TASL: ADAPTER LIBRARY FOR EVOLVING TOOLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-augmented language agents extend LLM capabilities by calling external tools, but API evolution can invalidate previously successful calls. Tool calls therefore need to be adapted to match the updated interfaces. Existing methods that rely on model generation to adapt calls still require repeated model inference for new calls affected by the same API change. To reduce repeated adaptation, we introduce **TASL** (Tool-Adapter Synthesis Library), a training-free framework that synthesizes persistent executable adapters for evolving APIs. TASL turns each API change into a reusable adapter, validating it via execution and round-trip checks so that subsequent calls apply the stored program directly, with no adaptation-related model inference. Model parameters remain fixed, and no task-answer supervision is used. Experiments across 7 domains, 11 tools, and 2 evolved API sets in ToolQA-D show that TASL achieves higher accuracy than all evaluated training-free baselines while using fewer adaptation-related LLM calls. Specifically, it improves mean exact-match accuracy by 13.2 and 7.2 percentage points over the strongest baseline with 4B and 14B backbones, respectively. Code will be released after acceptance.

1 INTRODUCTION

Recently, tool-augmented language agents (Shen et al., 2023; Paranjape et al., 2023; Wang et al., 2024b), which combine large language model (LLM) reasoning with external tool execution, have attracted increasing attention. By interacting with external tools, these agents can retrieve documents, query databases, and perform computations beyond text generation, thereby extending the capabilities of LLMs (Karpas et al., 2022; Gao et al., 2023; Nakano et al., 2021). They typically access tools through application programming interfaces (APIs), which define callable operations, argument names, and input formats (Song et al., 2023). However, as deployed tools evolve, changes to these interfaces can invalidate previously successful calls even when the underlying user task remains unchanged (Xavier et al., 2017; Brito et al., 2018). We view compatibility restoration under API evolution as involving 2 steps: identifying the target interface that replaces the outdated API and adapting affected calls to the new interface. In this work, we focus on the latter problem.

Prior work has explored how agents adapt tool calls. ToolEVO (Chen et al., 2025) collects interaction trajectories through Monte Carlo Tree Search and fine-tunes models to adapt tool calls in evolving environments. General agent frameworks also provide mechanisms for incorporating feedback: ReAct (Yao et al., 2023) interleaves reasoning and actions with environmental observations, while Reflexion (Shinn et al., 2023) retains verbal reflections on task feedback to guide subsequent attempts.

When call adaptation relies on model generation, environment feedback or stored verbal reflections can guide the correction, but the model must still perform inference to adapt each new call (Chen et al., 2025; Yao et al., 2023; Shinn et al., 2023). As illustrated on the left of Figure 1, when different tasks encounter the same API change, the LLM may have to make the same repair repeatedly. However, because the updated API interface remains in effect within the environment, the correction learned from one repair can be reused in subsequent tasks. This observation raises a natural question: instead of re-deriving the same correction for every new call, can an agent *synthesize it once, verify it once, and reuse it across future calls*?

We answer this question with **TASL** (Tool-Adapter Synthesis Library), a training-free framework centered on a persistent **Adapter Library**. As illustrated on the right of Figure 1, TASL uses

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

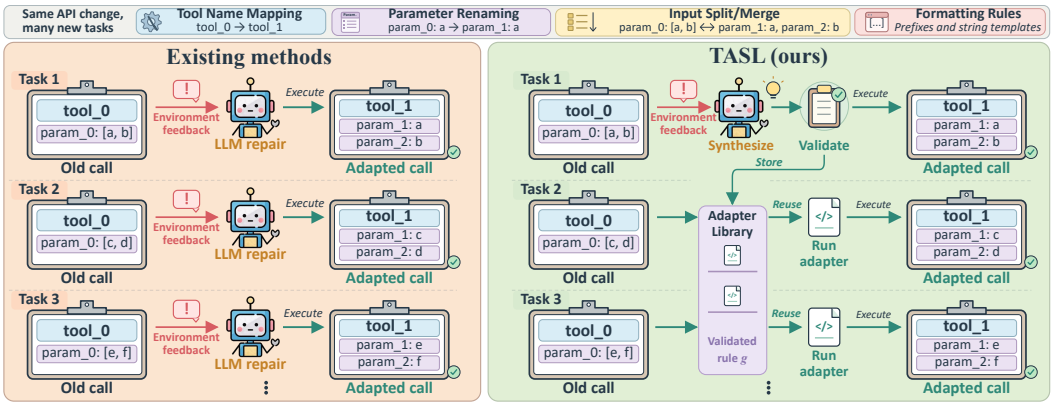

Figure 1: **API evolution and adapter reuse.** API descriptions collected by an agent can become outdated as deployed tools change their names, parameters, or input formats. Existing methods that rely on LLM-based repair repeat adaptation across tasks facing the same change; TASL stores a validated executable adapter and reuses it on subsequent calls.

environment feedback to synthesize an executable adapter and validates it before storing it in the library. Subsequent calls affected by the same API change can directly reuse the stored adapter. The backbone remains fixed, and successful reuse requires no adaptation-related model inference.

Our 3 contributions are:

- We formulate training-free tool-API adaptation as the synthesis and persistent reuse of executable API transformations. This separates one-time adapter acquisition from repeated call execution.
- We introduce a validation protocol that combines environment acceptance with forward–inverse consistency on current and historical inputs. This provides a task-label-free criterion for admitting candidate adapters to the library and detecting recoverability failures on observed inputs.
- We evaluate TASL on ToolQA-D with 4B and 14B backbones, improving mean accuracy over the strongest training-free baseline by 13.2 and 7.2 percentage points while reducing adaptation-related LLM calls. On a tool-fine-tuned backbone, TASL also remains competitive with ToolEVO without evolved-API training.

2 Related Work

Tool learning and API evolution. Tool-augmented language agents interact with external systems to retrieve information, execute computations, and complete tasks. Toolformer learns when and how to call APIs through self-supervised examples (Schick et al., 2023), and ToolLLM scales instruction tuning to a large collection of real-world APIs (Qin et al., 2024). ToolQA evaluates tool-assisted question answering over external data (Zhuang et al., 2023). API-Bank and τ-bench further evaluate executable API use and tool–agent–user interactions (Li et al., 2023; Yao et al., 2025). API changes introduce an additional challenge: Gorilla uses retrieved documentation to accommodate test-time API-document changes (Patil et al., 2024), while ToolEVO learns adaptive behavior through Monte Carlo Tree Search and supervised fine-tuning, and evaluates it on ToolQA-D (Chen et al., 2025). TASL also evaluates on ToolQA-D, addressing API changes through program synthesis and reuse at deployment time while keeping model parameters fixed.

Execution feedback and experience reuse. Execution feedback provides information for revising agent actions and generated programs (Yang et al., 2023; Le et al., 2022). Reflexion retains verbal feedback in episodic memory (Shinn et al., 2023), while CRITIC uses tool feedback to iteratively revise model outputs (Gou et al., 2024). ExpeL extracts transferable insights from experience (Zhao et al., 2024); Agent Workflow Memory induces reusable routines for subsequent tasks (Wang et al., 2025); and SkillTTA synthesizes context-specific skills from retrieved trajectories (Wang et al., 2026). TASL focuses on retaining API adaptation rules acquired from environment feedback for subsequent reuse.

Executable programs and reuse. CodeAct uses executable Python as an agent action representation (Wang et al., 2024c). LATM and CRAFT construct reusable tools (Cai et al., 2024; Yuan et al., 2024), and Voyager maintains a library of executable skills (Wang et al., 2024a). Execution also supplies evidence for code selection: LEVER trains a verifier conditioned on the natural-language input, the generated program, and its execution results (Ni et al., 2023). More recently, TTHE adapts a frozen LLM agent by evolving and persisting its surrounding executable harness using execution-derived feedback (Nie et al., 2026). TASL focuses on synthesizing executable argument transformations for outdated API calls.

3 TASL

3.1 Overview

TASL places a persistent adaptation layer between a frozen agent and its tools to adapt outdated calls to the deployed interfaces. As illustrated in Figure 2, TASL uses target-interface information supplied through environment feedback to query the Adapter Library. On a cache hit, the stored adapter directly transforms the call. On a cache miss, TASL synthesizes a forward transformation and a candidate left inverse. It then validates the pair through execution and round-trip checks and stores it only if both checks pass. Once stored, validated adapters are reused across calls, while unreliable entries are evicted based on execution outcomes. This design enables successful reuse without adaptation-related model inference.

3.2 Adapter synthesis from environment feedback

Since a single API change affects many subsequent calls, capturing the required adaptation as an executable program enables direct reuse without further LLM calls. We therefore synthesize persistent executable adapters from execution feedback.

At runtime, an outdated call fails and the environment returns a deprecation warning with target-interface information. Using both the failed call and this feedback, TASL synthesizes a reusable argument-transformation program that can be applied to subsequent calls affected by the same API change. Let f_0 denote an outdated tool API and $x \in \mathcal{X}_0$ the argument dictionary generated by the agent. TASL extracts the supplied interface information for the target API f_v from environment feedback r. It then queries $\mathrm{Cache}[k]$ using the composite key $k = (f_0, \mathrm{sig}(f_v))$, where $\mathrm{sig}(f_v)$ denotes the target-interface signature. A cache hit reuses the stored adapter. If the feedback is not recognized as an API change, ordinary agent execution continues. Feedback formats and parsing details are provided in Appendix H.

