SKILLTRACER: STRUCTURAL FAILURE ATTRIBUTION AND REFINEMENT OF AGENTIC SKILLS IN LONG-HORIZON WEB TASKS

Yuyang Li[1,2*], **Yiran Dou**[1,2*], **Jie-Jing Shao**[1,3], **Yueming Lyu**[1], **Ivor Tsang**[1,4], **Haiyan Yin**[1†]

[1]Centre for Frontier AI Research (CFAR), A*STAR, Singapore
[2]National University of Singapore (NUS), Singapore
[3]State Key Laboratory of Novel Software Technology, Nanjing University, China
[4]Nanyang Technological University (NTU), Singapore

ABSTRACT

Long-horizon web agents frequently fail without knowing where or why execution broke down. This issue is particularly pronounced in skill-based agentic web systems, where failures arise within composite skills whose internal decision processes are not directly traceable, making precise diagnosis and repair especially difficult over long horizons. We introduce **SkillTracer**, a framework that represents skills as attributed plan graphs structured by hierarchical nodes and verifiable edge transitions, enabling programmatic verification of execution progress. By decomposing skills into inspectable hierarchies, SkillTracer converts raw interaction traces into structural evidence, making execution breakdowns localizable to specific node-level decision points and attributable to failing components. This attribution signal facilitates targeted structural repair, allowing the agent to selectively revise failing components while preserving the integrity of valid substructures for partial reuse and adaptive recovery. Furthermore, SkillTracer synthesizes short-term traces with long-term historical evidence to construct a persistent skill graph, enabling failure patterns to drive continual refinement across episodes. Evaluated on challenging long-horizon benchmarks, SkillTracer achieves a 17.7% average improvement in success rate over strong baselines, with gains of up to 56.3% in cross-domain settings, demonstrating that structural attribution and skill repair are critical for reliable long-horizon web interaction.

1 INTRODUCTION

Web agents are increasingly deployed to automate complex, goal-directed interactions such as web navigation and multi-step service execution (Deng et al., 2024; Xue et al., 2025). These tasks require operating over long horizons under partial observability, where intermediate states may be transient. As web systems mediate real-world services, the reliability of long-horizon agents becomes critical.

Recent Large Language Models (LLMs) have reframed automation as an instruction-following reactive paradigm, acting as grounded controllers on the Accessibility Tree (AXTree) (Yao et al., 2023). However, this formulation treats steps as isolated decisions without a pre-defined task structure to anchor progress. Lacking an internal benchmark to distinguish minor execution variance from logical failure, small grounding errors often compound into trajectory drift over long horizons (Xue et al., 2025). Existing approaches attempt to mitigate this via reflection mechanisms (Shinn et al., 2023; Madaan et al., 2023). Yet, reflection operates on flattened execution traces that discard the internal decision boundaries of the original plan. Consequently, the agent suffers from **attribution ambiguity**: it cannot align negative outcomes with the specific internal decisions that produced

[*]Equal contribution
[†]Corresponding author

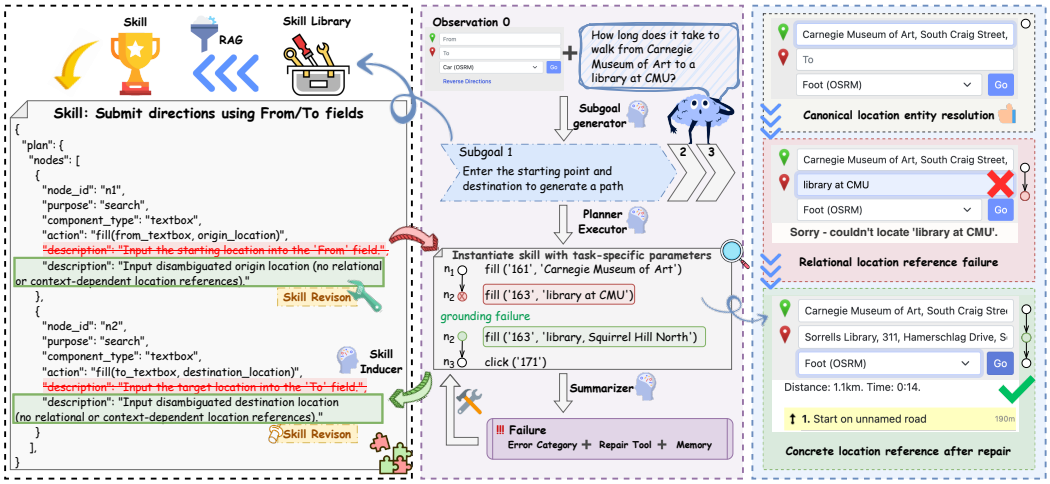

Figure 1: **Overview of the SkillTracer framework. Step 1 (Skill Retrieval):** A user query is decomposed into a subgoal, which retrieves a relevant skill from the skill library and instantiates it as a plan graph (**left**). **Step 2 (Execution and Failure Attribution):** The instantiated skill is executed to produce an interaction trace; execution failure is attributed to the first non-executable plan node using trace-based verification (**center**). **Step 3 (Skill Repair and Reuse):** The failing components are selectively repaired and re-executed, and the revised skill is written back to the library for partial reuse and continual improvement (**right**).

them, making it difficult to distinguish incorrect plans from transient grounding errors and preventing precise, localized repair.

To address long-horizon complexity, recent frameworks adopt **skill induction**, abstracting successful trajectories into reusable execution units (Sharma et al., 2022; Cai et al., 2024). Prevalent approaches model these skills as **monolithic program macros** (e.g., Python functions) (Zhou et al., 2024b; Zheng et al., 2025). Under this formulation, the skill's internal decision structure is opaque during execution. When a skill fails due to UI drift, the agent observes only a terminal signal without access to intermediate states. Without visibility into where the execution deviated, failure attribution is impossible beyond the skill level, reducing recovery to episodic retries rather than systematic refinement (Ouyang et al., 2025).

In this work, we propose **SkillTracer**, a framework that reimagines skills as editable plan graphs rather than atomic macros. Each induced skill is represented as an attributed graph where nodes denote subgoals and edges encode verifiable conditions. During execution, SkillTracer performs **programmatic verification** at each node to validate progress. When execution deviates, this explicit topology renders the logic fully traceable, allowing failures to be localized to the specific node where verification failed. This enables targeted structural repair—selectively revising failing substructures while preserving valid subplans—and aggregates verification outcomes into a **structural memory**, transforming transient failures into signals for systematic refinement.

Our contributions are four-fold:

(1) We introduce a **plan-graph representation** that preserves execution-time decision structure, enabling failures to be localized to specific subgoals rather than entire action sequences.

(2) We propose a **failure attribution mechanism** utilizing **structural memory** to map execution breakdowns to responsible plan nodes, identifying brittle decision patterns over time.

(3) We reformulate skills as **editable plan-graph snapshots** rather than monolithic macros, defining skills as manipulable objects that support localized repair and selective reuse.

(4) We evaluate SkillTracer on **long-horizon web interaction benchmarks**, demonstrating superior reliability with average success rate improvements of **43.1%**, **94.0%**, and **62.8%** over ASI, SkillWeaver, and AWM, respectively.

2 RELATED WORK

Skill Induction in Agentic System. A large body of work studies how agents discover reusable skills and compose them to solve long-horizon tasks (Shin et al., 2019; Sharma et al., 2022; Ellis et al., 2023; Cai et al., 2024; Wang et al., 2024; Grand et al., 2024). Prior approaches typically treat skills as atomic units, focusing on learning, selecting, or recombining skills to improve efficiency and generalization (Zhou et al., 2024b; Zheng et al., 2025). Representative examples include SkillWeaver (Zheng et al., 2025), which enables self-improvement by discovering and honing reusable skills, and PolySkill (Yu et al., 2025), which learns generalizable skills through polymorphic abstraction. Some recent efforts have also explored bootstrapping task spaces and test-time interaction to enhance this discovery process (Shen et al., 2025; Jiang et al., 2025). While effective at the level of skill composition, these methods often assume skills are internally opaque and do not support reasoning about failures within a skill. In contrast, SkillTracer models skills as structured, hierarchical plan graphs whose internal nodes can be inspected, modified, and partially reused, enabling localized repair rather than external recomposition.

