EVOTOOLAGENT: CO-EVOLVING AUTONOMOUS TOOL AGENTS AND INTERACTION FOR ROBUST LLM TOOL USE

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents increasingly rely on external tools to perform complex tasks. However, tool interfaces, output formats, and availability may continuously change during deployment, making it difficult for agents designed for static tool environments to adapt reliably. We argue that the key to dynamic tool adaptation lies not only in recovering from failed tool calls, but also in identifying the sources of failure and directing failure feedback to the appropriate system components. To this end, we propose EvoToolAgent, an attribution-driven framework for agentic system evolution in dynamic tool environments. EvoToolAgent performs component-level failure attribution to localize errors arising during task execution to specific system components, thereby providing a unified mechanism for both within-task local recovery and cross-task system evolution. In addition, we introduce EvoToolBench-D, a benchmark designed to systematically evaluate the adaptability of agents to tool changes and failures. Experimental results demonstrate that EvoToolAgent consistently outperforms baselines across different backbone models and dynamic tool conditions, highlighting the importance of explicit failure diagnosis and structured recovery for tool-using agents in dynamic environments. Anonymous code is available at `https://anonymous.4open.science/r/EvoToolAgent-EF73`.

1 INTRODUCTION

Large language model (LLM) agents are evolving from isolated function calling toward decision-making systems that orchestrate multiple external tools over long-horizon tasks. As shown in Figure 1(a), conventional tool-augmented agents typically rely on a central agent interacting with a fixed set of passive tools. Effective tool-augmented reasoning increasingly requires multi-step planning, heterogeneous tool use, and coordination among specialized agents (Wu et al., 2024b; Gu et al., 2024; Ma et al., 2024; Shen et al., 2024). Consequently, the challenge is no longer limited to selecting the correct tool and arguments, but extends to maintaining execution state, using intermediate feedback, and making reliable decisions in open environments.

Real-world tool environments are inherently dynamic. Tool interfaces, descriptions, outputs, and availability may change during deployment (Xu et al., 2026), and even semantically equivalent interface variations can substantially affect tool-selection behavior (Faghih et al., 2025). Agents may also encounter newly introduced or previously unavailable tools after training (Wölflein et al., 2025). As a result, policies learned from static tool specifications can fail under schema shifts, output changes, tool failures, or tool replacement.

Recent work addresses such changes through environment interaction, error feedback, and test-time exploration, allowing agents to revise tool knowledge and invocation strategies during deployment (Chen et al., 2025a;b). Process-level supervision further shows that intermediate execution signals can help detect and prevent error propagation in multi-step trajectories (Deng et al., 2025). However, these approaches primarily adapt tool-use behavior, reasoning trajectories, or environment knowledge, while providing limited guidance on *which internal system component should be modified when a task fails*.

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.

This distinction is important because the same task-level failure may arise from different sources, including tool-interface errors, incorrect arguments, inappropriate routing, faulty task decomposition, or misuse of valid tool evidence. Recent work has begun to study failure attribution in LLM multi-agent systems (Zhang et al., 2025b), but identifying a responsible agent or failure step does not directly determine how failures in dynamic tool environments should trigger component-specific recovery and subsequent system optimization.

To address this gap, we propose **EvoToolAgent**, an attribution-driven framework for agentic system evolution in dynamic tool environments. As shown in Figure 1(b), EvoToolAgent encapsulates tools as autonomous Tool Agents and co-evolves them with Task Agents, the Interaction Graph, Reflective Memory, and the Archive. Component-level attribution identifies the primary source of failure, after which EvoToolAgent applies minimal-scope recovery while preserving unaffected intermediate states.

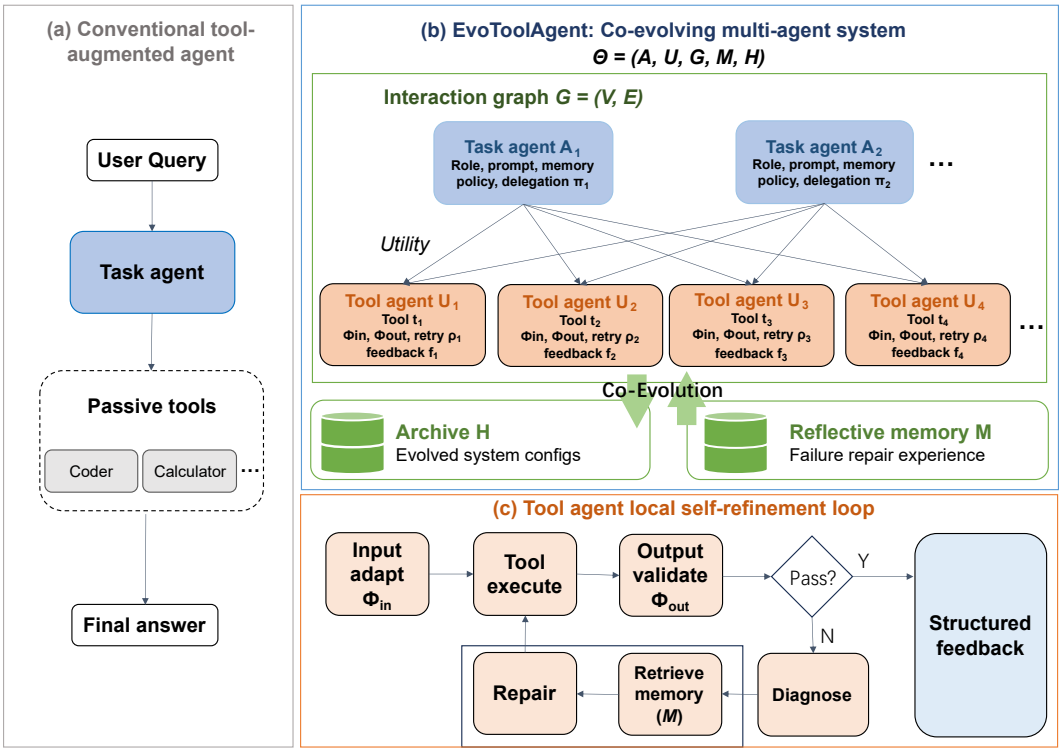


Figure 1: Overview of EvoToolAgent. (a) Conventional centralized tool use. (b) EvoToolAgent co-evolves Task Agents, Tool Agents, and their interactions. (c) Each Tool Agent performs local self-refinement for tool execution and recovery.

Our main contributions are as follows:

- We formulate dynamic tool adaptation as an **attribution-driven agentic system evolution** problem, where execution failures are localized to specific system components and used as structured signals for both local recovery and system optimization.
- We propose **EvoToolAgent**, which uses component-level attribution to connect minimal-scope recovery with cross-task system evolution.
- We introduce **EvoToolBench-D**, a benchmark covering schema shift, output shift, description noise, tool failure, tool replacement, and multi-step dependency, and jointly evaluate task correctness, tool execution, and failure recovery.

2 Related Work

Tool-Augmented Language Models and Agent Systems. Yao et al. (2023) established the reasoning–action–observation paradigm underlying conventional tool-augmented agents. Subse-

quent work extends tool use toward multi-step planning, heterogeneous APIs, and multi-agent collaboration (Qin et al., 2024; Patil et al., 2024; Wu et al., 2024b; Gu et al., 2024; Wang et al., 2024; Shen et al., 2024; Sun et al., 2024; Hong et al., 2024; Wu et al., 2024a; Zhang et al., 2025c). Most such systems, however, operate with relatively stable tool interfaces and predefined interaction structures.

Dynamic Tool Adaptation and Test-Time Agent Learning. Recent work adapts agents after deployment using execution feedback, environment interaction, memory, and test-time supervision (Chen et al., 2025b;a; Yu et al., 2025; Lou et al., 2026; Zhang et al., 2026; Faghih et al., 2025; Deng et al., 2025; Wang et al., 2025; He et al., 2025; Wölflein et al., 2025). LATS (Zhou et al., 2024) further explores alternative reasoning and action trajectories at test time. In contrast to trajectory-level adaptation or search, EvoToolAgent attributes failures to specific Agent–Tool components and uses these signals for targeted recovery and cross-task system updates.

Tool-Use Benchmarks. API-Bank (Li et al., 2023), ToolBench (Qin et al., 2024), ToolQA (Zhuang et al., 2023), and BFCL (Zhang et al., 2024) evaluate tool selection, API invocation, reasoning, and function calling, primarily under fixed or predefined interfaces. EvoToolBench-D complements them with controlled tool-environment perturbations and evaluates task correctness, tool execution, and failure recovery.

Automated Agent Design and System Evolution. Automated agent design optimizes prompts, roles, workflows, and multi-agent structures through search or evolution (Hu et al., 2025; Zhou et al., 2026; Zhang et al., 2025c; Hu et al., 2026; Zhang et al., 2025a; Yang et al., 2025; Zhou et al., 2025; Giloni et al., 2025; Zhang et al., 2025b). DSPy (Khattab et al., 2024) similarly optimizes language-model programs according to downstream objectives. EvoToolAgent instead uses explicit failure attribution to identify which system component should be modified, connecting localized recovery with subsequent system evolution.

Reflection, Memory, and Failure-Driven Optimization. Reflection and memory enable agents to reuse execution experience and refine future behavior (Shinn et al., 2023; Madaan et al., 2023; Gou et al., 2024; Guo et al., 2025; Yu et al., 2025; Lou et al., 2026; Ge et al., 2025; Salama et al., 2025; Kang et al., 2025; Xia et al., 2025; Chen et al., 2025c).

3 METHOD

3.1 OVERVIEW

We propose **EvoToolAgent**, an evolvable multi-agent framework for dynamic tool environments. The key idea is to treat tool-use failures as structured system feedback. The system first identifies the component responsible for a failure, performs localized recovery based on the attribution result, and further uses failure signals accumulated across tasks to selectively optimize the corresponding system components.

EvoToolAgent consists of a Task Agent, a set of autonomous Tool Agents, an Interaction Graph, Reflective Memory, and an Archive. We denote the system configuration as

$$\Theta = (A, U, G, M, H), \tag{1}$$

where A denotes the Task Agent responsible for task understanding, planning, and answer generation; U denotes the set of autonomous Tool Agents; G denotes the Interaction Graph between the Task Agent and Tool Agents; M denotes Reflective Memory that stores failure and recovery experience; and H denotes the Archive that maintains high-quality system configurations discovered during evolution. Given a task, the Task Agent delegates planned subtasks to Tool Agents through G; failures are attributed to the responsible component for localized recovery, and accumulated attribution signals subsequently guide system evolution.

3.2 AUTONOMOUS TOOL INTERACTION

EvoToolAgent encapsulates each underlying tool t_j as an autonomous Tool Agent:

$$u_j = \left(t_j, \phi_j^{\text{in}}, \phi_j^{\text{out}}, \rho_j\right), \tag{2}$$

where ϕ_j^{in} denotes the input adaptation policy, ϕ_j^{out} denotes the result validation policy, and ρ_j denotes the local recovery policy.

Upon receiving a request from the Task Agent, a Tool Agent sequentially performs input adaptation, tool execution, and result validation. Input adaptation handles missing arguments, renamed fields, and type changes, while result validation determines whether tool execution succeeded and whether the returned output supports the current subtask. As illustrated in Figure 1(c), each Tool Agent performs local self-refinement by diagnosing failed executions, retrieving relevant experience from Reflective Memory, and repairing the tool invocation.

A local repair step can be abstracted as

$$x_j^{(b+1)} = \rho_j \left(x_j^{(b)}, o_j^{(b)}, M \right), \tag{3}$$

where $x_j^{(b)}$ and $o_j^{(b)}$ denote the invocation arguments and tool feedback at the b-th attempt, respectively. Upon successful repair, the Tool Agent returns a validated result. After repeated failures, it returns structured failure information and transfers control to the system-level recovery procedure in §3.3.

The collaboration structure between the Task Agent and Tool Agents is represented by an explicit Interaction Graph $G = (V, E)$ with $V = A \cup U$. An edge $e_{ij} = (a_i, u_j)$ indicates that Task Agent a_i can delegate a subtask to Tool Agent u_j. Explicitly modeling this relation allows the system to distinguish between an unsuccessful invocation of the correct tool and incorrect routing to an inappropriate tool. The graph is updated from execution feedback; detailed Edge Utility computation and update rules are given in Appendix D.

3.3 Attribution-Driven Reflection and Recovery

Failures may arise beyond local tool execution, including planning, routing, and result utilization, motivating component-level attribution. EvoToolAgent performs component-level failure attribution over failed trajectories. We consider four primary failure sources $\mathcal{C} = \{C_A, C_U, C_G, C_F\}$, where C_A denotes task-understanding or planning errors in the Task Agent, C_U denotes execution or adaptation errors in Tool Agents, C_G denotes routing or dependency errors in the Interaction Graph, and C_F denotes errors in result utilization or final answer generation. Given a failed execution trajectory τ, the attribution module identifies the primary failure component:

$$c^* = \text{Attribute}(\tau), \qquad c^* \in \mathcal{C}. \tag{4}$$

Attribution considers the execution plan, tool-call status, environment errors, intermediate results, and validation states. The failure context, attribution result, recovery action, and recovery outcome are stored in Reflective Memory and retrieved when similar failures occur. Given the attribution result and retrieved experience, the system selects a recovery action $a_{\text{repair}} = \text{Recover}(\tau, c^*, M)$. The formal definitions of $\text{Attribute}(\cdot)$, $\text{Recover}(\cdot)$, and the associated operators are given in Appendix D.

