CAN TOOL-USING AGENTS KEEP ADAPTING AS TOOL ENVIRONMENTS CHANGE?

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents increasingly rely on external tools, yet existing tool-learning methods typically assume relatively stable tool environments. In practice, tools and their interfaces continuously evolve. We study dynamic tool learning, where an agent must retain unaffected tool-use capabilities while efficiently adapting behaviors invalidated by environmental changes. Our key observation is that multi-step tool-use behaviors naturally exhibit ordered and compositional structures. Inspired by natural selection, we propose CATE (Continual Adaptation to Tool Environments), which organizes tool-use experience as an evolving population of skills. CATE represents accumulated tool-use experience as reusable Skills that capture not only individual tool usage but also the ordering and dependencies among tools. Starting from single-tool Skills, it progressively develops compositional multi-step behaviors through interaction, recombination, and failure-driven refinement. When the environment changes, CATE localizes affected skills and adapts them through experience inheritance, preserving still-valid knowledge instead of relearning the entire tool space. To systematically evaluate continual adaptation, we introduce DynaToolBench, a controlled benchmark built on a stable MirrorAPI execution backend. DynaToolBench models tool replacement over three successive timesteps while ensuring that evaluation tasks remain solvable as the environment evolves. Experiments across evolving tool environments show that CATE maintains strong performance over time, effectively recovering capabilities affected by tool changes while largely preserving previously acquired behaviors.

1 INTRODUCTION

Large language models (LLMs) can extend their capabilities by interacting with external tools, APIs, and software services, making tool use an important component of LLM-based agents (Schick et al., 2023; Qin et al., 2024). Existing tool-learning methods acquire tool-use knowledge either by training the underlying model or by providing external knowledge such as tool documentation, demonstrations, interaction experience, and memory at inference time. For example, EasyTool rewrites heterogeneous tool documentation into concise instructions (Yuan et al., 2025), while DRAFT iteratively improves tool documentation through trial-and-error interaction and execution feedback (Qu et al., 2025). Most existing methods, however, assume that the tool environment remains fixed during learning and subsequent use.

This assumption becomes problematic when tool interfaces evolve, especially in multi-step tool use where later calls often depend on outputs returned by earlier ones. ToolExpNet, for example, models similarity and functional dependencies between tools to improve multi-tool selection and planning (Zhang et al., 2025), but such knowledge is still learned with respect to the interfaces currently available to the agent. When a tool is replaced by another interface with different parameters or return fields, part of the previously learned workflow may become outdated even though the underlying task remains solvable. As illustrated in Figure 1, the same movie-information workflow can remain valid after replacement while the identifier passed between consecutive calls changes from `id` to `reference_id`. Recent work such as ContDa begins to study adaptation in evolving toolsets (Wu et al., 2026); here, we focus specifically on *tool replacement*, asking how previously accumulated multi-tool experience can preserve still-valid knowledge while adapting the parts invalidated by the new interface. Importantly, such changes do not necessarily invalidate an entire multi-step behav-

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.

ior. A replaced tool may alter only one local invocation or dependency, while upstream decisions, downstream structure, and interaction experience remain useful. Reconstructing tool-use knowledge from scratch after every change would therefore discard still-valid experience and repeatedly relearn previously acquired behavior. This motivates a more local form of adaptation: the agent should identify which parts of its accumulated knowledge are affected, revise them against the new interface, and preserve the remainder whenever possible. The challenge is therefore not only to acquire knowledge about the new tool, but also to determine how that change propagates through previously learned tool-use dependencies and behaviors.

To address this problem, we represent accumulated tool-use experience as reusable *Skills* and propose **CATE** (**C**ontinual **A**daptation to **T**ool **E**nvironments), a training-free framework that maintains and evolves a SkillBank over time. A Skill represents an ordered tool-use behavior together with experience about when and how its tools should be used and coordinated. CATE starts from atomic Skills and discovers composite Skills through dependency-aware composition and interaction. Execution feedback continually refines their experience. When the environment changes, CATE identifies affected Skills, inherits its experience that remains useful, and updates the parts that no longer match the replacement interface. Viewed through the lens of natural selection, the SkillBank forms an evolving population of behaviors: useful experience is retained and inherited, while behaviors that no longer fit the current environment are revised through further interaction.

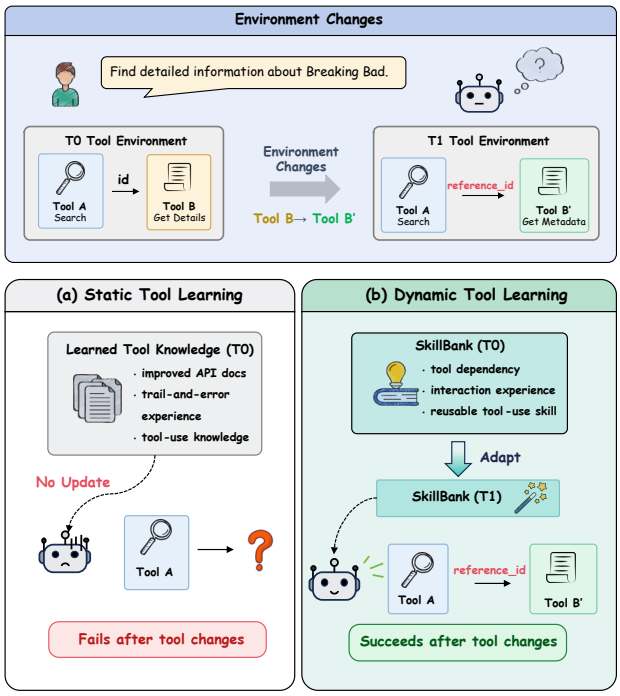


Figure 1: Illustration of tool replacement and dynamic adaptation. Above: solution-preserving interface change. Below: static versus adaptive tool knowledge.

Evaluating such adaptation also requires a controlled dynamic environment. Live APIs may fail, change unexpectedly, or become unavailable, making it difficult to separate adaptation failures from execution instability. Moreover, if a tool change removes the only valid solution to a request, failure cannot be attributed solely to the agent. We therefore introduce **DynaToolBench**, built on a stable MirrorAPI execution backend (Guo et al., 2025). DynaToolBench applies controlled, solution-preserving tool replacements over three successive timesteps, while keeping every evaluation request solvable in the current environment. This design isolates adaptation to changing tool interfaces from uncontrolled API instability, capability acquisition, and task unsolvability.

Experiments across successive tool replacements show that CATE maintains strong performance as the environment evolves, effectively recovering from interface changes while preserving still-valid tool-use experience. Our main contributions are:

- We formulate dynamic tool learning under solution-preserving tool replacement, where previously acquired multi-tool experience may become only partially invalid after an environment change.
- We propose CATE, a training-free framework that organizes tool-use experience as compositional Skills and locally adapts affected Skills through experience inheritance when tools are replaced.
- We introduce DynaToolBench, a controlled longitudinal benchmark with stable execution and solution-preserving replacements over three timesteps, enabling systematic evaluation of retention and adaptation.

2 Related Work

Tool Learning for LLM Agents. Existing tool-learning approaches improve tool use either by training the underlying model with demonstrations, synthetic trajectories, or execution feedback (Schick et al., 2023; Tang et al., 2023; Qin et al., 2024; Liu et al., 2025), or by improving the external tool knowledge provided to a fixed LLM. Among the latter, EasyTool rewrites tool documentation into concise instructions (Yuan et al., 2025), DRAFT refines documentation through trial-and-error interaction (Qu et al., 2025), and ToolExpNet models similarity and functional dependencies between tools for multi-tool selection and planning (Zhang et al., 2025). These methods mainly assume fixed tool environments. More recently, ContDa studies evolving toolsets through continual documentation adaptation (Wu et al., 2026). CATE instead focuses on solution-preserving tool replacement and adapts previously accumulated multi-tool experience as the underlying interfaces change.

