

SKILL-DISCO: Distilling and Compiling Agent Traces into Reusable Procedural Skills

Zhongxin Guo¹ Danrui Qi¹ Hanwen Gu² Peng Cheng¹ Yongqiang Xiong¹

¹Microsoft Research Asia ²Beijing Foreign Studies University
{zhongxin.guo, danruiqi}@microsoft.com

Abstract

Agents often repeatedly solve similar task instances from scratch, leading to unnecessary reasoning cost and long execution traces. Prior work has explored workflow reuse and executable skill induction, but it remains unclear which task scenarios admit procedural skills and how the shared procedural structure should be represented across successful traces. We study this problem in FSM-defined scenarios, where successful traces can be viewed as paths through partially observed instance-level transition graphs, and formulate procedural skills as reusable parameterized control-flow subgraphs. Based on this view, we introduce SKILL-DISCO, a distillation-and-compilation framework that distills reusable PFSM subgraphs from successful traces and compiles them into callable, executable, and verifiable procedural skills. Experiments on ALFWorld and WebArena show that SKILL-DISCO improves success rates and reduces agent turns across benchmarks and model scales, demonstrating the benefits of representing shared experience as reusable execution structures.

1 Introduction

LLM agents are increasingly used for interactive tasks that require many reasoning and acting steps (Yao et al., 2023; Shinn et al., 2023; Wang et al., 2024a; Yang et al., 2024). Yet even with stronger action representations such as Code-Act (Wang et al., 2024a), agents often solve each task independently, repeatedly rediscovering low-level action patterns shared across related tasks (Figure 1). This increases reasoning cost, execution length, and generalization brittleness (Shinn et al., 2023; Wang et al., 2024b, 2025b; Ni et al., 2026). Recent work addresses this issue by reusing self-reflections, workflows, or structured skill documents as context (Shinn et al., 2023; Wang et al., 2024b; Ni et al., 2026), and by inducing executable skills (Wang et al., 2023, 2025b; Zheng et al., 2025).

These results suggest that agents should accumulate reusable procedures rather than solve each task from scratch.

However, the foundations of procedural skill discovery remain under-specified. Reusable skills are meaningful when tasks share stable execution patterns, but become ill-defined for open-ended, instance-specific generation. Existing methods often synthesize skills directly from successful traces using LLMs. Without an explicit notion of shared structure, the resulting libraries can become fragmented and biased toward trace-specific surface patterns rather than reusable procedural logic.

We make the scope of procedural skill discovery explicit by focusing on *FSM-defined scenarios*, where execution dynamics are described by finite states, admissible actions, and deterministic transitions. In such scenarios, successful traces are paths in an unknown transition graph, and procedural skills correspond to repeated transition structures that help reach goal states. If this graph were known, task completion would reduce to graph search. In realistic agent settings, however, the graph is unavailable, and the agent only observes successful traces.

To capture shared structure across traces, we formalize procedural skills through a *Parameterized Finite-State Machine* (PFSM) view. A PFSM abstracts concrete states and actions into parameterized states and operators, so traces with different objects, states, or lengths can instantiate the same execution pattern. Under this view, each successful trace can be lifted into a parameterized trace graph, and a procedural skill is a reusable PFSM subgraph matched across traces under parameter binding. Thus, a skill is not merely a textual routine or an LLM-generated script, but a structurally grounded abstraction of shared execution logic.

This formulation does not assume that the latent PFSM is directly observable. In realistic agent environments, neither the full transition graph nor the

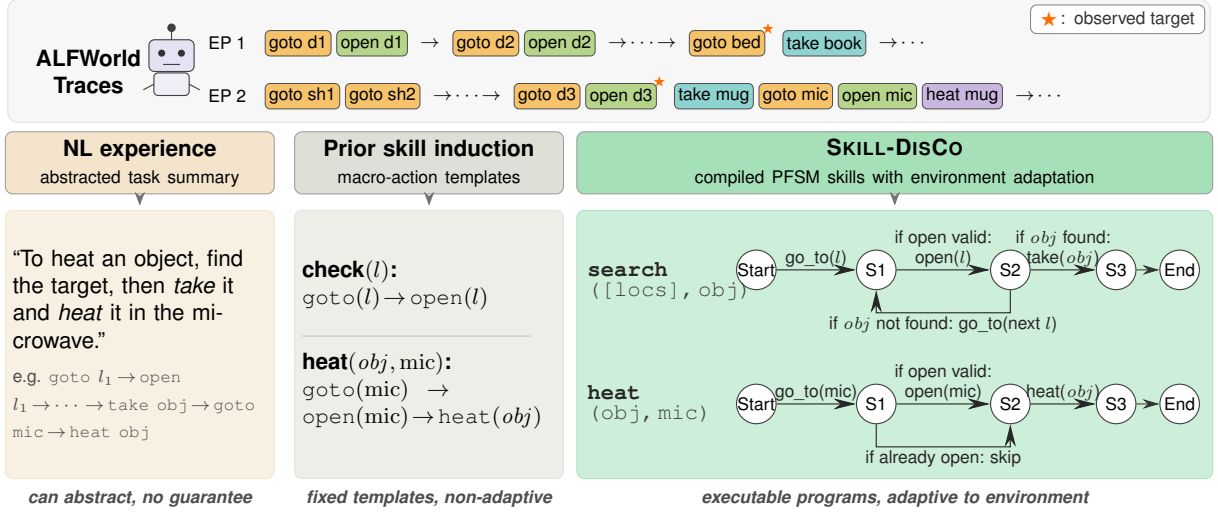


Figure 1: SKILL-DISCO distills environment-adaptive PFSM skills that branch on observations, transfer across episodes and model scales without re-induction.

lifting function from raw traces to PFSM subgraphs is given. SKILL-DISCO therefore approximates PFSM-based skill discovery by recovering reusable parameterized control-flow patterns from successful traces and validating them through skill compilation. This yields a distillation-and-compilation framework. Distillation discovers compact, high-coverage PFSM subgraphs that favor reusable structures over trace-specific routines. Compilation turns each discovered structure into a callable, executable, and verifiable skill through explicit specification and execution-grounded verification.

Our contributions are as follows:

(C1) We make explicit the scope of procedural skill discovery by focusing on FSM-defined scenarios, where successful traces are paths in an unknown transition graph and reusable skills have well-defined transition semantics.

(C2) We formulate procedural skills as reusable parameterized control-flow subgraphs that can match multiple successful traces under parameter binding. This provides structural targets for skill discovery, rather than treating each successful trace as an independent routine.

(C3) We introduce SKILL-DISCO, a distillation-and-compilation framework that approximates these structural targets and compiles them into verified and executable procedural skills.

(C4) Extensive experiments show that SKILL-DISCO improves both success rate and efficiency across different benchmarks and model scales. Further analysis shows that compact compiled skills reduce library redundancy, improve execution reli-

ability, and transfer procedural knowledge from stronger induction models to smaller execution models.

2 Preliminaries

2.1 FSM-Defined Scenarios

In this paper, we focus on scenarios whose execution dynamics can be formulated as finite-state machines. Such scenarios have a finite state space, a finite action space, and deterministic transitions: given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the next state is uniquely determined by $\delta(s, a)$. We refer to such scenarios as *FSM-defined scenarios*.

Definition 1 (FSM-Defined Scenario) A scenario is called an *FSM-defined scenario* if its execution dynamics can be represented as a finite-state machine

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \delta, \mathcal{S}_0, \mathcal{S}_{\text{goal}}),$$

where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\delta : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a deterministic transition function, $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of possible initial states, and $\mathcal{S}_{\text{goal}} \subseteq \mathcal{S}$ is the set of goal states. The corresponding transition graph is defined as $G^* = (V^*, E^*)$, where $V^* = \mathcal{S}$ and $E^* = \{(s, a, s') \mid s, s' \in \mathcal{S}, a \in \mathcal{A}, \delta(s, a) = s'\}$.

The deterministic transition function applies within each initialized instance; different instances may instantiate different FSMs rather than share one global graph. For example, ALFWorld instances vary in room layouts, object placements, and container states, while WebArena instances

vary in page and backend states. Within an initialized instance, however, the current state and action determine the next state.

For an FSM-defined scenario, if the complete transition graph G^* and the goal states are known, task completion can be reduced to finding a path from an initial state to a goal state on G^* . However, the complete transition graph is unavailable in many agent settings. Thus, we aim to infer reusable transition structures from past successful traces.

Definition 2 (Primitive Operator) A primitive operator is a tuple

$$op = (\mathcal{X}, \mathcal{Y}, \text{Pre}, \text{Post}),$$

where $\mathcal{X}$ is the input space, $\mathcal{Y}$ is the output space, $\text{Pre} : \mathcal{X} \rightarrow \{0, 1\}$ is a precondition predicate, and $\text{Post} : \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\}$ is a postcondition predicate. For any input $x \in \mathcal{X}$ satisfying $\text{Pre}(x) = 1$, executing op produces a unique output $y = op(x) \in \mathcal{Y}$ such that $\text{Post}(x, y) = 1$.

