EvoTool: Self-Evolving Tool-Use Policy Optimization in LLM Agents via Blame-Aware Mutation and Diversity-Aware Selection

Anonymous ACL submission

Abstract

LLM-based agents depend on effective tool-use policies to solve complex tasks, yet optimizing these policies remains challenging due to delayed supervision and the difficulty of credit assignment in long-horizon trajectories. Existing optimization approaches tend to be either monolithic, which are prone to entangling behaviors, or single-aspect, which ignore cross-module error propagation. To address these limitations, we propose EvoTool, a self-evolving framework that optimizes a modular tool-use policy via a gradient-free evolutionary paradigm. EvoTool decomposes agent's tool-use policy into four modules, including Planner, Selector, Caller, and Synthesizer, and iteratively improves them through three mechanisms. *Trajectory-Grounded Blame Attribution* uses diagnostic traces to localize failures to a specific module. *Feedback-Guided Targeted Mutation* then edits only that module via natural-language critique. *Diversity-Aware Population Selection* preserves complementary candidates to ensure solution diversity. Across four diverse benchmarks, EvoTool outperforms strong baselines by over 5 points on both GPT-4.1 and Qwen3-8B, while achieving superior efficiency and transferability.

1 Introduction

LLM-based agents augmented with external tools have become a central paradigm for solving complex tasks in reasoning (Wei et al., 2022; Zhou et al., 2023), decision-making (Yao et al., 2022; Xie et al., 2024), and domain-specific automation (Bran et al., 2023; Yang et al., 2024). These agents depend on an effective tool-use policy to coordinate interdependent competencies, including goal decomposition, tool selection, schema-valid argument construction, and the grounded synthesis of tool outputs (Qu et al., 2025). However, achieving reliable tool use in practice remains challenging, as real-world tasks often involve long-horizon, tightly coupled decision trajectories in which a single error in planning, selection, invocation, or synthesis can cause overall failure (Liu et al., 2023). At the same time, supervision is typically available only at the end of an interaction (Shinn et al., 2023), thereby collapsing multiple error sources into a single terminal signal, creating a severe credit assignment problem that obscures the specific cause of failure and hinders targeted policy improvement.

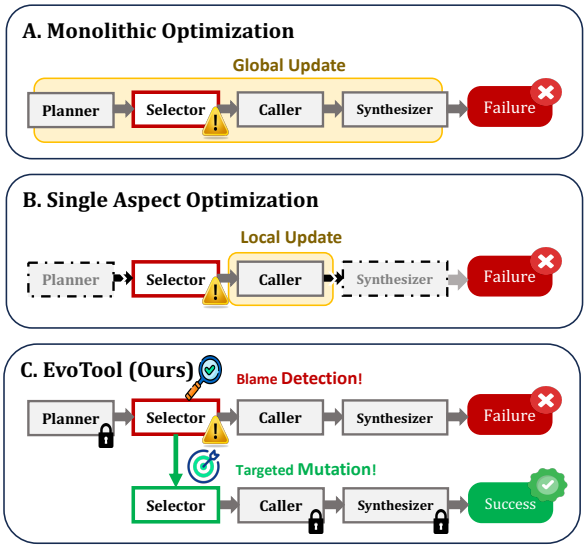

Figure 1: Limitations of monolithic and single-aspect tool-use policy optimization, and how EVOTOOL enables targeted, blame-aware policy updates.

Early approaches rely on hand-crafted prompting patterns or fixed control heuristics (Yao et al., 2022; Wang et al., 2023), which require substantial manual effort and lead to system collapse when unanticipated errors disrupt the rigid execution flow. More recent work explores automated policy optimization (Gao et al., 2025) but generally diverges into two extremes that fail to resolve the credit assignment dilemma. Monolithic Policy optimization methods (Yang et al., 2023; Fernando et al., 2023) apply global black-box search over the entire agent prompt, which can entangle heterogeneous behaviors across modules and induce regressions where

fixing one error destabilizes other capabilities. In contrast, single-aspect optimization methods refine individual components in isolation, such as planning or tool calling (Sun et al., 2023; Yuan et al., 2025), while ignoring cross-module error propagation in long-horizon trajectories. Consequently, no existing paradigm simultaneously achieves the targeted error correction and multi-module coordination needed for robust tool-use policy optimization.