Skill-Based Agent Learning. Agent experience has also been organized as reusable procedural knowledge. Voyager maintains composable executable skills (Wang et al., 2024), ExpeL extracts reusable experience from past interactions (Zhao et al., 2024), and Agent Workflow Memory induces reusable workflows from previous trajectories (Wang et al., 2025). These works demonstrate the value of persistent behavioral abstractions; CATE extends this idea to evolving tool environments, where such knowledge must be selectively preserved and adapted when the underlying tools change.

Benchmarks for Tool-Using Agents. Existing benchmarks evaluate API selection, execution, and multi-step tool use, including API-Bank, ToolBench, and RestBench (Li et al., 2023; Qin et al., 2024; Song et al., 2023), while recent work further considers complex function calling and stateful interaction (Patil et al., 2025; Lu et al., 2025). StableToolBench (Guo et al., 2024) improves reproducibility through stable tool-use evaluation, while MirrorAPI (Guo et al., 2025) provides a stable execution backend. DynaToolBench builds on this setting but introduces controlled, solution-preserving tool replacements across successive timesteps to evaluate retention and adaptation of previously acquired tool-use knowledge.

3 Problem Definition: Dynamic Tool Learning

Dynamic Tool Environments. We model a dynamic tool environment as a sequence $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_T$, where $\mathcal{E}_t = (\mathcal{T}_t, \mathcal{D}_t)$ contains the set of tools available at timestep t and their observable documentation. The initial environment $\mathcal{E}_0$ allows an agent to acquire reusable tool-use knowledge through interaction. As the environment evolves, tool identities or interfaces may change, making previously acquired knowledge only partially valid. Dynamic tool learning therefore requires the agent to preserve knowledge that remains useful while adapting behaviors affected by such changes.

Tool-Use Workflows and Skills. For a user query q, let $\Pi_t(q)$ denote the set of valid tool-use solutions under environment $\mathcal{E}_t$. A solution consists of an ordered sequence of tool calls, where later calls may depend on outputs produced by earlier ones. Consequently, replacing one tool may invalidate only its own invocation or may further affect downstream calls that consume its outputs.

Rather than retaining such experience solely at the level of individual tool documentation, we use a *Skill* as the persistent unit of reusable tool-use behavior. A Skill s is represented as

$$s = (g_s, e_s), \qquad g_s = (\tau_1, \ldots, \tau_L), \tag{1}$$

where g_s denotes the ordered sequence of tools involved in the behavior, and e_s denotes the accumulated procedural experience associated with it. Such experience may describe when the Skill is applicable, how its tools should be coordinated, how parameters are constructed, and how intermediate outputs are passed between dependent calls. When $L = 1$, s is an *atomic Skill* associated with a single tool; when $L > 1$, it is a *composite Skill* representing a reusable multi-tool behavior. Unlike ordinary tool documentation, which primarily describes the current interface and functionality of an individual tool, a Skill is an agent-maintained behavioral abstraction that can accumulate interaction experience, be reused across tasks, and be composed with other Skills.

Solution-Preserving Adaptation. A central difficulty in evaluating dynamic tool learning is distinguishing failure to adapt from a task becoming unsolvable after an environment change. We therefore consider *solution-preserving* changes, requiring that for every evaluation query q and timestep

Table 1: DynaToolBench statistics.

Subset	APIs	Tasks	Path Lengths	Avg. Len.	Affected T_1/T_2
DS	50	64	$60\times2,\ 4\times3$	2.06	40 / 62
DL	34	30	$20\times3,\ 7\times4,\ 3\times5$	3.43	28 / 30

t,

$$\Pi_t(q) \neq \emptyset. \tag{2}$$

The valid tool-use path may change across environments even though at least one solution remains available. Thus, a behavior that succeeded in $\mathcal{E}_{t-1}$ may become invalid in $\mathcal{E}_t$ while the underlying user task remains solvable. This setting allows performance degradation after a tool change to be attributed to outdated tool-use knowledge rather than to the disappearance of the required capability.

4 DynaToolBench

Existing tool-use benchmarks typically assume a fixed tool environment, making it difficult to evaluate whether agents can preserve and update tool-use knowledge as APIs evolve. We therefore construct **DynaToolBench**, an executable benchmark with controlled changes to the available API interfaces. Starting from a stable dependency-aware environment at T_0, we derive T_1 and T_2 through controlled API replacement while preserving task semantics and backend execution. Additional construction and filtering details are provided in Appendix A.

4.1 Benchmark Construction

We construct the initial T_0 benchmark from the G2 and G3 API groups of StableToolBench (Guo et al., 2024), which contain functionally related APIs suitable for multi-step tool composition. Candidate APIs are filtered and validated against the MirrorAPI backend (Guo et al., 2025) to ensure stable and reproducible execution.

We identify executable inter-API dependencies by matching output fields of upstream APIs to compatible inputs of downstream APIs, followed by semantic and execution verification. Valid dependency paths are sampled only after end-to-end execution succeeds, and the verified trajectories are then used to synthesize natural-language queries and their ground-truth tool trajectories. For LLM-assisted construction and verification, we use GLM-5.2. Each query–trajectory pair is further scored on five dimensions: *naturalness*, *coherence*, *necessity*, *dependency quality*, and *solvability*. Instances are removed when coherence, necessity, dependency quality, or solvability receives a score of 2 or lower, or when the task is judged to be an artificial composition of otherwise unrelated APIs. This separates mere executability from benchmark quality; full prompts and filtering details are provided in Appendix A.1.

The resulting benchmark contains two complementary subsets. **Dyna-Short (DS)** contains 64 tasks over 50 APIs and primarily tests short two-step compositions, whereas **Dyna-Long (DL)** contains 30 disjoint tasks over 34 APIs with three- to five-step solution paths. Statistics are summarized in Table 1.

4.2 Controlled Dynamic Evolution

Starting from T_0, we construct T_1 and T_2 through **cumulative API replacement**. At each transition, selected APIs are replaced by functionally equivalent alternatives with different exposed interfaces, which may alter API names, parameter schemas, output fields, and inter-API dependencies. The user request and underlying task semantics remain unchanged.

All timesteps share the same MirrorAPI backend, while a versioned adapter maps each timestep-specific API surface to consistent executable behavior. Before execution, the adapter converts agent-facing arguments to the backend schema and transforms the returned response back to the interface exposed at the current timestep. These mappings belong only to the execution layer and are never exposed to the agent. Thus, tasks remain solvable across environments, but agents must adapt their previously acquired tool-use knowledge to the changed interfaces.

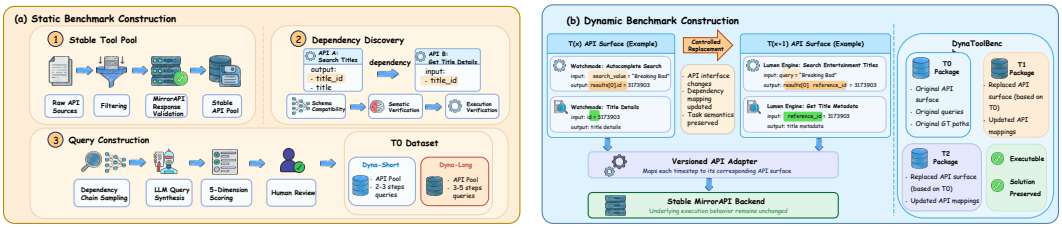


Figure 2: Construction of DynaToolBench. (a) Static benchmark construction from stable APIs and dependency-aware task generation. (b) Controlled dynamic evolution through solution-preserving API replacement.

5 CATE

Figure 3 presents an overview of CATE. CATE maintains a **SkillBank** of reusable tool-use behaviors, including atomic Skills associated with individual APIs and composite Skills that capture local relations between tools. Each Skill stores its underlying tool sequence together with accumulated experience about when and how the behavior should be used. At T_0, CATE initializes atomic Skills from tool documentation, expands them into useful compositions, and refines them through interaction. When the environment changes, unaffected Skills are preserved while experience associated with replaced tools is selectively inherited and adapted.

Composite Skills capture local reusable tool relations rather than complete task solutions. Longer trajectories can therefore retrieve and combine multiple relevant Skills, keeping individual Skills reusable across tasks.