For a multi-step task, if a failure occurs at plan node z_l, the system replans only the associated dependency subgraph:

$$P' = \text{ReplaceSubplan}(P, G_{\text{dep}}(z_l), \text{Replan}(G_{\text{dep}}(z_l))) \,. \tag{5}$$

ReplaceSubplan preserves the relative order of unaffected steps and replaces only the affected pending dependency subgraph with a dependency-consistent replanned sequence. Thus, local failures do not invalidate unaffected completed steps. Validated tool outputs are retained in an evidence set $\mathcal{V} = \{o_i : \text{Verify}(o_i) = 1\}$, and the final answer is generated as $\hat{y} = \text{Synthesize}(q, \mathcal{V})$. If sufficient evidence has already been obtained but final-answer validation fails, EvoToolAgent preserves $\mathcal{V}$, attributes the failure, and applies the corresponding minimal-scope recovery action rather than bypassing attribution.

3.4 Attribution-Driven System Evolution

While online recovery adapts the system within individual tasks, EvoToolAgent further uses failed trajectories accumulated during population evaluation to optimize the complete system configuration

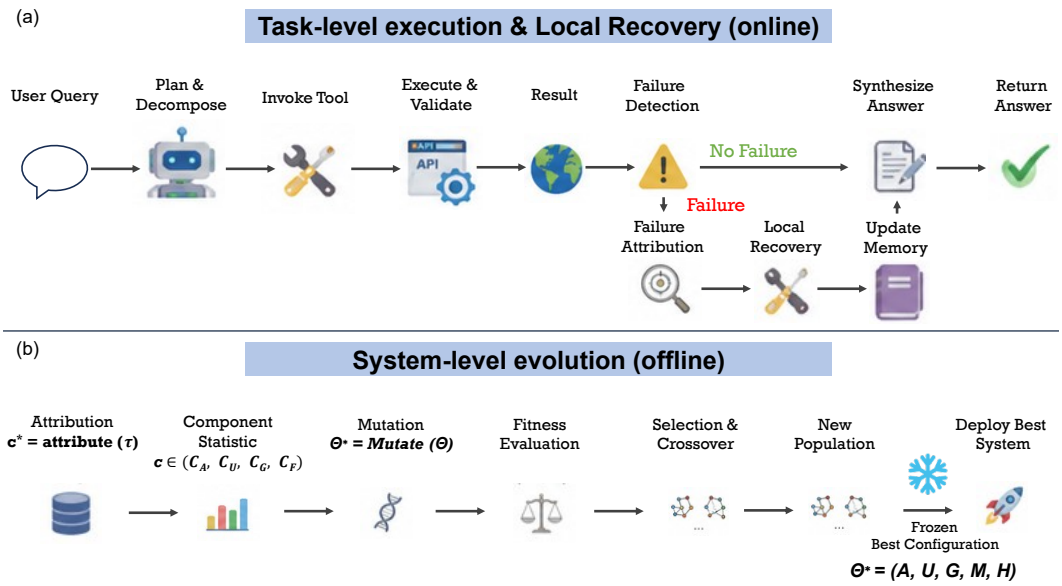


Figure 2: Two-level adaptation in EvoToolAgent: (a) attribution-driven online recovery and (b) attribution-guided offline system evolution.

$\Theta = (A, U, G, M, H)$. For each such trajectory τ, attribution identifies the responsible component $c^* = \text{Attribute}(\tau)$, and the aggregated attribution signal guides bounded targeted mutation during population update:

$$c^* = \text{Attribute}(\tau), \qquad \Theta' = \text{Mutate}_{c^*}(\Theta). \tag{6}$$

When $c^* = C_A$, mutation targets the planning and execution policies of the Task Agent; when $c^* = C_U$, it modifies the corresponding Tool Agent's adaptation, validation, or recovery policies; when $c^* = C_G$, it updates the Interaction Graph; and when $c^* = C_F$, it adjusts the result-utilization and answer-generation process. Appendix D specifies how $\text{Mutate}_{c^*}(\cdot)$ acts on each subsystem and how attribution signals connect online recovery with offline evolution. Thus, component-level attribution provides a shared signal across the two adaptation scales: it determines where to recover online and what to evolve offline.

4 EvoToolBench-D

In real-world deployments of tool-augmented large language models, tools are inherently non-static. API arguments may change across versions, returned results may adopt new data structures, services may become temporarily unavailable, and existing tools may even be replaced by new interfaces. Following prior work on evolving tool environments (Chen et al., 2025b), we evaluate not only whether an agent can complete tasks in a fixed tool environment, but also whether it can identify failures and recover execution after tool changes or failures.

To this end, we introduce EvoToolBench-D (Dynamic Tool-Augmented Agent Benchmark), a benchmark for evaluating tool-augmented agents in dynamic tool environments. EvoToolBench-D injects different types of perturbations into the tool-execution environment and records the resulting perturbations and failures. This design enables evaluation beyond final task accuracy, including tool-execution success and recovery from execution failures.

EvoToolBench-D contains 800 tasks across four complementary subsets covering dynamic question answering, API invocation, multi-tool workflows, and code repair. EvoToolQA-D contains 300 instances adapted from ToolQA (Zhuang et al., 2023), targeting questions that require external knowledge retrieval and tool operations. It covers heterogeneous data sources, including flight, coffee, Yelp, Airbnb, DBLP, SciREX, and agenda. EvoToolAPI-D contains 200 tasks derived from API-Bank (Li et al., 2023). API-Bank provides executable API environments and human-annotated tool-use dialogues; we use its samples to construct API invocation tasks that primarily evaluate tool

selection, argument generation, and argument adaptation under interface changes. EvoToolFlow-D contains 200 multi-tool tasks designed to evaluate planning and execution when dependencies exist among multiple tools. This subset uses convertible samples from ToolBench (Qin et al., 2024) as seeds and supplements them with synthetic tasks to reach the target size. Finally, EvoToolCode-D contains 100 synthetic code tasks covering code generation, Python execution, testing, and failure-driven code repair. This subset is designed to provide explicit execution feedback for evaluating tool failures and multi-step recovery. Detailed construction, conversion rules, and validation procedures are provided in Appendix B.

5 EXPERIMENTS

We evaluate three research questions:

- **RQ1: Dynamic Tool Adaptation.** Under the same backbone model, tool environment, and test-time budget, can EvoToolAgent complete tasks and recover from tool failures more effectively than existing tool-augmented agents?
- **RQ2: Component Contribution.** How do different recovery and collaboration mechanisms in EvoToolAgent affect task accuracy and dynamic tool recovery?
- **RQ3: Generalization.** Can an Agent–Tool system configuration evolved on the training and development sets generalize to unseen test tasks after being frozen?

To answer these questions, we use two complementary evaluation protocols. **Protocol A** fixes the global configuration of all methods while allowing method-specific local test-time adaptation under a unified execution budget. Unless otherwise specified, the main results are reported under Protocol A. **Protocol B** performs offline system evolution on the training and development sets and evaluates the resulting frozen configuration on the held-out test set, allowing us to study the generalization of system-level evolution.

5.1 EXPERIMENTAL SETUP

Implementation Details. We conduct experiments on EvoToolBench-D introduced in Section 4. For the main comparison under Protocol A, we use an independently constructed test set containing 160 tasks. All methods are evaluated under the same task-level perturbations defined in Section 4 and under a unified execution budget. During evaluation, methods may perform their own local adaptation, such as self-refinement or memory updates, but population-level evolution operations, including mutation, crossover, and selection, are disabled. For Protocol B, we use a standard train/dev/test split containing 560 training tasks, 80 development tasks, and 160 test tasks. EvoToolAgent performs system configuration search only on the training and development sets and selects the best configuration Θ^* according to development fitness. Unless otherwise specified, all methods use the same backbone model and inference configuration. We evaluate both open-source and proprietary backbones, including Qwen3.5-35B, Llama3-8B, and Claude Opus 5, to examine performance across different model families and scales.

Baselines. We compare EvoToolAgent against **Direct LLM**, **ReAct** (Yao et al., 2023), **Static Tool Agent**, **Self-Refine** (Madaan et al., 2023), **Reflexion** (Shinn et al., 2023), **Critic-Style** (Gou et al., 2024), **ToolEVO-Style** (Chen et al., 2025b), **MCTS-Only**, **Random Search** (Hu et al., 2025), **Prompt-Only Evolution** (Hu et al., 2025), **Tool-Only Evolution** (Hu et al., 2025), **Graph-Only Evolution** (Hu et al., 2025; Zhang et al., 2025a), and **Workflow Search** (Zhang et al., 2025c; Zhou et al., 2025). We additionally include component-restricted and workflow-search controls; their search spaces and Protocol-A execution behavior are specified in Appendix A.

5.2 MAIN RESULTS

We first compare EvoToolAgent with all baselines under Protocol A. All methods are evaluated on the same tasks with identical backbone models, task perturbations, and execution budgets, and results are averaged over three random seeds. We report task accuracy, recovery rate, and execution-level tool success (Exec. ETS); formal definitions are provided in Appendix B.9.

Table 1: Main results under Protocol A. Results are averaged over three random seeds. Best results are shown in **bold**, and second-best results are underlined.

Method	Acc.	Recovery	Exec. ETS
Direct LLM	12.5±1.1	0.0±0.0	–
ReAct	22.7±4.6	11.9±3.2	28.2±2.1
Self-Refine	21.7±0.7	20.2±0.7	28.1±1.5
Reflexion	41.5±10.5	35.2±11.9	68.9±2.6
Critic-Style	20.2±1.8	18.7±1.8	15.6±0.9
Static Tool Agent	42.1±11.0	35.8±12.4	68.9±2.6
ToolEVO-Style	16.9±3.2	3.6±0.1	46.5±0.4
MCTS-Only	24.4±3.2	12.3±0.5	35.0±1.1
Random Search	41.7±11.2	35.2±12.4	68.9±2.6
Prompt-Only Evolution	41.5±9.4	35.2±10.7	68.9±2.6
Tool-Only Evolution	41.7±10.6	35.2±11.8	68.9±2.6
Graph-Only Evolution	41.0±10.1	34.8±11.5	68.9±2.6
Workflow Search	10.6±0.0	5.3±0.0	21.7±1.4
EvoToolAgent	**50.6±0.0**	**50.0±0.6**	**71.6±0.7**

As shown in Table 1, EvoToolAgent achieves the best overall task accuracy (50.6) and recovery rate (50.0) under Protocol A. Compared with static tool-use and self-feedback baselines, EvoToolAgent more effectively converts tool-execution feedback into subsequent repair actions. The improvement is particularly pronounced in recovery rate, indicating that explicit recovery mechanisms are important when agents operate under dynamic tool changes and execution failures.

ReAct incorporates tool observations into subsequent reasoning, while Self-Refine and Reflexion use failure feedback for iterative refinement. EvoToolAgent differs by explicitly attributing failures to system components and connecting the resulting attribution to targeted repair operations, providing a structured mechanism for determining whether recovery should act on tool execution, routing, planning, or result utilization. To assess backbone robustness, we repeat Protocol A with Qwen3.5-35B, Llama3-8B, and Claude Opus 5 under the same evaluation setting. Table 2 reports the results.

Table 2: Results across different backbone models under Protocol A. Best results within each backbone are shown in **bold**, and second-best results are underlined.

Backbone	Method	Acc.	Recovery	Exec. ETS
Qwen3.5-35B	ReAct	22.7±4.6	11.9±3.2	28.2±2.1
	ToolEVO-Style	16.9±3.2	3.6±0.1	46.5±0.4
	EvoToolAgent	**50.6±0.0**	**50.0±0.6**	**71.6±0.7**
Llama3-8B	ReAct	6.0±0.7	1.7±0.8	34.8±2.0
	ToolEVO-Style	18.8±0.0	3.0±0.0	47.5±0.0
	EvoToolAgent	**38.3±0.7**	**37.0±1.0**	**66.5±0.8**
Claude Opus 5	ReAct	24.4±1.3	33.7±1.4	59.4±1.1
	ToolEVO-Style	27.1±2.0	40.5±0.7	65.3±0.3
	EvoToolAgent	**73.1±1.3**	**71.4±0.9**	**97.5±0.1**

Across all three backbones, EvoToolAgent achieves the highest task accuracy and recovery rate among the compared methods. The improvement is observed for both open-source models and the proprietary Claude Opus 5 backbone, suggesting that the effectiveness of the framework is not restricted to a single model family or scale. Notably, EvoToolAgent reaches 73.1 accuracy and 71.4 recovery with Claude Opus 5, while maintaining substantial gains over ReAct and ToolEVO-Style with Qwen3.5-35B and Llama3-8B.

5.3 Ablation Study

We ablate two execution-time components: the Recovery Loop, which performs bounded repair and partial replanning after detected failures, and Validators, which check intermediate observations and candidate answers.

Removing the Recovery Loop causes a substantial decrease in both task accuracy and recovery rate, with drops of 8.1 and 18.1 points, respectively. This result supports the role of structured repair and partial replanning in returning failed executions to a usable trajectory. Removing Validators has a smaller but consistent negative effect on both accuracy and recovery, suggesting that explicit validation provides an additional safeguard during dynamic execution.

Table 3: Ablation study of EvoToolAgent. Best results are shown in **bold**, and second-best results are underlined. Values in parentheses denote changes relative to the full model.

