COMPILING MIXED AGENT TRACES INTO GUARDED EXECUTABLE WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents often solve repeated tasks by reconstructing similar tool-use procedures with different inputs. Although storing reusable local functions can reduce low-level execution effort, the model still needs to reason about how to compose these functions for every new request. We propose Trace2Workflow, a framework that converts successful interaction traces into reusable executable workflows. Instead of repeatedly planning from scratch, the agent can directly reuse a learned workflow when its applicability conditions are satisfied. This reduces the reasoning burden of the model and avoids unnecessary LLM calls. When no workflow is applicable, the agent falls back to normal model-based reasoning, which preserves robustness on unseen tasks. We evaluate Trace2Workflow on ALFWorld and AppWorld under a continual setting. We compile previously successful task-level compositions, rather than only local skills, into guarded executable workflows whose bindings and observable effects are checked at runtime. These results show that reusable workflows can amortize repeated composition reasoning across tasks, reducing both model inference burden and execution cost without sacrificing task performance.

1 INTRODUCTION

Large language model agents have shown strong ability in interactive tasks that require planning, tool use, and multi-step reasoning. However, many agent tasks are highly repetitive. Different requests often share the same underlying procedure and differ only in their objects, parameters, or goals. Existing agents usually reconstruct the same tool composition from scratch for each new request. This repeated reasoning increases the number of LLM calls and leads to unnecessary inference cost.

Existing work reuses past experience through reasoning traces, executable code, workflow memory, and reusable skills (Yao et al., 2023; Wang et al., 2024b; 2025b;a; Ouyang et al., 2026; Zheng et al., 2025). Recent studies further explore adaptive context management and agent improvement through evolving memories or contexts (Zhang et al., 2026; Ouyang et al., 2026). Other works search over agent architectures and workflows (Hu et al., 2025; Zhang et al., 2025a; Shang et al., 2025). These methods reduce repeated low-level work, but an agent may still need to decide how to compose reusable components for each new task. We focus on this repeated task-level composition. Our key observation is simple: once an agent has solved a useful composition, the composition itself can become a reusable executable object. The agent can then reuse the decision structure instead of reconstructing it with the LLM.

We introduce Trace2Workflow, a framework that converts agent interaction traces into reusable executable workflows. The framework operates in three stages. First, it extracts parameterized local programs from observed interaction patterns and removes candidates that are contradicted by available evidence. Second, it composes validated local programs and primitive tools into guarded workflow graphs that preserve data dependencies between actions. Third, when a new request arrives, it binds the request to a compatible workflow and executes it directly. If the workflow is not applicable or an expected effect cannot be verified, control is returned to the LLM from the current state. The overall framework is illustrated in Figure 1.

We separate the evidence used for learning and validating workflows. Successful traces provide candidate compositions, while later executions test whether the generalized workflow remains valid under new inputs and object bindings. This separation prevents a workflow from being treated as

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

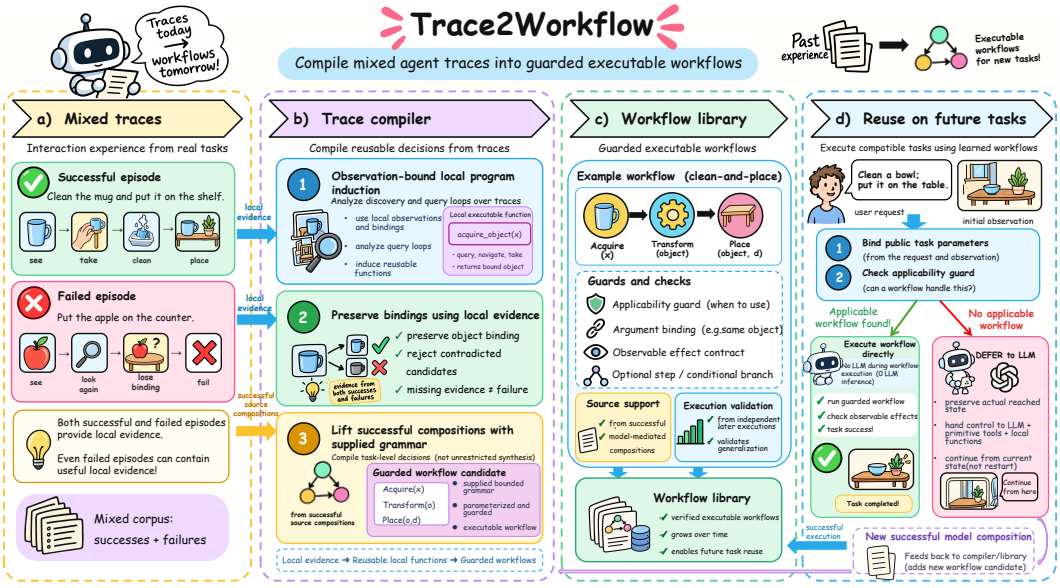

Figure 1: Overview of Trace2Workflow. Mixed traces provide evidence for local program induction. Validated local programs are compiled into guarded workflows that preserve data dependencies. On future tasks, compatible workflows execute without model inference, while failed execution defers to the LLM from the current state.

reliable simply because it was derived from one successful episode. As more tasks are solved, newly validated compositions can be added to the workflow library and reused in subsequent tasks.