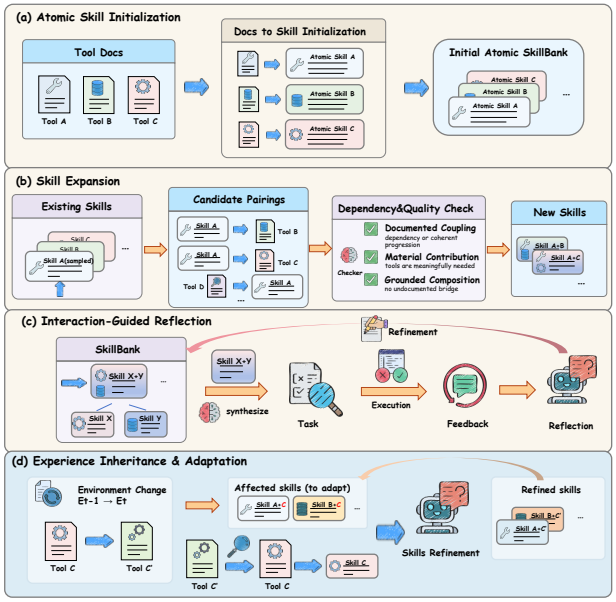


Figure 3: Overview of CATE. Panels (a–c) show SkillBank evolution at T_0; panel (d) shows experience inheritance and local adaptation after environment changes.

SkillBank Evolution at T_0. Starting from the atomic SkillBank, CATE acquires tool-use experience through two complementary processes: *Skill Expansion* discovers useful relations between tools, while *Interaction-Guided Reflection* improves Skills using execution experience.

Skill Expansion. Starting from an atomic Skill, CATE proposes a small set of tools that may naturally precede or follow it and verifies each candidate against the corresponding API documentation. As illustrated in Figure 3(b), a composition is retained only if the tools exhibit a grounded information dependency or functional progression, make non-redundant contributions to the resulting capability, and do not rely on undocumented assumptions. At T_0, expansion produces pairwise composite Skills that provide reusable local experience for longer tool-use trajectories.

Evolution Details. Atomic Skills are routed to Skill Expansion, whereas already composed Skills are consolidated rather than recursively expanded; composite Skills therefore contain at most two APIs in our experiments. For each Expansion attempt, CATE proposes at most three prefix and three suffix candidates, retains at most two validated composite Skills, and constructs three diagnostic tasks for each newly generated Skill. These tasks are executed in the actual tool environment and provide the interaction evidence used for subsequent refinement.

Interaction-Guided Reflection. A structurally plausible Skill does not necessarily imply reliable tool use. CATE therefore constructs diagnostic tasks from each explored Skill and its associated API documentation and executes them against the environment. These internally generated tasks are used only for Skill learning rather than benchmark evaluation. Execution feedback identifies whether failures arise from Skill retrieval, tool-use strategy, or execution, allowing the corresponding experience to be revised without rewriting the entire Skill. Reusable tool-specific evidence is also propagated back to participating atomic Skills, while partner-specific relations and incidental task details are discarded.

Reflection is checked every five evolution attempts rather than after every interaction. An active Skill becomes eligible when its fitness falls below 0.6 and at least three new interaction records have accumulated since its previous successful Reflection. Each update uses at most three recent failures and two successful cases, and revises only the diagnosed Skill field while preserving unaffected knowledge.

Interaction Evidence and Fitness. Generated diagnostic tasks are executed in the same tool environment used by the agent, and a separate training-time completion judge determines whether each request is successfully completed from the actual trajectory and final response. This signal is used only for Skill evolution and is independent of the formal evaluation judge. For each Skill, CATE maintains a fitness estimate updated from the success rate of its attributable interaction batch using an exponential moving average with coefficient $\alpha = 0.3$. The fitness score provides a compact signal of whether the current Skill remains reliable and is subsequently used to determine when additional Reflection is required.

Experience Inheritance and Adaptation. When the environment changes, CATE preserves Skills unrelated to replaced tools and selectively updates affected knowledge, avoiding both reconstruction from scratch and direct reuse of obsolete experience.

Atomic Experience Inheritance. For a newly introduced tool, CATE first initializes a fresh atomic Skill from its current documentation. Retired atomic Skills from the previous environment are then considered as potential experience sources. If a functionally related predecessor is identified, only reusable experience that remains supported by the new API documentation is transferred. The new Skill remains anchored to its current documentation, preventing obsolete API names, parameters, or interface assumptions from being inherited. In our replacement setting, CATE retrieves the three most similar retired atomic APIs as possible experience sources; the LLM may select one of them or NONE, and only documentation-supported execution notes are transferred.

Composite Skill Adaptation. CATE identifies composite Skills whose tool sequences contain a replaced API. For each affected position, candidate replacements are retrieved from newly introduced APIs and examined with the original documentation and the role of the removed tool in the Skill. The selected replacement is incorporated into the tool sequence and its execution guidance is revised for the new interface, while unaffected parts of the composition and its high-level capability are preserved whenever possible. Each affected composite Skill is adapted once per environment transition. For every removed API, the three most similar newly introduced APIs are retrieved as candidates from public documentation, after which the LLM selects the replacement using the complete old and candidate documentation and the affected step's role in the Skill. The adaptation procedure receives neither benchmark ground-truth trajectories nor formal evaluation outcomes.

The resulting SkillBank becomes the knowledge base for subsequent tool use and future environment changes, allowing CATE to inherit previously learned experience without relearning the entire tool space.

Skill Retrieval and Execution. During formal evaluation, the SkillBank is frozen and no new experience is written back. For each query, CATE retrieves the top-5 active Skills using Qwen3-Embedding-0.6B, where retrieval is based on the Skill title and when-to-apply description and Skills with identical tool sequences are deduplicated. Retrieved Skills serve as reusable guidance rather than rigid programs: the executor may reuse an entire Skill or only the relevant local experience, while current API documentation and actual tool observations remain authoritative. Execution is limited to at most six tool-use decision steps.

6 EXPERIMENTS

We evaluate CATE on DynaToolBench to answer three questions: (1) whether CATE maintains strong tool-use performance as the environment evolves; (2) whether continual adaptation helps recover from API replacements; and (3) how much additional cost is required for such adaptation.

6.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate all methods on the Dyna-Short (DS) and Dyna-Long (DL) subsets of DynaToolBench introduced in Section 4. Both subsets contain three environments, T_0, T_1, and T_2, with cumulative API replacements across timesteps.

Baselines. We compare CATE with representative methods for constructing and adapting external tool knowledge, including tool documentation and reusable tool-use experience. The continual baseline, ContDa (Wu et al., 2026), maintains and updates learned tool knowledge as the environment evolves. Among the non-continual baselines, Raw-Doc directly uses the current tool documentation, while EasyTool (Yuan et al., 2025) independently constructs concise tool instructions for each environment. Each method follows its original knowledge-update procedure under the same timestep-specific tool environments and task sets.

Backbone Models and Implementation Settings. We conduct experiments with Qwen3-4B and Qwen3-8B to examine whether continual tool learning remains effective across different model scales. Within each model-scale experiment, the same backbone is used for CATE evolution, dynamic adaptation, task execution, and the corresponding LLM-based baseline operations. The Qwen3 models are served locally with vLLM with reasoning mode disabled and temperature 0.3. CATE uses 100 evolution attempts at T_0, pairwise composite Skills (maximum length 2), Qwen3-Embedding-0.6B for Skill retrieval, top-5 evaluation-time Skill retrieval, and at most six rollout decision steps. All reported results are averaged over three independent repetitions. Additional settings are summarized in Appendix B.

Evaluation Protocol. We report LLM-judged task accuracy (%). For each task, the evaluator decomposes the user query into semantically independent requirements and scores each requirement as fulfilled (1.0), partially fulfilled or environment-limited (0.5), or failed (0.0), based on the executed tool trajectory, observed outputs, and final response. Formal evaluation uses an isolated Gemini-3.8-Flash judge with temperature 0. The judge does not receive the benchmark ground-truth trajectory, so any valid tool-use path can receive full credit.

