL https://doi.org/10.48550/arXiv.2604.24594.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 16022–16076. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.850. URL https://doi.org/10.18653/v1/2024.acl-long.850.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024, 2024a. URL https://openreview.net/forum?id=ehfRiF0R3a.

Ruoyao Wang, Peter A. Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pp. 11279–11298.

Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.EMNLP-MAIN.775. URL https://doi.org/10.18653/v1/2022.emnlp-main.775.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024b.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-agent-e: Self-evolving mobile assistant for complex tasks. *CoRR*, abs/2501.11733, 2025a. doi: 10.48550/ARXIV.2501.11733. URL https://doi.org/10.48550/arXiv.2501.11733.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *CoRR*, abs/2504.06821, 2025b. doi: 10.48550/ARXIV.2504.06821. URL https://doi.org/10.48550/arXiv.2504.06821.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025c. URL https://proceedings.mlr.press/v267/wang25bx.html.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. Skillopt: Executive strategy for self-evolving agent skills. *CoRR*, abs/2605.23904, 2026. doi: 10.48550/ARXIV.2605.23904. URL https://doi.org/10.48550/arXiv.2605.23904.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *CoRR*, abs/2510.04618, 2025. doi: 10.48550/ARXIV.2510.04618. URL https://doi.org/10.48550/arXiv.2510.04618.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: LLM agents are experiential learners. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, pp. 19632–19642. AAAI Press, 2024. doi: 10.1609/AAAI.V38I17.29936. URL https://doi.org/10.1609/aaai.v38i17.29936.

A Appendix

A.1 Theoretical Analysis of Routing Generalization

This appendix proves Proposition 1 and Corollary 1. We adapt the trajectory-consistency argument of Khardon (1999) to skill-level routing, allowing a stochastic reference router and imperfect agreement with demonstrations.

A.1.1 Analysis Setting

We fix a skill library $\mathcal{L}$, a distribution over task instances, and a skill execution mechanism. A deterministic routing program maps the invocation context $c_k = (x, h_{t_k})$ to an element of $\mathcal{L} \cup \{\bot\}$, where $\bot$ invokes the fallback policy. The context construction is fixed, and all episodes terminate within a finite execution budget.

Let $R^\star$ be a reference router, which may be stochastic and history dependent. Its task success probability is $S^\star = S(R^\star, \mathcal{L}) > 0$. The reference and learned routers share the same execution mechanism, so that given the same execution history and selected skill, including $\bot$, they induce the same conditional distribution over subsequent actions, observations, and termination. This allows parameter binding and environment transitions to be stochastic.

Let $\rho_1, \ldots, \rho_n$ denote the skill-level trajectories of n independent executions of $R^\star$ sampled conditional on task success. For each ρ, its routing decisions are denoted by $\{(c_k, \sigma_k)\}_{k=0}^{H_\rho - 1}$, with contexts constructed from the complete underlying interaction history. The library, execution mechanism, and program encoding are fixed before the sample is drawn.

Define the trajectory disagreement loss as

$$\ell_{\mathrm{tr}}(R, \rho) = \mathbb{1}\left[\exists k \in \{0, \ldots, H_\rho - 1\} : R(c_k; \mathcal{L}) \neq \sigma_k\right]. \tag{8}$$

Its population and empirical values are

$$q_+(R) = \mathbb{E}_{\rho|\mathrm{success}}\big[\ell_{\mathrm{tr}}(R, \rho)\big], \qquad \widehat{q}_+(R) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\mathrm{tr}}(R, \rho_i). \tag{9}$$

Thus, $q_+(R)$ measures the probability of at least one routing disagreement along a random successful reference trajectory. Disagreement need not imply task failure, since an alternative skill sequence may also succeed.

These assumptions hold the skill library and execution mechanism fixed so that the analysis focuses on the routing program itself. This setting matches how SCRIPT refines the router, since candidate routers are compared on the same cached routing decisions while the library stays fixed. Under this setting, differences in task success between routers arise only from their routing decisions, which are exactly what offline verification checks.