Definition 3 (Successful Trace) Given an FSM-defined scenario $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \delta, \mathcal{S}_0, \mathcal{S}_{\text{goal}})$, an agent trace is a finite execution record $\tau = (o_0, a_0, o_1, a_1, \dots, a_{T-1}, o_T)$, where o_t denotes the observation received by the agent at step t and $a_t \in \mathcal{A}$ is the action executed by the agent.

The trace is induced by an underlying state trajectory $(s_0, a_0, s_1, a_1, \dots, a_{T-1}, s_T)$, such that $s_{t+1} = \delta(s_t, a_t)$, and each observation o_t is generated from the underlying state s_t . We call τ a successful agent trace if the underlying execution starts from a valid initial state and reaches a goal state, i.e. $s_0 \in \mathcal{S}_0$ and $s_T \in \mathcal{S}_{\text{goal}}$.

Note that every action $a \in \mathcal{A}$ in an FSM-defined scenario is treated as a *Primitive Operator*, i.e. actions in a *Successful Trace* are all primitive operators. For clarity, we will simply refer to such primitive operators as actions throughout the paper.

2.2 PFSM-Based Procedural Skill Discovery

In a standard FSM, task completion corresponds to a concrete path from an initial state to a goal state. Such a path precisely records one execution, but is often too instance-specific because each transition is grounded in concrete states and actions. As a result, executions with the same control logic may appear as different paths when they involve different objects, locations, or intermediate states.

To capture reusable execution patterns, we introduce the notion of a *Parameterized Finite State Machine* (PFSM). A PFSM abstracts concrete states

and actions with parameters, allowing multiple concrete FSM paths to be represented by the same parameterized control-flow structure.

Definition 4 (Parameterized FSM (PFSM)) A *Parameterized Finite State Machine* is defined as

$$\tilde{\mathcal{M}} = (\tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \Theta, \tilde{\delta}, \tilde{\mathcal{S}}_0, \tilde{\mathcal{S}}_{\text{goal}}),$$

where $\tilde{\mathcal{S}}$ is a finite set of parameterized states, $\tilde{\mathcal{A}}$ is a finite set of parameterized actions, Θ is the parameter space, and $\tilde{\delta} : \tilde{\mathcal{S}} \times \tilde{\mathcal{A}} \times \Theta \rightarrow \tilde{\mathcal{S}}$ is a deterministic parameterized transition function. Each parameter assignment $\theta \in \Theta$ instantiates a parameterized transition into a concrete FSM transition. Here, a parameterized state $\tilde{s} \in \tilde{\mathcal{S}}$ represents an abstract execution state rather than a fully grounded environment state. A parameterized action $\tilde{a} \in \tilde{\mathcal{A}}$ represents an action schema whose arguments are instantiated by parameters.

Running example (Figure 1). Consider two successful traces from different tasks, both of which contain a find-and-take subprocedure:

```

 $\tau_1$ : go_to(drawer1)  $\rightarrow$  open(drawer1)  $\rightarrow$ 
      go_to(drawer2)  $\rightarrow$  open(drawer2)  $\rightarrow$ 
      go_to(desk1)  $\rightarrow$  go_to(bed)  $\rightarrow$ 
      take(book)  $\rightarrow \dots$ 
 $\tau_2$ : go_to(shelf1)  $\rightarrow$  go_to(shelf2)  $\rightarrow$ 
      go_to(drawer1)  $\rightarrow$  open(drawer1)  $\rightarrow$ 
      go_to(drawer2)  $\rightarrow$  open(drawer2)  $\rightarrow$ 
      take(mug)  $\rightarrow$  go_to(mic)  $\rightarrow$  open(mic)  $\rightarrow$ 
      heat(mug)  $\rightarrow \dots$ 

```

The complete traces serve different task goals, but both contain a find-and-take segment. These segments differ in length and concrete entities, which can be absorbed by parameter bindings: θ_1 assigns $\text{obj}=\text{book}$ and $[\text{locs}]=[\text{drawer1}, \text{drawer2}, \text{desk1}, \text{bed}]$, whereas θ_2 assigns $\text{obj}=\text{mug}$ and $[\text{locs}]=[\text{shelf1}, \text{shelf2}, \text{drawer1}, \text{drawer2}]$. Under these bindings, both segments instantiate the same parameterized control-flow graph. Here, $[\text{locs}]$ is the ordered list of candidate locations, and l denotes the current location selected from that list in each iteration:

```

repeat go_to( $l$ )
  if can_open( $l$ )
    then open( $l$ )
    check_target( $l, \text{obj}$ )
until found( $\text{obj}$ )
then take( $\text{obj}$ ).

```

This procedure corresponds to the $\text{search}([\text{locs}], \text{obj})$ PFSM in Figure 1, where $[\text{locs}]$ supplies the candidate locations,

obj the target, and l the current location. State S_1 represents arrival at l ; S_2 represents observing its contents after opening it when valid. If obj is absent, $S_2 \rightarrow S_1$ advances to the next l ; if visible, $S_2 \rightarrow S_3$ executes $\text{take}(\text{obj})$ and terminates. Thus, segments with different objects and search lengths are bound traversals of the same PFSM. After the shared segment, EP 2 continues with the separate $\text{heat}(\text{obj}, \text{mic})$ PFSM. All PFSM states denote equivalence classes of observable execution contexts rather than complete simulator states.

Given a set of successful traces, each trace can be lifted from a concrete FSM path to a subgraph of the PFSM transition graph. Such a subgraph captures the parameterized execution structure of a successful trace, including its abstract states, parameterized actions, and control-flow relations. We refer to this lifted subgraph as a *parameterized trace graph*. Since the underlying PFSM is unavailable, the lifting function $\phi : \tau_i \mapsto \tilde{G}_i$ does not admit a closed-form definition. We approximate it via a multi-stage pipeline described in Section 3.

Our goal is to discover procedural skills from successful traces. Intuitively, a procedural skill corresponds to a PFSM subgraph that is shared by many successful traces. Such a subgraph captures a recurring control-flow structure that helps the agent reach a goal across different concrete instantiations.

Problem 1 (Procedural Skill Discovery) *Given a set of successful traces $\mathcal{T}^+ = \{\tau_1, \tau_2, \dots, \tau_N\}$ and a lifting function $\phi : \tau_i \mapsto \tilde{G}_i$ that maps each trace to a parameterized trace graph, the goal is to discover a set of procedural skills $\mathcal{K} = \{K_1, K_2, \dots, K_m\}$, where each skill K_j is a parameterized control-flow subgraph that matches a subset of $\{\tilde{G}_i\}_{i=1}^N$ under parameter binding, i.e. $K_j \preceq \tilde{G}_i$ for i in some index set. Specifically, $K_j \preceq \tilde{G}_i$ means that, for some binding θ and branch choices, an executable traversal of K_j is isomorphic to a subgraph of $\tilde{G}_i$; optional branches of K_j need not all occur in one trace.*

Intuitively, a desirable skill set $\mathcal{K}$ should satisfy three properties: (i) **Coverage**, where each skill is supported by multiple successful traces rather than a single execution, (ii) **Utility**, where skills capture control-flow structures that help reach goal states, and (iii) **Compactness**, where skills are neither overly specific nor trivially generic. In realistic agent settings, neither the full PFSM $\tilde{\mathcal{M}}$ nor the lifting function ϕ is analytically available. SKILL-DISCO approximates both from successful traces

through the pipeline in Section 3.

3 The SKILL-DISCO Framework

Section 2 formulates procedural skill discovery as identifying reusable PFSM subgraphs from successful agent traces. Solving this problem directly is challenging because the complete PFSM $\tilde{\mathcal{M}}$ is unavailable, the lifting function ϕ from raw traces to parameterized trace graphs must be inferred, and exact subgraph matching under parameter binding is costly and noise-sensitive. To address these challenges, SKILL-DISCO discovers procedural skills through distillation and compilation.

3.1 Framework Overview

Figure 2 shows an overview of the SKILL-DISCO framework. The input to SKILL-DISCO is a set of successful traces $\mathcal{T}^+ = \{\tau_1, \tau_2, \dots, \tau_N\}$, where N denotes the number of successful traces. Distillation yields a set of consolidated PFSM subgraphs $\mathcal{K} = \{K_1, K_2, \dots, K_m\}$; compilation then turns these subgraphs into executable candidates and retains the verified subset $\mathcal{F} = \{f_1, f_2, \dots, f_\ell\}$, where $\ell \leq m$.