In light of this, we propose EvoTool, a novel self-evolving framework that optimizes a modular tool-use policy, comprising Planner, Selector, Caller, and Synthesizer, through a gradient-free evolutionary paradigm. Specifically, instead of relying on a single terminal output, we propose *Trajectory-Grounded Blame Attribution*, which exploits structured intermediate feedback from the tool environment to convert trace-level diagnostics into module-level responsibility signals, thereby identifying the component most likely to have caused the failure. Once the target module is identified, a *Feedback-Guided Targeted Mutation* mechanism then generates a trace-grounded natural-language critique with detailed feedback to selectively edit the blamed module specification, while keeping the remaining modules fixed.

Furthermore, since tool-use competence decomposes into multiple partially competing skills, greedily selecting a single 'best' candidate by global average can discard complementary behaviors and induce premature convergence. Instead, EvoTool adopts a *Diversity-Aware Population Selection* strategy, which retains a population of policy variants and selects candidates based on instance-level wins, thereby preserving complementary strengths while discouraging collapse to a narrow strategy. Together, EvoTool resolves key limitations of prior work by avoiding the instability of monolithic global edits and the incompleteness of single-aspect refinement, addressing credit assignment under delayed feedback to enable coordinated, targeted improvement across interdependent competencies. Experiments on four diverse benchmarks show that our approach consistently outperforms SoTA baselines by over 5 points on both open-source and closed-source models while achieving superior token efficiency and transferability across both datasets and models.

Main contributions are shown as follows:

- We propose **EvoTool**, a self-evolving framework that optimizes a modular tool-use policy using a gradient-free evolutionary paradigm.
- We introduce Trajectory-Grounded Blame Attribution and Feedback-Guided Targeted Mutation, which leverage structured trajectories and natural-language critique to produce localized updates to the responsible module.
- We propose Diversity-Aware Population Selection to preserve complementary candidates across heterogeneous tool-use competencies.
- We evaluate EvoTool on four diverse benchmarks, showing consistent performance gains over state-of-the-art baselines alongside superior token efficiency and robust transferability across models and datasets.

2 Related Work

Agent Tool-Use Policy Learning. Agent tool-use policy has progressed from static engineering to dynamic self-improvement (Jiang et al., 2025). Early systems relied on hand-crafted prompting patterns and fixed control heuristics (Yao et al., 2022; Wei et al., 2022; Wang et al., 2023; Shen et al., 2023; Lu et al., 2025), which demand excessive manual effort and often generalize poorly across domains and tools. Training-based approaches internalize tool use via SFT (Schick et al., 2023; Patil et al., 2024) or RL (Qian et al., 2025; Feng et al., 2025), but adaptation to evolving environments remains costly due to static weights and large data requirements. A complementary direction therefore explores training-free optimization, refining tool-use behavior online from interaction feedback without weight updates (Ramnath et al., 2025). However, these methods face a key trade-off: global edits can entangle heterogeneous behaviors in planning and execution (Yang et al., 2023; Guo et al., 2023), while isolated local optimizations (Qu et al., 2024; Du et al., 2024; Yuan et al., 2025) ignore cross-module error propagation in long-horizon trajectories. We therefore explore gradient-free search that couples blame attribution with targeted mutation to localize failures and repair the responsible module without destabilizing the overall policy.

Self-Evolving Agent Systems. Self-evolving agent systems enable agents to acquire and refine competencies over time by repeatedly acting, evaluating outcomes, and updating decision policies (Gao et al., 2025; Fang et al., 2025). This line of

work includes self-reflection and self-correction (Madaan et al., 2023; Shinn et al., 2023) that turn failures into natural-language or structured feedback (Yuksekgonul et al., 2024; Agrawal et al., 2025), as well as continual improvement loops that accumulate skills or update agent components (Khattab et al., 2023; Hu et al., 2024). However, many frameworks rely on greedy selection and converge prematurely to a narrow strategy, discarding diverse behaviors needed for heterogeneous task distributions (Fernando et al., 2023). Therefore, we explore diversity-aware population selection to maintain a heterogeneous candidate pool.