A.1.2 Success Preservation

Proof of Proposition 1. Given a reference execution, run the learned router on the same task with the same execution randomness. At each routing point, if both routers select the same skill from the same history, the skill produces the same interaction and the two histories remain identical. By induction, a learned router that agrees with every routing decision of a successful reference execution reproduces that execution and therefore also succeeds. Hence,

$$\begin{aligned} S(R, \mathcal{L}) &\geq \Pr_{R^\star}\left[\mathrm{success} \wedge \ell_{\mathrm{tr}}(R, \rho) = 0\right] \\ &= S^\star \Pr_{R^\star}\left[\ell_{\mathrm{tr}}(R, \rho) = 0 \mid \mathrm{success}\right] \\ &= S^\star\big(1 - q_+(R)\big). \end{aligned} \tag{10}$$

The learned router may also succeed after a disagreement, so the expression is a lower bound. □

A.1.3 Generalization over Routing Programs

We bound $q_+(R)$ uniformly over routing programs using the Occam argument of Blumer et al. (1987) in its Hoeffding form (Langford, 2005), which charges each program by its description

length. Let $\mathcal{R}_b$ contain the admissible deterministic routing programs represented by at most b bits under a pre-specified binary encoding, where $b \geq 1$. The encoding includes all learned routing logic and associated data, while fixed infrastructure is shared. There are fewer than 2^{b+1} binary strings of length at most b, and therefore

$$|\mathcal{R}_b| < 2^{b+1}. \tag{11}$$

For $\delta \in (0, 1)$, define

$$C(b, \delta) = (b + 1) \ln 2 + \ln \frac{b(b + 1)}{\delta}. \tag{12}$$

Theorem 1 (Program-dependent generalization)**.** *Under the setting in Appendix A.1.1, with probability at least $1 - \delta$ over the sampled demonstrations, the bound*

$$S(R, \mathcal{L}) \geq S^\star \left[1 - \widehat{q}_+(R) - \sqrt{\frac{C(b, \delta)}{2n}} \right]_+ \tag{13}$$

holds simultaneously for every $b \geq 1$ and every $R \in \mathcal{R}_b$, where $[u]_+ = \max\{u, 0\}$.

Proof. For each $b \geq 1$, allocate failure probability

$$\delta_b = \frac{\delta}{b(b + 1)} \tag{14}$$

to $\mathcal{R}_b$. For a fixed $R \in \mathcal{R}_b$, the losses $\ell_{\mathrm{tr}}(R, \rho_i)$ are independent Bernoulli variables with mean $q_+(R)$, so Hoeffding's inequality gives

$$\Pr\left[q_+(R) > \widehat{q}_+(R) + u\right] \leq e^{-2nu^2}. \tag{15}$$

Taking a union bound over $\mathcal{R}_b$ yields

$$\Pr\left[\exists R \in \mathcal{R}_b,\ q_+(R) > \widehat{q}_+(R) + u\right] \leq 2^{b+1} e^{-2nu^2}. \tag{16}$$

Setting $u = \sqrt{C(b, \delta)/(2n)}$ bounds this probability by δ_b. Thus, except on an event of probability at most δ_b,

$$q_+(R) \leq \widehat{q}_+(R) + \sqrt{\frac{C(b, \delta)}{2n}} \qquad \text{for all } R \in \mathcal{R}_b. \tag{17}$$

A union bound over all $b \geq 1$ gives total failure probability at most

$$\sum_{b=1}^{\infty} \delta_b = \delta \sum_{b=1}^{\infty} \frac{1}{b(b + 1)} = \delta. \tag{18}$$

Thus, Equation 17 holds simultaneously over all program lengths with probability at least $1 - \delta$. Applying Proposition 1 and the nonnegativity of success probabilities establishes the claim. □

Proof of Corollary 1. The first claim is Equation 13. If $\widehat{q}_+(R) + \sqrt{C(b, \delta)/(2n)} \leq \epsilon$, the bracketed term is at least $1 - \epsilon$, which gives $S(R, \mathcal{L}) \geq (1 - \epsilon)S^\star$. □

Combined with success preservation, this bound turns agreement with recorded routing decisions into a lower bound on task success. Because it holds uniformly over programs, it covers the router selected by SCRIPT, and for the same empirical agreement it gives a tighter guarantee to a shorter router. This motivates controlling routing disagreement and program length jointly, and the code cost in Appendix A.2 serves as a practical proxy for the encoding length b.