On a cache miss, TASL seeks an executable transformation $g : \mathcal{X}_0 \to \mathcal{X}_v$ that maps the old argument dictionary to one accepted by f_v. This follows the program-synthesis view of constructing executable programs from a specification (Gulwani et al., 2017). Here, TASL uses the target-interface information provided in environment feedback to guide synthesis, without requiring full documentation of the updated API. TASL asks the LLM to produce 2 Python functions in 1 code block: `adapt(args)` implements g, and `inverse_adapt(args)` implements a left inverse h. The prompt specifies a transformation over argument dictionaries with the same calling convention, not merely a replacement dictionary for the current input (Appendix E).

The paired formulation makes input recoverability an explicit synthesis objective. For example, if a call requests 2 columns, an adapter that emits only the first may still be executable. Its inverse, however, must reconstruct the original request. In contrast, a parameter introduced only by the new API need not appear in the reconstructed old dictionary. Thus the target property is recovery of the source input, not a bijection over every dictionary admitted by the new API. Candidate code is evaluated in the restricted Python environment described in Appendix K.

3.3 Validation through execution and round-trip consistency

To decide whether to admit a candidate adapter to the persistent library, TASL checks target-API acceptance of the transformed call and reconstruction of the source inputs. Let $\mathcal{B}_{f_0}$ be the historical

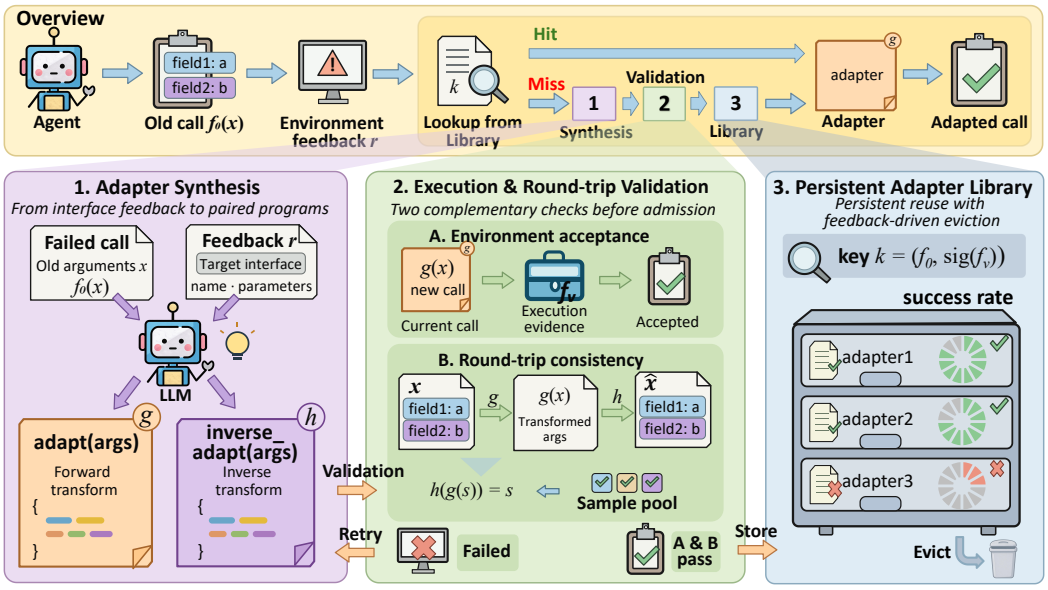

Figure 2: **The TASL framework.** Environment feedback supplies target-interface information for library lookup. On a cache miss, TASL (1) synthesizes paired transformations, (2) checks environment acceptance and round-trip consistency, and (3) stores validated adapters in the persistent library for reuse. Successful reuse requires no adaptation-related LLM inference; execution outcomes guide eviction.

sample pool containing successful (x_i, x_i') pairs, with its capacity specified in Appendix A. Define

$$\mathcal{S}_{f_0}(x) = \{x\} \cup \{x_i : (x_i, x_i') \in \mathcal{B}_{f_0}\}. \tag{1}$$

For a candidate pair (g, h), admission to the library requires

$$V(g, h; x, r) = \underbrace{\mathrm{Accepted}(\mathrm{Env}(f_v, g(x)))}_{\text{environment acceptance}} \wedge \underbrace{\bigwedge_{s \in \mathcal{S}_{f_0}(x)} [h(g(s)) = s]}_{\text{round-trip consistency}}. \tag{2}$$

The first predicate checks whether the adapted call executes under the target API, including its tool name, required arguments, and response format. The second predicate compares the reconstructed and original dictionaries for the current input and the historical inputs stored in the sample pool. Historical samples make admission less dependent on the structure of the first call. A candidate that works for a single field may fail when checked against a prior multi-field request. Because the pool is populated by real calls, these checks do not require manually labeled transformation examples or task answers. This use of an executable invariant is related to property-based testing (Claessen & Hughes, 2000). TASL evaluates the invariant on observed calls rather than randomly generated inputs.

Passing the round-trip consistency check shows that g does not map distinct source inputs in the checked set to the same output. If $g(s_1) = g(s_2)$, applying h gives $s_1 = s_2$, thereby excluding collisions among the checked inputs and preventing the transformation from silently dropping or duplicating observed arguments. Together, environment acceptance and round-trip consistency supply complementary evidence for reuse.

TASL uses execution feedback to iteratively revise candidate programs, an approach also explored in LLM self-debugging (Chen et al., 2024). If code execution, environment acceptance, or reconstruction fails, TASL passes the error and any reconstruction difference back to the model, guiding the next synthesis attempt. The synthesis loop permits at most K candidate attempts. Once both checks pass, the accepted pair is retained. If the budget is exhausted, adaptation fails and the episode terminates without falling back to ordinary ReAct correction. This keeps the evaluated recovery pathway attributable to the synthesis procedure. Algorithm 1 in Appendix D gives the procedure.

3.4 PERSISTENT ADAPTER MEMORY

This component has two coupled purposes. On the one hand, it persists validated adapters so that they can be applied directly to all subsequent calls without re-synthesis. On the other hand, it uses execution feedback to track each entry's reliability and evict entries that fall below a threshold, ensuring that only dependable adapters remain active.

The Adapter Library stores $(f_v, \texttt{source}, g, h, n_{\text{ok}}, n_{\text{fail}})$ in $\text{Cache}[k]$. Here, n_{ok} and n_{fail} denote the numbers of accepted and failed executions for the entry, respectively. Persisting source code makes the acquired rule inspectable. Storing callable functions allows direct application to new argument values. On a cache hit, TASL executes $g(x)$ and dispatches it to f_v. A successful call updates the execution counters and appends its old–new argument pair to the per-tool FIFO sample pool. No LLM is called for adaptation on this successful reuse path. The agent's ordinary task reasoning remains separate.

Persistence creates a need to manage unsuccessful reuse. TASL records accepted and failed executions. It evicts an entry when it has at least 3 observed calls and its empirical success rate falls below τ. Synthesis can then replace an unreliable rule instead of retaining it indefinitely.

Analysis of adaptation-call cost. Let N be the number of affected tool calls and M the number of distinct library keys encountered within the evaluated stream. If each key is successfully covered within K synthesis attempts and remains reusable, at most KM adaptation-related LLM calls are needed, compared with repeatedly calling a model to repair affected calls. More generally,

$$C_{\text{LLM}} \leq KM + R, \tag{3}$$

where C_{LLM} denotes the total number of adaptation-related LLM calls, excluding ordinary task-reasoning calls. R counts additional synthesis calls after initial per-key acquisition budgets, including those caused by eviction or exhausted budgets on earlier calls. Successful cache hits do not increase the number of adaptation-related LLM calls. Lookup is expected $O(1)$ with a hash-based cache. The cost of applying an adapter depends on its code and the size of its inputs. We report repair calls, total LLM calls, and token consumption alongside task accuracy in Section 4.2.

4 EXPERIMENTS

4.1 SETUP

Benchmark. We evaluate on **ToolQA-D** (Chen et al., 2025), a purpose-built and well-controlled benchmark for tool-API evolution in tool-augmented language agents. ToolQA-D offers a broad and diagnostically rich evaluation suite. It spans 7 domains, 11 tools, 3 API sets, and 700 easy and 730 hard questions (1,430 distinct questions). The suite comprises 42 domain–difficulty–version groups, yielding 4,290 question–version episodes per method and backbone. The unchanged Original API (version 0) corresponds to $\mathcal{P}_c$; Evolved-V1 (version 1; V1) corresponds to $\mathcal{P}_{s_{in}}$; and Evolved-V2 (version 2; V2) corresponds to $\mathcal{P}_{s_{OOD}}$. Since TASL does not use these sets for parameter training, we refer to them as V1 and V2. Perturbations cover diverse forms of tool evolution (Appendix B).

Compared methods. We compare TASL with ReAct (Yao et al., 2023), Reflexion (Shinn et al., 2023), and 2 baselines constructed for this study, Prompt-Adapt and Schema-Adapt. ReAct adapts tool calls using environmental feedback in the accumulated context. Reflexion augments the same ReAct loop with an additional LLM call that converts deprecation feedback into a verbal reflection to guide the subsequent call adaptation. Neither ReAct nor Reflexion retains adaptation memory across questions. Prompt-Adapt stores natural-language argument-transformation instructions, whereas Schema-Adapt stores updated JSON tool schemas. Both insert cached artifacts into subsequent prompts and rely on the model to generate new calls. TASL instead stores validated executable adapters that directly transform the arguments of subsequent calls. In ToolQA-D, environment feedback for API adaptation is delivered as deprecation warnings. All methods share the warning parser and tool environment. They have access to the same deprecation warnings, including the target tool name, parameter list, and available argument example. Like TASL, Prompt-Adapt and Schema-Adapt permit up to 3 synthesis attempts per adaptation episode. When a cached artifact fails on a later call, a fresh synthesis episode begins with the same 3-attempt budget. They differ from TASL

Table 1: **Exact-match accuracy on evolved ToolQA-D APIs.** Evolved Avg is the arithmetic mean of the V1 and V2 scores and excludes Original. Best scores within each backbone are bold; second-best scores are underlined. Reference accuracy for the Original API is shown once per backbone; full control results appear in Appendix C.