3.2 The Distillation Phase

The distillation phase transforms successful traces into reusable PFSM subgraphs by normalizing each trace, extracting subgoal-level operations, and consolidating recurring operations across traces.

Stage 1: Trace Normalization. Raw agent traces interleave reasoning text, actions, and observations, leaving their state-transition structure implicit. Stage 1 begins the pipeline’s approximation of $\phi : \tau_i \mapsto \tilde{G}_i$ by converting each successful trace τ_i into a structured, executable program p_i . The program preserves every executed action and its resulting observation, keeping the representation grounded in the observed trajectory without requiring recovery of the complete environment FSM.

We use Python-like control flow as a textual presentation of the concrete trace graph. Action-observation pairs encode visited transitions, sequential statements preserve their order, and loops and conditionals represent repeated or observation-conditioned paths. Compared with an explicit graph serialization, this programmatic form is easier for an LLM to generate and interpret while retaining the graph structure needed by later stages. Thus, p_i is an intermediate program representation

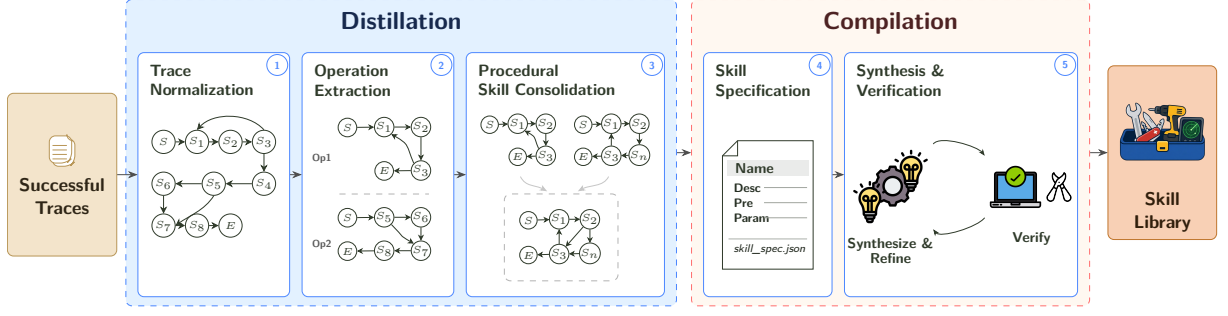


Figure 2: Overview of SKILL-DISCO. The *distillation* phase turns successful traces into reusable PFSM subgraphs; the *compilation* phase converts them into executable and verified skills.

used to approximate $\tilde{G}_i$. The Stage 1 panel of Figure 2 visualizes the state-transition graph encoded by such a normalized program.

Stage 2: Subgoal-level Operation Extraction.

The normalized graph $\tilde{G}_i$, represented by program p_i , may contain several reusable procedures. Treating the entire trace as one skill would bind these procedures to a task-specific execution, whereas treating every primitive transition as a skill would provide little procedural abstraction. Stage 2 therefore decomposes the normalized program p_i into intermediate-granularity operation subgraphs, each organized around a coherent subgoal.

This decomposition is performed by an LLM. Given the normalized program and its action-observation log, the LLM groups consecutive steps according to their semantic goal and control-flow relationship. For example, a household trace may contain one group for locating and retrieving an object and another for applying an appliance. For each resulting operation, the extractor returns $o = (\nu, \sigma, \mathbf{u}, c)$, where ν is an action-oriented identifier, σ summarizes the subgoal, $\mathbf{u} = (u_1, \dots, u_{|o|})$ records the primitive-action sequence, and c is the corresponding code fragment from p_i . Thus, the operation retains both its semantic intent and the transition structure required for later comparison across traces.

For program p_i , the extracted operations form $\mathcal{O}_i = \{o_1^{(i)}, \dots, o_{m_i}^{(i)}\}$. Unit-length fragments are discarded because they coincide with primitive transitions and provide no multi-step abstraction; the remaining candidates are $\mathcal{O}_{\text{multi}} = \bigcup_i \{o \in \mathcal{O}_i : |o| \geq 2\}$. The Stage 2 panel in Figure 2 illustrates the resulting segmentation with Op1 and Op2. For instance, EP 2 in Figure 1 can be separated into a find-and-take operation and a subsequent heating operation, showing how one trace yields multiple coherent candidates for the cross-trace consolida-

tion in Stage 3.

Stage 3: Procedural Skill Consolidation. Because the latent PFSM and its parameter bindings are unavailable, Stage 3 approximates exact subgraph matching by consolidating operations that express the same semantic goal and underlying procedure. To scale beyond a single LLM context while supporting online incremental consolidation as new traces arrive, it uses two passes:

(1) *Batch grouping.* The LLM partitions operations within each batch into non-overlapping groups based on their action-observation patterns, code structure, and subgoal semantics. Each batch thereby yields local candidate groups for reusable behavior.

(2) *Cross-batch consolidation.* A second LLM pass merges semantically equivalent local groups into a compact, non-redundant cluster set. It can run globally over all batch-level groups or incrementally by combining each new batch’s groups with the existing cluster set.

Each final cluster c_k approximates a reusable PFSM subgraph K_k . Formally, $K_k \preceq \tilde{G}_i$ means that an executable traversal of K_k , under some binding θ , matches a subgraph of $\tilde{G}_i$.

The Stage 3 panel in Figure 2 schematically illustrates the overall consolidation process: operations with related semantics but different concrete control-flow structures are consolidated into a reusable cluster that retains their compatible paths.

3.3 The Compilation Phase

The compilation phase turns each distilled PFSM subgraph into a callable skill by defining its interface and behavioral requirements, synthesizing an implementation, and verifying it through environment interaction.

Stage 4: Skill Specification. A subgraph K_k captures reusable behavior but does not specify

how an agent should invoke it. Stage 4 therefore converts it into a skill definition with (i) a typed signature and structured return type, (ii) an LLM-facing description, (iii) preconditions, postconditions, and side effects, and (iv) metadata such as abstraction level, expected primitive-action savings, and confidence. Together, these fields expose the parameter bindings and observable conditions needed for synthesis and verification.

In Figure 2, this stage turns the consolidated graph K into the specification `search([locs], obj)`, whose parameters identify the candidate locations and target object and whose postcondition requires the agent to hold the target.

Stage 5: Synthesis and Verification. Stage 5 synthesizes a Python implementation f_k that realizes the specification using environment primitives and returns a structured execution result. The implementation is then exercised on induction-split tasks, where verification records its invocation count, unhandled runtime exceptions, and self-reported completion rate. Evaluation tasks are never used in this process.

A skill is flagged for refinement if it raises an unhandled exception or, after at least three invocations, reports a success rate below the heuristic threshold of 30%. The resulting execution feedback is returned to the LLM to fix and refine the implementation, which is then re-verified. This iterative synthesis–refinement–verification loop, illustrated in Figure 2, continues until the skill satisfies the acceptance criteria or exhausts a configurable refinement budget (up to three refinement rounds in our experiments, excluding the initial synthesis). After refinement, skills that still raise unhandled exceptions or fall below the applicable success-rate threshold are discarded. The output is therefore a verification-filtered executable library; “verified” refers to these acceptance criteria rather than correctness on every possible invocation.

4 Experiments

We evaluate whether SKILL-DISCO can distill reusable procedural skills from successful traces and deploy them as reliable skills for interactive agents. Our experiments are organized around four questions: **RQ1.** Does SKILL-DISCO improve end-to-end task success rate and turn efficiency compared with base agents and prior skill-induction methods? (§4.2) **RQ2.** Do the procedural skills transfer across model families and scales? (§4.3)

RQ3. How are skills induced by the SKILL-DISCO pipeline, and how do they perform at inference time? (§4.4) **RQ4.** Do all components of SKILL-DISCO contribute to the observed gains? (§4.5)

4.1 Experimental Setup

Datasets. We evaluate on two interactive agent datasets with different execution structures: (1) *ALFWorld* (Shridhar et al., 2021) consists of text-based household tasks that require navigation, search, and object manipulation. (2) *WebArena* (Zhou et al., 2024) consists of realistic web-navigation tasks over self-hosted websites.

Splits. For each dataset, we use a strict *induction/evaluation* split: skills are derived only from the induction split and evaluated only on held-out tasks. For *ALFWorld*, the induction split consists of 200 tasks sampled from the official train set, and the evaluation split is the official unseen split with 134 tasks. For *WebArena*, which has no canonical induction split, we split the 812 tasks in half: the first 406 tasks for induction and the remaining 406 for evaluation. All baselines use the same splits as SKILL-DISCO.