Variant	Acc. ↑	Recovery ↑	Exec. ETS ↑
EvoToolAgent (Full)	**50.6**	**49.4**	71.9
w/o Recovery Loop	42.5 (−8.1)	31.3 (−18.1)	**90.9** (+19.0)
w/o Validators	50.0 (−0.6)	48.7 (−0.7)	69.5 (−2.4)

5.4 Generalization of Evolved Agent–Tool Systems

For Protocol B, the development-selected configuration Θ^* is frozen at test time: the Task Agent, Tool Agents, Interaction Graph, Reflective Memory, Archive, and Edge Utility are fixed, and no mutation, crossover, selection, memory write, or graph update occurs during held-out evaluation. Table 4 reports frozen-test performance.

Table 4: Generalization of the evolved Agent–Tool system from the development set to the held-out test set. Gen. Gap is the signed quantity $F_{\text{dev}} - F_{\text{frozen-test}}$; values closer to zero indicate smaller gaps.

Setting	Acc.	Fitness	Recovery	Gen. Gap
Dev Best (Θ^*)	46.25	0.4049	20.82	–
Frozen Test	43.75	0.3880	18.56	0.0169

Table 4 evaluates whether the configuration selected through offline evolution transfers to unseen tasks after all adaptive system states are frozen. The frozen configuration achieves a fitness of 0.3880 on the held-out test set, compared with 0.4049 for the development-selected configuration, corresponding to a generalization gap of 0.0169. Its task accuracy remains 43.75 after freezing. These results indicate that the configuration obtained through offline evolution retains most of its development-stage performance on unseen tasks without requiring continued mutation, crossover, memory writing, or graph updates at test time.

6 Discussion

This work studies how agents can effectively adapt to tool changes and execution failures in dynamic tool environments. Our results show that relying solely on the tool-use and reflection capabilities of language models is insufficient to robustly handle changes such as schema shifts, output shifts, tool failures, and tool replacements. Instead, component-level failure attribution and structured recovery play a critical role in dynamic tool adaptation.

To address this challenge, we introduce EvoToolAgent, which integrates autonomous Tool Agents, an explicit Interaction Graph, component-level failure attribution, Reflective Memory, and minimal-scope recovery. This design connects local tool adaptation within individual tasks with system-level evolution across tasks. We further introduce EvoToolBench-D, a dynamic tool benchmark that systematically evaluates task completion, tool execution, and failure recovery.

Our experiments demonstrate that EvoToolAgent consistently improves task accuracy and recovery performance across different backbone models and dynamic tool conditions. Ablation results further show that both the Recovery Loop and Validators contribute to robust execution, while Protocol B indicates that configurations obtained through offline evolution retain substantial performance when frozen and evaluated on unseen tasks. Together, these results suggest that explicitly modeling failure attribution and recovery provides a practical foundation for building more adaptive and evolvable tool-augmented agents.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic datasets, implementing methods, editing the research paper for readability, summarizing or analyzing existing literature, and formatting references. We have not used generative AI tools for developing theoretical models or conceptual frameworks; making mathematical claims or providing key elements of proofs of mathematical claims; proposing or refining hypotheses, designing research methods or experiments, providing feedback, or interpreting results; proposing the title or keywords of the research paper; creating or modifying scientific figures or images; writing sections of the research paper; discovering research topics or identifying research gaps; proposing experimental parameters; or suggesting the structure of the research paper. Cleaning and reformatting datasets, developing survey or interview questions, producing physical artifacts, and transcribing recordings of research materials are not applicable to this work.

We have reviewed all AI-assisted work. In particular, we manually checked AI-assisted literature summaries against the original sources, verified that formatted references match the cited works, and reviewed all AI-assisted code, generated data, and manuscript text for factual accuracy, consistency with our experiments, and readability. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI. The manuscript and code have been manually reviewed by the authors.

ETHICS STATEMENT

This work does not raise additional ethics concerns beyond standard machine learning research.

REPRODUCIBILITY STATEMENT

Code, data, and experiment configurations have been uploaded to an anonymous repository at `https://anonymous.4open.science/r/EvoToolAgent-EF73`.

ACKNOWLEDGMENTS

REFERENCES

Arthur Chen, Zuxin Liu, Jianguo Zhang, Akshara Prabhakar, Zhiwei Liu, Shelby Heinecke, Silvio Savarese, Victor Zhong, and Caiming Xiong. Grounded test-time adaptation for LLM agents. *arXiv preprint arXiv:2511.04847*, 2025a.

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 *International Conference on Learning Representations*, 2025b.

Justin Chen, Archiki Prasad, Swarnadeep Saha, Elias Stengel-Eskin, and Mohit Bansal. MAgICoRe: Multi-agent, iterative, coarse-to-fine refinement for reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 32663–32686, Suzhou, China, 2025c. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1660.

Yuchen Deng, Shichen Fan, Naibo Wang, Xinkui Zhao, and See-Kiong Ng. AgentPro: Enhancing LLM agents with automated process supervision. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 9981–10006, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.506.

Kazem Faghih, Wenxiao Wang, Yize Cheng, Siddhant Bharti, Gaurang Sriramanan, Sriram Balasubramanian, Parsa Hosseini, and Soheil Feizi. Tool preferences in agentic LLMs are unreliable. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20954–20969, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1060.

Yubin Ge, Salvatore Romeo, Jason Cai, Monica Sunkara, and Yi Zhang. SAMULE: Self-learning agents enhanced by multi-level reflection. In *Proceedings of the 2025 Conference on Empirical*

Methods in Natural Language Processing, pp. 16591–16610, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.839.

Amit Giloni, Chiara Picardi, Roy Betser, Shamik Bose, Aishvariya Priya Rathina Sabapathy, and Roman Vainshtein. CAIR: Counterfactual-based agent influence ranker for agentic AI workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 18940–18966, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.958.

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 *International Conference on Learning Representations*, 2024.

Yu Gu, Yiheng Shu, Hao Yu, Xiao Liu, Yuxiao Dong, Jie Tang, Jayanth Srinivasa, Hugo Latapie, and Yu Su. Middleware for LLMs: Tools are instrumental for language agents in complex environments. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7646–7663, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.436.

Yifu Guo, Jiaye Lin, Huacan Wang, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, and Mingguang Chen. SE-Agent: Self-evolution trajectory optimization in multi-step reasoning with LLM-based agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Yufei He, Ruoyu Li, Alex Chen, Yue Liu, Yulin Chen, Yuan Sui, Cheng Chen, Yi Zhu, Luca Luo, Frank Yang, and Bryan Hooi. Enabling self-improving agents to learn at test time with human-in-the-loop guidance. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1625–1653, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-industry.115.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025.

Yuntong Hu, Matthew Trager, Yuting Zhang, Yi Zhang, Shuo Yang, Wei Xia, and Stefano Soatto. Evolutionary generation of multi-agent systems. *arXiv preprint arXiv:2602.06511*, 2026.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory OS of AI agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25961–25970, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1318.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhang Zhang, Keshav Santhanam, Sri Vardhamanan, Stefano Haussmann, Hanna Miller, Syed Zaidi, Benjamin Van Durme, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations*, 2024.

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

Zhanzhi Lou, Hui Chen, Yibo Li, Qian Wang, and Bryan Hooi. Learning to learn-at-test-time: Language agents with learnable adaptation policies. *arXiv preprint arXiv:2604.00830*, 2026.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, and Aixin Sun. SciAgent: Tool-augmented language models for scientific reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15701–15736, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.880.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

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, 2024.

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.

Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. MemInsight: Autonomous memory augmentation for LLM agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33136–33152, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1683.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small LLMs are weak tool learners: A multi-LLM agent. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 16658–16680, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.929.

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, 2023.

Jimin Sun, So Yeon Min, Yingshan Chang, and Yonatan Bisk. Tools fail: Detecting silent errors in faulty tools. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14272–14289, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.790.

Hongru Wang, Rui Wang, Boyang Xue, Heming Xia, Jingtao Cao, Zeming Liu, Jeff Z. Pan, and Kam-Fai Wong. AppBench: Planning of multiple APIs from various APPs for complex user instruction. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15322–15336, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.856.

Jiayin Wang, Zhiqiang Guo, Weizhi Ma, and Min Zhang. How far can LLMs improve from experience? measuring test-time learning ability in LLMs with human comparison. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25677–25691, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1304.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. LLM agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26092–26130, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1266.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Conference on Language Modeling*, 2024a.

Qinzhuo Wu, Wei Liu, Jian Luan, and Bin Wang. ToolPlanner: A tool augmented LLM for multi granularity instructions with path planning and feedback. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 18315–18339, Miami, Florida, USA, 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.1018.

Yu Xia, Yiran Jenny Shen, Junda Wu, Tong Yu, Sungchul Kim, Ryan A. Rossi, Lina Yao, and Julian McAuley. SAND: Boosting LLM agents with self-taught action deliberation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 3062–3077, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.152.

Haoyuan Xu, Chang Li, Xinyan Ma, Xianhao Ou, Zihan Zhang, Tao He, Xiangyu Liu, Zixiang Wang, Jiafeng Liang, Zheng Chu, Runxuan Liu, Rongchuan Mu, Ming Liu, and Bing Qin. The evolution of tool use in LLM agents: From single-tool call to multi-tool orchestration. *arXiv preprint arXiv:2603.22862*, 2026.

Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, and Weinan Zhang. AgentNet: Decentralized evolutionary coordination for LLM-based multi-agent systems. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Xiao Yu, Baolin Peng, Vineeth Vajipey, Hao Cheng, Michel Galley, Jianfeng Gao, and Zhou Yu. ExACT: Teaching AI agents to explore with reflective-MCTS and exploratory learning. In *International Conference on Learning Representations*, 2025.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 75834–75852. PMLR, 2025a.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76583–76599. PMLR, 2025b.

Tianjun Zhang, Fan Yang, Peitian Huang, Jintian Zhang, Guanhua Zhou, Huan Liu, Banghua Li, Tong Zhang, Joseph E. Gonzalez, and Ion Stoica. Berkeley function calling leaderboard. *arXiv preprint arXiv:2408.04667*, 2024.

Yaolun Zhang, Xiaogeng Liu, and Chaowei Xiao. MetaAgent: Automatically constructing multi-agent systems based on finite state machines. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 75667–75694. PMLR, 2025c.

Yunxiang Zhang, Yiheng Li, Ali Payani, and Lu Wang. AdaMEM: Test-time adaptive memory for language agents. *arXiv preprint arXiv:2606.05684*, 2026.

Andy Zhou, Kevin Yan, Michiel Shlapentokh-Rothman, Hongxin Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies. In *International Conference on Learning Representations*, 2026.

Heng Zhou, Hejia Geng, Xiangyuan Xue, Li Kang, Yiran Qin, Zhiyong Wang, Zhenfei Yin, and Lei Bai. ReSo: A reward-driven self-organizing LLM-based multi-agent system for reasoning tasks. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15979–15998, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.808.

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, 2023.

CONTENTS OF APPENDIX

A Implementation Details **16**

A.1 Experimental Environment 16

A.2 Shared Runtime and Hyperparameters 16

A.2.1 Shared Runtime Configuration 16

A.2.2 Planner and Tool Retrieval 16

A.2.3 Adaptation Budget 17

A.2.4 Evolution Hyperparameters 17

A.2.5 Edge Utility 17

A.2.6 MCTS Configuration 18

A.3 Baseline Implementations 18

A.3.1 Baseline-Specific Defaults 19

A.3.2 Evolution-Named Baselines under Protocol A 20

A.3.3 Capabilities Removed from Individual Baselines 20

A.4 Protocol A: Fair Test-Time Adaptation 20

A.5 EvoToolAgent Execution Pipeline 21

A.6 Protocol B: Offline Evolution and Frozen Evaluation 21

A.6.1 Frozen-System Generalization Results 22

A.7 Core Algorithms 23

A.7.1 Tool Agent Self-Refinement 23

A.7.2 Component-Level Credit Assignment 23

A.7.3 Fitness Function 24

A.7.4 MCTS Tool Repair 24

A.7.5 Population Evolution 24

A.8 Tool Environment 25

A.9 Architecture Configuration 25

B EvoToolBench-D Details **25**

B.1 Benchmark Design 25

B.2 Benchmark Composition and Data Splits 26

B.2.1 Four Subsets 26

B.2.2 Synthetic Task Construction 26

B.2.3 Source-Dataset Conversion and Inclusion Criteria 27

B.2.4 Protocol-Specific Splits 27

B.3 Perturbation Distribution 28

B.3.1 Perturbation Definitions 28

B.4 Held-Out Test Set Statistics 28

B.5 Unified Task Schema 29

B.6 Tool Specifications . . . 29
B.6.1 Example Tool Schema . . . 29
B.7 Perturbations and Evaluated Components . . . 30
B.8 Research Questions and Benchmark Coverage . . . 31
B.9 Evaluation Metrics . . . 31

C Prompt Collection **32**
C.1 Task Agent . . . 32
C.1.1 Task Agent System Prompt (default) . . . 32
C.1.2 Tool Planning (`plan_tools`) . . . 32
C.1.3 Answer Synthesis (`synthesize_answer`) . . . 32
C.2 Tool Agent . . . 32
C.2.1 Tool Agent System Prompt (default) . . . 32
C.2.2 Parameter Adaptation (`adapt_input`) . . . 32
C.2.3 Failure Diagnosis (`diagnose`) . . . 32
C.3 Tool Parameter Inference and Repair . . . 32
C.3.1 Parameter Utility System Prompt . . . 32
C.3.2 Parameter Generation (`infer_params_with_llm`) . . . 33
C.3.3 Parameter Repair (`repair_params_with_llm`) . . . 33
C.4 Baseline Methods . . . 33
C.4.1 Direct LLM . . . 33
C.4.2 Zero-Shot ReAct . . . 33
C.4.3 Self-Refine — Critique . . . 33
C.4.4 Self-Refine — Refinement . . . 33
C.4.5 Reflexion — Reflection Injection . . . 33
C.4.6 Reflexion — Lesson Writing . . . 33
C.4.7 Baseline Shared Synthesis . . . 34
C.4.8 CRITIC-Style Correction . . . 34
C.4.9 Prompt-Only Evolution Candidates . . . 34
C.5 Architecture Layer (Failure Detection and Validation) . . . 34
C.5.1 Failure Detector (hybrid LLM branch) . . . 34
C.5.2 Observation Validator (hybrid LLM branch) . . . 34
C.6 Evolution Mutation Suffixes . . . 34

D Formal Specification: Attribution, Recovery, and Evolution **35**
D.1 Symbol Table . . . 35
D.2 Execution Trajectory . . . 35
D.3 Tool Agent Local Repair . . . 35
D.4 Interaction Graph Updates . . . 35

D.5 Failure Detection 36
D.6 Component Attribution: $\mathrm{Attribute}(\tau)$ and $\mathrm{Attribute}(\tau_{\leq l})$ 36
D.7 Reflective Memory 36
D.8 Recovery Operator: $\mathrm{Recover}(\tau, c^*, M)$ 36
D.9 Partial Replan and Evidence Preservation 37
D.10 Within-Task Recovery Loop 37
D.11 Attribution-Driven System Evolution 37
D.12 Worked Example 39

E Case Studies and Additional Results **39**
E.1 Case 1: Evidence-Locked Extractive Synthesis 39
E.2 Case 2: Multi-Tool Execution with Recovery 39
E.3 Case 3: Failure in a Multi-Step Workflow 40
E.4 Case 4: Tool Replacement 40
E.5 Case 5: Comparison with Non-Adaptive Tool Use 41
E.6 Case 6: Failure on an API-Calling Task 41

A Implementation Details

This appendix provides implementation details for EvoToolAgent, including the experimental environment, shared runtime configuration, adaptation and evolution hyperparameters, baseline implementations, evaluation protocols, core algorithms, and architecture settings.