Method	V1 (%)	V2 (%)	Evolved Avg (%)
Qwen3-4B; Original reference: 44.5%			
ReAct	40.8	12.7	26.8
Reflexion	40.3	22.3	31.3
Prompt-Adapt	<u>41.6</u>	22.7	<u>32.2</u>
Schema-Adapt	30.7	<u>23.6</u>	27.1
TASL	**45.2**	**45.4**	**45.3**
Qwen3-14B; Original reference: 46.2%			
ReAct	40.6	16.1	28.4
Reflexion	39.6	29.9	34.7
Prompt-Adapt	<u>43.1</u>	<u>37.2</u>	<u>40.2</u>
Schema-Adapt	27.1	28.6	27.9
TASL	**47.6**	**47.1**	**47.4**

only in that they cache text (instructions or schemas) that the model re-interprets at inference time, without execution-guided validation, sample pooling, or eviction. The main experiment therefore compares complete adaptation systems; Section 4.4 examines component contributions through ablation. Baseline implementation details and prompts appear in Appendix F.

Evaluation protocol. We use Qwen3-4B-Instruct and Qwen3-14B-Instruct (Yang et al., 2025) as backbones, hereafter abbreviated as Qwen3-4B and Qwen3-14B, respectively. The 2 evolved versions comprise 2,860 episodes; the remaining 1,430 form the unevolved control. The pipeline follows the ToolEVO evaluation setting (Chen et al., 2025). It uses the official external corpora, database and graph tools (Zhuang et al., 2023), and ReAct prompts and demonstrations. For semantic retrieval with sentence embeddings (Reimers & Gurevych, 2019), we use all-mpnet-base-v2 and NumPy cosine similarity. Episodes alternate reasoning, actions, and tool observations until the agent submits a final answer or reaches the step budget (at most 15; domain-specific limits are in Appendix A). Answers are scored by exact match after numeric normalization. Both backbones use greedy decoding and are served through vLLM (Kwon et al., 2023), with prefix caching enabled. For TASL, Prompt-Adapt, and Schema-Adapt, adaptation state persists across questions within each domain–difficulty–version group and is reset between groups. ReAct and Reflexion do not share adaptation state across questions. Questions follow the original benchmark order. No task-answer labels are used to synthesize or validate adapters.

4.2 Main Results and Analysis

To evaluate task performance under API evolution, we compare TASL with the 4 baselines on V1 and V2 using both backbones, with Original as an unevolved reference. Per-version accuracy averages domain–difficulty groups with equal weight.

As shown in Table 1, TASL achieves the highest accuracy on both V1 and V2 with both backbones. Its evolved mean accuracy reaches 45.3% at 4B and 47.4% at 14B. These scores exceed those of Prompt-Adapt, the strongest baseline by evolved mean, by 13.2 and 7.2 percentage points, respectively. TASL maintains similar accuracy across the 2 evolved versions, with its largest advantage on V2. On this version, it achieves 45.4% at 4B and 47.1% at 14B, outperforming Schema-Adapt and Prompt-Adapt, the strongest baselines at the respective scales, by 21.8 and 9.9 percentage points. Under Original, all methods use the same ordinary execution pathway for a given backbone and obtain 44.5% and 46.2%, respectively. Full domain-level and control results appear in Appendix C.

These results indicate that the complete synthesis–validation–reuse design improves API adaptation in the evaluated settings. The larger gains on V2 are consistent with the ability of executable transformations to repeatedly apply structural and formatting rules. Once an adapter passes validation,

Table 2: **Adaptation statistics on** V1 + V2**.** Counts are aggregated over $N = 2{,}860$ question–version episodes per method. Repair calls follow the definitions in the text. Call counts refer to LLM calls.

Method	LLM calls		Tokens (M)	
	Repair	Total	Input	Output
Qwen3-4B				
ReAct	13,132	23,237	64.0	2.88
Reflexion	17,051	27,599	67.0	3.49
Prompt-Adapt	394	12,650	34.8	1.69
Schema-Adapt	263	13,012	36.0	1.68
TASL	**244**	**10,604**	**27.3**	**1.37**
Qwen3-14B				
ReAct	10,288	19,718	53.3	2.47
Reflexion	17,189	27,113	66.0	2.86
Prompt-Adapt	365	12,721	35.1	1.67
Schema-Adapt	230	11,323	31.6	1.45
TASL	**221**	**9,738**	**25.9**	**1.26**

subsequent inputs can directly use the stored transformation. Section 4.4 further examines these mechanisms through ablation.

The effect of model scale also depends on the adaptation system. From 4B to 14B, Prompt-Adapt's evolved mean increases from 32.2% to 40.2%, whereas Schema-Adapt's remains roughly flat (27.1% to 27.9%). Larger models therefore do not necessarily use cached schemas more effectively. In evolved environments, the interface information in deprecation warnings and the call adaptation process may alter the agent's interaction trajectories. This may help the agent correctly answer some questions that it fails to answer under Original. Original accuracy therefore serves as a reference, but is not a strict upper bound on accuracy in evolved environments.

Beyond task performance, a second motivation for persistent reuse is reducing repeated inference cost. To examine this, we measure repair calls, total LLM calls, and input/output token counts for ReAct, Reflexion, Prompt-Adapt, Schema-Adapt, and TASL over the same 2,860 evolved-version episodes. For caching methods, repair calls include artifact generation and revision, including failed candidates and, for TASL, re-synthesis after eviction. For ReAct, call classification identifies in-context correction calls. For Reflexion, repair calls include reflection generation and in-context call correction. Repair-call counts exclude ordinary task reasoning and subsequent interpretation of cached text.

As shown in Table 2, TASL records the lowest repair-call counts, total LLM calls, and input/output token usage among the evaluated methods at both model scales. It uses 244 and 221 repair calls at 4B and 14B, respectively, approximately 1/54 and 1/47 of ReAct's counts. Its total LLM calls are 10,604 and 9,738, also below those of the other caching methods, Prompt-Adapt and Schema-Adapt. Input/output token usage is 27.3M/1.37M at 4B and 25.9M/1.26M at 14B.

These results show that, over the evaluated call sequences, TASL concentrates adaptation effort into relatively few synthesis and revision calls and serves subsequent calls through persistent reuse. Successful cache hits execute stored transformations without adaptation-related model inference.

4.3 Comparison with training-based adaptation

We further investigate whether TASL remains effective on a backbone fine-tuned for tool use and how it compares with training-based adaptation. Using Qwen3-4B on ToolQA-D, we construct Tool-SFT by fine-tuning on successful trajectories containing only valid tool calls in the Original environment. We evaluate Tool-SFT with ReAct (Yao et al., 2023) in V1 and V2, allowing it to adjust its calls in response to deprecation warnings. We then evaluate the same checkpoint with TASL. We compare both settings with ToolEVO (Chen et al., 2025), which is trained from the same initial backbone on successful interactions in V1. The 2 training pipelines use matched training questions and supervision budgets, and all methods share the evaluation environment and backbone

Table 3: **Comparison with training-based adaptation on Qwen3-4B.** Tool-SFT is fine-tuned on valid-call trajectories in the Original environment; ToolEVO is fine-tuned on successful V1 interactions from the same backbone, under matched supervision budgets. V1+V2 repair calls and Total LLM calls are both summed over the V1+V2 episodes ($N = 2{,}860$).

Method	Original (%)	V1 (%)	V2 (%)	Evolved Avg (%)	Repair calls	Total LLM calls
ToolEVO	55.9	55.0	51.8	53.4	4,310	12,133
Tool-SFT + ReAct	**57.6**	31.5	32.7	32.1	11,238	18,393
Tool-SFT + TASL	**57.6**	**55.9**	**56.2**	**56.0**	**287**	**7,617**

Table 4: **Component ablation (Qwen3-4B).** Exact-match accuracy on V1/V2 and their mean; Δ gives the reported change relative to the full-system row in this ablation.

Variant	V1 (%)	V2 (%)	Mean (%)	ΔV1 (pp)	ΔV2 (pp)	Synth. calls
Full TASL	**45.2**	**45.4**	**45.3**	—	—	244
w/o round-trip	40.6	41.9	41.2	-4.6	-3.5	191
w/o sample pool	42.7	42.0	42.3	-2.5	-3.4	227
w/o retry (K=1)	41.3	41.9	41.6	-3.9	-3.5	285
w/o cache	31.3	39.9	35.6	-13.9	-5.5	9,088

prompting format. All 3 methods are evaluated on Original, V1, and V2. Tool-SFT receives no training trajectories from evolved environments; TASL acquires adaptation rules from execution feedback at inference time. As shown in Table 3, adding TASL raises Tool-SFT accuracy from 31.5%/32.7% to 55.9%/56.2% on V1/V2. With only 287 repair calls (roughly $1/15$ of ToolEVO), Tool-SFT with TASL exceeds ToolEVO by 2.6 pp on Evolved Avg. These results show that TASL recovers evolved-environment performance on a tool-trained backbone at inference time without additional evolved-API training. Detailed training settings and per-domain results are provided in Appendices I and J.