Programmatic and Structured Skill Representations. To improve interpretability and compositionality, several works represent skills as low-level natural language descriptions in prompt libraries (Wang et al., 2025b; Zhu et al., 2025; Ouyang et al., 2025) or brittle action traces from successful executions (Wang et al., 2025b; Zheng et al., 2025). A more robust approach represents Skills as Programs, an area pioneered in interactive environments like Minecraft (Wang et al., 2023) and recently applied to web agents (Zheng et al., 2025) and software engineering (Chen et al., 2025). For instance, *Inducing Programmatic Skills for Agentic Tasks* (Wang et al., 2025a) encodes skills as executable programs derived from interaction traces. Related efforts similarly aim to lift behavior from raw trajectories into symbolic representations (Sarch et al., 2024; Wong et al., 2024). However, these learned programs are typically concrete implementations tied to a single context, failing to handle variations across different websites fulfilling the same function (Wang et al., 2023; Zheng et al., 2025). SkillTracer differs by explicitly modeling the internal structure of skills as editable plan graphs, allowing failures to be localized and repaired at specific nodes.

Planning under Partial Observability and Failure Recovery. Planning in web environments remains challenging due to the vast, unseen web and cascading execution errors (Xue et al., 2025). Existing agents often address failures through global replanning, recursive introspection (Qu et al., 2024), or repeated trial-and-error driven by reinforcement learning (Zhou et al., 2024b; Murty et al., 2024a;b). While some approaches incorporate episodic memory (Park et al., 2023), hierarchical working memory (Hu et al., 2024), or persistent external stores (Chhikara et al., 2025; Fang et al., 2025) to bias future decisions, they lack explicit representations connecting failures to internal plan structure. Other methods rely on LM-based synthesis or agent-driven exploration (Ou et al., 2024; Xu et al., 2025; Murty et al., 2024b) to provide "warm starts" on specific websites (Deng et al., 2024). SkillTracer addresses this gap by combining hierarchical plan representations with cross-episode structural memory, enabling agents to attribute failures to recurring plan-level patterns and systematically avoid brittle structures.

3 METHODOLOGY

3.1 SKILL-BASED WEB AGENT PROBLEM

Problem Formulation and Skill Representation. We formulate the web agent task as a sequential decision process where an agent accomplishes a natural-language query q by interacting with an environment state S_t (represented by the Accessibility Tree). At each step t, the agent selects a primitive action $a_t \in \mathcal{A}$ based on a policy π. However, long-horizon queries expose two fundamental challenges: *grounding* high-level intent to implicit UI elements, and *representation* of tasks as flat action sequences, which lack structural dependencies for error localization.

To address this, we introduce **skills** as executable abstractions. Standard approaches typically induce a monolithic skill $\mathcal{K}$ from a complete trajectory T via a policy π_s, updating the action space as $\mathcal{A} \leftarrow \mathcal{A} \cup \{\mathcal{K}\}$, where $\mathcal{K} = \pi_s(T)$. Functionally, this treats $\mathcal{K}$ as a high-level primitive that executes the entire sequence T atomically. In contrast, **SkillTracer** defines skills as modular, parameterized

templates scoped to specific **subgoals**. Given a verified trajectory segment $T^{(g)}$, the induction policy synthesizes a structural skill graph by lifting instance-specific literals into symbolic parameters θ. This transformation decouples interaction logic from data instances, enabling the skill to serve as a versatile, reusable plan template rather than a rigid linear macro.

Skill-based Execution Pipeline. We formalize skill execution as the context-aware instantiation of a skill graph. Given a subgoal g_i, the system first derives a state slice $S_t^{(i)} = \psi(S_t, g_i)$ containing only relevant UI elements. The most relevant skill $\mathcal{K}^*$ is retrieved, and its plan graph $G_{\mathcal{K}^*}$ serves as a structural skeleton for the execution pipeline, which coordinates four modules. First, the **Planner** adapts $G_{\mathcal{K}^*}$ to the current subgoal by pruning or extending the graph to produce an executable plan G_{g_i}. Next, the **Executor** grounds abstract plan nodes in G_{g_i} to concrete UI elements in S_t and executes induced actions sequentially. Simultaneously, the **Summarizer** aggregates the execution trace—including $S_t^{(i)}$, G_{g_i}, and state transitions $\{\Delta S\}$—to evaluate outcomes and trigger localized repair signals upon failure. Finally, successful trajectories are consolidated by the **Skill Inducer** to update the skill library, ensuring that refined graph structures are accumulated as persistent knowledge.

3.2 Plan Representation as an Attributed Graph

To enable precise failure attribution and targeted repair for skills, we shift from implicit execution traces to an explicit **internal plan representation**. We formalize the agent's strategy as a diagnosable **attributed plan graph**, where execution semantics are governed by structured node-level subgoals and state-dependent transitions. This formulation ensures that intermediate decisions are not merely logged, but remain inspectable and verifiable at runtime, providing the necessary topology for localized fault recovery.

Hierarchical Composition of Plan Graphs. We represent each *skill* not as a static macro or flat action sequence, but as a structured plan with explicit internal decision structure. Concretely, a skill is formulated as a directed attributed graph $G = \langle V, E \rangle$, where nodes encode atomic interaction decisions and edges specify their structural ordering and dependencies. Each node $v_t \in V$ represents a single interaction unit and is defined as a typed tuple:

$$G = \langle V, E \rangle, \qquad v_t = \langle \tau_t,\ \gamma_t,\ \alpha_t,\ \delta_t \rangle \in V. \tag{1}$$

Crucially, this factorization enforces a top-down grounding hierarchy. The *interaction intent* τ_t specifies the semantic objective of the step (e.g., navigation, selection, input), which constrains the admissible *UI component type* γ_t. The component type in turn restricts the set of valid *executable operations* α_t, preventing structurally invalid actions. The *description* δ_t then resolves the abstract node to a concrete AXTree element in the current page state S.

During execution, node instantiation is carried out by the planner policy π_p following this generative dependency. Rather than predicting attributes independently, π_p performs constrained inference: the selection of an interaction intent τ_t gates the permissible component types γ_t and executable operations α_t (e.g., syntactically precluding `text-entry` on `button` elements). This structure aligns the agent's decisions with the functional affordances of the environment and reduces grounding errors caused by incompatible action–element pairings.

Programmatic Verification of Edge Preconditions. Directed edges $e_{ij} \in E$ are gated by **programmatically verifiable** preconditions κ_{ij}, which deterministically validate the executability of successor nodes v_j against the current page state S_t. Unlike heuristic LLM critiques, these predicates yield binary, high-fidelity failure signals based on two atomic checks:

(a) **Component Availability** verifies the presence of the required component type:

$$\kappa_{ij}^{\text{comp}}(\mathfrak{s}_t; \gamma_j) = \mathbf{1}[\gamma_j \in \text{types}(\mathfrak{s}_t)]\,. \tag{2}$$

(b) **Anchor Text Presence** confirms the existence of a synthesized textual anchor c_{ij}:

$$\kappa_{ij}^{\text{text}}(\mathfrak{s}_t; c_{ij}) = \mathbf{1}[c_{ij} \in \text{texts}(\mathfrak{s}_t)]\,. \tag{3}$$

Verification aggregates these decidable constraints over the current edge set:

$$\mathcal{V}(\boldsymbol{\kappa}_t, \mathfrak{s}_t) \triangleq \begin{cases} 1, & \text{if } \boldsymbol{\kappa}_t = \emptyset \\ \bigwedge_{e_{ij} \in E_t} \kappa_{ij}(\mathfrak{s}_t), & \text{otherwise.} \end{cases} \tag{4}$$

Violated predicates provide precise diagnostics for localized repair, while valid subgraphs are preserved for future *structural reuse*. See Appendix B.1 for predicate definitions and implementation details.