We evaluate Trace2Workflow on ALFWorld (Shridhar et al., 2021) and AppWorld (Trivedi et al., 2024) under a continual setting. The learned workflow library is frozen during final evaluation. Compared with matched baselines that reuse the same local skills but still rely on the LLM for task-level composition, Trace2Workflow achieves stronger task performance while substantially reducing LLM calls. These results show that reusing workflow-level decisions can reduce repeated reasoning and inference cost. Our contributions are summaries as follows:

- We formulate reuse at the level of the composition between reusable parts. A successful task procedure is compiled into a parameterized workflow with explicit bindings, an applicability guard, and observable completion checks. A new request reuses the decision structure instead of rebuilding it.
- We propose Trace2Workflow, which induces local programs from successful and failed traces, verifies them against their own local goals, and compiles the validated programs into guarded workflows. A successful trace only creates a candidate, and later executions decide its status, so the workflow library corrects itself.
- Workflow execution does not call LLMs, and defers to it from the current state once a check fails. Trace2Workflow outperforms ReAct, CodeAct and Skill-DisCo with far fewer model turns on ALFWorld and AppWorld. It completes most ALFWorld tasks without LLM calls.

2 Problem Formulation

Traces and observable evidence. For a request q, each primitive action a_t produces an observation o_{t+1}. We record an episode as $\tau = (q, o_0, a_0, o_1, \ldots, a_{T-1}, o_T, y)$, where o_0 is the initial observation and y is the official task outcome. Traces also include generated code and tool results. The full environment state need not be observable. Our corpus $\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$ contains both successful and failed episodes. A successful episode may contain invalid actions, while a failed episode may still contain valid and reusable fragments. We therefore use local observable evidence to identify reusable fragments from both types of episodes.

Workflows as partial executable policies. A workflow is $W = (\mathbf{x}, V, \beta, g, \Phi)$. Here, $\mathbf{x}$ denotes formal inputs, and V is an ordered set of tool-call nodes. The binding rules β specify each node's arguments. An argument can be a constant, a workflow input, or a field returned by an earlier node. The predicate g defines when the workflow applies. The effects Φ specify its expected observable results. Nodes may depend on prior observations and call compiled local functions. No node calls the model. We retain each workflow's source records and later execution evidence. Let h be the observed interaction history. The binding θ assigns values from the request and public observations to $\mathbf{x}$. The executor returns

$$\mathrm{Exec}(W, \theta, h) = (\eta, h'), \qquad \eta \in \{\mathrm{SUCCESS}, \mathrm{DEFER}\}. \tag{1}$$

Here, h' extends h with all actions and observations from execution. SUCCESS means that the task is complete. On DEFER, the model continues from h' with the remaining primitive-action budget. The environment and budget are not reset.

The compilation problem. We process a task stream $q_1, \ldots, q_N$. Task q_i uses the workflow library $\mathcal{L}_{i-1}$. After the episode ends, the compiler updates it to $\mathcal{L}_i$ using only evidence available so far. Compilation follows a given program language and relies on observed bindings and effects. We do not assume access to hidden object locations or a complete environment transition graph. Our goal is to reduce LLM turns while maintaining task success. We evaluate success rate and LLM turns over all tasks.

3 TRACE2WORKFLOW

Given the trace corpus $\mathcal{D}$ defined in Section 2, Trace2Workflow learns reusable local programs and task-level workflows. The method has three stages. First, we extract and verify local programs. Second, we compile successful task compositions into parameterized workflows. Third, we match and execute workflows on new tasks and update the workflow library.

3.1 LOCAL PROGRAM INDUCTION VIA GOAL-BASED VERIFICATION

We evaluate each local program by its own completion condition. We do not use the final task outcome as the local program label.

Candidate extraction. We split the training corpus $\mathcal{D}$ into mining and validation episodes. We extract code fragments from both successful and failed episodes. Each fragment has explicit inputs and outputs. We preserve its tool calls, branches, and data dependencies. We replace task-specific values with input parameters. For each local goal, these fragments form a candidate set $\mathcal{P}_0$.

Local completion conditions. Each local goal has a precondition P and a completion check Q. The precondition $P(x, h)$ determines whether the program can run. The completion check $Q(x, h, z, h')$ evaluates the returned value z and the observed effects in h'. It returns

$$Q(x, h, z, h') \in \{1, 0, \bot\},$$

where 1 means completion, 0 means violation, and $\bot$ means insufficient evidence. We derive these checks from the local goal, task instructions, and tool specifications. All candidates for the same goal use the same checks.

Execution verification. For each local goal, we construct a validation set $\mathcal{T}$. Each test e provides an input x and an observed history h. We keep tests with $P(x, h) = 1$. We then execute candidate p with a fixed action budget b:

$$(z, h') = \mathrm{Run}_b(p, e). \tag{2}$$

We assign a local verdict:

$$c(p, e) = \begin{cases} +, & Q(x, h, z, h') = 1, \\ -, & Q(x, h, z, h') = 0, \\ ?, & Q(x, h, z, h') = \bot. \end{cases} \tag{3}$$

Program errors and budget exhaustion give a negative verdict. Missing evidence gives an unknown verdict.