Baselines. We compare against two categories of baselines. The first category contains base agents without skill augmentation: ReAct (Yao et al., 2023) using the AgentBench implementation (Liu et al., 2024), and CodeAct (Wang et al., 2024a). The second group consists of offline workflow- and skill-induction methods: AWM_{offline} (Wang et al., 2024b) and the offline variant of ASI, ASI_{offline} (Wang et al., 2025b). AWM and ASI were not evaluated on *ALFWorld* in their original papers. For *ALFWorld*, we adapt their original pipelines and reuse their prompts wherever possible, making only the environment-specific changes needed for evaluation. For all skill-augmented variants, each skill’s signature and description are appended in agents’ prompts to instruct LLMs to decide whether and how to invoke a skill or emit a primitive action.

Metrics. We report *success rate* (SR) and *average agent turns* (Avg. Turns) per episode. Per-episode token usage and inference cost for methods are reported in Appendix A.3.

4.2 End-to-End Performance

We first evaluate whether SKILL-DISCO improves complete task execution when added to existing interactive agents. Table 1 compares SKILL-DISCO with base agents and prior offline skill-induction

Table 1: End-to-end results on ALFWorld and WebArena. Parentheses show relative changes over the matching base agent. **Bold** marks the best result and underlining the second-best.

Method	SR (%) $\uparrow$	Avg. Turns $\downarrow$
<i>(a) ALFWorld</i>		
ReAct	82.0	19.3
CodeAct	<u>96.3</u>	<u>3.6</u>
AWM _{offline}	80.1	15.6
ASI _{offline}	70.1	19.0
SKILL-DISCO + ReAct	92.4 (+12.7%)	8.8 (−54.5%)
SKILL-DISCO + CodeAct	99.1 (+2.9%)	3.2 (−11.3%)
<i>(b) WebArena</i>		
ReAct	23.9	5.9
CodeAct	20.0	10.9
AWM _{offline}	21.2	5.9
ASI _{offline}	<u>24.6</u>	<u>5.7</u>
SKILL-DISCO + ReAct	29.1 (+21.6%)	5.1 (−13.1%)
SKILL-DISCO + CodeAct	22.9 (+14.8%)	8.5 (−22.0%)

methods using GPT-4o. We also use GPT-4o as the model for the SKILL-DISCO pipeline, where it performs both distillation and compilation to produce the skill library.

Higher task success rate. SKILL-DISCO improves SR over the matching base agent in all four settings. On *ALFWorld*, it raises ReAct from 82.0% to 92.4% and the already strong CodeAct from 96.3% to 99.1%; Appendix A.2 reports the five-run stability of this modest, near-saturated gain. On *WebArena*, it raises ReAct from 23.9% to 29.1% and CodeAct from 20.0% to 22.9%. The strongest SKILL-DISCO configuration achieves the highest SR on each benchmark; on *WebArena*, its 29.1% exceeds ASI_{offline} by 4.5 absolute points. All offline skill-induction methods use the same induction and evaluation splits.

Consistent reduction in agent turns. SKILL-DISCO reduces average turns in all four settings: 54.5%/11.3% (ReAct/CodeAct) on *ALFWorld* and 13.1%/22.0% on *WebArena*. A skill invocation transfers a multi-step subtask to an executable procedure, which can ground and execute several observation-conditioned primitive actions before returning control to the agent. Thus, the agent need not reason about and issue each primitive action separately. Table 3 shows that each call executes 5.6 primitive steps on average in *ALFWorld* and 2.8 in *WebArena*, with maxima of 33 and 36. Execution verification and runtime grounding further limit failed calls and the additional turns needed for recovery.

Table 2: Skill transfer across target models on ALFWorld and WebArena. Δ denotes relative change over the vanilla setting; **bold** marks the best result.

Model	SR (%) $\uparrow$		Avg. Turns $\downarrow$	
	Van.	+Sk. (Δ)	Van.	+Sk. (Δ)
<i>(a) ALFWorld</i>				
Qwen3.5-4B	61.2	91.0 (+48.8%)	9.0	5.0 (−44.4%)
Qwen3.5-9B	54.5	98.5 (+80.8%)	10.8	3.4 (−68.2%)
GPT-4o	96.3	99.1 (+2.9%)	3.6	3.2 (−11.3%)
GPT-4o-mini	71.6	94.8 (+32.3%)	7.1	4.4 (−38.0%)
GPT-5-chat	91.0	98.5 (+8.2%)	4.7	3.0 (−34.6%)
GPT-5-mini	91.8	98.5 (+7.3%)	4.9	3.3 (−33.3%)
<i>(b) WebArena</i>				
Qwen3.5-4B	10.1	18.7 (+85.3%)	8.6	6.8 (−20.4%)
Qwen3.5-9B	21.2	23.9 (+12.8%)	6.8	6.0 (−12.0%)
GPT-4o	23.9	29.1 (+21.6%)	5.9	5.1 (−13.1%)
GPT-4o-mini	18.0	18.0 (+0.0%)	7.7	7.1 (−7.7%)
GPT-5-chat	32.5	37.0 (+13.7%)	5.3	4.8 (−8.6%)
GPT-5-mini	32.5	33.7 (+3.8%)	5.4	4.8 (−10.1%)

Complementary to different agents. Within each benchmark, the same fixed library improves both ReAct and CodeAct, showing that SKILL-DISCO complements different agent architectures. Although SKILL-DISCO +CodeAct (22.9%) remains below base ReAct (23.9%) on *WebArena*, this is a cross-backbone comparison: relative to its matching CodeAct base, SKILL-DISCO improves SR by 14.8% and reduces turns by 22.0%. CodeAct is less effective on *WebArena* because its code interactions repeatedly incur large accessibility-tree observations.

4.3 Cross-Model Skill Transferability

We next test whether the skill libraries produced by SKILL-DISCO using GPT-4o can transfer to other models. For each benchmark, we deploy its library *without modification* across multiple target models: Qwen3.5-4B, Qwen3.5-9B, GPT-4o, GPT-4o-mini, GPT-5-chat, and GPT-5-mini. Based on Table 1, we use the skill library induced from the stronger SKILL-DISCO configuration on each benchmark: CodeAct for *ALFWorld* and ReAct for *WebArena*.

Skills transfer across target models. With the same skill library, every target model benefits: average turns drop in every cell (7.7% to 68.2%), while SR improves by 2.9% to 80.8% on *ALFWorld* and improves or remains unchanged on *WebArena*, with gains of up to 85.3%. These results show that the induced skills are robust across model families and scales.

Weaker models can benefit substantially from skill transfer. Although gains are not

Table 3: Skill-usage statistics. #Sk is the number of induced skills; Turns and Sk. turns are usage-log averages of total and skill-invoking turns; Avg./Max prim. steps are primitive steps collapsed per skill call; Err. is execution-error rate.

Bench	Method	#Sk	Turns ↓	Sk. turns	Avg. prim. steps	Max prim. steps	Err. (%) ↓
ALFWorld	ASI _{offline}	47	19.0	16.2	4.3	9	55.4
	SKILL-DISCO	5	3.2	2.6	5.6	33	0.0
WebArena	ASI _{offline}	146	5.7	1.3	2.5	8	33.9
	SKILL-DISCO	20	5.1	1.0	2.8	36	21.5

strictly monotone in model capacity, the largest jumps occur on the smaller open-source backbones: Qwen3.5-9B gains 80.8% on *ALFWorld* and Qwen3.5-4B gains 85.3% on *WebArena*, versus 2.9%–7.3% for GPT-4o and GPT-5-mini on *ALFWorld*. Strikingly, Qwen3.5-9B with the library produced using GPT-4o reaches 98.5% on *ALFWorld*, surpassing the unaugmented GPT-4o target agent at 96.3% while using a smaller model. This shows that SKILL-DISCO enables *procedural-knowledge distillation and transfer*: smaller models can benefit from skills produced by SKILL-DISCO using a stronger model.

4.4 Skill Induction and Inference-Time Performance

To answer RQ3, we examine both skill induction and deployment. The stage-wise profile in Appendix B.1 shows how distillation and compilation turn trace-specific operations into a compact, execution-filtered library, and reports their yields and refinement effort. Table 3 then compares the resulting libraries with ASI_{offline} at inference time, using CodeAct on *ALFWorld* and ReAct on *WebArena* as in Table 1.

SKILL-DISCO learns compact, high-coverage skills. SKILL-DISCO uses far fewer skills than ASI (5 vs. 47 on *ALFWorld*; 20 vs. 146 on *WebArena*). Nevertheless, its libraries cover 74.7%/73.8% of induction traces and contain an applicable skill for 98.5%/60.8% of held-out tasks. Each call also executes more primitive steps than ASI on average (5.6 vs. 4.3; 2.8 vs. 2.5) and achieves greater maximum step compression (33 vs. 9; 36 vs. 8). Thus, SKILL-DISCO captures reusable procedures rather than producing a large collection of narrow fragments.