To remain consistent with the terminology used in the main paper, we use **Protocol A** to denote the fair test-time adaptation setting used for the main comparison, and **Protocol B** to denote offline system evolution followed by frozen evaluation on unseen test tasks.

A.1 Experimental Environment

All experiments are conducted in a controlled execution environment. Open-source backbones are served through Hugging Face. The main experiments use Qwen3.5-35B as the primary backbone, with Llama3-8B included in the multi-model evaluation reported in the main paper. We evaluate Claude Opus 5 via the Anthropic API using the model identifier `claude-opus-5`. All experiments were conducted in September 2026. We use the same input prompts and evaluation protocol as for the other evaluated models to ensure a fair comparison.

Table 5 summarizes the main experimental environment.

Table 5: Experimental environment.

Item	Configuration
Python	≥ 3.10
Dependencies	`pyyaml`, `numpy`, `pandas`, `pytest`
Open-source LLM backend	Hugging Face
Proprietary LLM backend	Anthropic API
Primary model	`Qwen/Qwen3.5-35B`
Secondary model	`meta-llama/Llama-3-8B`
Proprietary model	`claude-opus-5`
Temperature	0.0
Maximum output tokens	512
Request timeout	600 seconds
Protocol A seeds	Three independent random seeds
Tool implementation	Controlled local executable tools
Evaluation set	EvoToolBench-D, 160 held-out test tasks
Remote execution	AutoDL / SeetaCloud, `CUDA_VISIBLE_DEVICES=0`

For Protocol A, all reported main-comparison results are obtained using three independent random seeds, and we report the mean and standard deviation over the three runs.

A.2 Shared Runtime and Hyperparameters

A.2.1 Shared Runtime Configuration

All compared methods use the same runtime configuration. In particular, all methods share the `retriever_topk_plus_llm` planner and use retrieval top-$k = 3$. At planning time, the Task Agent conditions tool selection on the top-k retrieved candidates rather than the full tool catalog that may appear in dataset metadata (Appendix A.2.2).

The maximum number of LLM calls per task is 24, and the maximum number of tool calls is 8. Tool Agent self-refinement is allowed for at most 3 rounds, and an individual tool invocation can be retried at most 3 times.

The shared runtime configuration is summarized in Table 6.

A.2.2 Planner and Tool Retrieval

Under `plan_mode=retriever_topk_plus_llm`, the runtime first scores all tools in the task-visible tool set and passes only the top-$k = 3$ candidates to the planning prompt as

Table 6: Shared runtime configuration used across compared methods.

Parameter	Value	Description
temperature	0.0	Applied to all LLM calls
max_llm_calls	24	Maximum LLM calls per task
max_tool_calls	8	Maximum tool calls per task
max_wall_time_sec	600	Maximum wall-clock time per task
plan_mode	retriever_topk_plus_llm	Shared non-oracle planner
retriever_top_k	3	Number of retrieved candidate tools
refine_policy	all	Tool Agent self-refinement scope
synthesis_mode	extractive	Final answer synthesis
evaluator	exact_match_judge	Task-level evaluation
task_order	seed_shuffle	Seed-dependent task ordering
tool_schema_version	v1	Base tool-schema version

{retrieved_tools} (Appendix C). The nine-tool catalog in Table 26 defines the executable environment, but the planner prompt does not enumerate all nine tools unless they are retrieved. ReAct and other tool-using baselines receive the same retrieved candidate list through {tools_list}.

A.2.3 Adaptation Budget

Table 7 summarizes the adaptation budget used in the main setting. These limits apply uniformly to all compared methods during Protocol A.

Table 7: Adaptation budget used during task execution.

Parameter	Value
self_refinement_budget	3
max_tool_retries	3
mcts_simulation_budget	0
max_memory_writes_per_task	2
allow_memory_write	true
allow_tool_usage_update	true
allow_local_edge_update	true

Reflective Memory stores failure-related experience only, with at most two memory writes per task. During Protocol A, local updates to Edge Utility are allowed, while Archive updates and population-level evolution are disabled. The shared mcts_simulation_budget$=0$ entry applies to non-MCTS methods; MCTS-Only and ToolEVO-Style override this with up to 20 simulations (Table 10), subject to the shared max_llm_calls$=24$ cap described below.

A.2.4 Evolution Hyperparameters

The following hyperparameters are used for system-level evolution under Protocol B. They control population size, search depth, and the multi-objective fitness weights used to rank candidate systems.

These weights define the development-set objective used to select Θ^* before frozen evaluation on the held-out test split.

A.2.5 Edge Utility

Interaction Graph updates are controlled through Edge Utility. The corresponding coefficients are shown in Table 9.

Edge Utility is updated locally during task execution when permitted by the evaluation protocol, and is frozen together with the rest of the system during Protocol B test evaluation.

Table 8: System-evolution hyperparameters used in Protocol B.

Parameter	Value
Population size	8
Generations	5
Tournament size	3
Elite ratio	0.25
Mutation rate	0.3
Crossover rate	0.3
Archive top-k	3
$\lambda_{\text{accuracy}}$	0.45
λ_{tool}	0.20
$\lambda_{\text{feedback}}$	0.15
$\lambda_{\text{recovery}}$	0.10
λ_{cost}	0.10

Table 9: Edge Utility coefficients.

Symbol	Value	Description
α	0.4	Success weight
β	0.3	Gain weight
γ	0.2	Quality weight
δ	0.1	Cost weight
η	0.2	Update rate
θ_{add}	0.6	Edge-addition threshold
θ_{drop}	0.2	Edge-removal threshold

A.2.6 MCTS Configuration

MCTS is used by the ToolEVO-Style and MCTS-Only baselines. Its configuration is summarized below.

Table 10: MCTS configuration used by MCTS-based baselines.

Parameter	Value
MCTS simulations	20
UCT coefficient c	1.4
Parameter aliases	`query`→`keyword`; `expression`→`operations`; `table_text`→`raw_table`; `code`→`source_code`

These settings are used only by the MCTS-based baselines and are disabled in the full EvoToolAgent system unless explicitly noted in the ablation study.

LLM-call accounting. One LLM call counts as one backbone generation request (planning, parameter inference/repair, MCTS expansion or rollout steps that invoke the model, synthesis, or hybrid failure-detector branches). Deterministic tool execution, schema validation, UCT selection over already-expanded nodes, and backpropagation do not count. All methods share `max_llm_calls`$=24$ per task. MCTS-Only and ToolEVO-Style attempt up to 20 simulations, but the search terminates early once the global call budget is exhausted; fewer than 20 simulations may therefore complete on a given task.

A.3 Baseline Implementations

We implement 13 comparison methods together with the full EvoToolAgent system, resulting in 14 evaluated configurations. The baselines cover direct generation, conventional tool use, self-

feedback, test-time search, and system-level optimization. Table 11 lists the registered implementation name, core mechanism, and evaluation purpose of each method.

Table 11: Baseline implementations used in the experiments.

Reg. Name	Paper Label	Core Mechanism	Purpose
`direct_llm`	Direct LLM	No external tools	Measures the basic effect of tool augmentation
`react`	Zero-shot ReAct	Zero-shot thought–action–observation loop	Conventional tool-augmented reasoning
`static_tool_agent`	Static Tool Agent	Fixed Tool Agents; no memory or evolution	Tool Agent abstraction only
`self_refine`	Self-Refine	Generate–critique–refine (3 rounds)	Generic self-feedback
`reflexion`	Reflexion	Task-level flat reflection memory	Generic reflection versus component-level memory
`critic_style`	Critic-Style	Tool validation and correction (3 rounds)	Validator-driven correction
`toolevo_style`	ToolEVO-Style	MCTS, heuristic repair, tool knowledge	Single-agent adaptation
`mcts_only`	MCTS-Only	Within-task MCTS; no persistent memory	Local search only
`random_search`	Random Search	Random mutations on dev tasks	Random search baseline
`prompt_only_evolution`	Prompt-Only Evolution	Mutates only the Task Agent prompt	Single-component system optimization
`tool_only_evolution`	Tool-Only Evolution	Mutates only Tool Agent policies	Single-component system optimization
`graph_only_evolution`	Graph-Only Evolution	Mutates only the Interaction Graph	Measures the contribution of graph evolution
`workflow_search`	Workflow Search	Search over workflow templates	Fixed workflow search
`evotoolagent_full`	EvoToolAgent	Full attribution-driven recovery	Full proposed system

Unless otherwise noted, all baselines share the same planner, tool environment, and execution budget described in Appendix A.2. Differences arise only from the adaptation or optimization mechanisms retained by each method.

A.3.1 Baseline-Specific Defaults

The baseline implementations additionally use the defaults shown in Table 12 when applicable.

Table 12: Baseline-specific default parameters. These bounded search settings apply to component-restricted evolution baselines when search is enabled outside Protocol A frozen test evaluation; they are *not* compute-matched to EvoToolAgent Protocol B evolution (Table 8).

Parameter	Default
Seed	Random seed (fixed per run)
Maximum steps	8
Temperature	0.0
Population size for evolution baselines	4
Generations for evolution baselines	2
Maximum development subset for baseline search	5 tasks

The component-restricted search baselines use smaller bounded search budgets and are intended as diagnostic controls rather than strict compute-matched comparisons. EvoToolAgent under Protocol B uses population size 8, 5 generations, and the full train/dev splits for configuration search.

A.3.2 Evolution-Named Baselines under Protocol A

Random Search, Prompt-Only Evolution, Tool-Only Evolution, Graph-Only Evolution, and Workflow Search are registered with restricted mutation or search operators (Table 11). Under Protocol A, however, all methods—including these evolution-named baselines—start from the same shared initial global configuration (Table 14) and are evaluated on the held-out 160-task test split with **cross-task population evolution entirely disabled**: no mutation, crossover, selection, archive updates, or bounded development-subset search runs during test evaluation. Within-task mechanisms permitted by each baseline (for example, ReAct stepping, MCTS repair, or Tool Agent self-refinement) remain active when part of that baseline's design. The evolution-named baselines therefore function as fixed-configuration diagnostic variants under Protocol A; their names indicate which subsystem would be mutated in bounded diagnostic search settings (Table 12), not additional test-time system optimization.

A.3.3 Capabilities Removed from Individual Baselines

Table 13 summarizes the main EvoToolAgent capabilities intentionally removed from representative baselines.

Table 13: Main EvoToolAgent capabilities absent from representative baselines.

Baseline	Missing EvoToolAgent Capabilities
ReAct	Autonomous Tool Agents, component-level credit assignment, memory, graph adaptation
Static Tool Agent	Self-refinement, memory, Edge Utility, evolution
Reflexion	Component-level targets and Tool-Agent-level memory
ToolEVO-Style	Multi-agent decomposition, Interaction Graph, component-level attribution, Archive
Prompt-Only Evolution	Tool Agent, graph, and memory evolution
EvoToolAgent	None

Under Protocol A, Direct LLM, ReAct, and Static Tool Agent do not perform online memory writes, MCTS-based search, or Tool Agent self-refinement.

A.4 Protocol A: Fair Test-Time Adaptation

Protocol A is the main comparison protocol used in the paper. Its goal is to compare different methods under the same initial system configuration and the same task-level adaptation budget.

All methods are evaluated only on the untouched 160-task test split. Tool self-refinement, Reflective Memory updates, tool-usage knowledge updates, and local Edge Utility updates are allowed when they are part of the corresponding method. Global mutation, crossover, selection, and population-level evolution are disabled.

Table 14: Protocol A evaluation rules.