Program selection. For $v \in \{+, -, ?\}$, let

$$N_v(p) = |\{e \in \mathcal{T} : c(p, e) = v\}|. \tag{4}$$

We keep candidates with no verified failure and enough positive evidence:

$$\mathcal{P}_{\text{keep}} = \left\{ p \in \mathcal{P}_0 \;\middle|\; N_-(p) = 0, \quad N_+(p) \geq m, \quad \frac{N_+(p)}{|\mathcal{T}|} \geq \rho \right\}. \tag{5}$$

Here, m is the minimum number of positive tests, and ρ is the minimum positive-test ratio. We select the lowest-cost candidate among the retained programs for the same goal. The selected program becomes a reusable local function.

3.2 Compiling Traces into Parameterized Workflows

Local functions reduce repeated low-level decisions. The model may still need to connect these functions for every task. So, we compile successful task compositions into reusable workflows.

Workflow construction. Let $\mathcal{F}$ be the local-function library. Given a successful trace τ, we build

$$W = \text{Compile}(\tau, \mathcal{F}) = (\mathbf{x}, V, \beta, g, \Phi). \tag{6}$$

Here, $\mathbf{x}$ contains workflow inputs. The ordered set $V = (v_1, \ldots, v_n)$ contains tool or local-function calls. The binding rules β define the argument source of each node. The guard g defines when the workflow can start. The set Φ contains node completion checks.

We turn each tool call or local-function call into one workflow node. We preserve the call order from the source program. We also preserve state-changing calls.

Argument binding. We trace each argument back to its source. Task-specific values become workflow inputs. Earlier node results become variable references. Fixed values remain constants. For argument k of node v_j,

$$\beta_{j,k} = \begin{cases} x_\ell, & \text{workflow input,} \\ z_i[r], \quad i < j, & \text{earlier result,} \\ c, & \text{constant.} \end{cases} \tag{7}$$

Here, x_ℓ is workflow input ℓ. The variable z_i is the result of node v_i. The index r selects one field or item. We infer these links from code dependencies and call logs. Each workflow execution uses results from the current task.

Guards and completion checks. We build g from input requirements and entry-node preconditions. We keep branch conditions from the source program. Each branch uses current workflow inputs or earlier node results. We build Φ from local-function checks and observable tool effects. These guards and checks control workflow execution on new tasks.

3.3 Selective Execution and Workflow Validation

Algorithm 1 connects workflow reuse with workflow acquisition. It takes the local-function library $\mathcal{F}$ from Section 3.1 and a task stream. It starts with an empty workflow library. Each task supplies execution evidence for later reuse.

Workflow selection. For task q_i, the matcher searches the workflow library $\mathcal{L}_{i-1}$. It binds inputs θ from the request and initial observations. It selects a workflow with complete bindings and a satisfied guard $g(\theta, h_0) = 1$. The executor then receives the workflow and its bindings. An unmatched task goes to the model. The model composes tools and local functions to solve it.

Workflow execution. The executor runs workflow nodes in order. Before each node, it checks the arguments and branch condition. After each node, it checks the returned result and observable effect. The executor stops at task success. A failed or unresolved check triggers DEFER. Missing arguments also trigger DEFER. The model then continues from the current history. It uses the remaining action budget.

Workflow validation. A successful source trace creates a candidate workflow. Later executions provide validation evidence. For workflow W on task q_i, we record

$$\nu_i(W) = \begin{cases} +, & W \text{ reaches task success,} \\ -, & \text{a check confirms an execution error,} \\ ?, & \text{the evidence is incomplete.} \end{cases} \tag{8}$$

Algorithm 1 Workflow learning and execution

Require: $\mathcal{F}$; task stream $(q_i)_{i=1}^{N}$
$\mathcal{L}_0 \leftarrow \varnothing$
for $i = 1, \ldots, N$ **do**
  Observe h_0 for q_i; $h \leftarrow h_0$
  $(W, \theta) \leftarrow \mathrm{Match}(\mathcal{L}_{i-1}, q_i, h_0)$
  if W matches **then**
    $(\eta, h) \leftarrow \mathrm{Exec}(W, \theta, h_0)$
    Record $\nu_i(W)$
  end if
  if the task remains incomplete **then**
    Continue with model$(\mathcal{F}, h)$
  end if
  Record the trace τ_i
  $\mathcal{L}_i \leftarrow \mathcal{L}_{i-1}$
  Update $\mathcal{L}_i$ with $\nu_i(W)$, if available
  if the model completes the task **then**
    $W_{\mathrm{new}} \leftarrow \mathrm{Compile}(\tau_i, \mathcal{F})$
    if compilation succeeds **then**
      Add W_{new} to $\mathcal{L}_i$ as a candidate
    end if
  end if
end for
return $\mathcal{L}_N$

The learner records this verdict before model fallback. A positive verdict activates the candidate. A negative verdict increases its failure count. An unknown verdict preserves its current state. The learner disables a workflow after two verified failures since its latest success.

Library update. After task q_i, we record its trace τ_i and update the library to $\mathcal{L}_i$. The learner uses $\nu_i(W)$ to update the status of an executed workflow. A successful model-led trace provides a new task composition. The compiler converts this composition into a candidate workflow. It preserves argument dependencies and branch conditions (Sect. 3.2). The learner adds the candidate to $\mathcal{L}_i$. Later tasks test its execution and determine its status.