Task-level scores are obtained by averaging requirement-level scores within each query, and the reported dataset accuracy is the mean task score over the evaluation set. A successful API response alone is not sufficient for credit: the judge must verify that the supporting observation concerns the requested entity and that identifiers passed between APIs use the namespace expected by the receiving tool. Incorrect tool selection, argument construction, identifier propagation, missing required actions, unsupported final claims, or entity mismatches receive zero rather than partial credit.

For each evolved environment T_i, T_i^- denotes performance immediately after the environment changes, while T_i^+ denotes performance after the corresponding method processes the new environment. Continual methods adapt knowledge inherited from the previous timestep, whereas non-continual methods process the current environment independently. In the main results, T_1 and T_2 denote T_1^+ and T_2^+, respectively. Detailed judging criteria and implementation are provided in Appendix B.6.

6.2 MAIN RESULTS

Table 2 reports the main results across two backbone models and both DynaToolBench subsets.

Overall, CATE achieves the highest average performance across all four model–subset settings, with a more pronounced advantage on Dyna-Long. The stronger results on DL suggest that accumulated tool-use experience is particularly useful for tasks involving longer tool chains and stronger inter-tool dependencies. More broadly, these results suggest that continually maintaining structured tool-

Table 2: Main results on DynaToolBench in LLM-judged task accuracy (%). T_1 and T_2 denote post-processing performance, and Avg. is the mean over the three timesteps.

Model	Subset	Type	Method	T_0	T_1	T_2	Avg.
Qwen3-4B	*DS*	Non-cont.	Raw-Doc	**76.43**	70.57	70.96	72.65
			EasyTool	73.96	66.15	66.80	68.97
		Continual	ContDa	75.78	70.77	**75.13**	73.89
			CATE (Ours)	76.04	**76.04**	72.92	**75.00**
	DL	Non-cont.	Raw-Doc	72.50	75.83	73.61	73.98
			EasyTool	67.78	70.83	67.06	68.56
		Continual	ContDa	71.67	76.94	**80.56**	76.39
			CATE (Ours)	**76.67**	**83.61**	79.72	**80.00**
Qwen3-8B	*DS*	Non-cont.	Raw-Doc	76.04	72.92	**74.48**	74.48
			EasyTool	72.14	67.45	67.45	69.01
		Continual	ContDa	73.24	70.70	69.99	71.31
			CATE (Ours)	**77.99**	**74.61**	72.66	**75.09**
	DL	Non-cont.	Raw-Doc	80.00	77.22	73.33	76.85
			EasyTool	73.33	63.33	68.06	68.24
		Continual	ContDa	80.06	76.72	78.39	78.39
			CATE (Ours)	**81.67**	**78.47**	**79.17**	**79.77**

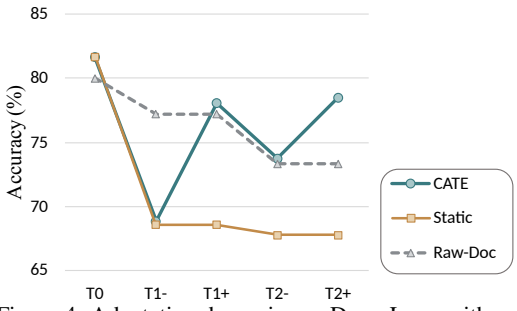


Figure 4: Adaptation dynamics on Dyna-Long with Qwen3-4B. T_i^- and T_i^+ denote performance immediately before and after adaptation.

Method	T_0	T_1	T_2	Avg.
w/o Evolution	80.00	77.22	73.33	76.85
w/o Adapt.	**81.67**	68.61	67.78	72.69
CATE	**81.67**	**78.47**	**79.17**	**79.77**

Table 3: Ablation on Dyna-Long with Qwen3-8B.

use experience can provide benefits beyond independently processing the current documentation or continually refining documentation alone.

6.3 Further Analysis

Adaptation Dynamics and Ablation. We first examine how CATE responds to environment changes and isolate the role of tool-knowledge evolution. Figure 4 reports performance immediately before and after adaptation on Dyna-Long with Qwen3-4B, while Table 3 compares CATE with two controlled variants on Dyna-Long with Qwen3-8B.

Environment changes initially disrupt previously learned tool-use patterns, while subsequent adaptation recovers a substantial portion of the lost performance. This behavior appears across successive replacements, indicating that CATE preserves still-valid experience while revising knowledge associated with affected tools and dependencies.

The ablation further shows that constructing the SkillBank already improves performance at T_0, but freezing it after tool replacement causes substantial degradation. In particular, stale experience can perform worse than discarding the learned SkillBank and using the current raw documentation. The full CATE model instead preserves the benefits of accumulated experience by continually updating affected knowledge.

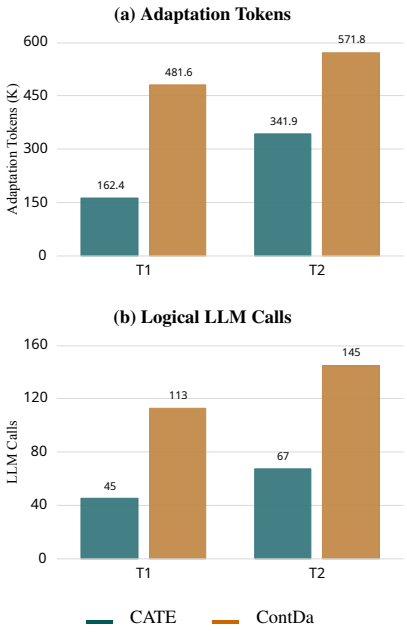


Figure 5: Adaptation cost on Dyna-Long with Qwen3-8B: adaptation tokens (a) and logical LLM calls (b).

Query	Find the U.S. streaming link for *Enemy Lines*, check whether it is phishing, and retrieve the domain's abuse-contact information.
Change	**API:** `viewdns::abuse lookup` → `Kestrel::find abuse contact` **Input:** `domain` → `hostname` **Output:** `email` → `mailbox_address`
Before T_1^-	**Guidance:** The valid prefix is preserved, but the replacement API is not yet covered, so the final lookup is omitted. **Sequence:** Streaming Sources → Phishing Check → **Missing lookup**
After T_1^+	**Guidance:** CATE preserves the valid prefix and updates only the affected tail for the new interface. **Sequence:** Streaming Sources → Phishing Check → **Kestrel abuse lookup**

Figure 6: Case study of localized adaptation under tool replacement.

Adaptation Cost. We further compare CATE and ContDa on Dyna-Long with Qwen3-8B in terms of the additional computation required at T_1 and T_2. Figure 5 reports adaptation tokens and logical LLM calls. We count only method-side knowledge updates performed after an environment transition; ordinary task execution and the independent formal-evaluation judge are excluded from both measures.

CATE incurs substantially lower adaptation overhead than ContDa under both measures at T_1 and T_2. The difference follows from their adaptation scope: rather than re-processing a broader set of tool knowledge after each environment change, CATE updates Skills associated with affected tools while preserving the remaining SkillBank. The results therefore show that localized adaptation reduces the additional LLM inference required to accommodate successive tool replacements.

Case Study. Figure 6 illustrates how this localized adaptation operates at the level of an individual tool-use behavior. After the abuse-contact API is replaced, the preceding streaming-source and phishing-check steps remain valid, while the final dependency becomes outdated. Before adaptation, the agent therefore preserves the valid prefix but fails to perform the required domain-contact lookup. CATE updates the affected Skill with the replacement API and its new input/output interface, recovering the missing final step without reconstructing the unaffected portion of the tool-use behavior.

7 Conclusion

We study tool-using agents in environments where APIs change over time and introduce CATE, a continual tool-learning framework that organizes tool-use knowledge as reusable skills. CATE expands atomic tool knowledge into useful dependencies through interaction, accumulates execution experience, and locally adapts affected skills when the environment changes while preserving still-valid knowledge. To support systematic evaluation, we further introduce DynaToolBench, which provides controlled multi-step tool replacements over stable and reproducible API environments. Experiments across two backbone models show that CATE maintains strong performance throughout successive environment changes while requiring lower adaptation overhead than continual documentation adaptation. Together, these results suggest that evolving structured tool-use experience provides a lightweight way for agents to remain effective as their tool environments change.