4.4 Component ablation

To attribute TASL's gains to individual mechanisms, we ablate one component at a time on Qwen3-4B. For each variant, we re-run the complete V1+V2 suite (2,860 question–version episodes per variant), holding all other components and the evaluation pipeline fixed. We evaluate 4 variants: (i) **w/o round-trip**—the inverse function is no longer required and the round-trip check is skipped (only environment acceptance remains); (ii) **w/o sample pool**—round-trip is checked only on the current call; (iii) **w/o retry**—synthesis gets 1 attempt (K=1) with no error feedback; (iv) **w/o cache**—verified adapters are never persisted, so every deprecated-tool occurrence re-synthesizes from scratch.

Table 4 shows a decrease in mean accuracy for each of the 4 reported removals. Persistent caching has the largest effect: without it, the reported mean falls from 45.3% to 35.6%, and synthesis calls rise from 244 to 9,088 (approximately $37\times$). Repeated synthesis exposes later questions to acquisition failures that a reusable adapter would avoid. Round-trip validation, historical samples, and retries also improve the reported mean.

4.5 Question-order sensitivity

TASL persists adapters across questions within a domain–difficulty–version group, so its accuracy may be path-dependent. Adapters synthesized for a tool in earlier questions can benefit later questions. To test whether persistent caching makes results depend on a particular ordering, we shuffle the question order within each group using 3 random seeds (101, 202, 303). We then re-run the full evaluation on V1 and V2 for TASL, Prompt-Adapt, and Schema-Adapt on Qwen3-4B. Neither ReAct nor Reflexion transfers adaptation memory across questions, so their results are independent of question order.

Table 5: **Question-order sensitivity (Qwen3-4B).** Mean±std exact-match accuracy across 3 random shuffles (seeds 101/202/303) of question order within each domain–difficulty group.

Method	V1 (%)	V2 (%)
Prompt-Adapt	41.7±0.5	22.7±0.2
Schema-Adapt	30.6±0.8	23.6±0.1
TASL	**45.2±0.3**	**45.4±0.2**

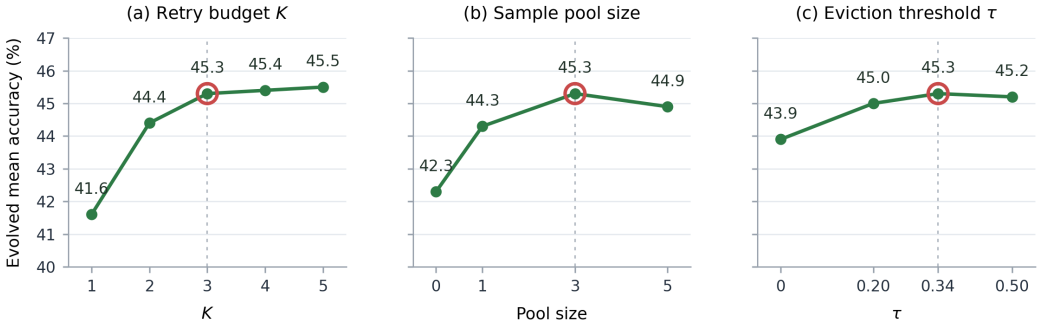

Figure 3: **Hyperparameter sensitivity on Qwen3-4B.** Evolved mean accuracy as we vary (a) retry budget K, (b) sample pool size, and (c) eviction threshold τ. The red circle marks the configuration used in the main experiment (K=3, pool = 3, τ=0.34).

Table 5 reports mean±std across the 3 shuffles. TASL varies by at most 0.3 percentage points (std) on either version (V1: $45.2\pm0.3\%$; V2: $45.4\pm0.2\%$), and no caching method exceeds 0.8 pp std. These results indicate that TASL's accuracy is relatively stable across different question orderings.

4.6 Hyperparameter sensitivity

We evaluate the sensitivity of 3 hyperparameters on Qwen3-4B under V1 and V2, holding all other settings fixed. Figure 3 reports the evolved mean. The results for the retry budget K (subplot a) show that 1 attempt is clearly insufficient (41.6%), while $K \geq 3$ yields similar accuracy (45.3–45.5%). We set K=3, where recovery is near saturation, to avoid extra cost. For sample pool size (subplot b), accuracy improves from 42.3% at pool = 0 to 45.3% at pool = 3, with pool = 5 slightly lower. We use pool = 3 to capture a small set of diverse inputs without growing indefinitely. For the eviction threshold τ (subplot c), accuracy is stable across $[0.20, 0.50]$ (45.0–45.3%), while τ=0 (no eviction) drops to 43.9%. We set τ=0.34 as a moderate value within the flat region.

Overall, accuracy is stable across a broad range of reasonable settings for all three hyperparameters, and the selected values fall within these flat regions rather than at isolated peaks. The main results therefore do not depend on fine-grained hyperparameter tuning.

5 Conclusion

We propose TASL, a training-free adaptation framework for evolving tool APIs. Its core idea is to retain adaptation rules acquired through call repair as persistent, reusable executable adapters, allowing subsequent calls to directly apply existing transformations. TASL synthesizes adapters from execution feedback. It uses target-API execution to check call acceptance and round-trip consistency checks on current and historical inputs to detect information loss. On ToolQA-D, TASL improves average accuracy across evolved API versions over the strongest evaluated training-free baseline by 13.2 and 7.2 percentage points with 4B and 14B backbones, respectively. Successful reuse avoids repeated adaptation-related model inference. These results demonstrate the value of retaining and reusing validated executable adaptation knowledge to improve agents' ability to handle API evolution.

AI USE STATEMENT

In this work, we used generative AI tools for assisting with translation and implementing methods. We have not used generative AI tools for supporting qualitative and thematic data analysis, interpreting results, helping develop theoretical models or conceptual frameworks, or designing or providing feedback on research methodology or experiments, and generating synthetic data sets, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, proposing or refining hypotheses, and cleaning and reformatting datasets are not applicable to this work. Additionally, we used generative AI tools for identifying relevant literature and creating or editing software code. We have reviewed all AI-assisted work. We reviewed and tested the generated or modified code. We also checked the analyses and interpretations against the experimental records. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 255–265, 2018. doi: 10.1109/SANER.2018.8330214. URL https://doi.org/10.1109/SANER.2018.8330214.

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*, volume 2024, pp. 54067–54089, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/ed91353f700d113e5d848c7e04a858b0-Abstract-Conference.html.

Guoxin Chen, Zhong Zhang, Xin Cong, Fangda Guo, Yesai Wu, Yankai Lin, Wenzheng Feng, and Yasheng Wang. Learning evolving tools for large language models. In *The Thirteenth International Conference on Learning Representations*, volume 2025, pp. 87140–87169, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/d8d1d325f822e07d16866780cc23deb2-Abstract-Conference.html.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 268–279, 2000. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 10764–10799, 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/fef126561bbf9d4467dbb8d27334b8fe-Abstract-Conference.html.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2):1–119, 2017. doi: 10.1561/2500000010. URL https://www.microsoft.com/en-us/research/publication/program-synthesis/.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics:*

ACL 2024, pp. 11143–11156, 2024. doi: 10.18653/v1/2024.findings-acl.664. URL https://aclanthology.org/2024.findings-acl.664/.

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. MRKL systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*, 2022. doi: 10.48550/arXiv.2205.00445. URL https://arxiv.org/abs/2205.00445.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328, 2022. doi: 10.52202/068431-1549. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, 2023. doi: 10.18653/v1/2023.emnlp-main.187. URL https://aclanthology.org/2023.emnlp-main.187/.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. WebGPT: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021. doi: 10.48550/arXiv.2112.09332. URL https://arxiv.org/abs/2112.09332.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 26106–26128, 2023. URL https://proceedings.mlr.press/v202/ni23b.html.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. TTHE: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026. doi: 10.48550/arXiv.2607.08124. URL https://arxiv.org/abs/2607.08124.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. ART: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023. doi: 10.48550/arXiv.2303.09014. URL https://arxiv.org/abs/2303.09014.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 126544–126565, 2024. doi: 10.52202/079017-4020. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/28e50ee5b72e90b50e7196fde8ea260e-Abstract-Conference.html.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410/.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551, 2023. doi: 10.52202/075280-2997. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In *Advances in Neural Information Processing Systems*, volume 36, pp. 38154–38180, 2023. doi: 10.52202/075280-1657. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/77c33e6a367922d003ff102ffb92b658-Abstract-Conference.html.

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. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. RestGPT: Connecting large language models with real-world RESTful APIs. *arXiv preprint arXiv:2306.06624*, 2023. doi: 10.48550/arXiv.2306.06624. URL https://arxiv.org/abs/2306.06624.

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. *Transactions on Machine Learning Research*, 2024a. URL https://openreview.net/forum?id=ehfRiF0R3a.

Jingxing Wang, Chenyu Zhou, Zhihui Fu, Jun Wang, Weiwen Liu, Weinan Zhang, and Jianghao Lin. Skills on the fly: Test-time adaptive skill synthesis for LLM agents. *arXiv preprint arXiv:2605.16986*, 2026. doi: 10.48550/arXiv.2605.16986. URL https://arxiv.org/abs/2605.16986.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6): 186345, 2024b. doi: 10.1007/s11704-024-40231-1. URL https://link.springer.com/article/10.1007/s11704-024-40231-1.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 50208–50232, 2024c. URL https://proceedings.mlr.press/v235/wang24h.html.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 63897–63911, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 138–147, 2017. doi: 10.1109/SANER.2017.7884616. URL https://doi.org/10.1109/SANER.2017.7884616.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu,

Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. doi: 10.48550/arXiv.2505.09388. URL https://arxiv.org/abs/2505.09388.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 23826–23854, 2023. doi: 10.52202/075280-1035. URL https://proceedings.nips.cc/paper_files/paper/2023/hash/4b175d846fb008d540d233c188379ff9-Abstract-Datasets_and_Benchmarks.html.