This loop connects our two levels of reuse. The model uses local functions to build a successful task procedure. The compiler stores that procedure as executable code. The agent can then reuse its decision structure with new inputs. The executor uses current observations to check each step.

3.4 Case Study

We use an ALFWorld example to show how Trace2Workflow reuses a learned task procedure on a new task. The source task is "put a clean apple in diningtable." Table 1 shows the source and unseen-task bindings; Figure 2 shows the compiled workflow reused across both tasks.

Table 1: Task bindings. Inputs and runtime values for source and unseen tasks.

	Source	Unseen
Object type	`apple`	`cloth`
Destination	`diningtable 1`	`cabinet 1`
Processor	`sinkbasin 1`	`sinkbasin 1`
Runtime object	`apple 1`	`cloth 1`
Object location	`countertop 1`	`cabinet 4`

Figure 2: Reusable workflow. Parameterized execution.

```
a ← acquire_object(target_type)
goto(processor)
clean(a.object, processor)
r ← goto(destination)
if needed:
  open_receptacle(destination)
put(a.object, destination)
```

The compiler turns task-specific values into workflow inputs. It also preserves data dependencies. The object returned by `acquire_object` becomes the input of `clean` and `put`. The workflow checks the current observation before opening the destination.

The unseen task is "clean some cloth and put it in cabinet." The workflow binds the new object and destination. It finds `cloth 1` in `cabinet 4`. It cleans the cloth in `sinkbasin 1`. It then moves to `cabinet 1`. The workflow observes that the cabinet is closed and opens it. It finally places the cloth in the cabinet. The workflow completes the task with 8 environment actions and 0 model calls. This case shows that Trace2Workflow reuses the task structure with new inputs, new intermediate results, and current observations.

Table 2: Main results on ALFWorld and AppWorld. **Bold** denotes the best result, and underlined denotes the second best. Turns is the average number of model turns per task. Zero-LM is the number of successful tasks completed without model calls.

Dataset	Split	Method	Success	SR ↑	Turns ↓	Zero-LM ↑
ALFWorld	Seen	ReAct	119/140	85.00	14.893	0
		CodeAct	104/140	74.29	3.236	0
		Skill-DisCo	131/140	93.57	2.721	0
		Trace2Workflow	**137/140**	**97.86**	**0.879**	**96**
	Unseen	ReAct	122/134	91.04	13.507	0
		CodeAct	101/134	75.37	3.209	0
		Skill-DisCo	124/134	92.54	2.776	0
		Trace2Workflow	**132/134**	**98.51**	**0.858**	**99**
AppWorld	Test-normal	ReAct	7/60	11.67	32.167	0
		CodeAct	47/60	78.33	20.517	0
		Skill-DisCo	44/60	73.33	21.833	0
		Trace2Workflow	**50/60**	**83.33**	**11.017**	**4**
	Test-challenge	ReAct	0/60	0.00	34.400	0
		CodeAct	37/60	61.67	26.500	0
		Skill-DisCo	33/60	55.00	26.817	0
		Trace2Workflow	**42/60**	**70.00**	**14.950**	**2**

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Datasets. We evaluate Trace2Workflow on ALFWorld (Shridhar et al., 2021) and AppWorld (Trivedi et al., 2024). ALFWorld contains multi-step household tasks in a text-based environment. AppWorld contains digital tasks across simulated applications. Agents solve these tasks through text actions or API calls.

Learning and evaluation protocol. For both benchmarks, we use the training split to learn local programs and workflows. Local programs are extracted from successful and failed traces and verified on separate validation episodes. Candidate workflows are built from successful training traces. No evaluation task is used for program or workflow construction. For ALFWorld, we evaluate on 140 Seen and 134 Unseen tasks. For AppWorld, we use 60 `test-normal` and 60 `test-challenge` tasks. During evaluation, the learned workflow library is fixed and no new workflow is added.

Baselines and implementation. We compare Trace2Workflow with ReAct (Yao et al., 2023), CodeAct (Wang et al., 2024b), and Skill-DisCo (Guo et al., 2026). All methods use Qwen3.5-27B as the backbone model. Experiments are run on two NVIDIA A800 GPUs. We use the same task settings and environment interfaces for all methods. Workflow execution does not call the language model. If no workflow applies or execution cannot continue, the model continues from the current state.

Metrics. We report Success, success rate (SR), average LLM turns (Turns), and zero-model completions (Zero-LM). Turns includes failed attempts and LLM fallback. Zero-LM counts successful tasks completed without LLM calls. Tool actions inside local programs and workflows are still counted in the task budget.

4.2 MAIN RESULTS

Table 2 reports the main results on ALFWorld and AppWorld. We evaluate both task success and model usage. Higher SR and Zero-LM are better, while lower Turns are better.

Overall performance. Trace2Workflow achieves the highest success rate and the lowest average model turns on all four evaluation sets. On ALFWorld Seen, it reaches a 97.86% success rate with only 0.879 model turns per task. On ALFWorld Unseen, it achieves 98.51% success with 0.858 turns. Compared with Skill-DisCo, the strongest baseline on ALFWorld, Trace2Workflow improves the success rate by 4.29 and 5.22 percentage points on Seen and Unseen, respectively. It also reduces model turns by 67.7% and 70.4%. The results show that workflow-level reuse provides ben-

Table 3: Ablation study of Trace2Workflow. Higher SR and Zero-LM are better, while lower Turns are better. **Bold** denotes the best result and underlined denotes the second best.