A.2 Implemented MDL Cost

The implementation scores $L = L_{\mathrm{code}} + L_{\mathrm{errors}}$ on training data only. The code cost charges each syntax node of the routing program, excluding docstrings, and each literal it contains,

$$L_{\mathrm{code}}(R) = 6|\mathcal{N}(R)| + \sum_{v \in \mathcal{V}(R)} \ell(v), \tag{19}$$

where $\mathcal{N}(R)$ and $\mathcal{V}(R)$ denote the syntax nodes and literals of R. A skill name costs $\log_2 M$ bits with $M = |\mathcal{L}|$, a literal from a shared vocabulary built from training contexts and skill identifiers costs a few bits that decrease with its frequency, and any other literal costs 8 bits per character. Reusable conditions are therefore cheap, while memorizing a specific training value is expensive.

The error cost counts disagreements with recorded routing decisions. Repeated occurrences of the same decision are grouped into one distinct decision $u \in \mathcal{U}$ with occurrences $\mathcal{I}_u$, so that each distinct decision receives equal weight,

$$L_{\text{errors}}(R; \mathcal{D}_R) = \lambda(1 + \log_2 M) \sum_{u \in \mathcal{U}} \frac{1}{|\mathcal{I}_u|} \sum_{j \in \mathcal{I}_u} \mathbb{1}[R(c_j; \mathcal{L}) \neq \sigma_j], \tag{20}$$

where λ controls the trade-off between routing errors and program length. We use $\lambda = 16$ in all experiments.

A.3 SCRIPT Algorithms

A.3.1 Skill Construction

Given successful trajectories $\mathcal{T}_{\text{train}} = \{\tau_1, \ldots, \tau_n\}$, an LLM segments and compares executions across trajectories, grouping action segments that accomplish the same local subgoal and abstracting each group into a reusable skill (Wang et al., 2025a):

$$\{\sigma_i\}_{i=1}^{|\mathcal{L}|} = \text{LLM}_{\text{abs}}\left(\mathcal{T}_{\text{train}}\right), \qquad \sigma_i = \left(d_i, n_i, \left\{v_i^{(m)}(\theta_i^{(m)})\right\}_{m=1}^{M_i}\right), \tag{21}$$

where d_i is a concise functional description, n_i contains usage notes, and each $v_i^{(m)}$ is an executable variant with parameters $\theta_i^{(m)}$. A variant may contain either a single primitive action or multiple actions. Segments realizing the same subgoal but exhibiting different action structures are retained as separate variants, while instance-specific entities and values are replaced by parameters.

At execution time, the router selects only the skill identity σ_k. An LLM then jointly selects an execution variant and binds its parameters from the current context:

$$(m_k, \theta_k) = \text{LLM}_{\text{bind}}(\sigma_k, x, h_{t_k}), \qquad \Delta h_k = v_k^{(m_k)}(\theta_k). \tag{22}$$

Thus, skill construction captures reusable local execution, while SCRIPT learns the global skill-selection and composition logic separately. The complete skill construction and program-guided execution procedures are summarized in Algorithms 1 and 2, respectively.

A.3.2 Failure attribution criteria.

Given a failed trajectory ρ^-, the attribution model identifies the affected decision point k and classifies the failure into one of three repair types. AddSkill is used when the required local subgoal is not covered by the current skill library; ModifySkill when the selected skill is appropriate but its local execution fails to accomplish the intended subgoal; and ModifyRouter when an appropriate skill already exists in the library but is selected at the wrong context, omitted, or composed in an incorrect order. The corresponding repair is then applied and validated following Algorithm 3.