Verified skills execute more reliably. At inference time, SKILL-DISCO reduces execution errors relative to ASI from 55.4% to 0.0% on *ALFWorld* and from 33.9% to 21.5% on *WebArena*. Its gains therefore come not only from invoking skills, but from invoking skills that execute reliably. Together

Table 4: Pipeline ablations. #Sk = skill library size; Relative SR Δ vs. the full pipeline in parentheses.

Configuration	#Sk	SR (%)	Turns
A1: no distill	43	53.0 (−46.5%)	11.5
A2: no compile	NA	97.0 (−2.1%)	3.6
Full SKILL-DISCO	5	99.1	3.2

with higher step compression, this reliability helps explain the SR and turn improvements in Table 1.

Taken together, stage-wise retention and trace/task coverage characterize *coverage*; library size and consolidation characterize *compactness*; and step compression and execution reliability characterize *utility*. These results connect the complete induction-to-deployment process to the three desirable properties introduced in Section 2.

4.5 Ablation Study

We evaluate the contribution of the two main phases of SKILL-DISCO, distillation and compilation, through ablations on *ALFWorld* using CodeAct with GPT-4o (Table 4).

Distillation is the main driver of success. Removing distillation and inducing skills per successful trace yields a larger 43-skill library but reduces SR from 99.1% to 53.0% and increases turns from 3.2 to 11.5. Without cross-trace consolidation the library is dominated by overlapping, trace-specific variants, making skill selection harder and causing wrong-variant invocations or fallback to long open-loop executions.

Compilation mainly improves execution compactness. Removing compilation and shipping the output of Stage 3 as natural-language procedure descriptions only modestly reduces SR (99.1 $\rightarrow$ 97.0%), but each episode must keep the full procedure text in context, inflating tokens (Appendix A.3). Compiling skills into callable code externalizes procedural execution from the prompt, yielding both efficiency and a small additional reli-

ability gain.

Appendix B.2 further evaluates the robustness of SKILL-DISCO across different models used by its pipeline.

5 Related Work

LLM-Based Agents. ReAct (Yao et al., 2023) interleaves reasoning with actions; CodeAct (Wang et al., 2024a), PAL (Gao et al., 2023), and Code as Policies (Liang et al., 2023) show executable code improves compositional reasoning (Wei et al., 2022). Yet procedural knowledge in successful traces remains ephemeral and rarely reused.

Reusing Agent Experience as Behavioral Imitation. A common line keeps the model fixed and reuses experience as *token-level context*: Reflexion (Shinn et al., 2023) prompts self-reflection; AWM (Wang et al., 2024b) induces textual workflows; Trace2Skill (Ni et al., 2026) writes structured skill documents. All store knowledge in *natural language*, which is harder to verify through execution before deployment and must be re-interpreted at inference. SKILL-DISCO instead compiles procedural knowledge into external executable code, verified and reusable across models.

Synthesizing Executable Tools, Workflows, and Skills. (1) *Tool and workflow generation.* LATM (Cai et al., 2023), CREATOR (Qian et al., 2023), and ToolGen (Wang et al., 2025a) synthesize or unify tools per task; AgentDistill (Qiu et al., 2025) produces MCP modules for training-free transfer. At the system level, ADAS (Hu et al., 2024), AgentSquare (Shang et al., 2024), AFlow (Zhang et al., 2024), and MaAS (Zhang et al., 2025) automate agentic-system or workflow design. All start from *explicit task specifications*; SKILL-DISCO mines a *corpus of execution traces*, grounding skills in recurring behavior. (2) *Skill induction from agent experience.* Voyager (Wang et al., 2023) commits one JavaScript function per successful task, leaving near-duplicate variants unconsolidated. SkillWeaver (Zheng et al., 2025) derives APIs from website affordances, not workload traces. ASI (Wang et al., 2025b) shows programs beat free-form text as skill representations but still operates per-trajectory. Concurrent NSI (Shao et al., 2026) induces logic-grounded programmatic skills with online honing. SKILL-DISCO instead uses PFSMs to represent procedural structure and aggregates semantically equivalent procedural fragments across the entire trace corpus, distilling them

into reusable skills before execution-grounded compilation and verification.

6 Conclusion

This paper studies procedural skill discovery by making shared execution structure across successful traces explicit. We formulate skills as reusable PFSM subgraphs under parameter binding, and introduce SKILL-DISCO, which distills compact, high-coverage structures from traces and compiles them into executable and verifiable skills. Experiments on *ALFWorld* and *WebArena* show that these structurally grounded skills improve success rate and turn efficiency under deterministic instance-level dynamics and a stable interaction interface, while transferring across target models.

Limitations

Environment scope. The current formulation assumes deterministic transitions conditioned on the full latent state within each initialized instance, together with a stable primitive action and observation interface. It accommodates partial observability and instance-specific objects, layouts, and element identifiers by grounding execution in current observations, but does not address arbitrary interface changes, unmodeled stochasticity, or noisy action outcomes. Extending the PFSM to POMDP settings would require observation- or belief-conditioned probabilistic guards and policy-level, repeated-trial verification; we leave this to future work.

Procedural tasks only. SKILL-DISCO compiles procedural skills as programs from interaction traces and is effective for tasks with reusable procedural structure (navigation, web automation, tool use). It is not designed for pure NLP tasks such as text generation or reading comprehension, where success depends primarily on linguistic understanding rather than executable procedures.

Pipeline quality depends on model capability and trace availability. SKILL-DISCO relies on the reasoning and code-generation ability of the model used for distillation and compilation, as well as a sufficient corpus of successful traces. Insufficient model capability can yield incorrect or overly specific skills, while domains in which even frontier models rarely succeed may provide too little signal for reliable corpus-level skill extraction. Execution-grounded verification mitigates but does not eliminate these dependencies.

References

- Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. [Large language models as tool makers](#). *arXiv preprint arXiv:2305.17126*.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. [PAL: Program-aided language models](#). In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.
- Shengran Hu, Cong Lu, and Jeff Clune. 2024. [Automated design of agentic systems](#). *arXiv preprint arXiv:2408.08435*.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. [Code as policies: Language model programs for embodied control](#). In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, et al. 2024. [AgentBench: Evaluating LLMs as agents](#). In *International Conference on Learning Representations*.
- Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjuan Jiang. 2026. [Trace2Skill: Distill trajectory-local lessons into transferable agent skills](#). *arXiv preprint arXiv:2603.25158*.
- Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. [CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models](#). In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939. Association for Computational Linguistics.
- Jiahao Qiu, Xinzhe Juan, Yimin Wang, Ling Yang, Xuan Qi, Tongcheng Zhang, Jiacheng Guo, Yifu Lu, Zixin Yao, Hongru Wang, Shilong Liu, Xun Jiang, Leqi Liu, and Mengdi Wang. 2025. [AgentDistill: Training-free agent distillation with generalizable MCP boxes](#). *arXiv preprint arXiv:2506.14728*.
- Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2024. [AgentSquare: Automatic LLM agent search in modular design space](#). *arXiv preprint arXiv:2410.06153*.
- Jie-Jing Shao, Haiyan Yin, Yueming Lyu, Xingrui Yu, Lan-Zhe Guo, Ivor Tsang, James Kwok, and Yu-Feng Li. 2026. [Lifting traces to logic: Programmatic skill induction with neuro-symbolic learning for long-horizon agentic tasks](#). In *International Conference on Machine Learning*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. [Reflexion: Language agents with verbal reinforcement learning](#). *Advances in Neural Information Processing Systems*, 36.
- Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. [ALFWorld: Aligning text and embodied environments for interactive learning](#). In *International Conference on Learning Representations*.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. [Voyager: An open-ended embodied agent with large language models](#). *arXiv preprint arXiv:2305.16291*.
- Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. 2025a. [ToolGen: Unified tool retrieval and calling via generation](#). In *International Conference on Learning Representations*.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. [Executable code actions elicit better LLM agents](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232. PMLR.
- Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. 2025b. [Inducing programmatic skills for agentic tasks](#). *arXiv preprint arXiv:2504.06821*.
- Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024b. [Agent workflow memory](#). *arXiv preprint arXiv:2409.07429*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). *Advances in Neural Information Processing Systems*, 35.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. [SWE-agent: Agent-computer interfaces enable automated software engineering](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. [ReAct: Synergizing reasoning and acting in language models](#). In *International Conference on Learning Representations*.
- Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025. [Multi-agent architecture search via agentic supernet](#). *arXiv preprint arXiv:2502.04180*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, et al. 2024. *AFLOW: Automating agentic workflow generation*. *arXiv preprint arXiv:2410.10762*.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, et al. 2025. *SkillWeaver: Web agents can self-improve by discovering and honing skills*. *arXiv preprint arXiv:2504.07079*.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. *WebArena: A realistic web environment for building autonomous agents*. In *International Conference on Learning Representations*.