3.3 Run-Time Failure Localization and Repair

To address execution anomalies dynamically within a single episode, the system employs a **Diagnostic Reasoning Module** that operates in real-time. Upon detecting a failure at step t, this module synthesizes a comprehensive evidence set $\mathcal{E}_t = \{\Delta S_t, \boldsymbol{z}_t, M_t\}$ to derive a localized structural repair operator Ω_t(detailed specifications of repair operators are listed in Appendix B.2). The evidence set consolidates three distinct diagnostic signals: **transition evidence** ΔS_t, capturing the differential state changes in the DOM induced by the preceding action; **verification evidence** $\boldsymbol{z}_t$, aggregating the deterministic boolean outcomes of the edge-feasibility predicates; and **memory evidence** M_t, which contextualizes the current error against immediate interaction history to prevent cyclic failures. By integrating these signals, the reasoning engine rectifies the plan topology on-the-fly, allowing the agent to recover from structural mismatches without restarting the entire task.

Diagnostic Memory. To ensure temporal consistency during runtime correction, the reasoning engine maintains a dual-layer memory $M_t = \langle M_t^s, M^\ell \rangle$. **Short-term memory** M_t^s records the history of localized failures and ineffective repairs within the current interaction episode. Formally, it is defined as a set of failure-repair tuples, preventing the agent from entering cyclic loops by penalizing redundant modifications to the same node v:

$$M_t^s \;=\; \{\langle v_k, \Omega_k \rangle \mid k < t,\ \text{Outcome}(v_k) = \texttt{False}\}\,. \tag{5}$$

Upon successful task completion, validated repairs are consolidated into **long-term memory** M^ℓ. This mechanism distills episodic experiences into persistent structural knowledge, allowing the agent to retrieve proven repair strategies for similar failure contexts in future sessions:

$$M^\ell \;\leftarrow\; M^\ell \cup \{\langle v, \Omega \rangle \in M_t^s \mid \text{Task}(Q) = \texttt{Success}\}\,. \tag{6}$$

Diagnosis Objective. The core decision process identifies the optimal repair strategy from generated candidates. Let $G'_t(\Omega) = \text{Apply}(G_t, \Omega)$ denote the candidate graph topology resulting from a repair Ω. The diagnostic module selects the repair Ω_t^* that maximizes logical consistency with the evidence set $\{\Delta S_t, \boldsymbol{z}_t, M_t\}$ while penalizing structural deviation from the original plan G_t:

$$\Omega_t^* \;\in\; \Big\{\Omega \;\Big|\; \max_{\Omega'} \Big[\mathcal{C}\Big(G'_t(\Omega') \mid \Delta S_t, \boldsymbol{z}_t, M_t\Big) \;-\; \lambda \cdot d\Big(G'_t(\Omega'), G_t\Big)\Big]\Big\}.$$

Here, $\mathcal{C}$ evaluates the consistency of the repaired graph against the transition (ΔS_t), verification ($\boldsymbol{z}_t$), and memory (M_t) signals, while the regularization term $d(\cdot, \cdot)$ enforces **minimal intervention** to preserve the valid portions of the existing plan.

3.4 Cross-Episode Skill Reuse and Adaptation

This section details how learned skills are systematically transferred across diverse tasks. In contrast to reactive agents that re-plan from scratch or execute fixed action sequences, *SkillTracer* leverages a library of structural templates that preserve logic while allowing for high-fidelity grounding.

Retrieval-based Skill Instantiation. A skill in *SkillTracer* is a reusable plan graph that captures the invariant structural logic of an interaction. Each skill is stored as a structured artifact:

$$\mathcal{K} = \langle \text{name}, \text{desc}, G_{\mathcal{K}} \rangle, \tag{7}$$

where name identifies the skill, desc summarizes its semantic intent, and $G_{\mathcal{K}}$ is a plan graph whose node attributes are parameterized as symbolic placeholders.

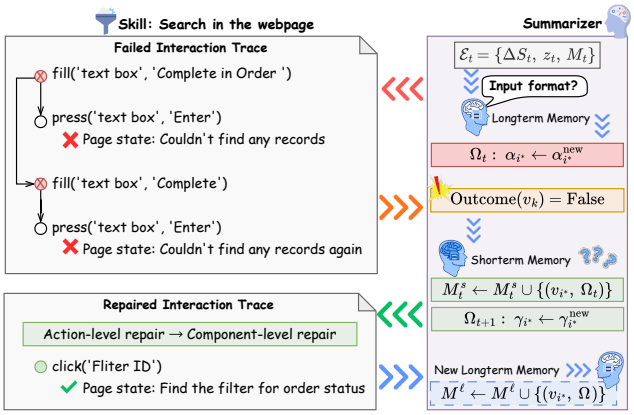

Figure 2: **An illustrative example of failure localization and structural repair from SkillTracer.** (a) A skill execution produces a failed interaction trace, where repeated action-level repairs lead to execution stagnation. (b) The summarizer aggregates execution evidence to detect stagnation and localize the failure to the responsible decision component. (c) The repair strategy is escalated to a component-level revision; ineffective repairs are recorded in short-term memory, and successful repairs are consolidated into long-term memory to guide future executions.

Table 1: Performance comparison across long-horizon web interaction domains, grouped by agentic framework category. Under the same GPT-4o backbone, SkillTracer consistently outperforms prior skill-induction-based systems, achieving average success rate improvements of 43.1%, 94.0%, and 62.8% over ASI, SkillWeaver, and AWM, respectively. These results indicate that attributed failure localization and structural repair provide robustness gains beyond those achievable by model scaling or skill induction alone.

Methods	**LLM Backbone**	**Domain**						**Avg.**
		Shopping	Admin	Reddit	GitLab	Map	Cross	
Single-Agent Systems								
WebRL (Qi et al., 2025)	Llama-3.1-70B	44.4	54.3	**78.9**	50.0	40.0	–	49.1
AgentOccam (Yang et al., 2025)	GPT-4-Turbo	40.6	45.6	61.3	37.8	46.8	14.6	43.1
AgentOccam + JUDGE (Yang et al., 2025)	GPT-4-Turbo	43.3	46.2	67.0	38.9	52.3	16.7	45.7
Multi-Agent Systems								
WebPilot (Zhang et al., 2025)	GPT-3.5	25.1	22.0	58.5	30.0	30.3	–	29.1
WebPilot (Zhang et al., 2025)	GPT-4o	36.9	24.7	65.1	39.4	33.9	–	37.2
Web Agentic Systems with Skill Induction								
AWM (Wang et al., 2025b)	GPT-4	30.8	29.1	50.9	31.8	43.3	–	35.5
SkillWeaver (Zheng et al., 2025)	GPT-4o	27.2	25.8	50.0	22.2	33.9	–	29.8
ASI (Wang et al., 2025a)	Claude-3.5-Sonnet	40.1	44.0	54.7	32.2	43.1	20.8	40.4
Agentic Systems with Hierarchical Plan-Graph (Ours)								
SkillTracer (Ours)	GPT-4o	**65.7**	**56.1**	68.7	**51.1**	**63.3**	**32.5**	**57.8**
		48.0%↑	**3.3%**↑	**12.9%**↓	**2.2%**↑	**21.0%**↑	**56.3%**↑	**17.7%**↑

Given a subgoal g, the planner retrieves a small set of candidate skills based on semantic similarity between g and the skill descriptions, and selects the most relevant one $\mathcal{K}^*$.

$$\{\mathcal{K}_1, \ldots, \mathcal{K}_k\} \sim \text{Top-}k(g, \{\text{description}(\mathcal{K})\}). \tag{8}$$

The optimal candidate $\mathcal{K}^*$ is selected, and its graph plan graph $G_{\mathcal{K}^*}$ is then instantiated by grounding its placeholders to the live page state S_t and query q.

Crucially, whereas conventional web agents encode skills as static action sequences or prompt-level macros that bind prematurely to page-specific elements, *SkillTracer* represents skills as reusable structural templates whose internal decision logic is preserved across executions. By separating invariant interaction structure from instance-specific grounding, skills can be instantiated on new pages while retaining verified decision dependencies, enabling reliable transfer across heterogeneous web environments.