A.4 Experimental Details

Benchmarks. ScienceWorld (Wang et al., 2022) evaluates multi-step scientific reasoning in a text-based interactive environment. We use 18 task types from nine categories covering changes of state, measurement, plant growth, genetics, chemistry, search, life stages, lifespan comparison, and inclined planes. For each task type and each official split, we select 10 variations at evenly spaced indices and use all available variations when fewer than 10 exist, which yields 170 training, 156 development, and 164 test instances. AppWorld (Trivedi et al., 2024) evaluates cross-application reasoning and state-changing operations across 9 simulated applications and 457 APIs. Each scenario contains three task variants, and we use 30 training scenarios with 90 tasks, 19 development scenarios with 57 tasks, and the full Test-Normal split of 56 scenarios with 168 tasks. On both benchmarks, the development split serves as the *Seen* setting and the test split as the *Unseen* setting following prior work (Song et al., 2024).

Algorithm 1 Learning skills and the routing program

Require: Successful trajectories $\mathcal{T}_{\text{train}}$, coding agent A, refinement budget B
Ensure: Skill library $\mathcal{L}$, routing program R, routing data $\mathcal{D}_R$

$(\mathcal{L}, \{\rho_i\}_{i=1}^n) \leftarrow \textsc{AbstractAndAnnotate}(\mathcal{T}_{\text{train}})$
$\mathcal{D}_R \leftarrow \textsc{ExtractRoutingPairs}(\{\rho_i\}_{i=1}^n)$
$R \leftarrow \textsc{InitializeRouter}(A, \mathcal{L}, \mathcal{D}_R)$
for $b = 1, \ldots, B$ **do**
  $E \leftarrow \textsc{VerifyCached}(R, \mathcal{D}_R)$
  $\mathcal{C} \leftarrow \textsc{ProposeRouterEdits}(A, R, E)$
  $\widehat{R} \leftarrow \arg\min_{R' \in \mathcal{C} \cup \{R\}} L(R'; \mathcal{D}_R)$
  if $L(\widehat{R}; \mathcal{D}_R) < L(R; \mathcal{D}_R)$ **then**
    $R \leftarrow \widehat{R}$
  end if
end for
return $(\mathcal{L}, R, \mathcal{D}_R)$

Algorithm 2 Program-guided skill execution

Require: Task x, initial observation o_0, skill library $\mathcal{L}$
Require: Learned router R and fallback policy F
Ensure: Interaction history h and skill-level trajectory ρ

$h \leftarrow (o_0)$; $\rho \leftarrow \langle\rangle$
while the task is not terminal and the execution budget remains **do**
  $c \leftarrow (x, h)$; $\sigma \leftarrow R(c; \mathcal{L})$
  if $\sigma = \perp$ **then**
    $\Delta h \leftarrow \textsc{FallbackStep}(F, x, h)$
  else
    $(m, \theta) \leftarrow \textsc{SelectVariantAndBind}(\sigma, x, h)$
    $(\Delta h, o') \leftarrow \textsc{ExecuteSkill}(\sigma, m, \theta, h)$
    $\rho \leftarrow \rho \,\|\, (c, \sigma, \theta, o')$
  end if
  $h \leftarrow h \,\|\, \Delta h$
end while
return (h, ρ)

Baselines. ReAct (Yao et al., 2023) interleaves reasoning and acting without retaining cross-task knowledge. ExpeL (Zhao et al., 2024) and ReasoningBank (Ouyang et al., 2025) retrieve textual insights or reasoning strategies distilled from trajectories. Trace2Skill (Ni et al., 2026) consolidates trajectory-local lessons into declarative skills, while Graph-of-Skills (Liu et al., 2026) performs dependency-aware retrieval over a skill graph. ACE (Zhang et al., 2025) maintains an evolving playbook. When a method was not designed for either benchmark, we adapt its learning mechanism while preserving its original experience representation. All methods receive the same task experience, LLM backbone, and environment interface.

Learning and evaluation protocol. Methods first receive experience from the training split and construct their artifacts following their native learning protocols. Trace2Skill, ACE, GoS, and SCRIPT use successful gold trajectories, whereas ExpeL and ReasoningBank acquire experience through their original iterative execution and reflection procedures. They are then evaluated without further updates on the development and held-out test splits. SCRIPT uses successful training trajectories to induce its skill library and routing supervision. Router candidates minimize the training-only MDL objective in Equation 5; development and test points never contribute to this score. In the additional evolution setting, each adaptable baseline continues learning on the development split using its original update mechanism before evaluation on the held-out test split. SCRIPT uses development failures for component-specific attribution and validated repair.