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://openreview.net/forum?id=WE_vluYUL-X.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, volume 2025, pp. 9965–10017, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/1b126cc38b8638e07bef37e7b2bb72bf-Abstract-Conference.html.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations*, volume 2024, pp. 40097–40125, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/af31604708f3e44b4de9fdfa6dcaa9d1-Abstract-Conference.html.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. ToolQA: A dataset for LLM question answering with external tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 50117–50143, 2023. doi: 10.52202/075280-2180. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/9cb2a7495900f8b602cb10159246a016-Abstract-Datasets_and_Benchmarks.html.

A HYPERPARAMETERS AND IMPLEMENTATION DETAILS

A.1 LLM INFERENCE CONFIGURATION

Table 6: LLM inference configuration.

Parameter	Value
Model (4B)	Qwen3-4B-Instruct
Model (14B)	Qwen3-14B-Instruct
Serving framework	vLLM (prefix caching enabled)
Decoding	Greedy (temperature $= 0$)
`max_model_len` (4B / 14B)	8192 / 16384
`max_tokens` (ReAct step)	256
`max_tokens` (synthesis call)	512
`max_tokens` (Reflexion reflection)	512
GPU memory utilization	0.75

A.2 TASL HYPERPARAMETERS

Table 7: TASL hyperparameters.

Parameter	Value	Notes
Max candidate attempts K	3	Includes the initial attempt; feedback on retries
Sample Pool size	3	FIFO of recent (old, new) pairs per old API
Eviction threshold	34%	After $\geq$3 calls with success $<$34%, evict entry
Sandbox bans	`import`, `open`, `eval`, `exec`, `__import__`	No file I/O or dynamic import
Code extraction	fenced `python` block regex	Dual-function block required
Sandbox timeout	none (synchronous)	Adapters are pure dict transforms

A.3 EVALUATION HYPERPARAMETERS

Table 8: Evaluation configuration.

Parameter	Value
Domains	agenda, scirex, flights, coffee, airbnb, yelp, dblp (7)
Difficulties	easy, hard
API versions	0 (Original), 1 (V1), 2 (V2)
Questions per group	$\sim$100 (coffee-hard: 130)
Distinct test questions	700 easy + 730 hard = 1,430
Total evaluation episodes	42 groups, 4,290 episodes / method / model
ReAct max steps	15 (agenda: 8, scirex: 6)
Maximum reflections per question, $R_{\max}$	3
Trajectory window for reflection	Up to the 6 most recent steps
Reflection calls consume the agent action budget	No
Retrieval model	all-mpnet-base-v2
Retrieval top-k	5 (oracle: 3)

Cache scope. For TASL, Prompt-Adapt, and Schema-Adapt, adaptation state persists across questions within each domain–difficulty–version group and is reset between groups. ReAct and Reflexion do not share adaptation state across questions. Easy/hard never share state, and versions 1/2 are treated as independent evolution eras.

B TOOL-EVOLUTION PERTURBATION MAPPING

Table 9 reproduces the old→new signature mapping defined by ToolQA-D's `api_vary.py`.

Table 9: Per-tool signature changes under V1 (version 1) and V2 (version 2).

Tool	V1 change	V2 change
RetrieveAgenda	renamed → `Fetch_Agenda_Data`; `keyword`→`Query`; new param `return_num`	renamed → `Call_Retrieve_On_Agenda`; `keyword`→`searchTerm`; new param `passage_num`
RetrieveScirex	renamed → `FetchScirexData`; `keyword`→`QueryText`	renamed → `CallRetrieveOnScirex`; `keyword`→`queryKeyword`
LoadDB	renamed → `InitializeDatabase`; `DBName`→`DatabaseName`	renamed → `Init_DB`; `DBName`→`databaseIdentifier`
FilterDB	renamed → `Apply_Database_Filters`; `condition`→`condition1`	renamed → `DoFilter_OnDatabase`; `condition`→`filterCriteria1`, `filterCriteria2, ...` (numbered criteria)
GetValue	renamed → `FetchValue_ByKey`; `column_name`→`column`; new param `ReturnResult`	renamed → `Extract_Value`; `column_name`→`fieldName1`, `fieldName2, ...`; new param `ReturnValue`
SQLInterpreter	renamed → `ExecuteSQLQuery`; `SQL`→`SQLCommand`; value prefixed with `"The SQL code is as follows: "`	renamed → `ProcessSQLQuery`; `SQL`→`SQL_Query`; value must start with `"This is the SQL code: "`
PythonInterpreter	renamed → `Execute_Python_Script`; `Python`→`PythonCode`; value prefixed with `"The Python code is as follows: "`	renamed → `Process_Python_Code`; `Python`→`python_execute_Code`; value must start with `"This is the Python code: "`
LoadGraph	renamed → `InitializeGraphData`; `GraphName`→`Graph_Name`	renamed → `Import_Graph`; `GraphName`→`Graph`
NodeCheck	renamed → `ValidateGraphNode`; `GraphName`→`Graph_Name`; `Node`→`graphNode`	renamed → `Inspect_TheNodes`; `GraphName`→`Graph`; `Node`→`Vertex`
NeighbourCheck	renamed → `Verify_NeighbourNodes`; `GraphName`→`Graph_Name`; `Node`→`graphNode`; new param `ReturnResult`	renamed → `Get_NeighbourList`; `GraphName`→`Graph`; `Node`→`Vertex`; new param `ReturnValue`
EdgeCheck	renamed → `ValidateGraphEdge`; `GraphName`→`Graph_Name`; `Node1, Node2` merged into template `NodeInfos="FirstNode[x], SecondNode[y]"`	renamed → `Inspect_TheEdges`; `GraphName, Node1, Node2` merged into template `CheckInfos="Graph[g], Vertex1[x], Vertex2[y]"`

C FULL PER-GROUP RESULTS

Tables 10–12 report exact-match accuracy for every domain×difficulty group. Domains are abbreviated (age=agenda, sci=scirex, fli=flights, cof=coffee, air=airbnb, yel=yelp, dbl=dblp; e=easy, h=hard).

Table 10: Original (unevolved control) per-group accuracy (AVG.e/AVG.h: macro-averages). All 5 methods share the same results within each backbone.

Method	age.e	age.h	sci.e	sci.h	fli.e	fli.h	cof.e	cof.h	air.e	air.h	yel.e	yel.h	dbl.e	dbl.h	AVG.e	AVG.h
Qwen3-4B																
All methods	**0.43**	**0.00**	**0.89**	**0.20**	**0.40**	**0.27**	**0.99**	**0.31**	**0.83**	**0.18**	**0.75**	**0.45**	**0.36**	**0.17**	**0.664**	**0.225**
Qwen3-14B																
All methods	**0.72**	**0.21**	**0.91**	**0.27**	**0.49**	**0.08**	**0.98**	**0.27**	**0.56**	**0.07**	**0.75**	**0.62**	**0.36**	**0.18**	**0.681**	**0.243**

Table 11: V1 (Evolved-V1) per-group accuracy (AVG.e/AVG.h: macro-averages).

Method	age.e	age.h	sci.e	sci.h	fli.e	fli.h	cof.e	cof.h	air.e	air.h	yel.e	yel.h	dbl.e	dbl.h	AVG.e	AVG.h
Qwen3-4B																
ReAct	0.48	**0.15**	**0.89**	**0.26**	0.32	0.09	0.80	0.05	0.92	0.22	**0.80**	0.26	**0.25**	**0.23**	0.637	0.179
Reflexion	0.43	0.00	0.85	**0.26**	0.30	0.24	0.84	0.24	**0.93**	0.19	0.54	0.42	0.23	0.18	0.589	0.218
Prompt-Adapt	0.45	0.10	0.86	0.18	0.30	0.20	**0.86**	**0.27**	0.92	0.22	0.75	0.45	0.18	0.08	0.617	0.214
Schema-Adapt	**0.53**	0.01	0.88	0.23	0.23	0.15	0.63	0.21	0.16	0.20	0.22	0.39	0.24	0.21	0.413	0.200
TASL	0.50	**0.15**	0.88	0.20	**0.47**	**0.26**	**0.86**	**0.27**	0.91	**0.24**	**0.80**	**0.48**	0.18	0.13	**0.657**	**0.247**
Qwen3-14B																
ReAct	**0.67**	0.21	0.91	0.26	0.25	0.01	0.82	0.01	**0.92**	0.18	0.61	0.34	0.20	0.30	0.626	0.187
Reflexion	0.13	0.20	0.89	0.23	0.30	0.06	0.81	0.05	0.90	0.20	0.60	0.56	**0.27**	**0.34**	0.557	0.235
Prompt-Adapt	0.65	**0.28**	0.90	0.25	0.40	0.12	**0.87**	**0.17**	0.90	0.26	0.58	0.45	0.17	0.03	0.639	0.223
Schema-Adapt	0.65	0.21	**0.93**	0.25	0.00	0.05	0.03	0.00	0.01	0.21	0.58	0.46	0.20	0.21	0.343	0.199
TASL	**0.67**	**0.28**	0.91	**0.27**	**0.52**	**0.15**	0.86	0.16	0.90	**0.30**	**0.63**	**0.63**	0.24	0.14	**0.676**	**0.276**

Table 12: V2 (Evolved-V2) per-group accuracy (AVG.e/AVG.h: macro-averages).