Skill Induction and Continual Adaptation. When a subgoal is successfully executed without invoking an existing skill, provided the resulting execution trajectory meets a minimum complexity

threshold $|\mathcal{T}_{1:T}| \geq 2$, the trace is distilled into a new skill artifact:

$$\mathcal{K}_{\text{new}} = \pi_s(Q, \mathcal{T}_{1:T}), \tag{9}$$

where π_s denotes the *skill inducer policy*. This policy abstracts the raw execution trace into a task-agnostic plan graph by replacing instance-specific entities, textual content, and numeric parameters with semantic placeholders.

If an existing skill $\mathcal{K}$ is selected but requires a repair signal Ω to complete the task, *SkillTracer* treats the corrected topology $G'_{\mathcal{K}}$ as a signal for **topological refinement**. Rather than a global re-induction, the system performs a surgical update to the skill representation:

$$\mathcal{K}' = \pi_s(\mathcal{K}, G_{\mathcal{K}}, G'_{\mathcal{K}}, \mathcal{T}_{1:T}). \tag{10}$$

Crucially, this update is governed by a **structural conservative objective**. We formalize the evolution of the skill library as an empirical risk minimization problem that balances robustness against structural drift:

$$\min_{\mathcal{K}'} \mathbb{E}_{\tau\sim\mathcal{D}}\big[\mathbb{I}(\text{fail}(\mathcal{K}', \tau))\big] + \lambda \mathbb{E}_{\tau\sim\mathcal{D}}\big[\Delta_{\text{struct}}(\mathcal{K}', \mathcal{K})\big], \tag{11}$$

The first term represents the intent to reduce failure risk over historical trajectories $\mathcal{D}$, while the second term serves as a **minimal intervention regularizer** to penalize excessive structural edit distance Δ_{struct} between the original and revised topologies. By framing refinement as the optimization of a topological prior rather than an unconstrained text-generation task, *SkillTracer* provides a principled mechanism for non-monotonic learning that preserves the functional integrity of the agent's knowledge base across heterogeneous domains. Under this formulation, the library aims to evolve through **stable refinement**; the agent intends to expand the skill's applicability to diverse page states while preserving the integrity of the verified interaction logic.

Complete algorithms and pseudocode are provided in Appendix B.3.

4 Experiments

4.1 Experimental Setup

Benchmarks and Evaluation. We evaluate **SkillTracer** primarily on the **WebArena** benchmark Zhou et al. (2024a), a rigorous evaluation suite designed to assess autonomous web agents in realistic environments. WebArena comprises 812 test tasks spanning five major real-world web domains. To further assess the generalization capability of our hierarchical plan graphs, we also report performance on *Cross-Domain* tasks, where agents must navigate across multiple websites to achieve a single objective.

Following standard protocols Zhou et al. (2024a); Wang et al. (2025a), we measure performance using **Success Rate (SR)** based on functional correctness. Following prior baselines, we employs program-based evaluators to assess execution trajectories, yielding deterministic binary outcomes and ensuring consistent comparison across methods. We utilize the standard observation space provided by BrowserGym Drouin et al. (2024), converting the accessibility tree (AXTree) into a text-based representation.

Baselines. We compare SkillTracer against three distinct categories of state-of-the-art web agents. **Single-Agent Systems** include **WebRL** (Qi et al., 2025), which applies reinforcement learning to optimize open-ended tasks, and **AgentOccam** (Yang et al., 2025), which leverages visual grounding and pruning. **Multi-Agent Systems** are represented by **WebPilot** (Zhang et al., 2025), utilizing a decoupling strategy to handle complex dependencies. Finally, as our most direct comparisons, **Skill-Induction Systems** include **AWM** (Wang et al., 2025b) for workflow induction, **SkillWeaver** (Zheng et al., 2025) for skill discovery, and **ASI** (Wang et al., 2025a) which emphasizes execution verification.

Backbone and Architecture. For the primary evaluation of SkillTracer, we utilize **GPT-4o** as the backbone Large Language Model (LLM) for the Planner, Executor, and Repairer modules. This ensures a fair comparison with the strongest baselines in Table 1, particularly those utilizing GPT-4o or Claude-3.5-Sonnet. Our action space $\mathcal{A}$ aligns with the standard WebArena default action space (detailed in Appendix A) to isolate the gains provided by our plan-graph representation rather than low-level action primitives.

Table 2: Comparison of token consumption during the **Skill Induction** phase. The relative reduction is shown in parentheses.

Method	Induction Cost (Tokens)
ASI	5,773
SkillTracer	**1,932** (↓ **66.5%**)

Implementation Details. We set the maximum number of interaction steps per episode to 10. For skill retrieval, we employ a RAG-based mechanism with a top-k setting of $k = 3$. The Planner is configured with a temperature of 0.3 to encourage stable and deterministic structure generation, while the Executor also uses a temperature of 0.3 to balance consistency and grounding robustness. All experiments were conducted on a standard workstation using commodity computing resources. (see Appendix C.1 for details).

4.2 Benchmark Results

Overall Performance. Table 1 summarizes performance across six domains. SkillTracer achieves a leading average success rate of **57.8%**, with robust single-domain scores: **65.7%** (Shopping), **56.1%** (Admin), **51.1%** (GitLab), **63.3%** (Map), and **68.7%** (Reddit). Crucially, in the Cross-domain setting, it significantly outperforms the strongest baseline (**20.8%**) with **32.5%** success. Overall, SkillTracer delivers an average gain of **+17.7 percentage points**, with a **56.3%** relative improvement in Cross-domain tasks, validating the efficacy of hierarchical, verifiable plan graphs for stable generalization.

Induction Overhead. Table 2 highlights that SkillTracer reduces token consumption during the skill induction phase by **66.5%** compared to ASI (1,932 vs. 5,773 tokens). This efficiency stems from two architectural design choices. First, instead of employing extensive multi-round verification to refine skills prior to storage, SkillTracer adopts a **continual improvement** paradigm. We treat the initial induction as a lightweight registration process, deferring optimization to the execution phase, where the localized repair mechanism naturally iterates on brittle graph nodes. Second, our framework triggers induction strictly upon the successful completion of **subgoals**, ensuring that skills have distinct semantic boundaries and controllable granularity. In contrast, methods that extract skills from full-horizon task trajectories often face challenges in segmentation ambiguity, which can result in prolonged primitives that consume more context while offering lower reusability.

Additional Results. Due to space constraints, we provide comprehensive supplementary analyses in the appendices. Qualitatively, Appendix D details a concrete case study illustrating skill generalization across diverse web environments. Quantitatively, we report system efficiency in Appendix C.2, alongside a comparative analysis of token consumption costs in Appendix C.4. Finally, extended ablation studies dissecting specific module contributions are presented in Appendix C.3.

5 Conclusion

We introduced **SkillTracer**, a framework that transforms web interaction from stochastic sequence generation into a principled, verification-aware control process by representing skills as hierarchical, attributed plan graphs. By decoupling invariant interaction logic from site-specific affordances, SkillTracer effectively shifts the agentic bottleneck from unconstrained text generation to the optimization of topological priors. This architectural shift enables precise, white-box failure localization and surgical structural repair, allowing the agent to maintain computational momentum without regressing to global re-planning. Our results demonstrate that robustness in web automation is a function of structural integrity rather than model scale, proving that symbolic plan-graph representations provide the necessary scaffolding for reliable, cross-domain generalization. Ultimately, SkillTracer establishes a scalable foundation for agents that do not merely execute tasks, but incrementally refine a verified world-model of digital interfaces.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-NMLP-2024-003), and the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech Funding Initiative, Career Development Fund (CDF) of the Agency for Science, Technology and Research (A*STAR) (No: C243512014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, the Agency for Science, Technology and Research, or the Infocomm Media Development Authority.

REFERENCES

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://openreview.net/forum?id=qV83K9d5WB.

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. Swe-exp: Experience-driven software issue resolution, 2025. URL http://arxiv.org/abs/2507.23361. arXiv preprint arXiv:2507.23361.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems (NeurIPS 2023)*, volume 36, 2024.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks?, 2024. URL https://arxiv.org/abs/2403.07718.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu Manager, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory, 2025. URL https://arxiv.org/abs/2508.06433. arXiv preprint arXiv:2508.06433.