Rule	Setting
Initial condition	Same initial global system configuration
Evaluation data	Held-out 160-task test set
Allowed	Local self-refinement, permitted memory updates, tool-usage updates, local Edge Utility updates
Disallowed	Global mutation, crossover, selection, population-level evolution
Random seeds	Three independent random seeds, reported as mean $\pm$ standard deviation

The rules in Table 14 ensure that all methods start from the same system configuration and are evaluated on the same held-out tasks, while still allowing method-specific local adaptation when that adaptation is part of the baseline design.

Algorithm 1 formalizes the fair comparison procedure under Protocol A.

Algorithm 1 Protocol A: Fair Test-Time Adaptation

Require: Test tasks $\mathcal{T}$ with $|\mathcal{T}| = 160$; methods $\mathcal{M}$; shared budget $\mathcal{B}$; shared runtime $\mathcal{R}$; seeds $\mathcal{S}$ with $|\mathcal{S}| = 3$
Ensure: Per-method metrics and execution trajectories

```
for each seed s ∈ S do
    Set random seed to s
    Shuffle task order using s
    for each method m ∈ M do
        Initialize m using shared runtime R and budget B
        Initialize permitted online state
        for each task q ∈ T do
            if m does not support online adaptation then
                Execute q with fixed state
            else
                Execute q with permitted local state
                Update permitted online state within budget B
            end if
            Evaluate the final answer against the gold answer
            Log the execution trajectory
        end for
        Compute metrics for method m under seed s
    end for
end for
Aggregate mean and standard deviation over S
return Accuracy, Recovery Rate, and execution-level tool success
```

In practice, each method is executed independently under the same task order, perturbation schedule, and budget limits, and the reported metrics are averaged over the three random seeds.

A.5 EvoToolAgent Execution Pipeline

EvoToolAgent uses a hybrid failure detector and component-level Structured Credit Assignment to route failures to the corresponding recovery mechanisms. The hybrid detector combines deterministic execution signals with optional model-based diagnosis.

The Recovery Controller allows at most three repair attempts and supports *parameter repair*, *output repair*, and *tool replacement*. Reflective Memory records only failure-related experience and allows at most two writes per task.

During Protocol A, Edge Utility may receive local utility updates. In contrast, Archive updates and population-level evolution are disabled. Algorithm 2 summarizes the end-to-end execution loop used by the full EvoToolAgent system.

This pipeline is shared across all EvoToolAgent variants evaluated in the main experiments; ablations remove individual modules from the same execution loop rather than replacing the underlying control flow.

A.6 Protocol B: Offline Evolution and Frozen Evaluation

Protocol B evaluates whether a system configuration learned through offline evolution generalizes to unseen tasks after the system is frozen.

The protocol consists of three phases. First, EvoToolAgent performs population-level system evolution using the training and development sets. Second, the system with the highest development

Algorithm 2 EvoToolAgent Task Execution

Require: Task q, system configuration Θ
Ensure: Execution trajectory ρ and final answer $\hat{y}$

```
Retrieve top-3 tools using retriever_topk_plus_llm
Construct the tool-execution plan
for each planned tool t_j do
    Infer and adapt invocation parameters
    Invoke Tool Agent u_j with self-refinement budget 3
    Apply the hybrid failure detector
    if a failure is detected then
        Perform Structured Credit Assignment
        Invoke Recovery Controller with at most 3 repair attempts
        Apply parameter repair, output repair, or tool replacement as required
    end if
    Validate the resulting observation
end for
Write failure-related experience to Reflective Memory, at most 2 entries
Perform local Edge Utility update if permitted
Synthesize the answer using validated evidence
if evidence is insufficient then
    Replan only the affected dependency subgraph
end if
return ρ, ŷ
```

fitness is selected as Θ^*. Third, Θ^* is frozen and evaluated on the held-out test set. Table 15 summarizes the allowed and disallowed operations in each phase, and Algorithm 3 describes the offline evolution and frozen-test procedure.

Table 15: Protocol B stages.

Phase	Data	Allowed Operations	Disallowed Operations
Evolution	Train (560) + Dev (80)	Mutation, crossover, selection, memory/graph updates	Use of test labels
Selection	Development set	Select Θ^* using development fitness	–
Frozen test	Test (160)	Read frozen system state and within-task self-refinement	Mutation, crossover, selection, Archive update, memory write, graph update

After evolution terminates, the selected configuration is frozen and evaluated without any further population-level updates. Only within-task self-refinement remains enabled during frozen testing.

A.6.1 Frozen-System Generalization Results

The best evolved system is obtained after four generations. Table 16 reports the development and frozen-test statistics.

Table 16: Development and frozen-test performance under Protocol B.

Split	Fitness	Acc.	Tool Success	Recovery	Invalid Tool Rate	Cost
Dev Best (Θ^*)	0.4049	46.25%	43.83%	20.82%	56.17%	0.2927
Frozen Test	0.3880	43.75%	40.45%	18.56%	59.55%	0.2722

The corresponding fitness generalization gap is 0.0169, and the best development configuration is obtained at generation 4.

Algorithm 3 Protocol B: Offline Evolution and Frozen Test

Require: Training tasks $\mathcal{T}_{\mathrm{tr}}$, development tasks $\mathcal{T}_{\mathrm{dev}}$, test tasks $\mathcal{T}_{\mathrm{te}}$
Ensure: Frozen-test metrics and generalization gap
Initialize population $\mathcal{P}$ with $|\mathcal{P}| = 8$
for $g = 1$ to 5 **do**
  for each candidate $\Theta \in \mathcal{P}$ **do**
    Evaluate Θ on training/development tasks
  end for
  Retain top 25% elites
  Perform tournament selection with tournament size 3
  Apply crossover
  Apply mutation with rate 0.3
  Update Archive
end for
$\Theta^* \leftarrow \arg\max_{\Theta \in \mathcal{P}} \mathrm{DevFitness}(\Theta)$
Freeze prompts, Tool Agents, graph, memory, Archive, and Edge Utility
for each task $q \in \mathcal{T}_{\mathrm{te}}$ **do**
  Execute q using frozen Θ^*
  Allow within-task self-refinement with budget 3
  Disallow global state updates and population-level evolution
  Evaluate the final answer
end for
Compute development and frozen-test fitness
Generalization Gap $\leftarrow \mathrm{DevFitness} - \mathrm{TestFitness}$
return Frozen-test metrics and Generalization Gap

A.7 Core Algorithms

A.7.1 Tool Agent Self-Refinement

Each Tool Agent retries invocation up to three times using memory-informed repair. Algorithm 4 formalizes this bounded local refinement loop.

Algorithm 4 Tool Agent Self-Refinement

Require: Tool t, invocation parameters θ, budget $B = 3$, memory M
Ensure: Structured feedback r
for $b = 1$ to B **do**
  $o \leftarrow \mathrm{Execute}(t, \theta)$
  if $\mathrm{ValidateOutput}(o)$ **then**
    return $\mathrm{Success}(o)$
  end if
  $c \leftarrow \mathrm{Diagnose}(\theta, o)$
  $h \leftarrow M.\mathrm{Retrieve}(\mathrm{context}, t)$
  $\theta \leftarrow \mathrm{Repair}(\theta, c, h)$
end for
return $\mathrm{Failure}(\mathrm{structured\ feedback})$

If all refinement attempts fail, the Tool Agent returns structured failure feedback to the system-level recovery procedure instead of triggering global replanning immediately.

A.7.2 Component-Level Credit Assignment

When local repair is insufficient, EvoToolAgent assigns the failure to one of four component types. Algorithm 5 outlines the hierarchical attribution rules used by Structured Credit Assignment.

The attributed component determines which recovery operator is invoked next and which system component receives evolutionary feedback after a batch of tasks is completed.

Algorithm 5 Component-Level Failure Attribution

Require: Execution trajectory ρ
Ensure: Failure component $c^* \in \{C_A, C_U, C_G, C_F\}$
if final-answer validation fails and all tool calls succeed **then**
  $c^* \leftarrow C_A$
else if tool calls contain parameter, schema, or parsing errors **then**
  $c^* \leftarrow C_U$
else if required tools are not invoked or the execution path is inappropriate **then**
  $c^* \leftarrow C_G$
else if correct tool output is unused or feedback is incomplete **then**
  $c^* \leftarrow C_F$
end if
Store failure type, context, cause, repair action, and target c^* in Reflective Memory
return c^*

A.7.3 Fitness Function

The fitness used during Protocol B combines task accuracy, tool success, structured-feedback quality, recovery, and normalized execution cost:

$$F(\Theta) = 0.45A + 0.20S + 0.15Q + 0.10R - 0.10C, \tag{7}$$

where A denotes task accuracy, S denotes tool-call success rate, Q denotes structured-feedback quality, R denotes recovery performance, and C denotes normalized execution cost. Specifically,

$$R = \begin{cases} 0, & N_{\text{failure}} = 0, \\ \min\left(1, \dfrac{N_{\text{recovered}}}{N_{\text{failure}}}\right), & N_{\text{failure}} > 0, \end{cases} \tag{8}$$

where N_{failure} counts logged execution-failure events that trigger recovery and $N_{\text{recovered}}$ counts those followed by successful continuation under the implemented recovery controller. The reported Recovery Rate metric uses the same convention (Appendix B.9).

The structured-feedback score Q assigns 0.25 to each valid structured-feedback field, with a maximum score of 1.0. The cost term C is normalized over token usage, tool calls, messages, and runtime.

A.7.4 MCTS Tool Repair

The MCTS-Only baseline and ToolEVO-Style repair use a simplified tree search over tool actions. Algorithm 6 describes the search procedure and UCT selection rule.

The UCT score is computed as

$$\mathrm{UCT}(v) = Q(v) + c\sqrt{\frac{\ln N(v_{\text{parent}})}{N(v)}}. \tag{9}$$

A.7.5 Population Evolution

Under Protocol B, population evolution uses elite retention and tournament selection. The top 25% candidates are retained as elites, and the remaining population is produced through tournament selection, mutation, and crossover.

Mutations operate on three system components:

- **Task Agent:** role, prompt, memory policy, and execution policy;
- **Tool Agent:** system prompt, input adapter, output validator, and retry policy;
- **Interaction Graph:** edge addition, edge removal, rewiring, and execution-order changes.

Algorithm 6 Simplified MCTS Tool Repair

Require: Task q, available tools $\mathcal{T}$, simulations $N = 20$, UCT coefficient $c = 1.4$
Ensure: Selected tool/parameter action

```
Initialize root state from q
for n = 1 to N do
    v ← root
    while v has children do
        Select child maximizing UCT
    end while
    Expand candidate tool actions from v
    Sample one expanded child
    Simulate the child state and obtain reward
    Backpropagate reward along the visited path
end for
return Action associated with the most visited root child
```

Crossover combines Task Agent components from one parent, Tool Agent components from another parent, and the Interaction Graph from the better-performing parent. Memory and Archive contents are merged during crossover.

A.8 Tool Environment

EvoToolBench-D currently contains nine categories of locally executable tools. Table 17 summarizes the tool families exposed to all compared methods.

Table 17: Locally executable tool categories in EvoToolBench-D.

#	Tool Category
1	Search
2	Calculator
3	Database lookup
4	Table parser
5	Python executor
6	Unit test
7	Weather
8	Date/time
9	Text parser

All tools are executed in a controlled local environment rather than through live online APIs. This ensures that all methods encounter identical and reproducible dynamic perturbations and prevents uncontrolled changes in external services from affecting the comparison.

A.9 Architecture Configuration

Table 18 summarizes the architecture configuration used by the full EvoToolAgent system.

B EvoToolBench-D Details

This appendix provides additional details of EvoToolBench-D, including its design motivation, subset composition, data splits, perturbation distribution, unified task schema, tool specifications, and evaluation metrics.

B.1 Benchmark Design

Existing tool-use benchmarks such as ToolQA (Zhuang et al., 2023), ToolBench (Qin et al., 2024), and API-Bank (Li et al., 2023) primarily evaluate task completion under static tool environments.

Table 18: EvoToolAgent architecture configuration.

Component	Setting	Description
`plan_router`	disabled	Plan-time routing disabled
`suitability_gate`	disabled	Pre-invocation suitability gating disabled
`coverage_loop`	enabled, strict	Enforces strict tool coverage
`failure_detector`	hybrid	Deterministic rules with optional LLM-based diagnosis
`repair_controller`	enabled; 3 attempts	Structured recovery controller
`repair_router`	`deterministic_parse_first`	Applies deterministic parsing before model-based repair
`validator`	deterministic, resynthesis budget $= 2$	Validates observations and final answers
`credit_assignment`	hierarchical, threshold $= 0.5$	Component-level Structured Credit Assignment
`memory`	failure-only, capacity $= 128$, top-$k = 3$	Reflective Memory
`synthesis`	extractive	Evidence-driven answer synthesis
`evidence_policy`	locked	Preserves validated evidence during synthesis
`graph.update_mode`	`utility_only`	Allows local Edge Utility updates only
`workflow.local_repair_scope`	`dependency_subgraph`	Restricts replanning to affected dependency subgraph

They do not explicitly evaluate whether an agent can recover when tool interfaces, outputs, availability, or dependencies change during execution.

EvoToolBench-D extends these settings by introducing dynamic perturbations that cover tool-schema changes, output-format shifts, transient tool failures, tool replacement, and multi-step dependency propagation. The benchmark is designed to support component-level evaluation of dynamic Agent–Tool systems rather than measuring only final task completion.

Table 19 summarizes how the benchmark design corresponds to the capabilities studied in EvoToolAgent.

Table 19: Mapping between evaluation requirements and EvoToolBench-D design.