Method	ALFWorld Unseen			AppWorld test-normal			AppWorld test-challenge		
	SR ↑	Turns ↓	Zero-LM ↑	SR ↑	Turns ↓	Zero-LM ↑	SR ↑	Turns ↓	Zero-LM ↑
Local Programs Only	96.27	2.806	0	80.00	18.350	0	63.33	23.217	0
Direct Workflow Compilation	97.01	1.214	84	81.67	13.267	2	66.67	17.483	1
w/o Local Verification	94.78	1.486	68	76.67	15.583	1	60.00	20.117	0
w/o Workflow Validation	96.27	**0.742**	91	80.00	**10.420**	0	65.00	**13.880**	**4**
Trace2Workflow	**98.51**	0.858	**99**	**83.33**	11.015	**4**	**70.00**	14.950	2

efits beyond reusing individual skills. Skill-DisCo can reuse procedural components, but the model still needs to decide how these components should be combined for each new task. In contrast, Trace2Workflow stores the composition itself as an executable workflow. Therefore, repeated planning decisions can be replaced by direct execution when a matching workflow is available. This leads to both fewer model calls and higher task reliability. The improvement on unseen tasks is especially important. Since workflows are learned from previous successful traces and evaluated on new tasks, the results indicate that Trace2Workflow can preserve reusable task structures while adapting to different objects and environments. The executor does not need to rediscover the same procedure from scratch, but only needs to verify workflow applicability and execution effects.

Results on AppWorld. Trace2Workflow also performs well on AppWorld. It achieves 83.33% success on test-normal and 70.00% on test-challenge. Compared with CodeAct, the strongest baseline in success rate, Trace2Workflow improves success by 5.00 and 8.33 percentage points, while reducing model turns by 46.3% and 43.6%, respectively. Compared with ALFWorld, the reduction in Zero-LM tasks is smaller on AppWorld. This difference reflects the characteristics of the two environments. ALFWorld contains repeated interaction patterns with stable object manipulation procedures, which are easier to compile into reusable workflows. AppWorld involves more diverse API interactions and application states, where workflow applicability is harder to guarantee. Even in this setting, Trace2Workflow still reduces LLM inference and improves task success, showing that executable workflow reuse remains effective beyond a single embodied environment. Trace2Workflow completes 96 and 99 ALFWorld tasks without LLM calls. On AppWorld, it completes 4 and 2 tasks without LLM calls. These show that the learned workflows can directly replace a substantial amount of repeated LLM reasoning while preserving robustness when reuse is not applicable.

4.3 Ablation Study

We study the contribution of each component in Trace2Workflow. The ablation results show that each component contributes to effective workflow reuse. Local Programs Only achieves 96.27% SR on ALFWorld Unseen, but it requires 2.806 model turns, much higher than 0.858 in Trace2Workflow. This shows that local programs can reuse low-level operations, while workflow reuse further removes repeated task-level composition. Direct Workflow Compilation reduces model turns to 1.214 and enables 84 Zero-LM tasks on ALFWorld Unseen. However, it still performs worse than the full method. This demonstrates the value of first learning reliable local programs before constructing complete workflows. Removing Local Verification reduces SR from 98.51% to 94.78% and Zero-LM tasks from 99 to 68. Removing Workflow Validation also decreases task success, such as from 70.00% to 65.00% on AppWorld test-challenge. These results show that reliable local programs and execution validation are both important for stable workflow reuse. Overall, Trace2Workflow benefits from combining local program learning with workflow-level reuse, which allows agents to reuse task-level decisions and reduce repeated model reasoning.

4.4 Reuse Granularity Analysis

We study how reuse granularity affects model inference. We compare four settings on ALFWorld valid_unseen: G0 uses no reusable component, G1 reuses local programs, G2 reuses partial workflows, and G3 reuses full workflows. All settings use the same backbone, training traces, and evaluation tasks. Figure 3(a) shows a clear decrease in model turns as reuse becomes more complete. The average model turns drop from 3.246 in G0 to 2.806 in G1, 1.731 in G2, and 0.858 in G3. Full workflows therefore remove most repeated task-level decisions. Figure 3(b) shows a clear trade-off

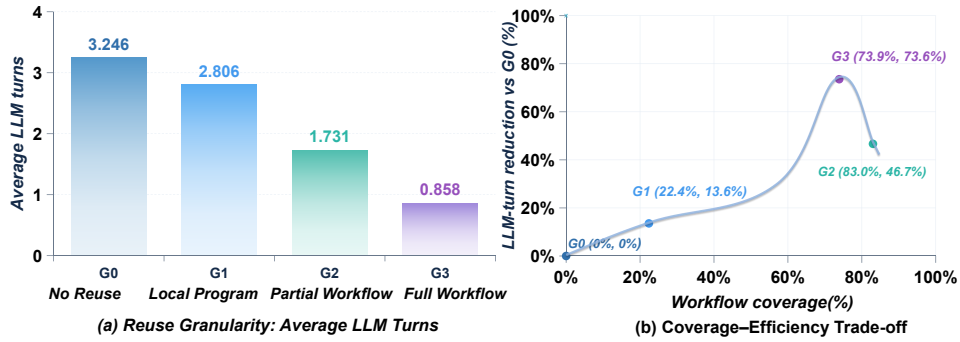

Figure 3: Reuse granularity analysis. (a) Larger reusable units reduce average model turns. (b) Reuse granularity shows a trade-off between coverage and inference reduction.