A Full Results: Token Usage and Inference Cost

A.1 API Pricing Assumptions

All per-episode inference costs reported in the paper are computed from provider-published token prices, retrieved from Artificial Analysis¹ on 23 May 2026 (on-demand rates). Cached input tokens (returned inside a context-caching session) are billed at 1/10 of the regular input rate; output tokens are billed at the full output rate. No batch-API discounts are applied.

A.2 Full Baseline Comparison with Token and Cost Statistics

Five-run stability. On *ALFWorld*, five independent GPT-4o + CodeAct runs yield $99.1 \pm 0.6\%$ SR with SKILL-DISCO, compared with $96.3 \pm 1.9\%$ for the base agent. Because the base agent is already near saturation, we regard this as a modest, stable gain rather than the main evidence for SKILL-DISCO. The broader evidence is the larger gains on non-saturated target models in Table 2 and the consistent reduction in agent turns across Tables 1 and 2.

A.3 Per-Episode Token and Cost Breakdown

Table 6 reports the full per-model token and cost breakdown for the *ALFWorld* settings in Table 2.

A.4 Full Ablation Breakdown

Table 7 expands the two-row ablation summary in Table 4 (§4.5) with the no-skill baseline and per-episode token and cost statistics.

¹<https://artificialanalysis.ai/models#pricing>

B Additional Evaluation and Analysis

We complement the end-to-end results with pipeline diagnostics, a pipeline-model ablation, and an analysis of runtime grounding and failure modes in WebArena.

B.1 Pipeline and Library Diagnostics

Table 8 maps each pipeline stage to its PFSM role and reports its yield. These are execution-grounded diagnostics of the latent lifting approximation: errors can be filtered by the next stage and ultimately by Stage 5 execution, rather than requiring access to a gold PFSM.

The final libraries retain common behaviors while dropping idiosyncratic trace fragments. Utility and realized compactness are measured at deployment in Table 3: calls replace multiple primitive steps, and their execution-error rates are 0.0% on *ALFWorld* and 21.5% on *WebArena*. The lower *WebArena* task coverage and higher error rate reflect its more heterogeneous tasks and more fragile accessibility-tree grounding. The final deployment gate is execution with the rule-based criteria in Stage 5.

B.2 Model Ablation for the SKILL-DISCO Pipeline

We vary only the model used by the SKILL-DISCO pipeline for distillation and compilation, then evaluate every resulting library with the same GPT-4o target agent on the *ALFWorld* unseen split. This separates pipeline-model capability from target-agent capability.

Every pipeline model yields a library that improves the non-saturated ReAct agent. CodeAct exposes a quality gradient because its 96.3% baseline leaves little headroom: frontier models improve it, GPT-4o-mini matches it, Qwen yields a small gain, and Llama-3.1-8B yields a 0.8-point decrease. The synthesized-to-final ratios also show that Stage 5 generally filters a larger share of candidates produced by weaker pipeline models.

B.3 Runtime Grounding and Failure Modes in WebArena

WebArena exposes an accessibility tree whose backend element IDs vary across episodes. Listing 8 therefore locates each control from its current accessible name (e.g., Submit, Title, or the forum name), recovers the current ID with `find_bid`, and only then emits the primitive action. This suc-

Table 5: Comparison with a GPT-4o backbone, including per-episode token usage and inference cost. “In-Tok. (Cached)” = avg. input tokens per episode with the prompt-cached portion in parentheses (cached tokens are billed at $1/10$ of the regular input rate). The same token convention is used in all subsequent tables.

Method	SR (%) $\uparrow$	Avg. Turns $\downarrow$	In-Tok. (Cached) $\downarrow$	Out-Tok. $\downarrow$	Cost (\$) $\downarrow$
<i>(a) ALFWorld</i>					
ReAct	82.0	19.3	35,600 (22,200)	626	0.0453
CodeAct	<u>96.3</u>	<u>3.6</u>	7,721 (6,232)	562	<u>0.0109</u>
AWM _{offline}	80.1	15.6	43,524 (31,534)	379	0.0416
ASI _{offline}	70.1	19.0	64,607 (57,054)	475	0.0379
SKILL-DISCO + ReAct	92.4 (+12.7%)	8.8 (−54.5%)	22,100 (11,300) (−37.9%)	607 (−3.0%)	0.0359 (−20.8%)
SKILL-DISCO + CodeAct	99.1 (+2.9%)	3.2 (−11.3%)	8,468 (6,303) (+9.7%)	368 (−34.5%)	0.0107 (−2.1%)
<i>(b) WebArena</i>					
ReAct	23.9	5.9	50,914 (8,605)	728	0.1152
CodeAct	20.0	10.9	75,962 (24,202)	806	0.1435
AWM _{offline}	21.2	5.9	49,025 (11,748)	564	0.1018
ASI _{offline}	<u>24.6</u>	<u>5.7</u>	56,662 (9,247)	603	0.1269
SKILL-DISCO + ReAct	29.1 (+21.6%)	5.1 (−13.1%)	51,068 (8,582) (+0.3%)	561 (−22.9%)	<u>0.1140</u> (−1.1%)
SKILL-DISCO + CodeAct	22.9 (+14.8%)	8.5 (−22.0%)	85,649 (27,078) (+12.8%)	736 (−8.7%)	<u>0.1606</u> (+11.9%)

Parenthesized values give relative Δ vs. the matching base agent (ReAct or CodeAct). **Best** and second-best values are highlighted for SR, Avg. Turns, and Cost within each benchmark.

Table 6: Per-episode token usage and inference cost on ALFWorld (CodeAct backbone, 134 unseen tasks). “ Δ ” is the relative change vs. the same model’s Vanilla setting (applied to In-Tok.).

Model	In-Tok. (Cached)			Out-Tok.			Cost (\$)		
	Vanilla	+Skill	Δ	Vanilla	+Skill	Δ	Vanilla	+Skill	Δ
Qwen3.5-4B	28,010 (24,566)	19,608 (14,505)	−30.0%	2,095	1,391	−33.6%	0.00049	0.00041	−16.3%
Qwen3.5-9B	27,754 (24,939)	10,152 (7,292)	−63.4%	1,382	610	−55.9%	0.00061	0.00036	−41.0%
GPT-4o	7,721 (6,232)	8,468 (6,303)	+9.7%	562	368	−34.5%	0.01090	0.01067	−2.1%
GPT-4o-mini	17,620 (16,020)	13,686 (11,271)	−22.3%	891	485	−45.6%	0.00102	0.00082	−19.6%
GPT-5-chat	10,606 (8,517)	7,728 (5,496)	−27.1%	677	391	−42.2%	0.01040	0.00739	−28.9%
GPT-5-mini	11,046 (6,979)	8,561 (6,189)	−22.5%	871	492	−43.5%	0.00293	0.00173	−41.0%

ceeds despite ID-level drift. If the same skill is selected from the wrong page, `find_bid("Submit", observation)` returns no match and the skill exits with Submit button not found instead of acting on a stale ID. The residual 21.5% error rate reflects element-not-found, wrong-page-state, precondition-mismatch, and skill-selection failures that name grounding alone cannot resolve.

C Pipeline Stage Prompts

We summarize the LLM prompts that drive each pipeline stage (`latest_pipeline/skill_mining/`). For brevity each listing keeps the system message’s role, key directives, and the JSON output schema; long motivating examples, repeated rationale, and formatting boilerplate from the source files are elided with “...”. Placeholders in {braces} are Python f-string slots filled at runtime. A few slots in Stages 4 and 5 are specialized per benchmark

(`benchmark_prompts.py`) – we leave them as placeholders here and omit the benchmark-specific fills for brevity.

C.1 Stage 1: Trace-to-Program Transpilation

Listing 1: Stage 1 prompt (trace normalization, from `prompt/trace_converter.md`).

```

1 [System]
2 You are a Senior AI Systems Architect
  . Convert a raw ReAct trace
3 (alternating THOUGHT / ACTION
  messages and environment
  observations)
4 into a structured Python program that
  preserves the reasoning while
5 making control flow explicit and
  logically stable.
6
7 Each reasoning block must follow the
  Unified Reasoning Template:
8 # A. Deterministic (logic-
  controlled)
9 if "<condition>" in observation:
10     action = "<derived action>"

```

Table 7: Full ablation breakdown on ALFWorld (CodeAct + GPT-4o).

Configuration	#Sk	SR (%)	Turns	In-Tok. (Cached)	Out-Tok.	Cost (\$)
Vanilla (no skills)	0	96.3	3.6	7,721 (6,232)	562	0.01090
A1: no distill	43	53.0	11.5	107,528 (97,901)	994	0.05849
A2: no compile	NA	97.0	3.6	19,083 (17,393)	559	0.01417
Full SKILL-DISCo	5	99.1	3.2	8,468 (6,303)	368	0.01067

Table 8: Stage-wise pipeline and library diagnostics. Source-fragment steps are measured at synthesis; deployed primitive steps per call appear in Table 3.