Gabriel Grand, Lionel Wong, Matthew Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://openreview.net/forum?id=TqYbAWKMIe.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model, 2024. URL https://arxiv.org/abs/2408.09559. arXiv preprint arXiv:2408.09559.

Minqi Jiang, Andrei Lupu, and Yoram Bachrach. Bootstrapping task spaces for self-improvement, 2025. URL https://arxiv.org/abs/2509.04575. arXiv preprint arXiv:2509.04575.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023.

Shikhar Murty, Dzmitry Bahdanau, and Christopher D. Manning. Nnetscape navigator: Complex demonstrations for web agents without a demonstrator, 2024a. URL https://arxiv.org/abs/2410.02907. arXiv preprint arXiv:2410.02907.

Shikhar Murty, Christopher Manning, Peter Shaw, Mandar Joshi, and Kenton Lee. Bagel: Bootstrapping agents by guiding exploration with language, 2024b. URL https://arxiv.org/abs/2403.08140. arXiv preprint arXiv:2403.08140.

Tianyue Ou, Frank F. Xu, Aman Madaan, Jiarui Liu, Robert Lo, Abishek Sridhar, Sudipta Sengupta, Dan Roth, Graham Neubig, and Shuyan Zhou. Synatra: Turning indirect knowledge into direct demonstrations for digital agents at scale. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS 2024)*, 2024. URL https://openreview.net/forum?id=KjNEzWRIqn.

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

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL https://arxiv.org/abs/2304.03442. arXiv preprint arXiv:2304.03442.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=oVKEAFjEqv.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve, 2024. URL https://arxiv.org/abs/2407.18219. arXiv preprint arXiv:2407.18219.

Gabriel Sarch, Lawrence Jang, Michael Tarr, William W. Cohen, Kenneth Marino, and Katerina Fragkiadaki. Vlm agents generate their own memories: Distilling experience into embodied programs of thought. *Advances in Neural Information Processing Systems (NeurIPS 2024)*, 37: 75942–75985, 2024.

Pratyusha Sharma, Antonio Torralba, and Jacob Andreas. Skill induction and planning with latent language. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 120–135. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.acl-long.120/.

Junhong Shen, Hao Bai, Lunjun Zhang, Yifei Zhou, Amrith Setlur, Shengbang Tong, Diego Caples, Nan Jiang, Tong Zhang, Ameet Talwalkar, and Aviral Kumar. Thinking vs. doing: Agents that reason by scaling test-time interaction, 2025. URL http://arxiv.org/abs/2506.07976. arXiv preprint arXiv:2506.07976.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems (NeurIPS 2019)*, volume 32, 2019.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023.

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, 2023. URL http://arxiv.org/abs/2305.16291. arXiv preprint arXiv:2305.16291.

Zhiruo Wang, Graham Neubig, and Daniel Fried. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *Forty-first International Conference on Machine Learning (ICML 2024)*, 2024. URL https://openreview.net/forum?id=DCNCwaMJjI.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. In *Second Conference on Language Modeling*, 2025a. URL https://openreview.net/forum?id=lsAY6fWsog.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=NTAhi2JEEE.

Lionel Wong, Jiayuan Mao, Pratyusha Sharma, Zachary S. Siegel, Jiahai Feng, Noa Korneev, Joshua B. Tenenbaum, and Jacob Andreas. Learning grounded action abstractions from language. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://openreview.net/forum?id=qJ0Cfj4Ex9.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025. URL https://openreview.net/forum?id=EEgYUccwsV.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents, 2025. URL https://arxiv.org/abs/2504.01382. arXiv preprint arXiv:2504.01382.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzefa Rangwala. Agentoccam: A simple yet strong baseline for LLM-based web agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=oWdzUpOlkX.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://arxiv.org/pdf/2210.03629.

Simon Yu, Gang Li, Weiyan Shi, and Peng Qi. Polyskill: Learning generalizable skills through polymorphic abstraction, 2025. URL https://arxiv.org/abs/2510.15863.

Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration. In *AAAI*, pp. 23378–23386, 2025. URL https://doi.org/10.1609/aaai.v39i22.34505.

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. 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. Webarena: A realistic web environment for building autonomous agents, 2024a. URL https://arxiv.org/abs/2307.13854.

Yifei Zhou, Qianlan Yang, Kaixiang Lin, Min Bai, Xiong Zhou, Yu-Xiong Wang, Sergey Levine, and Erran Li. Proposer-agent-evaluator(pae): Autonomous skill discovery for foundation model internet agents, 2024b. URL http://arxiv.org/abs/2412.13194. arXiv preprint arXiv:2412.13194.

He Zhu, Tianrui Qin, King Zhu, Heyuan Huang, Yeyi Guan, Jinxiang Xia, Yi Yao, Hanhao Li, Ningning Wang, Pai Liu, Tianhao Peng, Xin Gui, Xiaowan Li, Yuhui Liu, Yuchen Eleanor Jiang, Jun Wang, Changwang Zhang, Xiangru Tang, Ge Zhang, Jian Yang, Minghao Liu, Xitong Gao, Jiaheng Liu, and Wangchunshu Zhou. Oagents: An empirical study of building effective agents, 2025. URL https://arxiv.org/abs/2506.15741. arXiv preprint arXiv:2506.15741.

A ACTION SPACE

Table 3: Complete action space used throughout experiments.

Action Type	Description
`noop(wait_ms)`	Do nothing for specified time.
`click(elem)`	Click at an element.
`hover(elem)`	Hover on an element.
`fill(elem, value)`	Type into an element.
`keyboard_press(key_comb)`	Press a key combination.
`scroll(x, y)`	Scroll horizontally or vertically.
`select_option(elem, options)`	Select one or multiple options.
`memorize(elem)`	Memorize specific element on the page.
`goto(url)`	Navigate to a URL.
`go_back()`	Navigate to the previous page.
`go_forward()`	Navigate to the next page.
`new_tab()`	Open a new tab.
`tab_close()`	Close the current tab.
`tab_focus(index)`	Bring tab to front.
`send_msg_to_user(text)`	Send a message to the user.
`report_infeasible(reason)`	Notify user that the task is infeasible.

Action Space. We adopt the same primitive action space as prior WebArena-based baselines to ensure fair comparison, including standard navigation, interaction, and input operations (Table 6). To support tasks that require reasoning over intermediate results—such as comparing multiple items when the webpage does not provide built-in comparison tools—we introduce an additional `memorize` action. Unlike environment-affecting actions, `memorize(elem)` does not trigger any browser interaction. Instead, it takes an element identifier as input, and the system records a localized structural snapshot consisting of the target element together with its immediate parent and children (one level up and down in the AXTree). This cached information is injected into subsequent execution steps as auxiliary context, enabling cross-step comparison and reference without altering the underlying page state.

B METHODOLOGICAL SPECIFICATIONS AND ALGORITHMS

B.1 FORMAL CONSTRAINTS ON PLAN GRAPH EDGES

In the Plan Graph π_g, as introduced in Section 3.2, each edge is defined as $e = (N_u, N_v, C)$, where $N_u, N_v \in V$, and C is an optional precondition (denoted as $\perp$ if absent). To ensure the mechanical verifiability of the plan, preconditions C must satisfy the following strict constraints:

1. **Cardinality Constraint**: $|C| \leq 1$.
2. **Markov Property**: $Verify(S_t, C)$ depends only on the current state S_t and cannot reference history states $S_{<t}$ or actions $A_{<t}$.
3. **Predicate Restriction**: Only two types of predicates are allowed:

$$C \in \{exists\_component(\tau), exists\_content(\sigma)\} \tag{12}$$

4. **Generation:** The precondition C is produced by the planner conditioned on the current page state S_t, and is restricted to predicates that can be deterministically verified by a programmatic validator.
5. **Validation:** Preconditions are evaluated using deterministic string matching on the current page state S_t, without semantic expansion or approximate matching, ensuring reproducible and unambiguous verification.