3 Preliminaries

Task and Environment Formulation. We study LLM-based agents that solve tasks by interacting with an external tool environment. Let $x \sim \mathcal{D}$ denote a task instance drawn from a distribution $\mathcal{D}$, and let $\mathcal{T} = \{\tau_1, \ldots, \tau_K\}$ be a set of tools, each specified by a name, an argument schema, and a documentation. At step t, the agent observes a textual state s_t summarizing the interaction history and outputs an action a_t, which either performs intermediate reasoning, invokes a tool or terminates with an answer. The environment E then executes tool invocations and returns an observation o_t with tool outputs. A complete agent run yields a trajectory $\tau = \{(s_t, a_t, o_t)\}_{t=1}^{T}$ of variable length T, which terminates when the agent outputs a final prediction $\hat{y} = \hat{y}(\tau)$. We evaluate performance using a task-dependent success function $R(x, \hat{y}) \in [0, 1]$ that is instantiated by standard metrics such as pass@1, success rate, and F1 score. For convenience, we summarize each rollout as an episode record $e = (x, \tau, \hat{y}, R(x, \hat{y}))$.

Modularized Tool-Use Policy. We represent tool use as a modular policy with four roles: (i) a planner that decomposes the input into subgoals; (ii) a selector that decides whether and which tool to call given the current state s_t and subgoal; (iii) a caller that constructs valid arguments and executes the selected tool; and (iv) a synthesizer that integrates tool outputs into the final response. Accordingly, the overall tool-use policy is defined as a fixed modular composition: $\Pi = \pi_{\text{syn}} \circ \big(\pi_{\text{call}} \circ (\pi_{\text{sel}} \circ \pi_{\text{plan}})\big)$, where each module operates on the intermediate state produced by the previous one. All modules share the same base LLM with frozen weights W, conditioned on evolvable module specifications $\Theta = \{\theta_{\text{plan}}, \theta_{\text{sel}}, \theta_{\text{call}}, \theta_{\text{syn}}\}$ including prompts, tool templates, or lightweight formatting rules. We denote the instantiated policy by $\pi_{\Theta,W}$. Learning updates only Θ; the model weights W remain fixed.

Optimization Objective. Executing $\pi_{\Theta,W}$ in environment on input x induces a trajectory $\tau \sim (\pi_{\Theta,W}, E) \mid x$ and a terminal answer $\hat{y}(\tau)$. Our objective is to maximize expected task success by evolving Θ under frozen weights W:

$$J(\Theta; W) = \mathbb{E}_{x\sim\mathcal{D}}\, \mathbb{E}_{\tau\sim(\pi_{\Theta,W})|x}\big[\, R\big(x, \hat{y}(\tau)\big)\,\big]. \tag{1}$$

Therefore, the core challenge is: *how can we improve the tool-use policy from sparse, end-of-trajectory outcomes while (i) localizing which component caused the failure and (ii) updating only that component to avoid breaking other behaviors?*

4 Methods

In this section, we introduce EvoTool, a self-evolving framework designed to optimize the modular tool-use policy through a gradient-free evolutionary paradigm. EvoTool maintains a population of candidate module specifications and iteratively improves them using three mechanisms: (i) Trajectory-Grounded Blame Attribution to localize specific responsible module, (ii) Feedback-Guided Targeted Mutation to update only the blamed module using dense feedback signal, and (iii) Diversity-Aware Population Selection to preserve candidates with complementary competences.