Metrics. Let $\mathcal{T}$ be the set of evaluation tasks and $\mathcal{Z}$ the set of scenarios, where each scenario $z \in \mathcal{Z}$ contains tasks $\mathcal{T}_z$. For ScienceWorld, let $\mathcal{G}$ denote the set of task types and $\mathcal{T}_g$ the evaluated variations of type g. For task i, let $s_i \in [0, 1]$ be its normalized score and $y_i \in \{0, 1\}$ indicate

Algorithm 3 Failure attribution and targeted repair

Require: Failed development trajectory ρ^- for task x, skill library $\mathcal{L}$, router R
Require: Routing data $\mathcal{D}_R$, regression cases $\mathcal{Q}_{\text{dev}}$, coding agent A
Ensure: Updated $\mathcal{L}$, R, $\mathcal{D}_R$, and $\mathcal{Q}_{\text{dev}}$

$(z, \eta, k) \leftarrow \textsc{AttributeAndPropose}(A, \rho^-, \mathcal{L}, R)$
if $z = \textsc{ModifyRouter}$ **then**
  $\tilde{\boldsymbol{\sigma}} \leftarrow \textsc{ProposeSkillSequence}(\eta, \rho^-, k)$
  $(v, \rho^+) \leftarrow \textsc{ReplayFixedSequence}(x, \rho^-_{<k}, \tilde{\boldsymbol{\sigma}}, \mathcal{L})$
  if v improves the task score **then**
    $\Delta\mathcal{D}_R \leftarrow \textsc{ExtractRoutingPairs}(\rho^+)$
    $\mathcal{D}'_R \leftarrow \mathcal{D}_R \cup \Delta\mathcal{D}_R$
    $R' \leftarrow \textsc{RefineRouter}(A, R, \mathcal{D}'_R)$
    $u \leftarrow \textsc{VerifyRouter}(R', \mathcal{D}'_R, \mathcal{Q}_{\text{dev}})$
    if u passes **then**
      $R \leftarrow R'$; $\mathcal{D}_R \leftarrow \mathcal{D}'_R$
      $\mathcal{Q}_{\text{dev}} \leftarrow \mathcal{Q}_{\text{dev}} \cup \{(\rho^+, k)\}$
    end if
  end if
else ▷ $z \in \{\textsc{AddSkill}, \textsc{ModifySkill}\}$
  $\mathcal{L}' \leftarrow \textsc{ApplySkillEdit}(\mathcal{L}, z, \eta)$
  $(v, \rho^+) \leftarrow \textsc{TargetedReplay}(x, \rho^-_{<k}, \mathcal{L}', R)$
  if v improves the task score **then**
    $\mathcal{L} \leftarrow \mathcal{L}'$
    $\mathcal{Q}_{\text{dev}} \leftarrow \mathcal{Q}_{\text{dev}} \cup \{(\rho^+, k)\}$
  end if
end if
return $(\mathcal{L}, R, \mathcal{D}_R, \mathcal{Q}_{\text{dev}})$

whether it passes all task-level evaluation tests. We report

$$\begin{aligned} \text{Score} &= \begin{cases} \dfrac{100}{|\mathcal{G}|} \displaystyle\sum_{g \in \mathcal{G}} \frac{1}{|\mathcal{T}_g|} \sum_{i \in \mathcal{T}_g} s_i, & \text{ScienceWorld}, \\ \dfrac{100}{|\mathcal{T}|} \displaystyle\sum_{i \in \mathcal{T}} s_i, & \text{AppWorld}, \end{cases} \\ \text{SR} &= \frac{100}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} y_i, \\ \text{Scenario SR} &= \frac{100}{|\mathcal{Z}|} \sum_{z \in \mathcal{Z}} \prod_{i \in \mathcal{T}_z} y_i. \end{aligned} \tag{23}$$

On AppWorld, Success Rate and Scenario Success Rate correspond to the official Task Goal Completion and Scenario Goal Completion metrics (Trivedi et al., 2024). On ScienceWorld, a task succeeds when it reaches the maximum score, and each task type is treated as a scenario, so Scenario Success Rate is the percentage of task types in which every evaluated variation succeeds.