Phase	Metric	PFSM / library role	ALFWorld	WebArena
Stage 1	Successful traces used	Source traces τ_i	150/200	109/406
Stage 1	Valid normalized programs	Approx. trace graphs $\tilde{G}_i$	150/150	103/109
Stage 2	Multi-step operations	Candidate parameterized subgraphs	287	132
Stage 3	Consolidated clusters	Shared subgraphs K	7	25
Stage 3	Induction traces covered by a final skill	Coverage	112/150 (74.7%)	76/103 (73.8%)
Stage 3	Source-fragment steps, mean/max	Compactness proxy $ K $	12.25/40	4.32/18
Stage 4	Skill specifications	$(K, \theta, \text{pre/post})$ contracts	7	25
Stage 5	Verification pass rate	Execution-grounded filter	5/7 (71.4%)	20/25 (80.0%)
Stage 5	Mean refinement rounds per skill	Refinement effort (up to 3 rounds; initial synthesis excluded)	0.8	1.2
Stage 5	Final skills	Deployed library size	5	20
Evaluation	Tasks with an applicable skill	Held-out task coverage	132/134 (98.5%)	247/406 (60.8%)
Evaluation	Self-reported successful skill calls	Call reliability	248/348 (71.3%)	155/247 (62.8%)

```

11     context.append((observation,
12                     None, action))
13     observation, available_actions
14     = env.step(action)
15     # B. Cognitive (LLM-driven, for
16     adaptive / uncertain steps)
17     thinking, action = llm(task,
18                             context, observation,
19                             available_actions)
20     context.append((observation,
21                     thinking, action))
22     observation, available_actions =
23     env.step(action)
24
25 Rules: (i) use deterministic control
26 flow (if/for/while) for
27 repetitive
28 or stabilized reasoning; (ii) reserve
29 llm() for adaptive decisions;
30 (iii) update `context` and call `env.
31 step()` after every action;
32 (iv) preserve every ACTION from the
33 trace -- never drop mistakes or
34 recovery steps; (v) copy real
35 observation/thinking/action
36 strings as
37 inline `# ...` comments.
38
39 [User]
40 Please convert the following agent
41 trace history to Python code:
42 {raw_history}

```

C.2 Stage 2: Semantic Operation Extraction

Listing 2: Stage 2 prompt.

```

1 [System]
2 You are an expert at analyzing agent
3 traces represented as Python code
4 .
5 Extract semantic operations: groups
6 of consecutive steps that
7 together
8 accomplish one coherent sub-goal and
9 whose internal relationship can
10 be expressed as control flow (if/for/
11 while or fixed sequence).
12 A good operation has clear boundaries
13 , locality, and reusability.
14
15 For each operation emit:
16 name : snake_case verb_noun
17 description : the sub-goal
18 env_steps : every (action,
19 observation) pair -- prefer the
20 authoritative env-
21 step log over
22 the code, which
23 may
24 have dropped actions
25 during
26 conversion
27 code_snippet : the complete code
28 block (no truncation)
29 succeeded : true iff the final
30 observation confirms the sub-
31 goal
32
33 (e.g., "Nothing
34 happens." =>
35 false)
36
37 Output JSON: {"task_summary": "...",
38 "operations": [ ... ]}

```

Table 9: Model ablation for the SKILL-DISCO pipeline on ALFWorld. “Synth.” and “final” are the numbers of synthesized and Stage 5-verified skills.

Pipeline model	Synth.	Final	ReAct SR	ReAct turns	CodeAct SR	CodeAct turns
No skills	0	0	82.0	19.3	96.3	3.6
GPT-4o	7	5	92.4	8.8	99.1	3.2
GPT-5-chat	10	9	98.5	7.4	98.5	3.3
GPT-4o-mini	6	4	93.3	11.4	96.3	4.1
Qwen3.5-9B	9	3	93.3	11.8	97.0	3.8
Llama-3.1-8B-Instruct	5	2	89.6	13.3	95.5	5.0

```

19 [User]
20 ## Code to Analyze
21 ```python
22 {trace.code}
23 ```
24 {env_log_section}
25 Please extract all semantic
26 operations. The env-step log is
27 authoritative -- group its entries by
28 the code's structure but
include ALL actions from the log.

```

C.3 Stage 3: Operation Clustering (Two Passes)

Stage 3 first runs a *grouping pass* over mini-batches that proposes candidate reusable patterns, then a *consolidation pass* that merges overlapping proposals into a minimal cluster set.

Listing 3: Stage 3a prompt – grouping.

```

1 [System]
2 You are identifying reusable
operation patterns. Partition the
input
3 operations into non-overlapping
groups whose members share the
largest
4 reusable action/observation pattern
and semantic goal -- transferring
5 across different objects, locations,
pages, or task instances.
6
7 Rules: use `env_steps` to decide
membership (not names); ignore
8 incidental differences; split only
when the action pattern or goal
9 differs; ignore operations that did
not change state; prefer fewer,
10 larger, uniform groups -- when in
doubt, MERGE.
11
12 Output JSON: {"groups": [{"group_name":
", "description",
13 "representative_operations": [{"
operation_id", "name"}]}]}
14
15 [User]
16 ## Operations to Analyze ({len(batch)}
operations)
17 ```json
18 {json.dumps(batch, indent=2)}
19 ```

```

Listing 4: Stage 3b prompt – consolidation.

```

1 [System]
2 Merge the candidate operation groups
into a minimal, non-redundant
3 skill set. Assign every input group `
index` to exactly one final
4 cluster (no overlap, no missing index
).
5
6 Merge groups whose reusable skill,
goal, or induced state change is
7 similar, even if the wording, objects
, locations, or appliances
8 differ. Proceed in three steps: (1)
draft candidate clusters,
9 (2) merge duplicates/overlaps, (3)
emit the final JSON as the LAST
10 fenced `json` block in your response.
11
12 Output JSON: {"clusters": [{"name", "
description",
13 "indices": [0,3,7]}, ...]}
14
15 [User]
16 ## Groups to Consolidate ({len(groups)}
groups)
17 ```json
18 {json.dumps(group_summaries, indent
=2)}
19 ```

```

C.4 Stage 4: Skill Contract Definition

Stage 4 turns each cluster into a typed skill contract. The base prompt is built by `build_stage4_prompt(bench_cfg)` and contains two benchmark slots, `{action_construction_note}` and `{action_seq_example}`, that are filled per benchmark.

Listing 5: Stage 4 prompt (benchmark slots left as placeholders).

```

1 [System]
2 You design reusable skill APIs for
LLM agents. Skills are Python
3 functions of the form
4 def skill_name(param1, param2,
...):

```

```

5 `env` is a global -- never a
   parameter -- and is used as
6   observation, available_actions =
     env.step(action: str)
7 After a skill returns, the caller LLM
   sees ONLY the returned Dict;
8 it must always include at least `
   success`, the latest `observation`
9 and `available_actions`, and a
   process trace of (action,
   observation).
10
11 Design principles: (1) generalize --
   parameterize object names,
12 location lists, and action templates;
   (2) single responsibility --
13 one goal per skill; (3) action
   SELECTION not construction --
14 {action_construction_note}
   (4) declare every state change in `
   side_effects` and reflect it in
15 `canonical_action_sequence`; (5)
   compose freely -- assume no
16 prerequisite from the caller.
17
18 Output JSON contract:
19 {
20   "skill_name": "description", "
     docstring",
21   "parameters": [{"name", "type", "
     description", "required", "
     default"}],
22   "return_type": "Dict",
23   "preconditions", "postconditions",
     "side_effects",
24   "canonical_action_sequence": [
25     "Ordered action templates with
     {param} placeholders.",
26     {action_seq_example}
27     "Derive ONLY from SUCCEEDED
     traces; prefer the
     shortest
28     successful sequence; copy the
     exact action syntax."
29   ],
30   "abstraction_level": "primitive|
     composite|workflow",
31   "estimated_actions_saved": <int>,
     "confidence_score": <float>}
32
33 [User]
34 ## Operation Group
35 - Name / Description / #operations /
   #traces / avg_actions:
36   {cluster fields}
37
38 ## Representative Operations (real
   env.step() patterns)
39 ```json
40 {json.dumps(examples, indent=2)}
41 ```
42 Define a general, reusable skill that
   abstracts this pattern.

```

C.5 Stage 5: Skill Synthesis

Stage 5 synthesizes a self-contained Python implementation from the Stage 4 contract. The base prompt is built by

build_stage5_prompt(bench_cfg) and contains two benchmark slots, {first_action_rule} (how to obtain the first vs. later actions) and {identifier_note} (an env-specific gotcha).