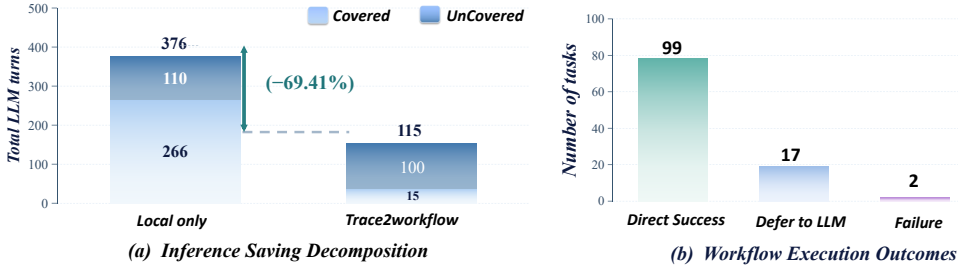

Figure 4: Inference saving analysis. (a) Most LLM-turn reduction comes from workflow-covered tasks. (b) Most matched workflows complete the task directly, while some defer to LLMs or fail.

between reuse coverage and inference reduction. G2 achieves the highest coverage at 83.0% and reduces model turns by 46.7%. G3 has a slightly lower coverage of 73.9%, but it achieves a larger reduction of 73.6%. This shows that increasing reuse coverage does not always lead to the largest inference saving. Partial workflows can apply to more tasks, while full workflows can replace more task-level decisions once they are matched. Trace2Workflow provides a good balance between two goals. It keeps high workflow coverage and also achieves a large reduction in LLM inference.

4.5 Analysis of Inference Savings

We analyze where the inference savings of Trace2Workflow come from. We partition the ALFWorld tasks into two groups. Workflow-covered tasks have a matched workflow. Uncovered tasks have no matched workflow. We use the same task partition for Local Only and Trace2Workflow. We then compare their total model turns. Figure 4(a) shows that Trace2Workflow reduces the total model turns from 376 to 110, a reduction of 70.74%. On workflow-covered tasks, the model turns decrease from 266 to 15. On uncovered tasks, they decrease from 110 to 95. Most of the savings therefore come from workflow reuse. Figure 4(b) shows the execution outcomes of the matched workflows. Among them, 99 complete the task directly, 17 defer to the LLM, and 2 fail. These results show that Trace2Workflow reduces repeated task-level decisions by executing learned workflows directly.

4.6 Workflow Transfer Across Models

We study whether the same workflow library can work with different executor models. We apply the learned workflows to four models without additional training. Figure 5(a) shows that the same workflow library improves the success rate across all executor models. It also enables many tasks to finish without model calls. Figure 5(b) shows a clear reduction in average model turns for every model. The reduction also holds for smaller models and DeepSeek-V3.2. These show that the learned workflows can transfer across different executor models and reduce repeated LLM inference.

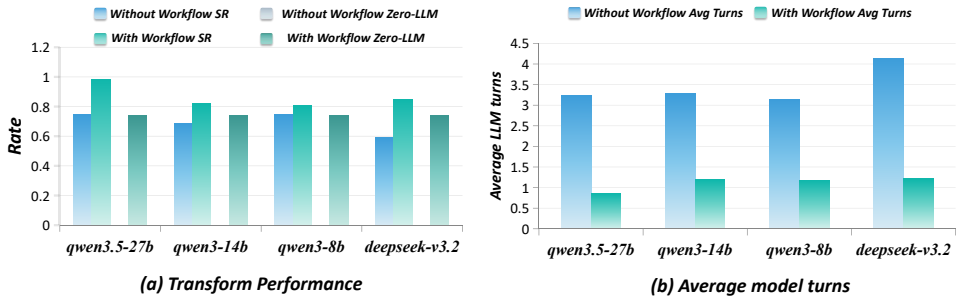

Figure 5: Workflow transfer across LLMs. (a) Workflow reuse improves task success and enables Zero-LLM execution across different LLMs. (b) Workflow reuse reduces average LLM turns.

5 Related Work

Reasoning and executable agents. Tool-augmented agents such as Toolformer (Schick et al., 2023) study how language models can learn to use external tools. ReAct (Yao et al., 2023) combines reasoning with tool actions, while CodeAct (Wang et al., 2024b) represents actions as executable code for flexible tool interaction. Reflexion (Shinn et al., 2023) further improves agent performance by leveraging feedback from previous attempts. AgentOccam (Yang et al., 2025) studies how action and observation design affects agent performance. These methods improve interactive problem solving, but LLMs still need to construct task-level decisions for each new request. Trace2Workflow instead reuses successful task-level workflows and directly executes them when possible.