Evaluation Requirement	Benchmark Design
Tool autonomy and parameter repair	Schema Shift + Tool Failure
Output validation and feedback quality	Output Shift
Failure recovery and memory	Tool Failure + Tool Replacement
Multi-agent collaboration	Multi-step Dependency + Flow subset
Interaction Graph adaptation	Flow subset + Tool Replacement
Code-oriented tool chains	Code subset with Python Executor + Unit Test
Generalization evaluation	Protocol B train/dev/test split with held-out test set

B.2 Benchmark Composition and Data Splits

B.2.1 Four Subsets

EvoToolBench-D contains 800 tasks divided into four complementary subsets: EvoToolQA-D, EvoToolAPI-D, EvoToolFlow-D, and EvoToolCode-D. Table 20 reports the size, source, and train/dev/test split of each subset.

Together, the four subsets cover single-tool QA, API invocation, multi-tool workflows, and code repair, while preserving a common 70/10/20 split ratio across subsets.

B.2.2 Synthetic Task Construction

EvoToolBench-D combines source-derived instances with synthetically generated tasks to reach the subset sizes in Table 20. Synthetic portions appear in EvoToolQA-D, EvoToolFlow-D, and all of

Table 20: Composition of EvoToolBench-D.

Subset	Size	Task Type	Source	Description	Train/Dev/Test
EvoToolQA-D	300	`dynamic_qa`	ToolQA + synthetic	Dynamic QA with external tools	210 / 30 / 60
EvoToolAPI-D	200	`api_calling`	API-Bank + ToolBench	API selection and invocation	140 / 20 / 40
EvoToolFlow-D	200	`multi_tool_flow`	ToolBench + synthetic	Multi-tool dependency workflows	140 / 20 / 40
EvoToolCode-D	100	`code_repair`	Synthetic	Code execution and repair	70 / 10 / 20
Total	**800**	–	–	–	**560 / 80 / 160**

EvoToolCode-D. Generative AI assisted in drafting synthetic instances, as disclosed in the AI-use statement; the archived submission does not include the exact generator model identifier or generation prompt template.

Each instance is serialized into the unified JSONL schema (Appendix B.5) with `input`, `gold_answer`, `required_tools`, and `expected_tool_path`. We do not claim human verification of every synthetic instance, contamination screening against evaluation backbones, or multi-stage expert QA.

EvoToolCode-D. The 100 tasks target code generation, Python execution, unit testing, and failure-driven repair using the local `python_executor` and `unit_test` tools.

EvoToolQA-D and EvoToolFlow-D. Synthetic tasks supplement ToolQA- and ToolBench-derived seeds to reach target subset sizes. They are written to require the local nine-tool surface and the perturbation types in Table 22.

Filtering. Tasks are retained only if they conform to the unified schema and can be executed in the controlled local tool environment (Appendix B.6). `gold_answer` is stored for offline evaluation only and is not exposed to agents (Table 25). We do not document a separate deduplication pipeline or manual review protocol in this release.

B.2.3 Source-Dataset Conversion and Inclusion Criteria

A source instance is **convertible** into EvoToolBench-D if it can be represented in the unified JSONL schema while preserving: (i) the natural-language task in `input`, (ii) a deterministic `gold_answer` checkable offline by the evaluator, (iii) multi-step structure when present via `expected_tool_path`, and (iv) perturbation metadata compatible with the local tool runtime.

Inclusion. ToolQA/API-Bank/ToolBench-derived tasks whose intended solutions can be expressed with the nine local executable tools in Table 26; dialogues or workflows with annotatable gold outputs; ToolBench workflow seeds whose dependency structure fits the Interaction Graph.

Exclusion. Tasks requiring live external APIs not emulated by the local runtime; instances without a deterministic offline gold answer; duplicates already covered at the target subset quota.

API mapping. Source API or tool names are mapped to the closest local tool category (for example, retrieval → `search`, tabular operations → `table_parser`, computation → `calculator` or `python_executor`). Original RapidAPI endpoints are not retained at evaluation time.

Validation. Converted tasks must satisfy schema constraints. `gold_answer` is checked offline by `exact_match_judge` during evaluation; construction-time local executability is required for code- and API-oriented subsets but is not separately re-audited task-by-task in this document.

B.2.4 Protocol-Specific Splits

Protocol B uses the full 560/80/160 train/dev/test split for offline system evolution and frozen generalization evaluation. Protocol A evaluates the compared methods only on the held-out 160-task test split and does not use test instances for hyperparameter search.

Table 21: Data splits used by the evaluation protocols.

Split	Size	Purpose
Train	560	Offline system evolution under Protocol B
Dev	80	Configuration selection and development evaluation
Held-out Test	160	Protocol A main comparison and Protocol B frozen evaluation

B.3 Perturbation Distribution

EvoToolBench-D contains six dynamic perturbation types together with an unperturbed condition. Table 22 reports their distribution over the full set of 800 tasks.

Table 22: Distribution of perturbation types in EvoToolBench-D.

Perturbation	Ratio	QA	API	Flow	Code	Total
Schema Shift	20%	68	48	48	28	**192**
None	20%	57	38	38	19	**152**
Description Noise	15%	47	30	30	19	**126**
Output Shift	15%	41	29	29	8	**107**
Tool Failure	15%	36	23	23	12	**94**
Tool Replacement	10%	32	22	22	8	**84**
Multi-step Dependency	5%	19	10	10	6	**45**

B.3.1 Perturbation Definitions

Table 23: Semantic definitions of the dynamic perturbations.

Perturbation	Simulated Scenario	Example Failure Type
Schema Shift	API argument names, types, or required fields change	`parameter_name_change`, `type_change`
Output Shift	Returned fields are renamed, nested, or missing	`nested_output_change`
Description Noise	Tool description becomes ambiguous or inconsistent with schema	`none`
Tool Failure	Initial failure, timeout, or empty response	`timeout`, `unavailable_tool`
Tool Replacement	Original tool becomes unavailable and must be replaced	`unavailable_tool`
Multi-step Dependency	Dependency propagation across multi-step execution	`wrong_tool_route`
None	No dynamic perturbation	`none`

B.4 Held-Out Test Set Statistics

The held-out test set contains 160 tasks stratified across subsets and perturbation types. Its composition is summarized in Table 24.

Table 24: Statistics of the 160-task held-out test set.

Dimension	Distribution
Subset	EvoToolQA-D: 60; EvoToolAPI-D: 40; EvoToolFlow-D: 40; EvoToolCode-D: 20
Task type	`dynamic_qa`: 60; `api_calling`: 40; `multi_tool_flow`: 40; `code_repair`: 20
Perturbation	Schema Shift: 32; None: 32; Description Noise: 24; Output Shift: 24; Tool Failure: 16; Tool Replacement: 16; Multi-step Dependency: 16
Source	ToolQA: 60; API-Bank: 40; Synthetic Flow/Code: 60

The test set is stratified across the seven perturbation conditions and four benchmark subsets to ensure coverage of all evaluated dynamic tool conditions.

B.5 Unified Task Schema

All EvoToolBench-D instances follow a unified JSONL representation. A representative example is shown below.

```
{
  "task_id": "toolqa_easy-agenda-0000",
  "source": "toolqa",
  "subset": "EvoToolQA-D",
  "task_type": "dynamic_qa",
  "input": "When did Grace attend Broadway Show on 2022/02/17?",
  "gold_answer": "8:00 PM",
  "required_tools": ["search", "calculator"],
  "expected_tool_path": ["search", "calculator"],
  "available_tools": ["search", "calculator", "table_parser", "..."],
  "perturbation_type": "description_noise",
  "failure_type": "none",
  "difficulty": "medium",
  "tool_versions": {"search": "v1"},
  "metadata": {"requires_recovery": false}
}
```

Table 25 summarizes the fields relevant to evaluation and whether they are exposed to the language model.

Table 25: Selected fields in the unified EvoToolBench-D task schema.

Field	Purpose	Exposed to LLM
`input`	User task	Yes
`gold_answer`	Evaluation target	No; Judge only
`required_tools`	Dataset construction and analysis	No
`available_tools`	Available tool set	Yes
`perturbation_type`	Dynamic perturbation	Indirectly via tool config.
`failure_type`	Fine-grained failure annotation	Partially through tool behavior

In particular, `gold_answer` and `required_tools` are never provided to the agent during Protocol A evaluation. The former is used only by the evaluator, while the latter is retained for dataset construction and analysis.

B.6 Tool Specifications

The benchmark contains nine locally executable tool categories. Tool specifications are represented using versioned schemas so that controlled interface changes can be introduced.

B.6.1 Example Tool Schema

A simplified version of the Search v1 specification is shown below.

```
{
  "tool_name": "search",
  "version": "v1",
  "description":
    "Search factual information from a knowledge base.",
  "input_schema": {
    "type": "object",
    "required": ["query"],
```

Table 26: Tool specifications used in EvoToolBench-D.

Tool	v1 Spec.	v2 Spec.	Function
Search	search.json	search_v2.json	Knowledge retrieval
Calculator	calculator.json	calculator_v2.json	Mathematical computation
Table Parser	table_parser.json	table_parser_v2.json	Table parsing
Python Executor	python_executor.json	–	Python execution
Unit Test	unit_test.json	–	Unit-test validation
Database Lookup	database_lookup.json	–	Database queries
Weather	weather.json	–	Weather lookup
Text Parser	text_parser.json	–	Text parsing
Date/Time	date_time.json	–	Date/time operations

```
    "properties": {
      "query": {
        "type": "string",
        "description": "Search query string"
      }
    }
  },
  "output_schema": {
    "type": "object",
    "properties": {
      "answer": {"type": "string"}
    }
  },
  "failure_modes": [
    "no_result",
    "missing_required_field",
    "timeout"
  ]
}
```

B.7 Perturbations and Evaluated Components

Different perturbations are designed to stress different components of the Agent–Tool system. Table 27 links each perturbation type to the primary component responsible for handling it and the expected recovery behavior.

Table 27: Mapping from perturbations to the primary EvoToolAgent components they evaluate.

Perturbation	Evaluated Component	Expected Behavior
Schema Shift	Tool Agent ϕ^{in} + self-refinement	Argument adaptation and parameter repair
Output Shift	Tool Agent ϕ^{out} + structured feedback	Output normalization and validation
Description Noise	Credit assignment	Planning and failure attribution
Tool Failure	Recovery Loop + Reflective Memory	Retry and failure recovery
Tool Replacement	Recovery Controller + Edge Utility	Selection of an alternative tool
Multi-step Dependency	Coverage Loop + Interaction Graph	Complete multi-step execution

Together, these mappings ensure that each perturbation type primarily evaluates a distinct recovery or attribution mechanism rather than only end-task accuracy.

B.8 Research Questions and Benchmark Coverage

Table 28 summarizes how different baseline groups and benchmark subsets support the experimental analysis. Each row links a research question to the representative baselines and benchmark subsets used to answer it.

Table 28: Mapping between research questions, representative baselines, and benchmark subsets.

RQ	Research Question	Representative Baselines	Relevant Subsets
RQ1	Effectiveness of tool-augmented execution	Direct LLM, ReAct, Static Tool Agent	All subsets
RQ2	Contribution of recovery and reflection mechanisms	Self-Refine, Reflexion, Critic-Style	All subsets
RQ3	Generalization and system-level adaptation	ToolEVO-Style, MCTS-Only, EvoToolAgent	QA + API + held-out evaluation
Aux.	Component-restricted diagnostic search (not compute-matched)	Random/Prompt/Tool/Graph Evolution, Workflow Search	Flow + Code

This mapping is used to organize the main comparison, ablation study, and Protocol B generalization analysis reported in the paper.

B.9 Evaluation Metrics

We report metrics at the task, recovery, and tool-execution levels. Their definitions are summarized in Table 29. Unless otherwise noted, higher values are preferred for accuracy-, success-, recovery-, and fitness-related metrics, while lower values are preferred for cost and invalid-call metrics.

Table 29: Evaluation metrics used in EvoToolBench-D.

Metric	Symbol	Direction	Definition
Accuracy	Acc.	↑	Correctly completed tasks / total tasks
Tool Success Rate	TS	↑	Successful tool calls / total tool calls (same success criterion as Exec. ETS)
Recovery Rate	Rec.	↑	Eq. equation 8: recovered failure events / logged failure events; 0 when no failure events are logged
Feedback Quality	FQ	↑	Completeness of structured failure feedback
Invalid Tool Call Rate	ITR	↓	Invalid tool calls / total tool calls
Cost	C	↓	Normalized token, call, message, and runtime cost
Fitness	F	↑	Weighted system-level objective defined in Appendix A.7
Execution-level Tool Success	Exec. ETS	↑	Macro-mean over tasks of successful tool invocations / total tool invocations; success requires local runtime completion and deterministic observation validation (Verify_o); no LLM judge
Generalization Gap	–	signed	$F_{\mathrm{dev}} - F_{\text{frozen-test}}$ under Protocol B; values closer to zero indicate smaller gaps; negative values mean frozen-test fitness exceeds development fitness

Feedback Quality assigns 0.25 to each valid field of the structured failure feedback, with a maximum value of 1.0. The Fitness metric follows the weighted objective defined in Appendix A.7. Task Accuracy uses the offline judge only: $\mathrm{Success}_{\mathrm{eval}}(\tau) = \mathbb{1}[\mathrm{Judge}(\hat{y}, y) = 1]$ with `exact_match_judge` and gold answer y unavailable to online execution (Appendix D). Hybrid LLM branches in the failure detector and observation validator route recovery but are not used to score Exec. ETS or Accuracy.