Method	age.e	age.h	sci.e	sci.h	fli.e	fli.h	cof.e	cof.h	air.e	air.h	yel.e	yel.h	dbl.e	dbl.h	AVG.e	AVG.h
Qwen3-4B																
ReAct	0.10	0.02	0.30	0.08	0.15	0.05	0.20	0.08	0.15	0.03	0.10	0.05	**0.30**	**0.17**	0.186	0.069
Reflexion	0.40	0.08	0.60	0.20	0.08	0.03	0.35	0.15	0.50	0.20	0.25	0.15	0.08	0.06	0.323	0.124
Prompt-Adapt	**0.48**	0.10	0.10	0.02	0.28	0.18	0.15	0.22	0.35	0.05	0.62	0.40	0.18	0.05	0.309	0.146
Schema-Adapt	0.15	0.03	0.40	0.12	0.30	0.18	0.60	0.25	0.55	0.18	0.18	0.15	0.13	0.08	0.330	0.141
TASL	**0.48**	**0.15**	**0.88**	**0.22**	**0.48**	**0.24**	**0.86**	**0.26**	**0.91**	**0.24**	**0.81**	**0.48**	0.19	0.15	**0.659**	**0.249**
Qwen3-14B																
ReAct	0.25	0.05	0.35	0.08	0.10	0.02	0.35	**0.15**	0.20	0.05	0.35	0.15	0.10	0.06	0.243	0.080
Reflexion	0.55	**0.25**	0.65	0.22	0.20	0.15	0.45	0.05	0.35	0.18	0.40	0.50	0.10	0.13	0.386	0.212
Prompt-Adapt	0.70	0.20	0.30	0.10	**0.55**	**0.18**	0.85	0.07	0.50	0.25	0.55	0.55	**0.30**	0.11	0.536	0.209
Schema-Adapt	0.67	0.08	0.55	0.10	0.22	0.05	0.48	0.02	0.50	0.13	0.35	0.32	0.23	**0.30**	0.429	0.143
TASL	**0.71**	**0.25**	**0.91**	**0.27**	0.52	0.16	**0.86**	0.07	**0.90**	**0.30**	**0.63**	**0.63**	0.22	0.17	**0.679**	**0.264**

D COMPLETE ADAPTATION ALGORITHM

Algorithm 1 TASL adaptation for an outdated call $f_0(x)$

Require: environment feedback r, executor Env, attempt budget $K = 3$, FIFO pools of size 3, threshold $\tau = 0.34$

$(f_v, \textit{plist}, e_v) \leftarrow \text{ExtractInterface}(r)$; $k \leftarrow (f_0, \text{sig}(f_v))$; *diagnostics* $\leftarrow$ `""`
if Cache$[k]$ exists **then**
  $x' \leftarrow \text{Cache}[k].g(x)$; *obs* $\leftarrow \text{Env}(f_v, x')$; update counters
  if *obs* accepted **then**
    $\mathcal{B}_{f_0}.\text{push}((x, x'))$; **return** *obs* ▷ no adaptation LLM call
  else
    if $n_{\text{ok}} + n_{\text{fail}} \geq 3$ **and** $\frac{n_{\text{ok}}}{n_{\text{ok}}+n_{\text{fail}}} < \tau$ **then**
      evict Cache$[k]$
    else
      return failure ▷ retain entry; no re-synthesis
    end if
  end if
end if
for $j = 1$ **to** K **do**
  $(g, h) \leftarrow \text{LLM.synthesize}(f_0, x, r, \textit{diagnostics})$
  $x' \leftarrow g(x)$ in restricted execution; *obs* $\leftarrow \text{Env}(f_v, x')$
  if *obs* accepted **and** $h(g(s)) = s$ for all $s \in \mathcal{S}_{f_0}(x)$ **then**
    store $(f_v, \texttt{source}, g, h, n_{\text{ok}}, n_{\text{fail}})$ under k; $\mathcal{B}_{f_0}.\text{push}((x, x'))$
    return *obs*
  else
    diagnostics $\leftarrow$ execution error or reconstruction difference
  end if
end for
return failure ▷ no fallback to ReAct correction

E SYNTHESIS PROMPT TEMPLATE

Below is the exact system+user template used on a cache miss in our ToolQA-D experiments. In this setting, environment feedback is provided as a deprecation warning (illustrated for `FilterDB→Apply_Database_Filters`).

```
You are an expert Python programmer helping a tool-using AI agent survive
a breaking API change. You will be given: (1) the OLD-style call the agent
just attempted, (2) the deprecation error message, which names the NEW tool
and shows ONE example of its NEW argument format.
Write TWO Python functions inside a single code block:

1. `def adapt(args: dict) -> dict:` -- converts an arbitrary OLD-style
   argument dict (same keys as the OLD call, possibly different values)
   into the exact NEW-style argument dict, matching the key names, any
   numbering, merged/templated strings, and required literal text prefixes
   shown in the example EXACTLY.

2. `def inverse_adapt(args: dict) -> dict:` -- the inverse of adapt: takes
   a NEW-style argument dict and converts it BACK to the OLD-style argument
   dict. This function must satisfy:
       inverse_adapt(adapt(old_args)) == old_args
   for ANY valid old_args. This ensures no parameter values are lost during
   conversion. For example, if adapt splits "price, service" into
   column1="price", column2="service", then inverse_adapt must rejoin them
   back into column_name="price, service".
```

```
Return ONLY a fenced ```python code block containing both functions, no
other text.

OLD call attempted: FilterDB({"condition": "Date=2001-12-06"})
Deprecation error received: Error: FilterDB[condition] is deprecated and no
longer available. Use Apply_Database_Filters[condition1] instead. param
example: {"condition1": "Date=2001-12-06"} instead.

Write the `adapt` and `inverse_adapt` functions now.
```

Each attempt retains the original call $f_0(x)$ and deprecation warning r. *diagnostics* is initially empty and contains the latest execution error or reconstruction difference on retries. On retry, the following is appended: `Your previous attempt was wrong: <error incl. round-trip diff>. Fix it.`

F Compared methods and baseline prompts

All 5 methods use the same frozen backbone, ReAct interaction loop, and tool environment, with access to the same deprecation warnings. They differ in how feedback is used to adapt calls and in whether and how adaptation artifacts are retained and reused.

ReAct. This memory-free control creates no adaptation artifacts. The raw warning text is appended to the agent scratchpad, and the model repairs the failed call using the accumulated context within the 15-step episode budget. No cross-question cache is maintained. When a later question calls the same outdated tool, the model derives the correction again from scratch.

Reflexion. We adapt Reflexion to API-deprecation feedback within the current episode. Upon a deprecation warning, an additional LLM call generates a verbal reflection from the current question, recent interaction history, and the raw warning. The reflection replaces the raw warning in the scratchpad and guides the subsequent call adaptation. Reflection memory is retained only within the current question and is not shared across questions.

Prompt-Adapt. When a source tool f_0 first triggers a deprecation warning, the model is prompted to generate a short natural-language paragraph describing how to transform old-style arguments into the new call. The paragraph is cached under f_0. On each subsequent deprecation of the same source tool, the paragraph is prepended to the prompt. The model interprets it to produce new arguments. This interpretation is performed by the same LLM call that generates the next action. Each adaptation episode permits at most 3 attempts ($K = 3$).

Schema-Adapt. This method follows Prompt-Adapt, except that the cached artifact is a rewritten JSON tool schema (the new name and parameter list with 1 example), rather than a natural-language paragraph. On a cache hit, the cached schema is injected, and the model interprets it again to generate the new call. Each adaptation episode likewise permits at most 3 attempts ($K = 3$).

TASL. This is our proposed method. Rather than caching text that the model must reinterpret on every cache hit, TASL synthesizes an executable forward transformation and its inverse. It validates the pair through target-API execution and round-trip checks on current and pooled historical inputs, then stores the callable programs. On a cache hit, the stored program is applied directly to the arguments without any adaptation-related model inference. Validation, sample pooling, up to 3 candidate attempts, and eviction are described in Section 3.

Reflexion reflection prompts. Reflection is triggered only by the first warning for an old API within a question, provided that the reflection budget has not been exhausted. Each old API can trigger at most 1 reflection; repeated warnings for the same old API are passed through unchanged. Reflections, the reflection counter, and the record of previously reflected-on APIs are reset between questions. Reflection uses environmental deprecation feedback without task-answer correctness labels. Reflection budgets and the trajectory-window configuration are specified in Appendix A.

Reflexion uses the following system and user prompt templates to generate reflections. In the user template, `{question}` is the current question. `{recent_trajectory}` contains the recent Thought/Action/Observation steps within the configured trajectory window. `{warning}` is the raw deprecation warning returned by the environment.