Listing 6: Stage 5 prompt (benchmark slots left as placeholders).

```

1 [System]
2 You synthesise the Python body of a
   skill defined by the Stage 4
3 contract. `env` is global; signature
   contains only domain parameters.
4
5 Critical rules:
6 1. Use ONLY `env.step()` and the
   Python stdlib -- no helper
7   functions, no simulation, no
   fabricated observations.
8 2. {first_action_rule}
9 3. Match the trace's action syntax
   exactly; when constructing,
10   reproduce the verb / preposition /
   argument pattern from the
11   reference traces.
12 4. Branch on the LATEST `observation`
   , not on a self-computed flag.
13 5. Honour every parameter; implement
   every step in
14   `canonical_action_sequence` and
   only those plus declared
15   side effects.
16 6. Append every (action, observation)
   pair to `process_trace`.
17 7. Return a Dict including at least `
   success`, latest `observation`,
18   latest `available_actions`, and `
   process_trace` (plus any extra
   keys the caller needs).
19
20 Environment note: {identifier_note}
21
22 Output JSON: {"implementation": "def
   skill_name(...): \n ...",
23   "example_usage": "
   result = skill_name
24   (...)"
25
26 [User]
27 ## Skill Definition
28 {skill.skill_name / description /
   parameters / pre & postconditions
   }
29
30 ## Representative Code Examples (
   ground truth env.step() patterns)
31 ```json
32 {json.dumps(examples, indent=2)}
33 ```
34 Synthesise the implementation. Select
   actions from
35 `available_actions`, follow the
   code_snippet patterns exactly,
   and
36 return the Dict so the caller can
   continue without blind spots.

```

D Skill Library Examples

We show one representative skill for each benchmark from the final compiled skill catalogs. Listing 7 is the highest-coverage skill in the ALFWorld library: it iterates over a list of candidate locations, opens any closed containers, and returns as soon as the target item is observed. Listing 8 is a representative WebArena skill that drives the multi-step submission workflow on a Reddit clone (open form → fill URL / title / body → select forum → submit). Unlike the generic tuple-return interface in the prompts above, the WebArena adapter used by this catalog returns the latest accessibility-tree observation directly from `env.step; find_bid(...)` grounds each action against that observation.

Listing 7: ALFWorld skill: `systematic_search_locations`.

```
1 def systematic_search_locations(
2     locations, target):
3     """Systematically search through
4     a list of locations to find a
5     specific target item. The skill
6     navigates to each location,
7     opens
8     it if necessary, and examines its
9     contents to locate the
10    target.
11
12    Args:
13        locations: A list of location
14        names to search through.
15        target: The name of the item
16        to search for.
17
18    """
19    process_trace = []
20    found_in = None
21    for location in locations:
22        # Step 1: Go to the location
23        action = f"go to {location}"
24        observation,
25        available_actions = env.
26        step(action)
27        process_trace.append((action,
28        observation))
29        # Step 2: Check if the
30        location is closed and
31        open it if necessary
32        if "closed" in observation:
33            action = next(
34                (a for a in
35                 available_actions
36                 if "open" in a.lower
37                 () and location
38                 in a.lower()),
39                f"open {location}",
40            )
41            observation,
42            available_actions =
43            env.step(action)
44            process_trace.append((
45                action, observation))
46        # Step 3: Inspect the
47        contents of the location
```

```
27         if target in observation:
28             found_in = location
29             return {
30                 "success": True,
31                 "found_in": found_in,
32                 "observation":
33                     observation,
34                 "available_actions":
35                     available_actions
36             },
37             "process_trace":
38                 process_trace,
39         }
40     # If the target is not found in
41     any location
42     return {
43         "success": False,
44         "found_in": None,
45         "observation": observation,
46         "available_actions":
47             available_actions,
48         "process_trace":
49             process_trace,
50     }
```

Listing 8: WebArena skill: `create_submission_workflow` (Reddit site).

```
1 def create_submission_workflow(
2     title: str,
3     body: str,
4     forum_name: str,
5     url: Optional[str] = None):
6     """Complete workflow for creating
7     a new submission/post.
8     Navigates
9     to the submission form, fills in
10    required fields (URL if
11    present,
12    title, body), selects the target
13    forum, and submits the form.
14    Flow: click Submit -> fill fields
15    -> select forum -> click
16    Create.
17
18    Args:
19        title: Title of the
20        submission to post.
21        body: Main content/body of
22        the submission.
23        forum_name: Target forum/
24        subreddit label to select
25        .
26        url: Optional URL to include
27        if the form has a URL
28        field.
29
30    """
31    process_trace = []
32    matched_elements = []
33    failure_reason = None
34    observation = env.step('noop(0)')
35    # Step 1: Click "Submit" to open
36    the submission form
37    submit_bid = find_bid('Submit',
38    observation)
39    if not submit_bid:
40        return {'success': False, '
41            observation': observation
42            ,
43            'process_trace':
44                process_trace,
```

```

25         'matched_elements':
26             matched_elements,
27         'failure_reason': "
28             Submit button not
29             found."}
30     observation = env.step(f"click('{
31         submit_bid}')")
32     process_trace.append((f"click('{
33         submit_bid}')", observation))
34     matched_elements.append('Submit')
35     # Step 2: Optionally fill URL
36     field
37     if url:
38         url_bid = find_bid('URL',
39                             observation)
40         if url_bid:
41             act = f"fill({url_bid!r},
42                         {str(url)!r})"
43             observation = env.step(
44                 act)
45             process_trace.append((act,
46                                   observation))
47             matched_elements.append('
48                 URL')
49     # Step 3: Fill Title field
50     title_bid = find_bid('Title',
51                           observation)
52     if not title_bid:
53         return {'success': False, '
54                 observation': observation
55                 ,
56                 'process_trace':
57                     process_trace,
58                 'matched_elements':
59                     matched_elements,
60                 'failure_reason': "
61                     Title field not
62                     found."}
63     act = f"fill({title_bid!r}, {str(
64         title)!r})"
65     observation = env.step(act)
66     process_trace.append((act,
67                           observation))
68     matched_elements.append('Title')
69     # Step 4: Open forum-selection
70     dropdown and pick the target
71     forum
72     combobox_bid = find_bid('Choose
73     one', observation) \
74         or find_bid('Forum
75         ', observation)
76     if not combobox_bid:
77         return {'success': False, '
78                 observation': observation
79                 ,
80                 'process_trace':
81                     process_trace,
82                 'matched_elements':
83                     matched_elements,
84                 'failure_reason': "
85                     Forum selection
86                     combobox not
87                     found."}
88     observation = env.step(f"click('{
89         combobox_bid}')")
90     process_trace.append((f"click('{
91         combobox_bid}')", observation
92         ))
93     forum_bid = find_bid(forum_name,

```

```

60     observation)
61     if not forum_bid:
62         return {'success': False, '
63                 observation': observation
64                 ,
65                 'process_trace':
66                     process_trace,
67                 'matched_elements':
68                     matched_elements,
69                 'failure_reason': f"
70                     Forum '{
71                     forum_name}' not
72                     found."}
73     observation = env.step(f"click('{
74         forum_bid}')")
75     process_trace.append((f"click('{
76         forum_bid}')", observation))
77     matched_elements.append(
78         forum_name)
79     # Step 5: Fill Body field
80     body_bid = find_bid('Body',
81                         observation)
82     if not body_bid:
83         return {'success': False, '
84                 observation': observation
85                 ,
86                 'process_trace':
87                     process_trace,
88                 'matched_elements':
89                     matched_elements,
90                 'failure_reason': "
91                     Body field not
92                     found."}
93     act = f"fill({body_bid!r}, {str(
94         body)!r})"
95     observation = env.step(act)
96     process_trace.append((act,
97                           observation))
98     matched_elements.append('Body')
99     # Step 6: Click "Create
100     submission" to finalize
101     create_bid = find_bid('Create
102     submission', observation) \
103         or find_bid('Submit'
104                     , observation)
105     if not create_bid:
106         return {'success': False, '
107                 observation': observation
108                 ,
109                 'process_trace':
110                     process_trace,
111                 'matched_elements':
112                     matched_elements,
113                 'failure_reason': "
114                     Create-submission
115                     button not found
116                     ."}
117     observation = env.step(f"click('{
118         create_bid}')")
119     process_trace.append((f"click('{
120         create_bid}')", observation))
121     matched_elements.append('Create
122     submission')
123     success = 'submitted' in
124         observation.lower() \
125         or 'created' in
126         observation.lower()
127     return {
128         'success': success,
129         'observation': observation,

```

```
95         'process_trace':  
96             process_trace,  
97         'matched_elements':  
98             matched_elements,  
99         'failure_reason': None  
100         if success  
            else 'No success  
                confirmation detected  
                .',  
    }
```