C PROMPT COLLECTION

This appendix lists the LLM prompts used by EvoToolAgent and the compared baselines. Placeholders: {task_input} / {input} denote the task text; {tool_name} denotes the tool name; other placeholders are defined per subsection.

C.1 TASK AGENT

C.1.1 TASK AGENT SYSTEM PROMPT (DEFAULT)

Type: System. **Purpose:** Define the Task Agent identity for tool planning and result synthesis.

```
You are a task agent that plans and synthesizes tool
results.
```

C.1.2 TOOL PLANNING (PLAN_TOOLS)

Type: User. **Purpose:** Select a minimal tool set from retrieved candidates and return a JSON plan. At runtime, {retrieved_tools} contains the top-$k = 3$ tools returned by retriever_topk_plus_llm (Table 6); the planner does not see the full nine-tool catalog unless a tool is retrieved.

```
Task: {task_input}
Choose tools only from the retrieved candidates:
{retrieved_tools}
Prefer the minimal set. For factual/agenda questions
use only search when retrieved. Use calculator only when
arithmetic is required and calculator is retrieved.
Return JSON: {"plan": ["tool_name", ...]}
```

C.1.3 ANSWER SYNTHESIS (SYNTHESIZE_ANSWER)

Type: User. **Purpose:** Extract the final short answer from ranked tool outputs.

```
Task: {task_input}
Tool outputs (ranked): {context}
Rules:
1) Reply with the FINAL ANSWER ONLY (short
phrase/number/name).
2) If a tool output contains '| answer: X', copy X exactly.
3) Prefer the first useful tool value; do not invent.
4) Never refuse or explain; never restate the question.
```

C.2 TOOL AGENT

C.2.1 TOOL AGENT SYSTEM PROMPT (DEFAULT)

```
You are a tool agent responsible for reliable tool
invocation.
```

C.2.2 PARAMETER ADAPTATION (ADAPT_INPUT)

```
Adapt params for {tool_name}: {raw_params}
```

C.2.3 FAILURE DIAGNOSIS (DIAGNOSE)

```
Diagnose failure for {tool_name}
```

C.3 TOOL PARAMETER INFERENCE AND REPAIR

C.3.1 PARAMETER UTILITY SYSTEM PROMPT

```
You emit only JSON tool parameters.
```

C.3.2 PARAMETER GENERATION (INFER_PARAMS_WITH_LLM)

```
Generate parameters for tool '{tool_name}'.
Task: {task_input}
Schema required fields: {required}
Schema properties: {properties}
Example (structure only): {example}
Return ONLY a valid JSON object with the required keys. No
markdown, no explanation.
```

C.3.3 PARAMETER REPAIR (REPAIR_PARAMS_WITH_LLM)

```
Repair JSON params for tool '{tool_name}'.
Task: {task_input}
Current params: {current_params}
Error: {error}
Required fields: {required}
Hints: {hints}
Suggested fallback: {fallback}
Return ONLY a corrected JSON object.
```

C.4 BASELINE METHODS

C.4.1 DIRECT LLM

```
Answer the following task directly:
{input}
```

C.4.2 ZERO-SHOT REACT

Type: User. **Purpose:** Zero-shot thought–action–observation loop with the same shared backbone, top-$k = 3$ retrieved tool context, and global execution budget as other tool-using methods (Table 6). {tools_list} is populated from the same retrieved candidates as {retrieved_tools} above; no few-shot demonstrations are used.

```
Task: {input}
Available tools: {tools_list}
Previous observation: {observation}
Respond with Thought, Action, Action Input. Use
Action=Final to finish.
```

C.4.3 SELF-REFINE — CRITIQUE

```
Critique this answer: {answer}
Task: {input}
```

C.4.4 SELF-REFINE — REFINEMENT

```
Improve answer based on feedback:
Answer: {answer}
Feedback: {feedback}
```

C.4.5 REFLEXION — REFLECTION INJECTION

```
Past reflections:
{lessons}
```

C.4.6 REFLEXION — LESSON WRITING

```
Task failed. Context: {input}
Answer: {final_answer}
Write a lesson.
```

C.4.7 BASELINE SHARED SYNTHESIS

```
Task: {input}
Tool outputs: {flat_outputs}
Reply with the final answer only. Prefer values found
in tool outputs. Do not repeat the question. Do not add
explanation.
```

C.4.8 CRITIC-STYLE CORRECTION

```
Fix answer: {answer}
Critique: {error}
```

C.4.9 PROMPT-ONLY EVOLUTION CANDIDATES

One of the following replaces the default Task Agent system prompt during search:

```
Plan tools step by step.
Verify tool outputs before answering.
Decompose the task before tool use.
You are a reflective planner.
```

C.5 ARCHITECTURE LAYER (FAILURE DETECTION AND VALIDATION)

C.5.1 FAILURE DETECTOR (HYBRID LLM BRANCH)

System:

```
You are a failure classifier. Be conservative.
```

User:

```
Task: {task_input}
Tool: {tool_name}
Output: {output}
Is this output sufficient to help answer the task? Reply
JSON {"failure":bool,"confidence":float,"reason":str}
```

C.5.2 OBSERVATION VALIDATOR (HYBRID LLM BRANCH)

System:

```
Observation validator
```

User:

```
Task: {task_input}
Observations: {tool_outputs}
Are observations sufficient? JSON {"sufficient":bool,"confidence":float,"reason":str}
```

C.6 EVOLUTION MUTATION SUFFIXES

During evolution, the following suffixes may be appended to system prompts:

- Task Agent: `Reflect on tool failures before answering.`
- Tool Agent: `Validate schema strictly. Always emit required JSON keys.`
- Generic mutation marker: `[mutated]`

D FORMAL SPECIFICATION: ATTRIBUTION, RECOVERY, AND EVOLUTION

This section makes the operators introduced in §4–6 fully explicit: what each symbol denotes, how each function is computed, and how online recovery and offline evolution connect. Notation follows the main paper.

D.1 SYMBOL TABLE

Table 30: Symbols for attribution, recovery, and evolution.

Symbol	Type	Meaning
q	task	User query / instruction
τ	trajectory	Complete execution record of one task attempt
$\Theta = (A, U, G, M, H)$	system config	Task Agent, Tool Agents, Graph, Memory, Archive
$\mathcal{C}$	set	Failure-component space $\{C_A, C_U, C_G, C_F\}$
c^*	element of $\mathcal{C}$	Primary attributed failure component
M	memory bank	Reflective Memory (set of entries μ)
$P = (z_1, \ldots, z_L)$	plan	Ordered tool-invocation plan
$\mathrm{Pending}(P)$	queue	Plan nodes not yet executed
$G_{\mathrm{dep}}(z_l)$	subgraph	Dependency subgraph rooted at failed node z_l
$\mathcal{V}$	evidence set	Validated tool observations for synthesis
B	budget	Per-task recovery budget (3 attempts in our implementation)
a_{repair}	action	Selected recovery action
Θ'	config	Mutated system after targeted evolution
y	label	Gold answer (evaluation only; not visible online)

Table 31: Component semantics (failure categories).

Component	Subsystem	Typical failure modes
C_A	Task Agent A	Wrong decomposition, tool choice, or synthesis
C_U	Tool Agent U	Schema mismatch, param error, parse/execution error
C_G	Interaction Graph G	Missing edge, wrong route, incomplete coverage
C_F	Feedback / utilization	Ignored valid output, incomplete feedback, ungrounded answer

D.2 EXECUTION TRAJECTORY

A trajectory τ for task q is a structured record:

$$\tau = \big(q,\ P,\ \{(z_l, u_l, o_l, s_l, e_l)\}_{l=1}^{L},\ \hat{y},\ y,\ \mathcal{E}\big),$$

where P is the planned tool nodes, u_l and o_l are submitted parameters and raw outputs, $s_l \in \{\text{SUCCESS}, \text{FAILURE}\}$, e_l is the validation error string, $\hat{y}$ is the predicted answer, y is the gold answer stored for benchmark evaluation, and $\mathcal{E}$ is the recovery-event log.

Offline benchmark evaluation uses the gold answer only:

$$\mathrm{Success}_{\mathrm{eval}}(\tau) = \mathbb{1}\big[\mathrm{Judge}(\hat{y}, y) = 1\big].$$

The gold answer y is used only for offline benchmark evaluation and is never exposed to failure detection, attribution, recovery, or test-time execution (Table 25).

D.3 TOOL AGENT LOCAL REPAIR

Each Tool Agent $u_j = (t_j, \phi_j^{\mathrm{in}}, \phi_j^{\mathrm{out}}, \rho_j)$ performs input adaptation, execution, and result validation. Local repair is defined in Eq. equation 3; after repeated failures, the Tool Agent escalates to the system-level recovery procedure in §3.3.

D.4 INTERACTION GRAPH UPDATES

The Interaction Graph $G = (V, E)$ with $V = A \cup U$ records Task Agent–Tool Agent delegation relations. Edge Utility updates preserve effective Task–Tool interactions while weakening consistently low-quality connections; update rules and coefficients are provided in Appendix A.

D.5 Failure Detection

Before component attribution, a failure detector $\mathrm{Detect}(\cdot)$ maps each tool step to a failure stage ϕ, confidence γ, and reason δ:

$$(\phi, \gamma, \delta) = \mathrm{Detect}(q, z_l, o_l, s_l, e_l),$$

where ϕ belongs to

$$\begin{aligned}\Phi = \{&\text{tool_selection}, \text{argument_construction}, \text{schema_validation}, \text{tool_execution},\\ &\text{output_parsing}, \text{observation_validation}, \text{multi_step_planning},\\ &\text{answer_synthesis}, \text{final_answer_validation}\}.\end{aligned}$$

The hybrid detector applies deterministic rules first and invokes an LLM classifier only when rule confidence is ambiguous.

D.6 Component Attribution: $\mathrm{Attribute}(\tau)$ and $\mathrm{Attribute}(\tau_{\le l})$

Attribution operates only on online execution signals and never uses the gold label y. For an intermediate step l, write the prefix trajectory as

$$\tau_{\le l} = \big(q,\ P,\ \{(z_i, u_i, o_i, s_i, e_i)\}_{i=1}^{l},\ \hat{y},\ \mathcal{E}\big).$$

Intermediate failures call $\mathrm{Attribute}(\tau_{\le l})$ using only executed steps, detector outputs, and validated evidence accumulated so far. Rules that require a synthesized final answer—in particular, final-answer validation failure—are inactive until $\hat{y} = \mathrm{Synthesize}(q, \mathcal{V})$ has been produced.

Let $\mathrm{TS}(\tau)$ denote that all tool steps succeeded, $\mathrm{TW}(\tau)$ that final-answer validation failed, $\mathrm{PE}(\tau)$ that a parameter/schema/parsing error is present, and $\mathrm{PF}(\tau)$ that at least one tool step failed. Online final-answer failure is defined by the evidence-conditioned validator:

$$\mathrm{TW}(\tau) = \mathbb{1}\big[\mathrm{Verify}_y(\hat{y}, q, \mathcal{V}) = 0\big].$$

The complete attribution rules used in our implementation are the ordered if–else if hierarchy in Algorithm 5, which defines c^* from online failure signals only.

A stage-to-action map $\pi : \Phi \to (\mathrm{comp}, a_{\mathrm{repair}})$ refines detection into executable recovery hints. Structured credit returns

$$\mathrm{SC}(\tau) = \big(\phi, \delta, \mathrm{comp}, \gamma, a_{\mathrm{repair}}\big),$$

with confidence gating $a_{\mathrm{repair}} = \texttt{stop}$ if $\gamma < \theta_{\mathrm{credit}}$, otherwise $\pi(\phi)_a$ ($\theta_{\mathrm{credit}} = 0.5$ in our implementation).

D.7 Reflective Memory

Each memory entry stores $\mu = (\text{type}, \text{context}, \text{cause}, \text{repair}, \text{target}, \text{tool}, \text{task_type})$. After a failed attempt,

$$M \leftarrow M \cup \{\mu\}, \quad \mu.\text{target} = c^*, \quad \mu.\text{cause} = \delta.$$

Before recovery, the system retrieves

$$\mathcal{M}_{\mathrm{ret}} = \underset{\mu \in M,\, |\mathcal{M}_{\mathrm{ret}}| \le k}{\arg\max} \ \mathrm{sim}(\mu, q, c^*, \phi),$$

using keyword overlap and failure-type matching ($k = 3$, capacity 128). Recovery uses retrieved entries:

$$a_{\mathrm{repair}} = \mathrm{Recover}(\tau, c^*, M) \quad \text{with} \quad \mathcal{M}_{\mathrm{ret}} \subseteq M.$$

D.8 Recovery Operator: $\mathrm{Recover}(\tau, c^*, M)$

The recovery action space is

$$\begin{aligned}\mathcal{A}_{\mathrm{repair}} = \{&\texttt{local_retry}, \texttt{tool_replace}, \texttt{reparse},\\ &\texttt{resynthesize}, \texttt{partial_replan}, \texttt{stop}\}.\end{aligned}$$

Given detection (ϕ, γ, δ), structured credit $\mathrm{SC}(\tau)$, memory hints $\mathcal{M}_{\mathrm{ret}}$, and remaining budget B_{rem},

$$a_{\mathrm{repair}} = \begin{cases} \texttt{stop} & B_{\mathrm{rem}} = 0 \ \vee \ \gamma < \theta_{\mathrm{rec}} \ \vee \ \text{same-failure limit reached} \\ \pi(\phi)_a & \text{otherwise, refined by the repair router and } \mathcal{M}_{\mathrm{ret}} \end{cases}$$

with $\theta_{\mathrm{rec}} = 0.5$, $B_{\max} = 3$. The attributed component c^* determines the minimal recovery scope $\mathrm{scope}_{\min}(c^*)$, while the failure stage ϕ and map π determine the concrete repair action. Table 32 summarizes the implemented stage-to-action routing.