Algorithm 1 SkillTracer: Subgoal-Conditioned Retrieval, Planning, Local Repair, and Skill Induction

Require: Task context Q, environment $\mathcal{E}$, skill library $\mathcal{K}$, memories $M_t = \langle M_t^s, M^l \rangle$
Ensure: Updated skill library $\mathcal{K}$

$t \leftarrow 0, \mathcal{B} \leftarrow \emptyset$ (task-level temporary skill buffer)
while TASKSOLVED$(Q) = 0$ **do**
  $g_t \sim \pi_g(\cdot \mid S_t, Q)$ (generate current subgoal)
  $\mathcal{C}_t \leftarrow \mathrm{RAG}_k(g_t, \mathcal{K})$ (retrieve top-k skills)
  $G_t \sim \pi_p(\cdot \mid S_t, Q, g_t, \mathcal{C}_t, M_t)$ (plan graph; may adopt a skill)
  SUBGOALDONE $\leftarrow 0$
  while SUBGOALDONE $= 0$ **do**
    $(\mathcal{T}_{1:T}, \mathcal{E}_t, \text{SUBGOALDONE}) \leftarrow \text{EXECUTE}(G_t, \mathcal{E}, M_t)$
    $\Omega_t \leftarrow \text{SUMMARIZE}(\mathcal{T}_{1:T}, \mathcal{E}_t, G_t, M_t)$ (derive repair signal from evidence)
    if $\Omega_t \neq \varnothing$ **then**
      $S_t \leftarrow \text{ROLLBACK}(S_t, \mathcal{T}_{1:T})$ (return to checkpoint)
      $G_t \leftarrow \mathrm{Apply}(G_t, \Omega_t)$ (localized structural repair)
    end if
  end while
  if $|\mathcal{T}_{1:T}| \geq 2$ **then**
    $\mathcal{B} \leftarrow \text{SKILLINDUCER}(g_t, \mathcal{T}_{1:T}, \Omega_t, M_t)$ (attribute & refine skill after subgoal completion)
  end if
  $t \leftarrow t + 1$
end while
$\mathcal{K} \leftarrow \text{MERGE}(\mathcal{K}, \mathcal{B})$ (commit on task completion)
SAVE$(\mathcal{K})$
return $\mathcal{K}$

B.2 Structural Repair Operators

Based on the diagnostic evidence, the reasoning engine generates a repair operator Ω defined as a targeted modification to a specific attribute of a plan node. These operators address granularity mismatches at different levels of the plan hierarchy:

$$\Omega = \begin{cases} \tau_{i^*} \leftarrow \tau_{i^*}^{\text{new}}, & \text{Level 1: Rectify Intent/Stage} \\ \gamma_{i^*} \leftarrow \gamma_{i^*}^{\text{new}}, & \text{Level 2: Correct Component Type} \\ \alpha_{i^*} \leftarrow \alpha_{i^*}^{\text{new}}, & \text{Level 3: Adjust Executable Action} \\ \delta_{i^*} \leftarrow \delta_{i^*}^{\text{new}}, & \text{Level 4: Refine Semantic Description} \end{cases} \tag{13}$$

Operators at higher levels (τ, γ, a) imply topological changes that necessitate a **cascading invalidation** of downstream nodes to preserve logical consistency. In contrast, low-level refinements (δ) constitute localized semantic corrections, allowing the executor to resume the existing plan immediately without regenerating the remaining graph structure.

B.3 Algorithm

Algorithm 1 formalizes the resulting SkillTracer lifecycle.

C Experimental Setup and Extended Analysis

C.1 Implementation Details

Model Hyperparameter Configuration Table 4 summarizes the implementation configuration for each agent module. While the current instantiation uses a unified backbone model and decoding setup across all roles for fair comparison and simplicity, the framework itself is designed to support *role-specific model configurations*. In particular, different agent components (e.g., planning, execution, or summarization) can be assigned distinct language models, decoding temperatures, or context budgets, enabling flexible trade-offs between reasoning quality, stability, and computational cost in future extensions.

Checkpoint-Based Rollback Mechanism. This mechanism 2 is built on the assumption that URL transitions provide a reliable structural boundary: once the URL changes, the page state is refreshed

Table 4: Implementation details of each agent module. All modules share the same backbone model and decoding configuration.

Agent Module	Model	Temperature	Max Output Tokens	Max Input Tokens
Subgoal Generator	GPT-4o	0.3	4096	128k
Planner	GPT-4o	0.3	4096	128k
Executor	GPT-4o	0.3	4096	128k
Summarizer	GPT-4o	0.3	4096	128k
Skill Inducer	GPT-4o	0.3	4096	128k

Algorithm 2 Checkpoint-Based Rollback Mechanism

Require: Current subgoal u_g, executed trajectory T_t, basic URL U_g
Ensure: Rollback action list $\mathcal{A}_g$
Initialize current URL $U_t \leftarrow U_g$ ▷ Initialize URL-level checkpoint
Initialize temporary action buffer $\mathcal{A}_t \leftarrow \emptyset$ ▷ Buffer actions since last URL change
for each executed pair $(u, a) \in T_t$ **do**
  if $u \neq U_t$ **then**
    Reset $\mathcal{A}_t \leftarrow \emptyset$ ▷ URL change invalidates prior actions
    Update $U_t \leftarrow u$ ▷ Advance checkpoint to new page
  else
    Append action a to $\mathcal{A}_t$ ▷ Action is reversible within page
  end if
end for
if subgoal completed **then**
  if $U_g \neq U_t$ **then**
    $\mathcal{A}_g \leftarrow \mathcal{A}_t$ ▷ Keep only actions after last URL transition
  else
    $\mathcal{A}_g \leftarrow \mathcal{A}_g \cup \mathcal{A}_t$ ▷ Merge actions within unchanged page
  end if
end if

and any prior transient UI operations become invalid. By treating URL changes as stable checkpoints, the agent avoids replaying stale interactions and reduces accumulated execution noise caused by intermediate page manipulations. During execution, actions are buffered as $\mathcal{A}_t$ only while the URL remains unchanged; when a URL transition is detected, the buffer is reset and the checkpoint is advanced. After a subgoal completes, the buffered actions are consolidated into $\mathcal{A}_g$, representing the minimal set of page-local operations required to reproduce the final state. Upon failure and subsequent repair, execution resumes by reapplying $\mathcal{A}_t$, ensuring that post-repair actions are grounded in a clean, consistent page state while preserving progress made since the last reliable checkpoint.

C.2 Efficiency Analysis

Table 5 reports the average token cost per task and the average number of steps among successful trajectories. Importantly, the token accounting includes *both* workflow action generation and skill induction, reflecting the full end-to-end cost of SkillTracer. Across domains, token usage varies substantially with task complexity and interface heterogeneity: Map has the lowest average token cost (81,036) with 4.9 steps on successful runs, while Cross-domain requires the highest token budget (187,193) and the longest successful trajectories (7.2 steps). Overall, SkillTracer uses 112,601 tokens per task on average, with 5.4 steps per successful trajectory.

C.3 Ablation Study

Ablation on Skill Reuse. Table 6 shows that removing skill reuse leads to a clear drop in task success on both Shopping (65.7% to 45.16%) and Shopping-Admin (56.1% to 48.39%), accompanied by higher token cost and longer successful trajectories. Without reusable plan templates, the agent relies more heavily on on-the-fly planning, resulting in increased computation and reduced execution reliability.

Table 5: Efficiency statistics of **SkillTracer** across domains. Token cost is reported as average *tokens per task*, and includes both workflow action generation and skill induction. Steps are averaged over *successful* trajectories only.

Domain	Token Cost (per task)	Avg. Steps (success only)
Shopping	82,370	4.2
Admin	112,422	6.7
Reddit	146,833	5.8
GitLab	120,138	5.1
Map	81,036	4.9
Cross	187,193	7.2
Avg.	111,910	5.4

Table 6: Ablation study analyzing the impact of skill reuse on task success, token efficiency, and execution length. Results are reported on Shopping and Shopping-Admin domains.