AI USE STATEMENT

Generative AI tools were used to provide feedback on research methodology and experiments, assist with implementation and troubleshooting, and help interpret experimental outputs. They were also used for literature search and summarization, language editing, LaTeX formatting, code editing, and figure-layout ideation. All AI-assisted outputs were reviewed by the authors; experimental results, citations, code changes, and scientific claims were checked against the underlying data, implementation, and primary sources. The authors take responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

REFERENCES

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. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.664.

Zhicheng Guo, Sijie Cheng, Yuchen Niu, Hao Wang, Sicheng Zhou, Wenbing Huang, and Yang Liu. StableToolBench-MirrorAPI: Modeling tool environments as mirrors of 7,000+ real-world APIs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 5247–5270. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.273.

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. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.187.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. In *International Conference on Learning Representations*, 2025.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.65.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025.

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 *International Conference on Learning Representations*, 2024.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. From exploration to mastery: Enabling LLMs to master tools via self-driven interactions. In *International Conference on Learning Representations*, 2025.

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.

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, 2023.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. ToolAlpaca: Generalized tool learning for language models with 3000 simulated cases, 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. *Transactions on Machine Learning Research*, 2024.

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 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025.

Bin Wu, Edgar Meij, and Emine Yilmaz. Beyond static toolsets: Self-evolving LLM tool agents via continual documentation adaptation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 21519–21539. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1082.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. EASYTOOL: Enhancing LLM-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 951–972. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.44.

Zijing Zhang, Zhanpeng Chen, He Zhu, Ziyang Chen, Nan Du, and Xiaolong Li. ToolExpNet: Optimizing multi-tool selection in LLMs with similarity and dependency-aware experience networks. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 15706–15722. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.811.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

A DynaToolBench Construction Details

This appendix provides additional details of the benchmark construction pipeline illustrated in Fig. 2. We first construct a static multi-tool benchmark at T_0, and then derive controlled dynamic environments by evolving the agent-facing API surface while preserving the underlying task semantics and executable backend. Unless otherwise specified, GLM-5.2 is used for the LLM-assisted construction and verification steps described below. We provide the core prompt instructions together with their corresponding construction stages; fixed formatting requirements and instance-specific API content are omitted for readability.

A.1 Static Benchmark Construction

Stable API Pool. As illustrated in Fig. 2(a), we construct the initial API pool from the G2 and G3 groups of StableToolBench (Guo et al., 2024), which provide APIs suitable for constructing cross-API dependencies. We first remove APIs that cannot be reliably matched to the executable environment. The remaining APIs are invoked through MirrorAPI (Guo et al., 2025), and APIs that fail response validation or cannot be executed reliably are discarded. This produces the stable API pool used for subsequent benchmark construction.

Dependency Discovery. We next identify executable dependencies between APIs. Candidate API pairs are first obtained through schema compatibility between upstream outputs and downstream inputs. Because schema matching alone can introduce spurious relations, each candidate is further checked by GLM-5.2 for semantic consistency: the upstream response must provide information

that is meaningfully required by the downstream API. Finally, the candidate dependency is instantiated and executed through MirrorAPI. Only dependencies that pass both semantic and execution verification are retained.

For semantic verification, GLM-5.2 receives the documentation of the upstream and downstream APIs together with a real upstream response. The main instruction is:

> **Dependency verification.** Determine whether API A's real response contains task-relevant information that can be used to construct a call to API B. A dependency exists only when at least one task-relevant input of B cannot be determined without information from A's response. The source and target field names do not need to match, and the source value may be transformed when necessary. Do not treat pagination or control arguments such as `page`, `limit`, `offset`, or `sort` as meaningful dependencies. If a valid dependency exists, identify the source response field, the corresponding target parameter, and an executable input for API B; otherwise, report that no meaningful dependency exists.

The resulting dependency graph is used to sample multi-step API chains. Dyna-Short (DS) contains shorter trajectories of 2–3 API calls, whereas Dyna-Long (DL) contains longer trajectories of 3–5 API calls.

Query Construction and Filtering. For each sampled chain, we first execute the complete trajectory to ensure that the dependency chain is realizable end-to-end. Only after the complete trajectory has executed successfully do we synthesize the corresponding natural-language query.

GLM-5.2 receives the verified trajectory, API information, dependency mappings, and execution evidence. The main query-generation instruction is:

> **Query synthesis.** The supplied ground-truth tool chain has already been executed successfully from end to end with real responses. Synthesize one natural user query for which this verified chain is a reasonable solution. All steps should contribute to one coherent user goal. Do not expose API or tool names, endpoint names, parameter names, JSON paths, intermediate results, or explicit execution instructions. State what the user wants rather than how the system should execute it. If no coherent natural query can justify the complete chain, reject the chain instead of forcing a query.

This ordering ensures that queries are constructed from executable solutions rather than generating queries first and subsequently searching for a matching tool chain.

We further assess each query–trajectory pair along five dimensions: *naturalness*, *coherence*, *necessity*, *dependency quality*, and *solvability*. Each dimension is scored on a five-point scale:

- **Naturalness**: whether the request resembles a plausible user query rather than a synthetic composition of API capabilities;
- **Coherence**: whether all requested information contributes to a single unified user goal;
- **Necessity**: whether each API in the trajectory contributes necessary information for satisfying the query;
- **Dependency quality**: whether adjacent API calls exhibit a meaningful semantic information dependency rather than only superficial schema compatibility; and
- **Solvability**: whether the task can be initiated from the information available in the query and completed using the verified trajectory.

The quality judge is explicitly instructed not to equate executability with benchmark quality:

> **Benchmark quality assessment.** The ground-truth tool chain has already succeeded end-to-end. Execution success only establishes that the chain can run; it does not imply that the resulting task is natural, coherent, necessary, or suitable for evaluation.
>
> Score the query–trajectory pair from 1 to 5 in terms of naturalness, coherence, necessity, dependency quality, and solvability. Every API should contribute to the user's actual objective, adjacent calls should exhibit meaningful information dependencies, and the query should contain sufficient information to initiate execution. Reject tasks that appear to have been constructed merely to connect otherwise unrelated APIs.

Instances are removed when *coherence*, *necessity*, *dependency quality*, or *solvability* receives a score of 2 or lower, or when the instance is judged to be an artificial composition of otherwise unrelated API calls. A final sanity check further verifies query quality, trajectory consistency, and executability. The retained instances constitute the T_0 datasets for DS and DL.

A.2 Controlled Dynamic Evolution

Starting from T_0, we construct two cumulative environment transitions, yielding T_1 and T_2, as illustrated in Fig. 2(b). We focus on API replacement, where the agent-visible interface of a selected API is replaced while its underlying functionality remains available.

Controlled Replacement. For each selected API, we construct an alternative interface that preserves its functional role but changes parts of its observable surface, including the tool/API identity, parameter schema, response fields, or their organization. When an affected API participates in a multi-step trajectory, the associated dependency mapping is updated accordingly.

For example, an upstream value exposed as `results[0].id` may instead appear as `results[0].reference_id`, requiring the downstream interface to consume `reference_id` rather than `id`. Such transformations alter the agent-visible interface and the dependency required to solve the task while preserving the underlying information flow.

The original user query is kept unchanged across timesteps. We update only the API surface and the corresponding ground-truth trajectory, and verify that the task remains executable after the transformation. Consequently, environment evolution changes how a task must be solved without changing what the user asks for.

Versioned API Adapter. To isolate interface evolution from changes in external services, all timesteps share the same underlying MirrorAPI backend. For each evolved interface, we define a versioned mapping between the agent-facing API and its corresponding backend API.

Before execution, the adapter converts the agent-visible arguments to the backend argument schema; after execution, it transforms the backend response into the response schema exposed at the current timestep. The mapping belongs only to the execution layer and is not exposed to the agent.