We measure efficiency by token consumption. *Learning tokens* denote all LLM tokens consumed during artifact construction on the training split and are reported as the average per training sample, while *evolution tokens* denote all LLM tokens consumed during additional adaptation on the development split. For routing-based methods, routing accuracy is the fraction of recorded decision contexts on which the router selects the skill used in the successful trajectory,

$$\text{Acc}(R) = \frac{1}{|\mathcal{D}|} \sum_{(c,\sigma) \in \mathcal{D}} \mathbb{1}\big[R(c; \mathcal{L}) = \sigma\big], \tag{24}$$

where $\mathcal{D}$ is the set of recorded routing decisions being evaluated.

A.5 ScienceWorld Results by Task Type

Table 6 reports the train-only Score and Success Rate for each ScienceWorld task type before evolution.

Table 6: Performance by ScienceWorld task type. Success Rate is abbreviated as Succ. All values are percentages.

Task type	ReAct		ReasoningBank		ExpeL		Trace2Skill		GoS		ACE		**SCRIPT**	
	Score	Succ.	Score	Succ.	Score	Succ.	Score	Succ.	Score	Succ.	Score	Succ.	Score	Succ.
boil	70.56	33.3	97.41	88.9	60.37	44.4	86.67	77.8	75.93	33.3	83.33	66.7	88.89	88.9
freeze	51.85	11.1	55.56	11.1	60.74	33.3	65.56	33.3	64.07	22.2	59.26	22.2	**76.89**	**55.6**
find-living-thing	100.00	100.0	100.00	100.0	90.83	80.0	100.00	100.0	100.00	100.0	100.00	100.0	100.00	100.0
find-plant	97.50	90.0	95.00	80.0	81.67	70.0	91.67	90.0	95.83	80.0	100.00	100.0	100.00	100.0
grow-fruit	41.96	20.0	33.75	10.0	67.86	50.0	34.29	10.0	28.75	10.0	28.39	0.0	64.20	50.0
grow-plant	62.59	50.0	60.53	40.0	86.41	80.0	61.47	30.0	69.21	30.0	75.41	50.0	67.10	50.0
chemistry-mix	64.29	50.0	91.67	87.5	91.67	87.5	89.58	87.5	78.87	75.0	83.63	75.0	**100.00**	**100.0**
chemistry-mix-paint	77.78	66.7	57.78	44.4	51.11	33.3	84.44	77.8	76.67	66.7	92.22	88.9	**100.00**	**100.0**
lifespan-longest	95.00	90.0	70.00	40.0	100.00	100.0	100.00	100.0	87.50	80.0	100.00	100.0	95.00	90.0
lifespan-shortest	70.00	40.0	70.00	40.0	85.00	70.0	87.50	80.0	90.00	80.0	**95.00**	**90.0**	80.00	70.0
life-stages-1	48.40	40.0	37.47	0.0	77.87	60.0	87.40	80.0	100.00	100.0	89.00	80.0	100.00	100.0
life-stages-2	69.50	25.0	69.50	25.0	14.50	0.0	70.50	25.0	68.50	25.0	79.00	50.0	**100.00**	**100.0**
incl.-plane-angle	27.00	10.0	54.50	40.0	68.00	60.0	68.00	50.0	57.00	40.0	92.00	90.0	**100.00**	**100.0**
incl.-plane-friction	78.50	70.0	78.00	70.0	85.50	80.0	77.00	70.0	77.50	60.0	**100.00**	**100.0**	95.00	90.0
mendelian-known	56.70	50.0	83.20	80.0	81.20	80.0	42.70	30.0	48.30	40.0	58.50	50.0	**100.00**	**100.0**
mendelian-unknown	18.20	0.0	16.20	0.0	31.80	20.0	34.10	20.0	**39.70**	**30.0**	28.60	10.0	29.30	10.0
melting-point	85.43	70.0	91.70	80.0	74.03	70.0	48.23	30.0	77.13	50.0	74.57	20.0	81.80	70.0
use-thermometer	45.94	30.0	78.54	70.0	89.79	80.0	84.69	70.0	91.35	70.0	100.00	100.0	100.00	100.0
Average / Overall	**64.51**	**48.17**	**68.93**	**52.4**	**72.13**	**63.4**	**72.99**	**59.1**	**73.68**	**54.9**	**79.94**	**66.5**	**87.68**	**80.49**

Table 7: Use of learned skills and the fallback agent on ScienceWorld, with 156 Seen tasks and 164 Unseen tasks.