Self-Evolving Optimization Loop As shown in Figure 2, EvoTool optimizes tool-use policy by iteratively refining module specifications $\Theta = \{\theta_{\text{plan}}, \theta_{\text{sel}}, \theta_{\text{call}}, \theta_{\text{syn}}\}$ while keeping base LLM weights frozen. We maintain a population $\mathcal{P} = \{\Theta^{(i)}\}_{i=1}^{N}$ of candidate specifications. In each generation, we sample a parent $\Theta \in \mathcal{P}$ and execute the instantiated tool-use policy $\pi_{\Theta,W}$ on a mini-batch of task instances drawn from training pool S_{train}, collecting the corresponding episode records $e = \big(x, \tau, \hat{y}, R(x, \hat{y})\big)$. To translate these interaction traces into policy updates, we first identify the target module $\pi^* \in \Pi$ using a trajectory-grounded blame score $b_\pi(e)$. We then construct dense, natural-language feedback $F(e, \pi^*)$ to edit only the corresponding module specification, producing a child candidate Θ' that differs from its parent in exactly one component. The child Θ' is added to $\mathcal{P}$ only if it outperforms the parent on the mini-batch. Finally, we update the parent sampling distribution using diversity-aware selection

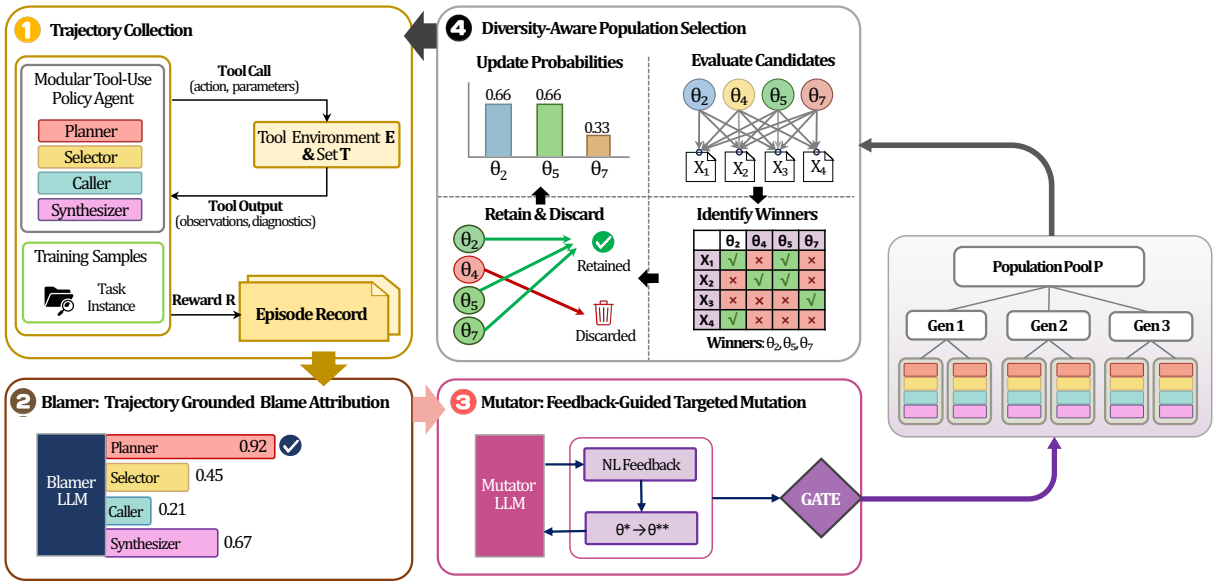

Figure 2: Overall architecture of EvoTool. EvoTool optimizes a modular tool-use policy through a self-evolving loop consisting of (1) trajectory collection from the tool environment, (2) trajectory-grounded blame attribution to identify the responsible module, (3) feedback-guided targeted mutation to update that module using natural language feedback, and (4) diversity-aware population selection over candidate policies to retain complementary candidates.

based on evaluation on a held-out set for population selection S_{sel}. We repeat this loop for a fixed budget and return the best-performing candidate in $\mathcal{P}$. Algorithm details are in Appendix A.1.