Method	Metric	Shopping	Shopping-Admin
SkillTracer	Success (%)	65.7	56.1
	Token Cost	82,370	112,422
	Steps (Succ.)	4.2	6.7
w/o Skill	Success (%)	45.16	48.39
	Token Cost	120,541	163,705
	Steps (Succ.)	5.1	6.9
Δ	Success (%)	$\uparrow$ 20.5	$\uparrow$ 7.7
	Token Cost	$\downarrow$ 31.7%	$\downarrow$ 31.3%
	Steps (Succ.)	$\downarrow$ 21.4%	$\downarrow$ 3.0%

This degradation highlights the role of skills as structural priors over interaction patterns. By reusing validated plan graphs, the agent avoids repeatedly rediscovering stable interaction structures, reducing both planning overhead and exposure to grounding errors. In their absence, planning becomes more reactive and brittle, leading to longer trajectories and lower overall success.

Ablation on Failure Localization and Repair. As reported in Table 7, disabling the repair mechanism substantially reduces success rates on Shopping (65.7% to 44.4%) and Shopping-Admin (56.1% to 42.8%). Notably, the trajectory lengths of successful episodes remain relatively stable, suggesting that the overhead of localized failure detection and structural repair is negligible. These results demonstrate that SkillTracer achieves significant gains in reliability without sacrificing computational efficiency, highlighting the framework's operational sustainability.

Without explicit failure localization, execution errors propagate unchecked, forcing the agent to either terminate prematurely or continue with an invalid plan structure. The repair mechanism enables targeted correction at the node and attribute level, preventing cascading failures and preserving valid portions of the plan graph. Its removal therefore directly undermines robustness, even when successful trajectories appear only marginally longer.

C.4 Token Consumption Comparison Analysis

Token breakdown. As detailed in Table 8, **SkillTracer** incurs a higher token overhead than ASI in the Shopping domain (82,370 vs. 59,613). This increase is a deliberate architectural consequence rather than an inefficiency. Unlike baselines that prioritize monolithic action generation in a single pass, SkillTracer explicitly allocates computational budget to *intermediate state analysis* and *structural memory maintenance* for error diagnosis. Crucially, this additional expenditure is not consumed by redundant retries, but is invested in **structured verification and localized repair**. By reasoning over page transitions and retaining diagnostic evidence, SkillTracer minimizes blind exploration—a critical advantage in dynamic environments like Shopping where layout shifts are frequent. Empirically, this trade-off proves highly favorable: the moderate increase in token usage translates into a substantial gain in reliability, achieving a **65.7% success rate** that significantly out-

Table 7: Ablation study on the impact of failure localization and structural repair. We report task success rate and average successful trajectory length on Shopping and Shopping-Admin. Removing the repair mechanism substantially degrades both effectiveness and execution efficiency.

Method	Metric	Shopping	Shopping-Admin
SkillTracer	Success (%)	65.7	56.1
	Steps (Succ.)	4.2	6.7
w/o Repair	Success (%)	44.4	42.8
	Steps (Succ.)	4.3	7.2
Δ	Success (%)	↑ 21.3	↑ 13.3
	Steps (Succ.)	↑ 2.4%	↑ 7.5%

Table 8: Token cost breakdown on **Shopping**. We decompose total token usage into workflow action generation and skill induction.

Method	Action Gen.	Skill Induction	Total Tokens
ASI	53,840	5,773	59,613
SkillTracer	80,438	1,932	82,370

performs skill-induction baselines. This confirms that the extra tokens are effectively amortized by the marked improvements in robust task completion.

D Case Study

We examine an **address-editing skill** (Figure 3) induced from successful form-filling trajectories, where specific entities (e.g., *city*, *postal code*) are abstracted into symbolic placeholders. Unlike soft hints, the retrieved skill is directly instantiated as the plan graph and executed under standard verification protocols. Edges enforce explicit preconditions on the page state; for instance, the transition $n_4 \rightarrow n_5$ (State selection) mandates the existence of a `combobox` component. If a target website implements this field as a *textbox*, the precondition fails deterministically at n_5, marking it as a breakpoint. Crucially, rather than discarding the skill, the summarizer preserves the valid prefix (n_1–n_4) and issues a **localized repair** solely for n_5 (e.g., modifying the component type). This in-place adaptation demonstrates robust **cross-site generalization**, allowing partial skill reuse despite structural UI mismatches.

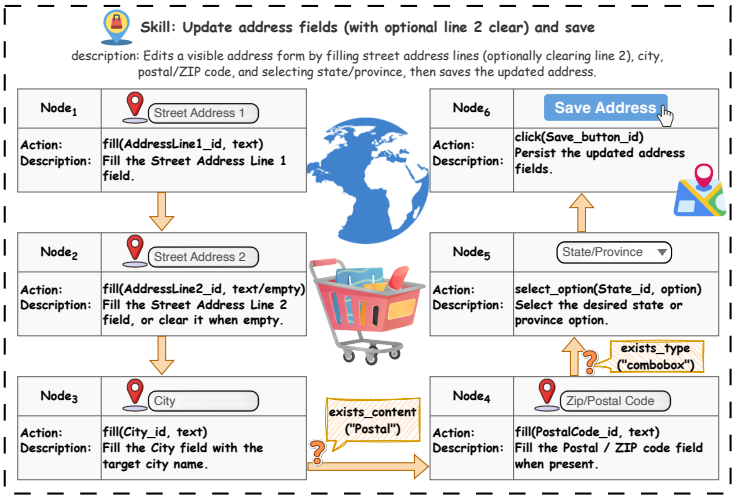

Figure 3: A structured plan-graph representation of a skill in SkillTracer. Explicit node attributes and state-dependent edge conditions make execution verifiable and failures localizable to individual plan nodes.

E SKILL EXAMPLE

In D, we illustrate skill reuse and repair using a schematic example to highlight the overall workflow. To complement that presentation, this appendix provides a *real* skill instance extracted from actual execution traces during evaluation. Unlike the abstracted case study in the main text, the following example shows the exact skill structure stored in the skill library, including its plan graph, node semantics, and execution order.

Skill Example: Skill instance

```
'''skill
{ "id": "s6",
  "name": "Update address fields and save",
  "description": "Edits a visible address form by filling street
    address lines, city, postal/ZIP code, and selecting
    state/province, then saves the updated address.",
  "plan": {
    "nodes": [
      { "node_id": "n1",
        "purpose": "commit_input",
        "component_type": "textbox",
        "action": "fill(address_line1_textbox_id,
    address_line1_text)",
        "description": "Fill the Street Address Line 1 field with the
    provided address line 1 text."},
      { "node_id": "n2",
        "purpose": "commit_input",
        "component_type": "textbox",
        "action": "fill(address_line2_textbox_id,
    address_line2_text_or_empty)",
        "description": "Fill the Street Address Line 2 field with the
    provided value; use an empty value to clear the field when
    supported."},
      { "node_id": "n3",
        "purpose": "commit_input",
        "component_type": "textbox",
        "action": "fill(city_textbox_id, city_text)",
        "description": "Fill the City field with the provided city
    text."},
      { "node_id": "n4",
        "purpose": "commit_input",
        "component_type": "textbox",
        "action": "fill(postal_code_textbox_id, postal_code_text)",
        "description": "Fill the Postal/ZIP code field with the
    provided postal code text."},
      { "node_id": "n5",
        "purpose": "commit_input",
        "component_type": "combobox",
        "action": "select_option(state_province_combobox_id,
    state_province_option_text)",
        "description": "Select the desired state/province option from
    the state/province dropdown."},
      { "node_id": "n6",
        "purpose": "commit_input",
        "component_type": "button",
        "action": "click(save_address_button_id)",
        "description": "Click the Save/Submit button to persist the
    updated address fields."}],
    "edges": [
      {"from": "n1","to": "n2"},
      {"from": "n2","to": "n3"},
      {"from": "n3","to": "n4","preconditions":
    ["exists_content('Postal')"]},
```

```
      {"from": "n4","to": "n5","preconditions":
    ["exists_type('combobox')"]},
      {"from": "n5","to": "n6"}],
    "next_step_decision": {"should_plan_next": false}
  }
}
'''
```

F LONG-TERM MEMORY ENTRY

This section presents a concrete example of *long-term memory entries* maintained by SkillTracer. Long-term memory stores stabilized structural knowledge distilled from repeated executions, including repaired plan graphs, failure patterns, and reusable skill abstractions.