Experience reuse and executable skills. Many methods reuse past agent experience through memory and skill learning. Agent Workflow Memory (Wang et al., 2025b) stores workflows for later tasks, and Agent S (Agashe et al., 2025) keeps both task-level and subtask-level experience. Voyager (Wang et al., 2024a), ASI (Wang et al., 2025a), and SkillWeaver (Zheng et al., 2025) study reusable executable skills and skill improvement from experience. Recent works also explore more adaptive skill organization and repeat-task consistency through approaches such as TROVE (Wang et al., 2026) and Habits (Weber & Taneja, 2026). Skill-DisCo (Guo et al., 2026) extracts and verifies procedural skills from traces, while TraceCompiler (Yadouni & Li, 2026) compiles traces into mostly deterministic workflows. They reduce repeated work, but the model may still need to select, combine, or adapt stored knowledge. Our method further reuses the task-level composition itself.

Workflow and agent architecture search. Recent work also studies automatic workflow construction. ADAS (Hu et al., 2025) searches over agent designs using evaluation feedback. AFlow (Zhang et al., 2025b) represents workflows in code and improves them through execution feedback. Agentic Compilation (Chundru, 2026) also explores compiling agent behaviors to reduce inference cost. AgentSquare (Shang et al., 2025) searches over planning, reasoning, tool-use, and memory modules. MAAS (Zhang et al., 2025a) searches multi-agent structures for different tasks. These methods mainly search for better agent or workflow structures. Trace2Workflow focuses on a different problem. Given a useful composition observed in past traces, we compile it into a parameterized executable workflow. The workflow can then be reused without internal model calls.

Relation to our work. Trace2Workflow is most related to executable skill and trace-compilation methods. The key difference is the reuse level. Local skills reuse part of a task, while our workflows also reuse the composition between these parts. We further separate local program verification, workflow construction, and later workflow validation.

6 Conclusion

We present Trace2Workflow, a framework for converting agent traces into reusable executable workflows. It first learns and verifies local programs from successful and failed traces. It then compiles successful task compositions into parameterized workflows. Experiments on ALFWorld and AppWorld show that Trace2Workflow improves task success while reducing model calls. Future work can explore more adaptive workflow acquisition and reuse strategies.

AI USE STATEMENT

We used AI tools to polish the language and improve the clarity and readability of this paper. The authors take full responsibility for the final content.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. Agent s: An open agentic framework that uses computers like a human. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 22924–22946, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/394c7c30ea87b5c3521b4d9e9d419071-Paper-Conference.pdf.

Jagadeesh Chundru. Agentic compilation: Mitigating the LLM rerun crisis for minimized-inference-cost web automation, 2026. URL https://arxiv.org/abs/2604.09718.

Zhongxin Guo, Danrui Qi, Hanwen Gu, Peng Cheng, and Yongqiang Xiong. Skill-disco: Distilling and compiling agent traces into reusable procedural skills, 2026. URL https://arxiv.org/abs/2606.26669.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long 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. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM Agent Search in Modular Design Space. In *International Conference on Learning Representations*, 2025. URL https://mlanthology.org/iclr/2025/shang2025iclr-agentsquare/.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning, 2021. URL https://arxiv.org/abs/2010.03768.

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, 2024. URL https://arxiv.org/abs/2407.18901.

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.

Tianxing Wang, Mingming Zhao, Shuai Huang, Huiyang Xu, Chaoyue Niu, Shengzhong Liu, and Fan Wu. TROVE: Adaptive agent skill orchestration via trace-grounded route validation and editing, 2026. URL https://arxiv.org/abs/2609.05019.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/wang24h.html.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks, 2025a. URL https://arxiv.org/abs/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.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 13–19 Jul 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

Travis Weber and Rohit Taneja. Making agents more consistent: Skills should form habits for repeat tasks, 2026. URL https://arxiv.org/abs/2609.25299.

Salma El Yadouni and Guanyi Li. Tracecompiler: Skill-guided mining and compilation of llm agent traces into mostly deterministic workflows, 2026. URL https://arxiv.org/abs/2608.02680.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik A Chaudhari, George Karypis, and Huzefa Rangwala. Agentoccam: A simple yet strong baseline for llm-based web agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 97533–97565, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f2c6e459b95694a24ac69c469a4ee746-Paper-Conference.pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

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. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10008343.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

A Additional Experimental Results

We report detailed results for reuse granularity, inference savings, and workflow transfer on ALFWorld `valid_unseen`. Table 4 provides the values used in Figure 3. G0–G3 correspond to no reuse, local programs, partial workflows, and full workflows, respectively. Coverage is the fraction of tasks with a matching reusable component, and turn reduction is measured against G0. Partial workflows achieve the highest coverage (83.0%), while full workflows achieve a larger turn reduction (73.6%) at a lower coverage (73.9%), indicating that full workflows replace more model decisions when applicable.

Table 4: Reuse granularity on ALFWorld `valid_unseen`. Turn reduction is relative to G0.

Setting	Coverage (%)	Avg. Turns ↓	Turn Reduction (%) ↑
G0: No Reuse	0.0	3.246	0.0
G1: Local Program	22.4	2.806	13.6
G2: Partial Workflow	83.0	1.731	46.7
G3: Full Workflow	73.9	0.858	73.6

Table 5 combines the model-turn decomposition and workflow execution outcomes shown in Figure 4. Using the same task partition for both methods, Trace2Workflow reduces total model turns from 376 to 110, with most savings coming from workflow-covered tasks (266 to 15 turns). Among matched tasks, 99 succeed directly, 17 defer to the model, and 2 fail. When a workflow returns DEFER, the model continues from the current environment state.