Table 32: Stage-to-action routing used by the recovery router.

Failure stage ϕ	Scope c^*	Action a_{repair}
`schema_validation`, `tool_execution`, `argument_construction`	C_U	`local_retry`
`output_parsing`	C_U	`reparse`
`tool_selection`, `multi_step_planning`	C_A / C_G	`partial_replan` / `tool_replace`
`answer_synthesis`, `final_answer_validation`	C_F / C_A	`resynthesize` / `partial_replan`

In particular, a C_U failure may trigger `local_retry` for invocation/schema errors or `reparse` for output-parsing errors. The default component-level scopes remain: C_U →`local_retry`/`reparse`; C_G →`tool_replace`; C_A →`partial_replan`; C_F →`resynthesize` from $\mathcal{V}$.

D.9 Partial Replan and Evidence Preservation

For plan node z_l, the dependency subgraph is $G_{\mathrm{dep}}(z_l) = (\{z_l\} \cup \mathrm{Descendants}(z_l), E_{\mathrm{dep}})$. Partial replan replaces only the pending dependent subplan:

$$P' = \mathrm{ReplaceSubplan}(P, G_{\mathrm{dep}}(z_l), \mathrm{Replan}(G_{\mathrm{dep}}(z_l))) \,.$$

ReplaceSubplan preserves the relative order of unaffected steps and replaces only the affected pending dependency subgraph with a dependency-consistent replanned sequence. In implementation, $\mathrm{Replan}(\cdot)$ conditions on q, $\mathcal{V}$, and retrieved memory $\mathcal{M}_{\mathrm{ret}}$. Execution proceeds over the current pending plan, so replanned dependent nodes are not skipped.

Validated observations form the evidence set $\mathcal{V} = \{o_l : \mathrm{Verify}_o(o_l) = 1\}$ and the answer is $\hat{y} = \mathrm{Synthesize}(q, \mathcal{V})$. If $\mathrm{Verify}_y(\hat{y}, q, \mathcal{V}) = 0$ but $\mathcal{V}$ already contains sufficient validated evidence, recovery preserves $\mathcal{V}$ and enters the same attribution/recovery loop rather than bypassing attribution.

D.10 Within-Task Recovery Loop

Algorithm 7 summarizes the online recovery procedure.

D.11 Attribution-Driven System Evolution

During Protocol B, each population candidate is evaluated on training and development tasks. Failed trajectories contribute attribution signals that guide targeted mutation during population update, rather than generating one independent mutant per failed trajectory.

For a failed trajectory τ, the attributed component is

$$c^* = \mathrm{Attribute}(\tau).$$

Targeted mutation replaces only the subsystem corresponding to c^*:

$$\mathrm{Mutate}_{c^*}(\Theta) = \begin{cases} \mathrm{Mutate}_A(A) \,\|\, (U, G, M, H) & c^* = C_A \\ \mathrm{Mutate}_U(U) \,\|\, (A, G, M, H) & c^* = C_U \\ \mathrm{Mutate}_G(G) \,\|\, (A, U, M, H) & c^* = C_G \\ \mathrm{Mutate}_F(A) \,\|\, (U, G, M, H) & c^* = C_F \end{cases}$$

Algorithm 7 Within-task recovery loop.

Require: task q, system Θ, budget $B_{\max}$
Ensure: trajectory τ, answer $\hat{y}$

$P \leftarrow \mathrm{Plan}(q, \Theta)$; $\mathcal{V} \leftarrow \emptyset$; $B_{\mathrm{rem}} \leftarrow B_{\max}$
while $\mathrm{Pending}(P) \neq \emptyset$ **do**
  $z_l \leftarrow \mathrm{Next}(P)$
  Execute tool step z_l and run failure detection
  if step failure detected **then**
    $c^* \leftarrow \mathrm{Attribute}(\tau_{\leq l})$
    $\mathcal{M}_{\mathrm{ret}} \leftarrow \mathrm{Retrieve}(M, q, c^*, \phi)$
    while $B_{\mathrm{rem}} > 0$ **do**
      $a \leftarrow \mathrm{Recover}(\tau, c^*, M)$
      if $a = \texttt{stop}$ **then**
        exit recovery loop
      end if
      apply a on $\mathrm{scope}_{\min}(c^*)$; update P if $a = \texttt{partial_replan}$
      $B_{\mathrm{rem}} \leftarrow B_{\mathrm{rem}} - 1$
    end while
  end if
  add validated outputs to $\mathcal{V}$
end while
$\hat{y} \leftarrow \mathrm{Synthesize}(q, \mathcal{V})$
if $\mathrm{Verify}_y(\hat{y}, q, \mathcal{V}) = 0$ and $B_{\mathrm{rem}} > 0$ **then**
  $c^* \leftarrow \mathrm{Attribute}(\tau)$
  $\mathcal{M}_{\mathrm{ret}} \leftarrow \mathrm{Retrieve}(M, q, c^*, \phi)$
  $a \leftarrow \mathrm{Recover}(\tau, c^*, M)$
  if $a \neq \texttt{stop}$ **then**
    apply a on $\mathrm{scope}_{\min}(c^*)$; update P if $a = \texttt{partial_replan}$
    if $a = \texttt{partial_replan}$ **then**
      while $\mathrm{Pending}(P) \neq \emptyset$ **do**
        $z_l \leftarrow \mathrm{Next}(P)$; execute z_l; add validated outputs to $\mathcal{V}$
      end while
    end if
    if $a \in \{\texttt{resynthesize}, \texttt{partial_replan}\}$ **then**
      $\hat{y} \leftarrow \mathrm{Synthesize}(q, \mathcal{V})$
    end if
    $B_{\mathrm{rem}} \leftarrow B_{\mathrm{rem}} - 1$
  end if
end if
update Reflective Memory with any recovery event in $\mathcal{E}$
return $\tau, \hat{y}$

Mutation domains include Task Agent prompts and policies (C_A), Tool Agent adapters and validators (C_U), graph edges (C_G), and synthesis policies (C_F).

Let $\mathcal{S}^{(k)}$ denote the multiset of attributed components aggregated from failed trajectories produced while evaluating generation k. With population size $K = 8$, elite fraction 25%, crossover, and mutation rate $p_m = 0.3$ (Algorithm 3),

$$\mathcal{C}^{(k)} = \mathrm{Elite}(\mathcal{P}^{(k)}) \cup \mathrm{Crossover}(\mathcal{P}^{(k)}) \cup \mathrm{Mutate}(\mathcal{P}^{(k)}; \mathcal{S}^{(k)}),$$

$$\mathcal{P}^{(k+1)} = \mathrm{Select}_K\left(\mathcal{C}^{(k)}\right).$$

Here $\mathrm{Mutate}(\mathcal{P}^{(k)}; \mathcal{S}^{(k)})$ generates offspring by applying $\mathrm{Mutate}_{c^*}(\cdot)$ with probability p_m, using attribution targets drawn from $\mathcal{S}^{(k)}$. Fitness for selection is

$$F(\Theta) = 0.45\,A + 0.20\,S + 0.15\,Q + 0.10\,R - 0.10\,C.$$

Attribution drives *where* to mutate; fitness drives *which* configurations are retained in Archive H.

D.12 Worked Example

Task: a schema-shifted calculator call returns `type_error`; search succeeds with `114.6`; synthesis initially returns a tool name.

1. **Detect:** $\phi =$ schema_validation, $\gamma = 0.95$.
2. **Attribute:** $c^* = C_U$.
3. **Recover:** $a_{\text{repair}} =$ `local_retry` with JSON parameter repair.
4. **Evidence:** search output passes Verify_o; $\mathcal{V} = \{114.6\}$.
5. **Synthesize:** $\hat{y} = \text{Synthesize}(q, \mathcal{V}) = 114.6$.
6. **If training:** attribution statistics from failed trajectories guide $\text{Mutate}_{C_U}(U)$ during population update.

E Case Studies and Additional Results

This appendix provides representative execution cases from EvoToolAgent. The case studies are drawn from the held-out test set and illustrate successful evidence synthesis, recovery from intermediate failures, multi-step execution failures, and tool replacement.

E.1 Case 1: Evidence-Locked Extractive Synthesis

This case is drawn from EvoToolQA-D under an output-shift perturbation.

Table 33: Case 1: EvoToolQA-D with output shift.

Field	Value
Task ID	`toolqa_easy-coffee-0012`
Subset / Perturbation	EvoToolQA-D / Output Shift
Input	What was the opening price of coffee on 2018-03-06?
Gold Answer	120.1
Prediction	**120.1**

The execution trajectory is summarized as follows:

1. `search(query=...)` returns `120.1`.
2. `database_lookup(SQL=...)` returns a placeholder output, {`rows: [{value: "database_lookup"}]`}.
3. The extractive synthesis module prioritizes the validated search result and selects the first valid numerical value, `120.1`.
4. No recovery event is triggered because the initial search step succeeds.

This example illustrates how evidence-locked extractive synthesis can preserve a valid tool observation and avoid being distracted by noisy outputs from subsequent tools.

E.2 Case 2: Multi-Tool Execution with Recovery

This case illustrates recovery after intermediate semantic failures in a multi-tool execution trajectory.

The execution trajectory proceeds as follows:

1. `database_lookup` returns a placeholder output, {`value: "database_lookup"`}.
2. `date_time` produces an output that is semantically unrelated to the task.
3. `calculator(expression="180-90")` returns `90.0`.
4. `python_executor` prints an intermediate value.

Table 34: Case 2: Multi-tool execution with recovery.

Field	Value
Task ID	`toolqa_easy-agenda-0007`
Subset	EvoToolQA-D
Recorded Perturbation	None
Input	How long did Brian attend Happy Hour on 2022/06/20?
Gold Answer	90.0
Prediction	**90.0**

5. `search` performs an additional retrieval step.
6. The Recovery Loop performs repair operations and preserves the valid calculator result.
7. The final synthesized answer is `90.0`.

This example illustrates how the Recovery Loop can recover from semantically invalid intermediate tool outputs and redirect execution toward a valid computational path. It provides a concrete example of the mechanism examined in the Recovery Loop ablation in the main paper.

E.3 Case 3: Failure in a Multi-Step Workflow

The following example is drawn from EvoToolFlow-D and contains a description-noise perturbation.

Table 35: Case 3: Multi-step workflow under description noise.

Field	Value
Task ID	`syn_flow_0020`
Subset / Perturbation	EvoToolFlow-D / Description Noise
Input	Search bitcoin price, parse numbers from text, average them.
Expected Output	Multi-step aggregated result
Result	Failure

The trajectory contains the following steps:

1. `text_parser` fails to extract the required information directly from the task description.
2. `table_parser` also fails.
3. `search(bitcoin price)` returns `50000.0`.
4. `calculator((50000)/1)` returns `50000.0`.
5. Additional calls to `table_parser`, `database_lookup`, and `search` are performed.
6. The complete chain of extracting multiple numerical values and computing their average is not successfully completed.

This case shows a representative failure mode in multi-step workflows: an inappropriate early tool choice can prevent subsequent execution from converging to the required dependency chain.

E.4 Case 4: Tool Replacement

This case illustrates a tool-replacement perturbation in EvoToolQA-D.

When the original tool becomes unavailable or its interface can no longer support execution, the Recovery Controller attempts tool replacement. In this example, computation that would normally be performed by the calculator must instead be transferred to the Python executor while preserving the remaining search-based evidence.

Successful replacement allows execution to continue along an equivalent path, whereas unsuccessful replacement results in a broken tool chain. This case illustrates the motivation for explicit tool replacement in the Recovery Controller.

Table 36: Case 4: Tool replacement in EvoToolQA-D.

Field	Value
Task ID	`toolqa_easy-agenda-0038`
Perturbation	Tool Replacement
Original Tool Family	Calculator + Search
Replacement Tool Family	Python Executor + Search
Task Type	Agenda query

E.5 Case 5: Comparison with Non-Adaptive Tool Use

We further examine a numerical ToolQA example for which EvoToolAgent obtains the correct answer through external tool evidence.

Table 37: Case 5: Example comparison on `toolqa_easy-coffee-0030`.

Method	Execution Behavior	Outcome
EvoToolAgent	Search returns `114.6`	**114.6**
ReAct	Multi-step tool interaction	Final synthesis fails
Direct LLM	No external tool	No grounded tool evidence

For this task, EvoToolAgent obtains the numerical value `114.6` from search and preserves it through extractive evidence synthesis. The case illustrates the benefit of grounding answer generation in validated tool observations rather than relying solely on parametric knowledge or an unconstrained tool-interaction loop.

E.6 Case 6: Failure on an API-Calling Task

The final case is drawn from EvoToolAPI-D and illustrates a remaining failure mode.

Table 38: Case 6: Failed API-calling example.

Field	Value
Task ID	`apibank_Dictionary-level-1-3`
Subset	EvoToolAPI-D
Perturbation	Tool Failure
Input	Can you give me the definition of the word “resilience”?
Result	Failure

The execution trajectory proceeds as follows:

1. `database_lookup(resilience)` returns a placeholder output.
2. `date_time` and `calculator` produce semantically irrelevant outputs.
3. `python_executor` prints the raw database output.
4. The system fails to recover the requested dictionary definition.

The hybrid failure detector identifies semantically inconsistent tool outputs, but detection alone does not guarantee successful recovery when no subsequent execution path produces the required evidence. This example illustrates a remaining failure mode of the current system.