System prompt.

```
You are a tool-using AI agent reflecting on a failed tool call.
You will be shown: the question you are solving, the sequence of
Thought/Action/Observation steps taken so far, and the error
observation returned by the environment (a deprecation warning that
names the NEW tool and shows ONE example of its new argument format).

Write a short reflection (3-6 sentences) that:

1. States what specifically went wrong in plain words (which tool was
   deprecated, what the environment asked for instead).
2. Extracts the EXACT new tool name and the new argument names from
   the warning's param example, including any required literal text
   prefixes (e.g. 'The SQL code is as follows:') and whether
   comma-separated values must be split into separate numbered
   parameters.
3. Gives a concrete corrected version of the failed call.

Be specific and concise; do not restate the whole trajectory.
Write ONLY the reflection text, no markdown formatting, no code blocks.
```

User prompt.

```
Question: {question}

Trajectory so far:
{recent_trajectory}

Latest observation (the failure to reflect on):
{warning}

Write your reflection now.
```

Prompt-Adapt artifact-creation prompt.

```
A tool call failed because the API changed.
OLD call: <old_api>(<old_args>)
Environment warning: <raw warning text>
Write ONE short paragraph of standing instructions telling the agent how to
convert any OLD-style arguments for <old_api> into the NEW tool call, so it
can repair this and all future occurrences itself. Do not solve the task;
only describe the argument transformation.
```

Schema-Adapt artifact-creation prompt. The same per-episode 3-attempt budget applies. The prompt is identical except that the model is asked to "rewrite the JSON schema (name + parameter list with one example) of the NEW tool." The resulting schema is injected instead of the paragraph.

G REAL ADAPTER CASE STUDIES

All examples below are extracted from actual run logs. Formatting (line breaks) was adjusted for presentation. The excerpts reproduce the model-generated code as logged.

G.1 SIMPLE RENAME (LOADDB→INITIALIZEDATABASE; V1, COFFEE)

The first attempt succeeds; 360 subsequent calls require no adaptation-related LLM inference.

```python
def adapt(args: dict) -> dict:
    new_args = {}
    if "DBName" in args:
        new_args["DatabaseName"] = args["DBName"]
    return new_args

def inverse_adapt(args: dict) -> dict:
    old_args = {}
    if "DatabaseName" in args:
        old_args["DBName"] = args["DatabaseName"]
    return old_args
```

G.2 RENAME + NEW REQUIRED PARAMETER (RETRIEVEAGENDA→FETCH_AGENDA_DATA; V1, AGENDA)

The inverse correctly ignores the newly introduced `return_num` (absent in old calls).

```python
def adapt(args: dict) -> dict:
    new_args = {}
    if "keyword" in args:
        query = args.get("keyword", "")
        new_args["Query"] = query
        new_args["return_num"] = 5  # default value as not specified in input
    return new_args

def inverse_adapt(args: dict) -> dict:
    old_args = {}
    if "Query" in args:
        old_args["keyword"] = args["Query"]
    if "return_num" in args:
        pass  # return_num is not part of old_args, ignore it
    return old_args
```

G.3 CAUGHT BY ROUND-TRIP VALIDATION: LOSSY FILTERDB DRAFT (V2, FLIGHTS)

First draft (accepted by environment, rejected by round-trip check):

```python
def adapt(args: dict) -> dict:
    new_args = {}
    condition = args.get("condition", "")
    if condition:
        first = condition.split(", ")[0]
        new_args["filterCriteria1"] = first
    return new_args

def inverse_adapt(args: dict) -> dict:
    criterion = args.get("filterCriteria1", "")
    return {"condition": criterion}
```

On `{"condition": "FlightDate=2022-06-05, IATA_Code=AA"}`, the first draft produces `{"filterCriteria1": "FlightDate=2022-06-05"}`, which the target API accepts. The inverse reconstructs only `{"condition": "FlightDate=2022-06-05"}`, dropping `IATA_Code=AA`; the round-trip check therefore fails. The revised adapter splits every comma-separated condition into numbered fields `filterCriteria1, filterCriteria2, ...` and rejoins them in the inverse function:

```python
def adapt(args: dict) -> dict:
    condition = args.get("condition", "")
    parts = condition.split(", ")
    new_args = {}
    for i, part in enumerate(parts, 1):
        new_args[f"filterCriteria{i}"] = part
    return new_args

def inverse_adapt(args: dict) -> dict:
    criteria = []
    i = 1
    while f"filterCriteria{i}" in args:
        criteria.append(args[f"filterCriteria{i}"])
        i += 1
    return {"condition": ", ".join(criteria)}
```

In this example, the call passes the environment check despite the loss of a filtering condition. The round-trip check detects this information loss and prevents the lossy adapter from entering the persistent library. Passing round-trip validation does not guarantee semantic correctness, but it guarantees that no input information is lost during transformation for the checked samples.

G.4 Field rename with mandatory prefix (SQLInterpreter→ExecuteSQLQuery; V1)

The evolved API renames `SQL` to `SQLCommand` and requires the value to begin with a fixed literal prefix. The adapter prepends the prefix on adaptation and strips it on reconstruction.

```python
def adapt(args):
    sql = args.get("SQL", "")
    return {"SQLCommand": "The SQL code is as follows: " + sql}

def inverse_adapt(args):
    cmd = args.get("SQLCommand", "")
    prefix = "The SQL code is as follows: "
    if cmd.startswith(prefix):
        cmd = cmd[len(prefix):]
    return {"SQL": cmd}
```

G.5 Template merge (EdgeCheck→ValidateGraphEdge; V1)

This listing illustrates the transformation from 2 node arguments to a fixed-format string and its inverse, together with the separate `GraphName` field. The inverse parses the template with ordinary string operations.

```python
def adapt(args: dict) -> dict:
    graph = args.get("GraphName", "")
    node1 = args.get("Node1", "")
    node2 = args.get("Node2", "")
    return {
        "Graph_Name": graph,
        "NodeInfos": f"FirstNode[{node1}], SecondNode[{node2}]",
    }

def inverse_adapt(args: dict) -> dict:
    graph = args.get("Graph_Name", "")
    node_infos = args.get("NodeInfos", "")
    node1 = node2 = ""
    if "FirstNode[" in node_infos and "], SecondNode[" in node_infos:
        rest = node_infos.split("FirstNode[", 1)[1]
```

```
        node1, rest2 = rest.split("], SecondNode[", 1)
        node2 = rest2.rsplit("]", 1)[0]
    return {"GraphName": graph, "Node1": node1, "Node2": node2}
```

H WARNING PARSER

In our ToolQA-D implementation, the interface-extraction operation in Algorithm 1 is implemented by parsing deprecation warnings with the following regular expressions (Python syntax):

```
NEW_NAME  = re.compile(r"[Uu]se\s+([A-Za-z_][A-Za-z0-9_]*)\s*\[")
PARAMS    = re.compile(r"\[([^\]]*)\]")            # split on ", " -> param list
EXAMPLE   = re.compile(r"param example:\s*(\{.*?\})")
```

If NEW_NAME does not match, the observation is passed through unchanged (no adaptation triggered). Cache entries are indexed by $k = (f_0, \mathrm{sig}(f_v))$. A stored entry is reused when both the source tool and target signature match.

I TRAINING AND EVALUATION PROTOCOL FOR THE TRAINING-BASED ADAPTATION COMPARISON

Data construction. We use the training questions released with ToolEVO (Chen et al., 2025) across the 7 ToolQA-D domains and both difficulty levels. Before trajectory collection, we reserve a validation split stratified by domain and difficulty, and exclude official test questions from training, validation, and demonstrations. Both collection pipelines use the same initial Qwen3-4B checkpoint and original API descriptions. Tool-SFT trajectories are collected in the Original environment, whereas ToolEVO trajectories are collected in V1. Both pipelines use MCTS with 20 trees per question, 5 branches per expansion, and a maximum path length of 15 steps. Collection demonstrations contain valid original-API usage without error-recovery or API-migration examples. ToolEVO retains its reflection and tool-update mechanisms during interaction. V2 is excluded from SFT trajectory collection.

Trajectory selection. A Tool-SFT trajectory is retained only if it contains at least 1 task-tool call, reaches a correct final answer, and contains no invalid calls, execution errors, deprecation feedback, or tool-update actions. Paths containing error-recovery steps are discarded in full. Valid tool responses containing no matching results are not automatically classified as execution errors. ToolEVO retains successful trajectories together with any intermediate errors, reflections, and tool updates. For the controlled comparison, we use questions with eligible trajectories in both pipelines. We randomly select the same number of trajectories per question, up to 4. Reference answers are used for training-trajectory scoring and selection but are not supplied to the model when generating actions.

Supervised fine-tuning. Each selected trajectory is converted into state–action examples. The input contains tool descriptions, the question, and preceding actions and observations; the target consists of the next Thought, Action, and Action Input. Newly acquired tool descriptions enter subsequent inputs only after they become available in the interaction. Loss is applied only to the target and its termination token; input and padding positions are masked. Observations resulting from the target action and terminal correctness feedback are excluded from that example. Both models are independently fine-tuned from the same initial checkpoint using the same optimization configuration. We match cumulative supervised target tokens. The common budget is 8 times the smaller single-pass target-token count of the 2 selected datasets. Each pipeline follows a fixed data order and stops upon reaching this budget. Any final-batch overshoot is recorded.

Prompting and inference. SFT training and backbone evaluation share one input encoder, tokenizer, chat template, and history-truncation policy. Thinking is disabled, and backbone prompts contain no few-shot demonstrations. The Tool-SFT + ReAct (Yao et al., 2023) baseline follows a

Table 13: Shared training configuration for Tool-SFT and ToolEVO.

Item	Configuration
Initial model	`Qwen3-4B-Instruct`
Fine-tuning method	Full-parameter supervised fine-tuning
Learning rate	2×10^{-5}
Learning-rate schedule / warmup ratio	Cosine / 0.03
Weight decay	0
Maximum input / target length	3,072 / 1,024 tokens
Numerical precision	BF16
Memory optimization	DeepSpeed ZeRO-2 and gradient checkpointing

Both models were trained on 8 NVIDIA H20 GPUs, each with 96 GB of GPU memory.