Long-Term Memory: Long-Term Memory Example

```
{
  "task": "5 blue Cronus yoga pants with size 33 arrived, update the
    stock",
  "subgoal": "filter the products grid using the Filters panel to
    locate the blue Cronus yoga pants in size 33",
  "original_node": {
    "purpose": "filter",
    "component_type": "textbox",
    "action": "fill('1026', '-33-Blue')",
    "description": "Fill the 'SKU' filter with text that encodes the
    blue color and size 33 variant (as represented in SKU naming)."},
  "error_evidence": {
    "evidence": "The plan step n6 references placeholder
    identifiers/values (sku_filter_textbox_id,
    blue_size_33_sku_query_text) that are not grounded in the current
    AXTree. In the visible Filters panel, the actual SKU field is
    present as [1031] textbox 'SKU' (empty), so the specified action
    cannot be instantiated/executed as written.",
    "summary": "No executable action was produced because n6's
    target/control id and input value are not tied to any real,
    visible element/value on the page; the correct grounded SKU
    textbox is bid 1031."},
  "stuck_repair": [{
      "repair_tool": {
        "tool": "REPLACE_ACTION",
        "arguments": {
          "node_id": "n6",
          "new_action": "fill('1031', '-33-Blue')"}},
      "insight": "Retarget the fill action to the actually-visible
    SKU textbox (bid 1031) and use a partial SKU string likely to
    match the Blue size-33 variant."}],
  "success_repair": {
    "repair_tool": {
      "tool": "REPLACE_ACTION",
      "arguments": {
        "node_id": "n6",
        "new_action": "fill('1026', '-33')"}},
    "insight": "The detected failure corresponds to a recurrent
    repair pattern: the action is correctly grounded to a visible SKU
    textbox, but the input format is invalid. The page does not
    support jointly encoding multiple attributes in a single SKU
    query, requiring the size and color constraints to be applied
    separately rather than combined."}
}
```

G PROMPT

To ensure reproducibility and clarify the role of language models in our system, we present the concrete prompts used for two critical modules: the *Summarizer* and the *Skill Inducer*. These prompts explicitly define the expected inputs, outputs, and decision scope of each module, enforcing structured reasoning rather than free-form generation.

The *Summarizer* prompt is responsible for analyzing execution evidence and producing a localized repair signal when failures occur. The prompt constrains the model to reason over observable evidence, verification signals, and memory, and to emit a single, well-scoped repair operation.

The *Skill Inducer* prompt governs how validated execution trajectories are abstracted into reusable skills. It specifies how concrete parameters are lifted into semantic placeholders, how plan graphs are normalized, and how updates are applied to existing skills. Together, these prompts define the interface between execution traces and the evolving skill library.

Prompt for Skill Inducer

```
[SYSTEM]
You are the Skill Inducer module in a web-agent system.
Your role:
Given a successfully completed subgoal, abstract or refine exactly
    ONE reusable SKILL
from the executed plan.
You ONLY perform:
- Structural abstraction
- Parameter generalization
- Semantic de-tasking
[INPUTS]
(1) Query: Global task intent, used only as a semantic reference
    frame.
(2) Subgoal: The successfully completed intermediate objective.
(3) Executed Evidence:
A consolidated execution record including:
- The successfully executed plan nodes (in order)
- Grounded action semantics
- Injected skill (if any)
[TASK]
From the executed evidence, produce exactly ONE skill by choosing ONE
     of the following:
1. Abstract a NEW SKILL
   Use when:
   - The executed plan represents a novel interaction pattern, OR
   - The semantic scope of the subgoal diverges from the injected
    skill
2. REFINE the INJECTED SKILL
   Use when:
   - The executed plan closely matches the injected skill structure,
    AND
   - The subgoal semantics align with the injected skill description
[TASK RULES]
- The skill MUST preserve the structural form of a Plan Graph.
- Actions MUST remain symbolic and parameterized
- Do NOT include task-specific entities, identifiers, or values
[OUTPUT]
The output must include:
- id
- name
- description
- plan (nodes, edges, next_step_decision)
Output only the skill.
```

Prompt for Summarizer

```
[SYSTEM]
You are the Repair Agent in a web-agent system.
Your role:
Given a detected execution failure, produce a single localized repair
     for the current plan node.
Do NOT replan globally.
[INPUTS]
(1) Query: Original user task intent.
(2) Subgoal: Current intermediate objective.
(3) Plan: Plan graph for the current subgoal.
(4) Executed Evidence:
A consolidated execution record including:
- Executed plan nodes and symbolic actions (current + previous rounds
    )
- AXTree0 Slice and delta AXTree trajectory aligned with actions
- Active memory (grounded elements generated by memorize())
- Short-term memory (repeated failure patterns)
- Long-term memory (successful past repairs)
(5) Error Signal: A concise description of the detected execution
    failure.
(6) Action Rules: Allowed symbolic actions.
[TASK]
1. From the judgeable nodes implied by the executed evidence, select
    exactly one causal node responsible for the failure.
2. Classify the failure into exactly one category:
   - Intent-Stage Misalignment
   - Interaction Grounding Failure
   - Execution Strategy Failure
   - Ungrounded Information Output
3. Justify the diagnosis using concrete signals from the executed
    evidence.
4. Select exactly one repair tool and propose a localized edit to the
     causal node:
   - REPLACE_PURPOSE
   - REPLACE_COMPONENT_TYPE
   - REPLACE_ACTION
   - REFINE_DESCRIPTION
5. Assess whether the subgoal description was sufficiently precise:
   - If not, output one concise rewrite suggestion.
   - Otherwise output Null.
[OUTPUT]
Produce a concise structured summary containing:
- error_node
- error_type
- evidence (key signals only)
- repair_tool + arguments
- insight (one sentence)
- subgoal_guidance (one sentence or Null)
Output only the summary.
```

H AXTree Structure

The AXTree is a hierarchical representation of the web page derived from the browser's accessibility API. Each node encodes a UI element with its component type, displayed value, and WAI-ARIA role attributes.

AXTree Snapshot: Gitlab - main page

```
... Ignore 3,507 characters.
1,151 characters selected bellow:
[582] main ''
    [591] button 'main', hasPopup='menu', expanded=False
      StaticText 'main'
    [798] list ''
      [799] listitem ''
        [800] link 'a11yproject.com'
    [803] button 'Author', hasPopup='menu', expanded=False
      StaticText 'Author'
    [844] Section ''
      [845] searchbox 'Search by message'
    [847] link 'Commits feed'
      [848] image ''
    [850] list ''
      [851] listitem ''
        StaticText '14 Mar, 2023'
        StaticText ''
        StaticText '2 commits'
      [854] listitem ''
        [855] list ''
          [856] listitem ''
            [858] link "Eric Bailey's avatar"
              [859] image "Eric Bailey's avatar"
            [862] link 'Update...'
            [864] button 'Toggle commit description'
              [865] image ''
            [867] link 'Eric Bailey'
            StaticText 'authored'
            [868] time 'Mar 14, 2023 1:04am UTC'
              StaticText '2 years ago'
            [871] button 'Unverified'
            StaticText 'ed37a2f2'
            [875] button 'Copy commit SHA', live='polite',
    relevant='additions text'
              [876] image ''
            [877] link 'Browse Files'
              [878] image ''
          [879] listitem ''
            [881] link "Eric Bailey's avatar"
              [882] image "Eric Bailey's avatar"
... 23,622 characters remaining.
```

Each AXTree entry corresponds to a single accessible UI node extracted from the browser's accessibility API. An entry is identified by a stable node index and is annotated with its *component type* (e.g., `button`, `textbox`, `listitem`), its *visible text or value* when available, and auxiliary accessibility attributes such as ARIA roles or state flags (e.g., `expanded`, `hasPopup`). The hierarchical indentation encodes parent–child containment relations on the page, allowing structural queries such as component existence, text presence, and relative nesting to be evaluated deterministically during planning and verification.