4.1 Trajectory-Grounded Blame Attribution

Failures in long-horizon, multi-tool settings are often heterogeneous and multi-causal: a single negative outcome can originate from diverse sources, such as poor goal decomposition, incorrect tool selection, schema violations, or ungrounded synthesis. Without an explicit diagnostic mechanism, a self-evolving agent cannot reliably determine which module to update, degenerating optimization into global blind search. Therefore, EvoTool converts each episode record e into a localized repair target by assigning module-level blame to isolate a single component for mutation. Prompt details on Blamer LLM can be found in Appendix A.4.

Given an episode record $e = (x, \tau, \hat{y}, R(x, \hat{y}))$, we first extract structured diagnostic events from the trajectory $\tau = \{(s_t, a_t, o_t)\}_{t=1}^{T}$, including tool-choice outcomes, argument validity signals, tool execution outcomes, and synthesis-grounding signals. We next provide the full episode e together with this diagnostic summary to a *Blamer* LLM, which outputs module-wise blame scores $b_\pi(e) \in [0, 1]$, where large $b_\pi(e)$ indicates stronger evidence that module policy π is responsible for the observed failure or suboptimality. We then select the module π^* with the highest blame score as the mutation target and pass it, along with the episode record (e, π^*), for a later targeted mutation.

4.2 Feedback-Guided Targeted Mutation

Given a blamed episode e and its selected target module π^*, EvoTool produces a child policy by editing only the corresponding module specification while freezing the remaining components. This targeted design directly mitigates the sparsity and delay of $R(x, \hat{y})$ by leveraging the full interaction trace as supervision: we prompt a reflective *mutator* LLM to translate the blamed episode into natural-language feedback $F(e, \pi^*)$ that both explains the error mode from the perspective of the selected module and proposes a concrete, localized edit to that module's specification given the current specification and trajectory evidence. We then apply this edit to form a child candidate Θ', keeping all other modules fixed. By restricting updates to a single blamed module and using rich textual feedback grounded in trajectories, EvoTool minimizes unintended regressions in unrelated competencies while enabling interpretable, trace-grounded improvements. Prompt details for Mutator LLM can be found in Appendix A.5.