Table 5: Inference savings and workflow execution outcomes on ALFWorld `valid_unseen`. Panel (a) reports total model turns; panel (b) reports task counts.

(a) Model-turn decomposition			
Method	Covered	Uncovered	Total
Local Programs Only	266	110	376
Trace2Workflow	15	95	110
(b) Workflow execution outcomes			
Outcome	Direct Success	DEFER to LLM	Failure
Number of tasks	99	17	2

Table 6 provides the values used in Figure 5. Applying the same frozen workflow library improves success rates and reduces average model turns for all four executor models. The same 99 tasks succeed without model calls across all models, showing that these workflows can execute independently of the executor model.

Table 6: Workflow transfer across executors on ALFWorld `valid_unseen`. WF denotes workflow reuse, SR denotes success rate, and Zero-LM counts successful tasks completed without LLM calls.

	SR (%) ↑		Avg. Turns ↓		Zero-LM ↑	
Model	w/o WF	w/ WF	w/o WF	w/ WF	w/o WF	w/ WF
Qwen3.5-27B	74.63	98.51	3.246	0.858	0	99
Qwen3-14B	68.66	82.09	3.276	1.194	0	99
Qwen3-8B	74.63	80.60	3.142	1.179	0	99
DeepSeek-V3.2	58.96	85.07	4.134	1.209	0	99

B Prompt Templates

We report the prompts used in the model-assisted parts of Trace2Workflow. Workflow matching, input binding, execution checks, and workflow validation do not use model calls.

B.1 AGENT AND CODEACT PROMPT

Agent and CodeAct Prompt

```
You are a helpful assistant assigned with the task of
problem-solving. To achieve this, you will be using an
interactive coding environment equipped with a variety of
tool functions to assist you throughout the process.

At each turn, you should first provide your step-by-step
thinking for solving the task. Your thought process should
be enclosed using "<thought>" tag.

After that, you have two options:

1) Interact with a Python programming environment.
Use:
<execute>
...
</execute>

2) Directly provide a solution.
Use:
<solution>
...
</solution>

You have {max_total_steps} chances to interact with the
environment or propose a solution. You can only propose a
solution {max_propose_solution} times.

{tool_desc}

---

{in_context_example}

---

{task_prompt}
```

B.2 LOCAL PROGRAM EXTRACTION

Local Program Extraction

```
You are given an agent interaction trace.

Extract reusable local programs from the trace.
Each program should complete one clear local goal.

For each candidate:
1. Identify the local goal.
2. Keep relevant tool calls and code.
3. Preserve execution order and data dependencies.
4. Replace task-specific values with input parameters.
5. Define explicit outputs.
6. Preserve useful fragments from successful and failed episodes.

Do not use the final task outcome as the local-program label.
The candidate will be verified separately.

Task:
{task}
```

```
Trace:
{trace}

Tool specifications:
{tool_desc}

Return:
Local goal:
Function signature:
Inputs:
Outputs:
Program:
Dependencies:
```

B.3 LOCAL COMPLETION CHECK

Local Completion Check

```
You are given a local goal and tool specifications.

Construct:

1. Precondition P(x,h):
   Check whether the program can start.

2. Completion check Q(x,h,z,h'):
   Check whether the local goal is completed.

Use only observable information from inputs, history,
returned values, tool outputs, and environment effects.

Q returns:
1       verified completion
0       verified violation
UNKNOWN insufficient evidence

Use the same checks for all candidates of the same local goal.
Do not use the final task outcome as the local verdict.

Local goal:
{local_goal}

Task instruction:
{task_instruction}

Tool specifications:
{tool_desc}

Return:
Precondition:
Completion check:
Required evidence:
Executable pseudocode:
```

B.4 WORKFLOW COMPILATION

Workflow Compilation

```
You are given a successful agent trace and a library of
verified local functions.

Compile the task procedure into a reusable workflow.
```

```
Rules:
1. Use one node for each tool or local-function call.
2. Keep the source execution order.
3. Keep state-changing calls.
4. Convert task-specific values into workflow inputs.
5. Preserve dependencies between node outputs and later inputs.
6. Keep fixed procedure values as constants.
7. Preserve branch conditions.
8. Build an applicability guard.
9. Attach observable completion checks.
10. Use results from the current execution.
11. No workflow node calls the language model.

Task:
{task}

Successful trace:
{trace}

Generated code:
{code}

Call log:
{call_log}

Verified local functions:
{local_function_library}

Return:
Workflow inputs:
Nodes:
Argument bindings:
Branch conditions:
Applicability guard:
Completion checks:
```

B.5 Model Continuation after Workflow Defer

Model Continuation after Workflow Defer

```
Continue solving the current task from the current state.

Use the interaction history below as the state already reached
by previous execution.
Keep completed actions and observed results.
Use the remaining action budget.

Task:
{task_prompt}

Current interaction history:
{current_history}

Remaining action budget:
{remaining_budget}

Available tools:
{tool_desc}
```