Formally, if an evolved API $\tilde{A}$ corresponds to backend API A, the runtime performs

$$(\tilde{A}, \tilde{x}) \xrightarrow{\text{adapter}} (A, x) \xrightarrow{\text{MirrorAPI}} y \xrightarrow{\text{adapter}} \tilde{y},$$

where $\tilde{x}$ and $\tilde{y}$ follow the current agent-facing interface. We validate the resulting mappings to ensure that interface evolution does not alter the underlying executable functionality.

Timestep Construction. API replacements are divided across two successive transitions. The first replacement group is introduced at T_1 and remains active thereafter, while the second group is additionally introduced at T_2. The environments therefore evolve cumulatively:

$$T_0 \rightarrow T_1 \rightarrow T_2.$$

The replacement groups are constructed to provide comparable coverage of affected benchmark tasks across transitions. Each timestep package contains the corresponding agent-facing API pool, unchanged user queries, and timestep-specific ground-truth trajectories. This controlled construction allows us to evaluate whether previously learned tool knowledge can be selectively updated as the environment changes.

A.3 Representative Dynamic Instances

Table 4 shows two representative instances from the final benchmark. The first corresponds to the example in Fig. 2(b): both APIs required by the task are replaced at T_1, while the user query remains unchanged. The second illustrates cumulative evolution, where the upstream API is replaced at T_1 and the downstream API is additionally replaced at T_2.

Table 5 further illustrates the versioned adapter for the examples above. The adapter preserves execution through the same MirrorAPI backend while presenting timestep-specific interfaces to the agent.

User Query	Env.	Ground-Truth API Chain	Dependency Mapping
"I want to find information about the show Breaking Bad. First search for it by name, then get the full details for the main Breaking Bad title from the search results."	T_0	`Watchmode: Autocomplete Search` → `Watchmode: Title Details`	`results[0].id` → `id`
	T_1	`Lumen Engine: Search Entertainment Titles` → `Lumen Engine: Get Title Metadata`	`results[0].reference_id` → `reference_id`
	T_2	`Lumen Engine: Search Entertainment Titles` → `Lumen Engine: Get Title Metadata`	`results[0].reference_id` → `reference_id`
"I want to watch Breaking Bad. Can you find out which streaming services it's available on?"	T_0	`Watchmode: Autocomplete Search` → `Watchmode: Title Streaming Sources`	`results[0].id` → `id`
	T_1	`Lumen Engine: Search Entertainment Titles` → `Watchmode: Title Streaming Sources`	`results[0].reference_id` → `id`
	T_2	`Lumen Engine: Search Entertainment Titles` → `Lumen Engine: Find Streaming Sources`	`results[0].reference_id` → `identifier`

Table 4: Representative DynaToolBench instances across environment timesteps. The user query remains unchanged while the agent-facing API chain and inter-API dependency mappings evolve. The second instance illustrates cumulative evolution: the upstream API is replaced at T_1, followed by the downstream API at T_2.

Agent-Facing API	Backend API	Input Mapping	Representative Output Mapping
`Lumen Engine: Search Entertainment Titles`	`Watchmode: Autocomplete Search`	`query` → `search_value`	`results[*].id` → `results[*].reference_id`
`Lumen Engine: Get Title Metadata`	`Watchmode: Title Details`	`reference_id` → `id`	`id` → `identifier`; selected response fields are reorganized
`Lumen Engine: Find Streaming Sources`	`Watchmode: Title Streaming Sources`	`identifier` → `id`	Backend response is exposed through the corresponding T_2 interface

Table 5: Representative versioned adapter mappings. The mappings belong to the execution layer and are hidden from the agent, allowing different agent-facing interfaces to reuse the same stable MirrorAPI backend.

B Implementation and Experimental Details

This appendix provides implementation and experimental details of CATE. We organize the description according to the components in Fig. 3, followed by baseline implementations, inference settings, and the evaluation protocol. For the LLM-assisted components, we provide the core prompt instructions together with their corresponding stages; fixed JSON schemas, formatting constraints, and instance-specific API content are omitted for readability.

B.1 CATE Implementation Details

Skill Representation and Initialization. As illustrated in Fig. 3(a), CATE initializes one atomic Skill for every API in the initial environment. Atomic initialization does not require an LLM call. The public API name and description provide the initial Skill title and *when-to-apply* field, and the Skill is bound to the corresponding single-API sequence.

A Skill maintains a human-readable *title*, a *when-to-apply* description for retrieval, a *principle* encoding reusable decision knowledge, *tool guidance* describing operational experience, and the associated ordered tool sequence. During interaction, Skills additionally accumulate execution evidence used for subsequent fitness updates and Reflection.

T_0 Evolution Budget. At T_0, each run is given a fixed budget of **100 evolution attempts**. Each attempt samples one active Skill/Chain and provides one opportunity for exploration or consolidation. For DynamicToolBench, an atomic parent is always routed to *Skill Expansion*, whereas an already composed Skill is routed to *Consolidation*. Thus, composite Skills are not recursively expanded in our experiments and contain at most two APIs.

Importantly, the budget counts evolution opportunities rather than only successful expansions. An Expansion attempt that does not produce a valid new Skill still consumes one of the 100 attempts.

Skill Expansion. Fig. 3(b) shows the Expansion process. Starting from an atomic Skill, CATE first asks the LLM to propose tools that may naturally precede or follow the current API. The proposed tool names are taken directly from the current Tool Pool and are used as candidates after closed-set validation; we do not perform a second embedding-based re-retrieval step. At most three prefix and three suffix candidates are proposed.

The core candidate-generation instruction is:

> **Expansion candidate proposal.** Expand the current tool chain by exactly one tool. A prefix tool should produce information that materially enables, determines, constrains, identifies, or parameterizes the current tool. A suffix tool should meaningfully consume, transform, or act on information produced by the current tool. Do not propose tools merely because they share a topic or could form two independent subtasks. Exact field-name matching is not required when a documented semantic dependency exists. Prefer plausible candidates at this stage; a later gate independently verifies their feasibility.

A second LLM call acts as a stricter dependency and quality gate:

> **Expansion feasibility verification.** Evaluate each candidate independently. A candidate is valid when it forms either (1) a documented sequential information dependency, where the earlier operation materially determines or parameterizes the later call, or (2) a coherent functional progression in which both tools provide distinct, non-redundant, and necessary results for one unified user goal. Reject topical co-occurrence, unnecessary pre-steps, redundant operations, and relations requiring undocumented transformations or external knowledge.

For each accepted pair, CATE constructs a composite Skill:

> **Composite Skill construction.** Describe the supplied tool sequence as one reusable Skill. Use *when-to-apply* only for retrieval applicability, *principle* for the core dependency or decision rule, and *tool guidance* for execution order, argument construction, output-to-input dependencies, and completion conditions. Preserve the supplied tool order and do not invent undocumented behavior.

At most two new composite Skills are materialized in one Expansion attempt. For every new Skill, CATE generates three diagnostic tasks and executes them against the actual tool environment before incorporating their evidence into the SkillBank.

The task-generation instruction emphasizes that the complete pair must be necessary for the generated request:

> **Expansion task generation.** Generate realistic user requests that naturally justify using the complete supplied tool sequence. Every tool must make a necessary contribution. For a sequential dependency, preserve the documented information flow and do not expose intermediate values that should be obtained by the earlier call. For a functional progression, each tool must satisfy a distinct meaningful part of the user goal. If the complete sequence cannot be justified naturally, reject the generation rather than fabricate a borderline task.

Consolidation. When a composite Skill is sampled again, CATE does not extend its tool sequence. Instead, it generates additional interaction tasks around the existing Skill to consolidate its applicability and execution experience. When useful, semantically nearby Skills may be provided as contrastive alternatives.

The core instruction is:

> **Consolidation task generation.** Generate realistic and diverse user requests for the target Skill using only the supplied Skill and API documentation. The target tool sequence must be a valid solution and every tool should make a necessary contribution. Do not expose Skill names, API names, the intended sequence, or downstream identifiers that should naturally be obtained during execution. Reject the generation rather than fabricate unsupported tasks.