Split	Skills (%)	Fallback agent (%)
Seen	88.9	11.1
Unseen (before evolution)	82.1	17.9
Unseen (after evolution)	**86.2**	**13.8**

As shown in Table 7, attribution-guided repair increases skill-based execution on Unseen episodes from 82.1% to 86.2% and reduces fallback-agent usage from 17.9% to 13.8%, a 4.1-point shift in each case. Because repairs are learned from development experience only, this shift indicates that the repaired skills generalize to previously unseen test episodes.

A.6 Prompts Used in SCRIPT

Skill Induction Prompt

The abstraction model receives the task instruction, multiple successful runs of the same task under different environment variations, environment rules, and optional revision feedback. The prompt enforces the following principles:

1. **Observable, progress-oriented segmentation.** A skill is a contiguous action sequence that achieves one meaningful subgoal and leaves the environment observably closer to task completion. Adjacent actions that jointly create one effect remain together, while setup actions are retained in the skill they enable. Every primitive step must appear exactly once and in its original order.
2. **Effect-based abstraction.** Segments are grouped primarily by semantic effect rather than surface action similarity. Executions with the same action structure use parameter slots for differing entities or values. Executions with the same effect but materially different action structures become variants of one skill. Segments with different effects remain separate skills.
3. **Reusable parameters and task-decided slots.** Task- or environment-specific objects, locations, thresholds, directions, and values are replaced by typed slots, with the same slot reused whenever multiple actions refer to the same entity. A parameter belongs to `task_slots` whenever the task determines its value. Values chosen from the runtime environment are not marked as task slots.

The model returns only structured JSON. Each skill contains `skill_id`, `name`, action `templates`, optional `variants`, typed `parameters`, `task_slots`, an observable `when_to_run` condition, optional `usage_notes`, and optional `repeat_until` rules. For each demonstration, a `sequences` en-

try lists the ordered skill invocations, concrete parameter values, selected variant, and covered step span. These sequences form the skill-level trajectories used to construct the routing dataset.

Router Authoring Prompt

The coding agent implements one shared Python policy, `choose(cands, task, history)`, that returns the identifier of the next skill or abstains with `None`. Here `cands` contains the available skill library, `task` is the current task instruction, and `history` is the complete ordered execution history. The router may return only an existing candidate skill; it cannot invent skills or emit primitive actions. A fixed prelude handles the trivial cases of one or zero candidates.

The prompt treats the labeled context–skill pairs in $\mathcal{D}_R$ as evidence for a policy shared across task variations rather than cases to memorize. It organizes router authoring into four stages:

1. **Inspect before coding.** The agent first examines the label distribution over all routing points, representative complete histories, task differences across variations, and facts shared by or separating neighboring skill labels. Summary and diagnostic operations analogous to `overview`, `explain`, `common`, and `diff` help derive conditions from recorded decisions rather than a few examples.
2. **Validate observable tests.** Before using a substring, regular expression, or parser, the agent evaluates it over the routing points and checks its distribution jointly with the correct labels. This prevents constant predicates or parsers from silently making later branches unreachable.
3. **Repair locally from attributed errors.** The verifier reports accuracy, incorrect points, unreachable returns, and the source line that produced each decision. The agent inspects the misclassified point and confused skill pair, edits the responsible fragment, and rescans the cached decisions. Accuracy-decreasing edits are reverted, and shared semantic conditions are preferred to variation-specific exceptions.
4. **Compress without changing decisions.** Behavior-preserving edits remove dead branches, merge conditions with the same return value, and eliminate redundant tests. Compression is accepted only when all cached predictions remain unchanged and the code description length decreases.

Candidate routers must compile, execute without exceptions on every cached point, and return only a candidate skill identifier or `None`. Among these routers, the retained version minimizes the MDL score in Equation.