The target-length limit includes EOS. Inference budgets and TASL hyperparameters follow Appendix A.

Thought–Action–Observation loop in Original, V1, and V2. Execution errors and deprecation warnings are appended to the interaction history, allowing the model to revise subsequent calls within the step budget. The tool interface includes `UpdateTool`, enabling model-generated tool descriptions to be added to subsequent prompts. Tool-SFT + TASL uses the same fine-tuned checkpoint and calls TASL upon encountering a deprecation warning. This checkpoint performs both task reasoning and adapter synthesis through role-specific prompts, without parameter updates during evaluation. ToolEVO retains its model-driven correction and tool-update operations. All methods receive the same environment feedback; adapter synthesis and validation access neither reference answers nor hidden API-transformation implementations.

Evaluation and persistent state. All 3 methods are evaluated on the complete official test set of 1,430 questions under Original, V1, and V2. This yields 4,290 question–version episodes per method. V1 and V2 together account for 2,860 episodes. Evaluation uses greedy decoding, a maximum of 15 backbone actions per question, and at most 256 output tokens per backbone action. Tool backends, observation-processing rules, action parsing, and answer scoring are shared across methods. TASL permits up to 3 candidate synthesis attempts per adaptation episode, uses a historical sample pool of size 3, and applies an eviction threshold of 0.34 after at least 3 observed calls. Questions follow the original benchmark order. The adapter library and sample pools persist within each domain–difficulty–version group and are reset between groups. Every group begins with empty persistent state, and initial acquisition costs are included. No state is transferred from training or validation to testing, or between API versions.

Metrics and repair-call accounting. We report exact-match accuracy for each domain–difficulty group. Easy and Hard scores are macro-averages over their respective 7 domains, and each version score averages all 14 groups with equal weight. Evolved Avg averages the V1 and V2 scores and excludes Original. Aggregation uses unrounded scores, with rounding applied only for presentation. Repair calls and total LLM calls are counted across the 2,860 evolved-version episodes. Total LLM calls include both task reasoning and adaptation. For Tool-SFT + ReAct and ToolEVO, repair calls include model calls classified as feedback-driven call correction or tool-description updating. For Tool-SFT + TASL, repair calls include adapter synthesis and revision, including failed candidates and re-synthesis after eviction. Ordinary task-reasoning calls are excluded from repair-call counts. Successful adapter reuse incurs no adaptation-related model call. The repair-call metric measures adaptation-related model calls rather than total inference tokens or execution time.

J Detailed per-domain results for the training-based adaptation comparison

Tables 14–16 report per-group exact-match accuracy for the comparison experiment. Domains are abbreviated as in Appendix C.

Table 14: Original per-group accuracy for the training-based adaptation comparison (Qwen3-4B).

Method	age.e	age.h	sci.e	sci.h	fli.e	fli.h	cof.e	cof.h	air.e	air.h	yel.e	yel.h	dbl.e	dbl.h	AVG.e	AVG.h
Qwen3-4B																
Tool-SFT	**0.62**	0.24	0.95	**0.40**	**0.62**	**0.46**	**0.99**	0.34	0.92	0.36	**0.88**	0.62	**0.48**	0.18	**0.780**	**0.371**
ToolEVO	0.52	**0.26**	**0.97**	0.36	0.54	0.40	0.98	**0.36**	**0.95**	**0.38**	0.82	**0.64**	0.44	**0.20**	0.746	**0.371**

Table 15: V1 per-group accuracy for the training-based adaptation comparison (Qwen3-4B).

Method	age.e	age.h	sci.e	sci.h	fli.e	fli.h	cof.e	cof.h	air.e	air.h	yel.e	yel.h	dbl.e	dbl.h	AVG.e	AVG.h
Qwen3-4B																
ToolEVO	**0.66**	**0.21**	0.87	0.34	0.52	0.43	0.94	0.45	**0.91**	0.29	0.81	0.58	0.39	**0.30**	0.729	**0.371**
Tool-SFT + ReAct	0.36	0.08	0.70	0.21	0.18	0.05	0.60	0.16	0.72	0.18	0.55	0.26	0.22	0.14	0.476	0.154
Tool-SFT + TASL	0.55	0.15	**0.94**	**0.36**	**0.60**	**0.44**	**0.96**	**0.46**	0.88	**0.33**	**0.88**	**0.59**	**0.43**	0.25	**0.749**	0.369

Table 16: V2 per-group accuracy for the training-based adaptation comparison (Qwen3-4B).

Method	age.e	age.h	sci.e	sci.h	fli.e	fli.h	cof.e	cof.h	air.e	air.h	yel.e	yel.h	dbl.e	dbl.h	AVG.e	AVG.h
Qwen3-4B																
ToolEVO	**0.65**	0.14	0.90	**0.35**	0.46	0.34	0.96	0.43	0.85	0.27	0.83	0.55	0.30	0.22	0.707	0.329
Tool-SFT + ReAct	0.40	0.12	0.58	0.22	0.24	0.10	0.56	0.20	0.65	0.24	0.52	0.30	0.27	0.18	0.460	0.194
Tool-SFT + TASL	0.64	**0.18**	**0.93**	0.34	**0.59**	**0.40**	**0.97**	**0.44**	**0.92**	**0.31**	**0.87**	**0.56**	**0.44**	**0.28**	**0.766**	**0.359**

K EXECUTION SAFETY

Evaluation setup. Generated adapters execute with a restricted `__builtins__` dictionary. It exposes only pure data-structure builtins (`dict`, `list`, `str` methods, `int`, `len`, `range`, ...) and blocks `import`, `open`, `eval`, `exec`, and `__import__`. In our benchmark runs, generated adapters are pure dictionary/string transformations that terminate in microseconds, so no timeout was imposed (Table 7). No hang or resource-exhaustion event was reported in the benchmark runs.

L CONTROLLED ORACLE VALIDATION

This section reports a supplementary diagnostic experiment that isolates the adaptation pipeline from downstream task execution. StableToolBench highlights the importance of controlling API-response instability when evaluating tool use (Guo et al., 2024). Our oracle diagnostic controls the API rules to separate adaptation behavior from downstream task execution. Its scores measure whether the tool is correctly called under the implemented API rules, not end-to-end answer exact match. To isolate the detect→synthesize→cache pipeline from confounds (ReAct reasoning ability, domain data distributions), we re-implemented ToolEVO's 11 tools $\times$ 2 evolved versions in a lightweight oracle. This implementation covers signature mappings and legality checks. The semantics of "how tools evolve" and "what counts as correct" are identical for all methods.

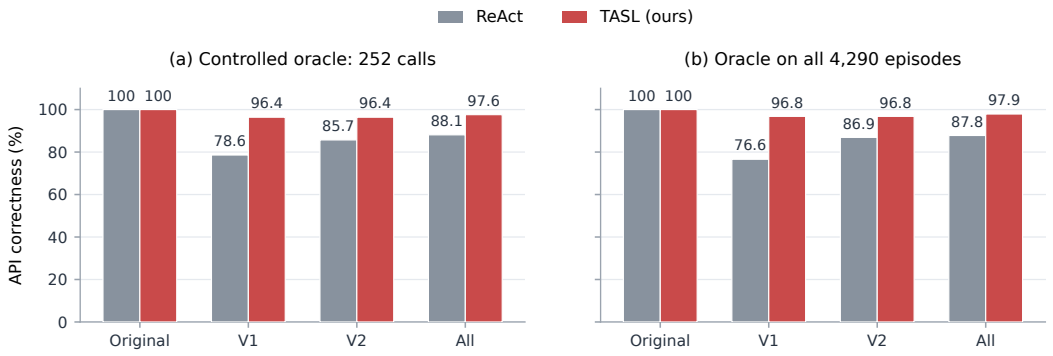

Figure 4: **Controlled oracle validation.** (a) Reported results on 252 controlled calls grouped by split: TASL obtains 96–100% accuracy with 83.7% fewer LLM calls. (b) Oracle results over 4,290 question–version episodes. This diagnostic checks the implemented perturbation rules rather than generalization to an independent tool ecosystem.

Controlled calls (84 calls in each of Original, V1, and V2; 252 total; Figure 4a): TASL attains 96.4–100% accuracy across splits (overall 97.6% vs. ReAct 88.1%) while cutting LLM calls by 83.7% (41 vs. 252). TASL makes 21 and 20 synthesis calls on V1 and V2, respectively. On Original, it triggers 0 synthesis calls (100%, 0 calls). Per-tool highlights: `EdgeCheck` (template merge) and `SQLInterpreter` (literal prefix) go from 0/6 (ReAct) to 6/6 (TASL) with 1 synthesis call each. `GetValue` (comma-split) requires re-synthesis (10 calls) as round-trip checks reject lossy first drafts.

Full test set under the oracle (4,290 question–version episodes; Figure 4b): Both ReAct and TASL achieve 100% on Original. TASL reaches 96.8–100% per split (97.9% overall vs. ReAct 87.8%). It does not trail ReAct in any of the 42 groups, providing mechanism-level support for the end-to-end results.

M Reproducibility

Taken together, the preceding appendices provide complete experimental details: hyperparameters, perturbation mappings, per-group results, prompts, and the full adaptation algorithm. All experiments run on local checkpoints with greedy decoding. Per-question raw logs (JSONL) record every ReAct step, synthesis event, cache hit, and eviction. The controlled oracle experiments (Appendix L) run in minutes on 1 GPU. The full end-to-end suite evaluates 4,290 question–version episodes per method per model. The complete codebase will be released upon publication.