Expansion therefore changes the structural contents of the SkillBank, whereas Consolidation accumulates additional experience for an existing composition.

Interaction Evidence and Fitness. As shown in Fig. 3(c), generated tasks are executed through the same tool environment used by the agent. The agent receives retrieved Skills and current API documentation, chooses one tool call at a time, and bases subsequent actions on actual observations.

A separate training-time completion judge determines whether the generated task was successfully completed. This signal is used only for Skill evolution and is independent of the formal evaluation judge described in Appendix B.6.

Its core instruction is:

> **Training completion judge.** Judge whether the agent actually completed the user's request using the executed trajectory and final answer. Return success only when all essential parts of the request are completed, the result is supported by actual observations, no essential failed or missing step prevents completion, and the final answer is consistent with the obtained results. Do not require a particular reference tool path.

For each Skill s, fitness is updated once per attempt from the success rate of the attributable interaction batch:

$$f_s^{(r)} = (1-\alpha) f_s^{(r-1)} + \alpha \hat{p}_s^{(r)}, \qquad \alpha = 0.3, \tag{3}$$

where $\hat{p}_s^{(r)}$ denotes the success rate of the tasks attributed to Skill s in evolution attempt r.

Interaction-Guided Reflection. Reflection is performed periodically rather than after every interaction. The scheduler reaches a Reflection checkpoint every five evolution attempts; at T_0, an actual Reflection pass is triggered when the checkpoint is supported by productive new interaction evidence.

A Skill is eligible for Reflection when it is active, its fitness is below 0.6, and at least three new interaction records have accumulated since its previous successful Reflection. For compact updates, each Reflection considers at most three recent failure cases and two successful cases.

Reflection first diagnoses the dominant reusable deficiency:

> **Reflection analysis.** Identify the primary generalizable deficiency that explains the supplied failures. Infer a reusable Skill-level problem rather than a solution to one specific task. Use only the supplied interaction evidence and API documentation, preserve behavior supported by successful cases, and focus on the affected field: retrieval applicability, decision principle, or execution guidance. Do not memorize task-specific identifiers, values, or outputs.

A second call revises only the diagnosed field:

> **Reflection rewrite.** Revise only the target Skill field using the diagnosis and recent evidence. Improve reusable behavior rather than memorizing individual cases. Preserve every unaffected Skill field and do not invent undocumented tool capabilities, parameters, outputs, or dependencies.

CATE additionally applies a lightweight Selection check after Skill mutation. A Skill is deactivated only when post-update evidence remains consistently weak and the same failure pattern persists. Selection is disabled during the direct replacement-adaptation stage and is not treated as a separate learning phase.

Experience Propagation to Atomic Skills. Expansion experience is also propagated back to the atomic Skills involved in a discovered pair. Only reusable knowledge about the atomic API itself is retained; partner-specific composition anecdotes are excluded.

The core summarization prompt is:

> **Atomic experience update.** Refine the atomic Skill using only reusable knowledge supported by newly accumulated Expansion experience. Useful knowledge may include applicability conditions, argument construction, output interpretation, downstream-useful return information, and reliable failure boundaries. Do not memorize individual queries, concrete argument values, observations, or partner-tool relationships. API documentation remains authoritative. If the new evidence contains no reusable atomic-tool knowledge, leave the Skill unchanged.

B.2 Local Adaptation under Environment Changes

Fig. 3(d) illustrates adaptation after a tool replacement. CATE compares adjacent public API environments and identifies active Skills whose tool sequences contain removed APIs. Unaffected Skills are retained without modification.

For replacement experiments, CATE does not run another general evolution loop at T_1 or T_2. Instead, each affected composite Skill is adapted *once*. For every removed API, the three most similar newly introduced APIs are retrieved using cosine similarity over their public documentation. The similarity ranking provides candidates only; the LLM makes the final replacement decision from the complete old and new API documentation.

The replacement prompt is:

> **Affected-Skill replacement.** One or more tools in the Skill are no longer available. For each removed tool, choose exactly one replacement from its supplied Top-3 candidate set using the complete old and candidate API documentation and the role of the affected step inside the Skill. Similarity rank is only a semantic reference and is not the correct answer.
>
> This is replacement rather than Skill reinvention. Preserve the Skill title, principle, and *when-to-apply* description; do not add or delete steps. Update execution guidance only as required by the selected APIs' documented parameters, outputs, and information flow.

The selected replacement is inserted at the affected sequence position programmatically. Thus, for composite Skills, the LLM selects the replacement and rewrites *tool guidance*, while the high-level Skill identity and unaffected sequence positions are protected. Invalid structured outputs may be retried up to three times, but the adaptation budget remains one successful adaptation operation per affected Skill.

Importantly, this replacement stage uses only the adjacent public API snapshots and the existing Skill state. Benchmark ground-truth trajectories and formal evaluation outcomes are not provided to the adaptation procedure.

Atomic Experience Inheritance. Every newly introduced API first receives a fresh atomic Skill constructed from its current public documentation. CATE then retrieves the three most similar retired atomic APIs and asks the LLM whether one of them provides transferable experience. The model may also select `NONE`.

The corresponding instruction is:

> **Atomic experience inheritance.** Given one newly initialized atomic Skill and several retired atomic Skills, identify the retired tool whose primary functional role most closely matches the new API, or select `NONE`. Similarity rank is only a semantic reference. Transfer only a few concise notes that are explicitly supported by the new API documentation, such as parameter caveats, usage tips, output interpretation, failure modes, or preconditions. Preserve the new Skill's title, principle, *when-to-apply*, and tool sequence. Do not transfer old-only capabilities or interface assumptions.

Atomic migration therefore updates only *tool guidance*; the retrieval identity of the new API remains anchored to its current documentation.

Skill Field	Before T_1 Adaptation	After T_1 Adaptation
Composite Skill: Comprehensive Title Information Retrieval		
Title	Comprehensive Title Information Retrieval	*unchanged*
Tool sequence	`ott details::title details` → `ott details::additional title details`	`Cedar Hub::get streaming title details` → `ott details::additional title details`
Tool guidance	"First, use `ott details::title details` with a valid IMDb ID to retrieve basic information such as title, genre, synopsis, and streaming availability. Then, use `ott details::additional title details` with the same IMDb ID ..."	"First, use `Cedar Hub::get streaming title details` with a valid IMDb ID to retrieve basic information such as title, genre, synopsis, and streaming availability. Then, use `ott details::additional title details` with the same IMDb ID ..."
Preserved fields	*title*, *when-to-apply*, and *principle* are unchanged.	
New Atomic Skill: get streaming title details		
Title	get streaming title details	*unchanged*
Tool sequence	`Cedar Hub::get streaming title details`	*unchanged*
Tool guidance	(empty at initialization)	Additional learned notes include: `title_ref` corresponds to the previous IMDb identifier; the response structure may differ; and the IMDb identifier should be validated before use.

Table 6: Representative Skill changes from the Qwen3-8B Dyna-Short T_1 run. For the composite Skill, only the affected API and interface-dependent guidance change. For the new atomic Skill, the current interface identity is preserved and only transferable execution notes are appended. Guidance text is shortened only for presentation.

Representative Skill Change. Table 6 shows an actual adaptation record from the Qwen3-8B Dyna-Short run at the $T_0 \rightarrow T_1$ transition. The table reports stored Skill fields rather than a natural-language interpretation of the change.

B.3 Skill Retrieval and Evaluation-Time Execution

During formal evaluation, the SkillBank is frozen and no new interaction experience is written back. For each user query, CATE retrieves the top-5 active Skills by cosine similarity using Qwen3-Embedding-0.6B. Skills with identical underlying tool sequences are deduplicated before retaining the top-5 retrieved Skills. The embedding representation of a Skill contains only its *title* and *when-to-apply* description; *principle* and *tool guidance* are used after retrieval rather than as retrieval features.