4.3 Diversity-Aware Population Selection

To prevent the population from collapsing into

	Model / Method	ToolBench				RestBench			τ-Bench			BFCL			Overall
		G1	G2	G3	Avg	TM	SP	Avg	Ret	Air	Avg	Sin	Mul	Avg	Avg
GPT-4.1	**Hand-crafted policies**														
	ReAct	68.2	64.7	57.9	63.6	72.4	74.3	73.4	59.8	35.9	47.9	74.8	37.2	56.0	60.6
	CoT	53.5	50.8	48.4	50.9	59.8	63.5	61.7	34.2	25.3	29.8	43.2	21.4	32.3	44.5
	Plan-and-Solve	63.3	59.2	55.3	59.3	64.8	69.2	67.0	58.4	36.7	47.6	57.2	25.3	41.3	54.4
	Monolithic optimization														
	OPRO	69.3	67.3	58.9	65.2	73.8	76.4	75.1	58.2	36.8	47.5	82.5	35.3	58.9	62.1
	PromptBreeder	68.5	66.5	54.7	63.2	74.1	75.2	74.7	56.4	31.3	43.9	81.3	36.2	58.8	60.5
	EvoPrompt	70.1	69.8	59.2	66.4	76.3	77.4	76.9	60.8	36.3	48.6	**84.2**	39.9	62.1	63.8
	Single-aspect optimization														
	AdaPlanner	62.2	58.5	48.8	56.5	66.2	70.1	68.2	62.2	38.8	50.5	69.2	41.1	55.2	57.5
	EASYTOOL	80.3	75.1	66.2	73.9	81.4	83.5	82.5	51.2	29.9	40.6	76.1	36.0	56.1	64.4
	DRAFT	82.1	76.8	68.4	75.8	84.3	85.2	84.8	48.4	29.2	38.8	77.8	32.0	54.9	64.9
	ANYTOOL	73.3	68.8	61.0	67.7	75.2	78.4	76.8	60.4	36.2	48.3	77.1	39.3	58.2	63.3
	Ours														
	EVOTOOL	**83.5**	**78.2**	**71.5**	**77.7**	**86.2**	**86.1**	**86.2**	**64.8**	**39.1**	**52.0**	83.9	**42.3**	**63.1**	**70.6**
Qwen3-8B	**Hand-crafted policies**														
	ReAct	60.0	55.0	47.5	54.2	63.7	63.2	63.5	33.1	14.4	23.8	70.6	33.3	52.0	49.0
	CoT	47.1	43.2	39.7	43.3	52.6	54.0	53.3	14.6	8.2	11.4	41.9	20.3	31.1	35.7
	Plan-and-Solve	55.7	50.3	45.3	50.4	57.0	58.8	57.9	32.0	15.0	23.5	55.5	24.0	39.8	43.7
	Monolithic optimization														
	OPRO	61.0	57.2	48.3	55.5	64.9	64.9	64.9	31.9	15.0	23.5	75.2	33.5	54.4	50.2
	PromptBreeder	60.3	56.5	44.9	53.9	65.2	63.9	64.6	30.6	11.3	21.0	73.9	34.4	54.2	49.0
	EvoPrompt	61.7	59.3	48.5	56.5	67.1	65.8	66.5	33.8	14.7	24.3	75.7	35.9	55.8	51.4
	Single-aspect optimization														
	AdaPlanner	54.7	49.7	40.0	48.1	58.3	59.6	59.0	34.8	**16.4**	25.6	67.1	36.0	51.6	46.3
	EASYTOOL	70.7	61.8	54.3	62.3	71.6	68.0	69.8	20.9	10.3	15.6	73.8	28.2	51.0	51.1
	DRAFT	72.2	65.3	56.1	64.5	74.2	72.4	73.3	17.8	8.9	13.4	72.5	27.1	49.8	51.8
	ANYTOOL	64.5	58.5	50.0	57.7	66.2	66.6	66.4	23.5	14.6	19.1	71.8	30.3	51.1	49.6
	Ours														
	EVOTOOL	**73.5**	**66.5**	**58.6**	**66.2**	**75.9**	**73.2**	**74.6**	**35.9**	15.7	**25.8**	**76.9**	**36.4**	**56.7**	**57.0**

Table 1: Main results on two backbone models. We report benchmark-standard metrics on ToolBench (G1/G2/G3), RestBench (TMDB/Spotify), τ-Bench (Retail/Airline), and BFCL (Single/Multi-turn), with the overall average. TM = TMDB, SP = Spotify, Ret = Retail, Air = Airline, Sin = Single-turn, Mul = Multi-turn.

a single mode and forgetting previously mastered behaviors, EVOTOOL explicitly preserves candidates with complementary competencies. Instead of greedily selecting parents based on global average performance, we employ an instance-wise winner criterion on a held-out set S_{sel}. After each generation, we evaluate all candidates $\Theta^{(i)} \in \mathcal{P}$ on S_{sel}. A candidate is retained only if it achieves the highest score $R(x, \hat{y})$ on at least one instance $x \in S_{\text{sel}}$. Candidates that never achieve instance-level dominance are removed, as they do not represent distinct competencies under the evaluation distribution. For the remaining population, we sample parents for subsequent generations proportional to their winner frequency (the fraction of instances where they outperform all others), thereby concentrating updates on broadly effective policies while retaining specialists that cover distinct regions of the task distribution. Implementation details can be found in Appendix A.6.

5 Experiment

5.1 Experiment Setup

Benchmarks and Metrics. We evaluate EVOTOOL on four established tool-use benchmarks following their standard setups. On **ToolBench** (Qin et al., 2023), which tests large-scale API generalization over RapidAPI, we report *Pass Rate* on the G1/G2/G3 subsets. On **RestBench** (Song et al., 2023), which evaluates sequential tool use over real REST APIs, we report *Success Rate* on the TMDB and Spotify subsets. On **τ-Bench** (Yao et al., 2024), which evaluates stateful, long-horizon agent–user interactions in the retail and airline domains, we report *Pass@1* to assess single-rollout success. On **BFCL** (Patil et al., 2025), which assesses function-calling for tool invocation, we report *Accuracy* on the selected single-turn and multi-turn subsets. For each benchmark, we additionally report the within-benchmark average over its subsets and an overall average across benchmarks. Details on the selected benchmarks can be found in the Appendix A.2.