The retrieved Skills are treated as reusable experience rather than rigid complete programs. An execution may use a full retrieved Skill or reuse only relevant local guidance from it. The current API documentation and actual tool observations remain authoritative.

The core execution instruction is:

> **Evaluation execution.** Use the retrieved Skills as reusable experience for solving the user query. Reuse relevant applicability conditions, argument-construction rules, output fields, and output-to-input transitions while ignoring Skill steps irrelevant to the current request. Follow the current API documentation when constructing arguments. Execute one tool at a time and use actual observations to determine the next action. Do not invent tool outputs or unsupported facts, and stop once the available evidence is sufficient to answer the user.

We allow at most six tool-use decision steps as a rollout safety bound; the agent terminates earlier when sufficient evidence has been obtained.

B.4 Baseline Implementations

We compare CATE with Raw-Doc, EasyTool (Yuan et al., 2025), and ContDa (Wu et al., 2026). All methods operate on the same timestep-specific API environments and evaluation queries and use the same Qwen3 backbone within each model-scale comparison. Method-specific knowledge construction is followed by the same downstream execution and evaluation protocol.

Raw-Doc. Raw-Doc directly uses the current public API documentation without constructing or retaining learned tool-use experience. At every timestep, it is rebuilt from the current API pool and therefore provides a non-learning reference for the effect of the current interface alone.

EasyTool. EasyTool (Yuan et al., 2025) converts raw API documentation into concise tool instructions describing functionality and usage. We preserve its original instruction-generation procedure. Across environment transitions, instructions whose underlying public API document remains unchanged are reused, while changed or newly introduced APIs are processed from their current documentation. This reuse is an implementation optimization and does not introduce interaction-derived continual experience.

ContDa. ContDa (Wu et al., 2026) is the continual documentation-adaptation baseline. It carries learned documentation across timesteps and updates the evolving Tool Pool through relation-guided exploration and relation-aware adjustment. We preserve its original update procedure and prompts and modify only the benchmark-loading and execution interfaces required to operate on DynaToolBench.

The comparison therefore covers four forms of external tool knowledge: current raw documentation, concise API-level instructions, continually adapted documentation, and CATE's structured interaction-derived Skills.

B.5 Models and Experimental Settings

We conduct the main experiments with Qwen3-4B and Qwen3-8B. Within each model-scale experiment, the same backbone is used for CATE evolution, dynamic adaptation, and task execution, and is also used for the corresponding LLM-based baseline operations. The Qwen3 models are served locally with vLLM, with explicit reasoning mode disabled. We use temperature 0.3 for evolution, adaptation, and execution.

All reported results are averaged over three independent repetitions.

Table 7 summarizes the main CATE settings.

B.6 Evaluation Protocol and LLM Judge

For each evolved environment T_i ($i > 0$), we evaluate the inherited method state immediately after the environment transition, denoted T_i^-, and evaluate it again after the method has processed the new environment, denoted T_i^+. The continual evaluation order is

$$T_0 \text{ processing} \rightarrow T_0 \text{ evaluation} \rightarrow T_i^- \rightarrow \text{knowledge update} \rightarrow T_i^+.$$

Thus, T_i^- isolates the immediate effect of environment change, while T_i^+ measures the result after adaptation. The T_1 and T_2 values reported in the main tables correspond to T_1^+ and T_2^+ unless otherwise specified.

Non-continual methods process the current environment without inheriting learned cross-timestep experience, whereas continual methods retain their method-specific state from the previous timestep.

LLM Judge. Formal DynamicToolBench evaluation uses an isolated LLM judge. In the reported Qwen3 experiments, we use Gemini-3.8-Flash with temperature 0. The judge receives the user query, actual executed tool calls and observations, and the final answer. The benchmark ground-truth trajectory is deliberately excluded, so any valid tool-use path can receive full credit.

Configuration	Value
Backbone LLMs	Qwen3-4B / Qwen3-8B
Evolution / execution temperature	0.3
Reasoning mode	disabled
Experimental repetitions	3
Skill embedding model	Qwen3-Embedding-0.6B
Evaluation retrieval	top-5 Skills
Maximum rollout steps	6
T_0 evolution budget	100 attempts
Composite Skill length	2 APIs
Expansion proposal limit	≤ 3 prefix + ≤ 3 suffix
New composite Skills per Expansion	≤ 2
Diagnostic tasks per generated Skill	3
Fitness EMA coefficient α	0.3
Reflection checkpoint	every 5 attempts
Reflection fitness threshold	0.6
Minimum new Reflection evidence	3 interactions
Reflection evidence window	up to 3 failures + 2 successes
Replacement candidates	top-3 newly introduced APIs
Dynamic composite adaptation	once per affected Skill
Atomic inheritance candidates	top-3 retired APIs + `NONE`
Structured-output retry limit	3

Table 7: Main implementation settings used for CATE on DynaToolBench.

The judge first decomposes a query q into the smallest set of semantically independent user-facing requirements, $\mathcal{R}(q) = \{r_1, \ldots, r_{m_q}\}$. Internal tool-use steps are not counted as separate requirements unless they correspond to an explicit user request.

For each requirement, the judge assigns

$$s_{q,j} \in \{0,\ 0.5,\ 1\}, \tag{4}$$

where the three values correspond to the following cases:

- 1.0**—Fulfilled:** the requirement is correctly satisfied in the final answer and supported by actual execution evidence;
- 0.5**—Partial or environment-limited:** a correct subset is supported, or the agent selected a directly relevant tool with reasonable arguments but the returned environment response prevented full completion;
- 0.0**—Failed:** the requirement is unsatisfied because of an agent error, including a wrong or irrelevant tool, incorrect argument or identifier namespace, missing required action, wrong entity, unsupported claim, hallucination, or a final answer inconsistent with the observations.

The main judge instruction is:

> **Requirement-level evaluation.** Identify the smallest set of semantically independent user-facing requirements in the query. Evaluate each requirement using only the actual tool trajectory, observations, and final answer.
>
> Assign 1.0 when the requirement is fully and correctly satisfied with supporting execution evidence. Assign 0.5 only when a genuine correct subset is supported or when a semantically appropriate tool call with reasonable arguments is prevented from fully succeeding by the environment. Assign 0.0 when the failure is caused by an agent error, including an incorrect tool, wrong arguments or identifiers, missing required action, incorrect entity, unsupported claim, or contradiction with the observations.
>
> Do not require a particular reference tool path. A successful API status alone is not evidence that the user's requirement was satisfied. Before scoring, verify that the supporting observation concerns the requested entity and that identifiers passed between APIs use the namespace expected by the receiving tool.

For a query with m_q requirements, the task-level score is

$$S(q) = \frac{1}{m_q} \sum_{j=1}^{m_q} s_{q,j}. \tag{5}$$

For an evaluation set Q, the reported LLM-judged task accuracy is

$$\mathrm{TaskAcc}(Q) = \frac{1}{|Q|} \sum_{q \in Q} S(q) \times 100. \tag{6}$$

This requirement-level protocol gives partial credit only for genuine partial completion or environment limitations. In particular, an incorrect tool choice, wrong identifier propagation, unsupported final claim, or entity mismatch receives zero rather than partial credit.

B.7 Adaptation Cost Measurement

For the adaptation-cost comparison in the main text, we count only computation used to update method-side knowledge after an environment transition. For CATE, this includes composite affected-Skill replacement and atomic experience inheritance. For ContDa, we count the corresponding documentation-adaptation operations.

We report two measures: the total number of LLM tokens consumed during adaptation and the number of logical LLM calls. Task execution and the independent formal-evaluation judge are excluded from both measures. Consequently, the reported cost reflects the additional computation required to process an environment change rather than ordinary inference cost.

C Limitations

Our study focuses on controlled tool evolution over a finite collection of APIs, which enables reproducible evaluation but does not cover the full diversity and frequency of changes encountered in deployed tool ecosystems. Future work could evaluate continual tool learning in larger and more naturally evolving environments with real-world API updates. In addition, CATE is currently studied primarily in textual and API-based environments. Extending the modular evolution paradigm to multimodal tools, interactive software environments, and embodied agents is another promising direction.