Baselines. We compare EVOTOOL against three streams of tool-use policy design. **(1) Hand-crafted tool-use policy** that relies on manually designed prompting patterns and fixed control heuristics, including REACT (Yao et al., 2022), chain-of-thought prompting (CoT, Wei et al., 2022), and PLAN-AND-SOLVE (Wang et al., 2023) pipelines.

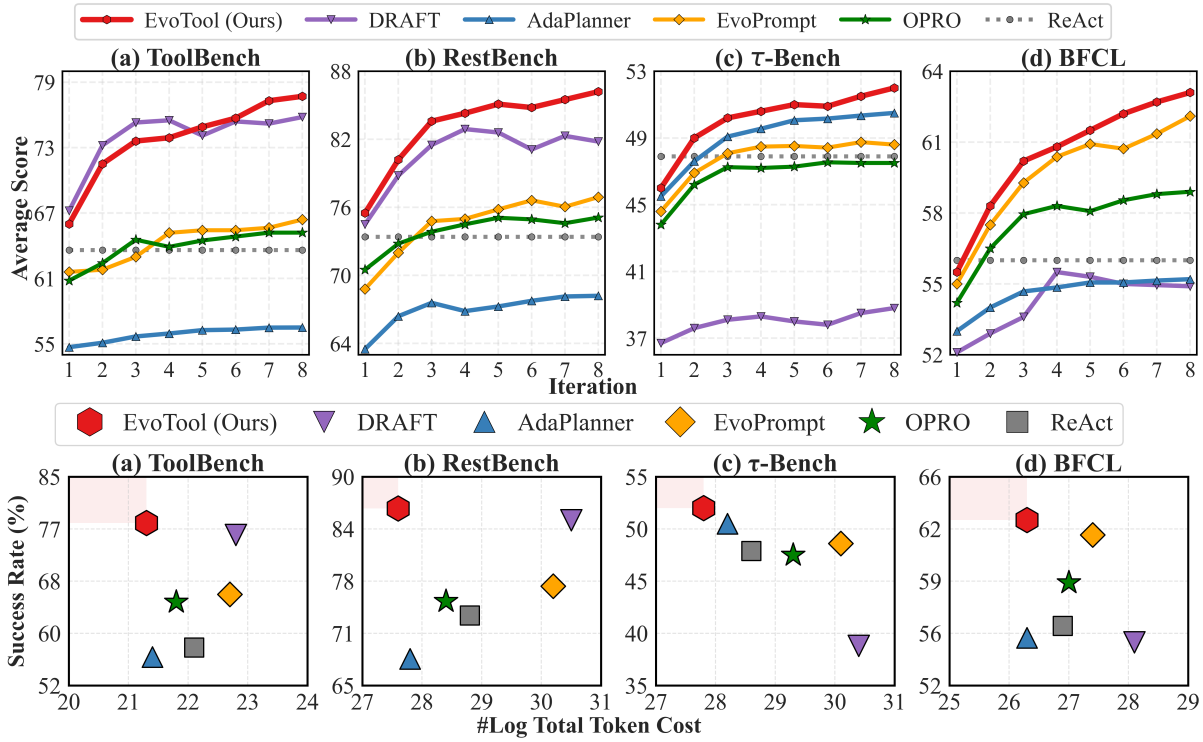

Figure 3: **Learning dynamics and efficiency comparison.** (a) Learning curves across evolution iterations on four benchmarks. (b) Performance versus log token cost under GPT-4.1.

(2) Monolithic tool-use policy optimization that treats agent prompt as a single global policy object optimized via black-box prompt search, including OPRO (Yang et al., 2023), PROMPTBREEDER (Fernando et al., 2023), and EVOPROMPT (Guo et al., 2023). **(3) Single-aspect (local/partial) tool-use policy optimization** that optimize individual component of tool-use policy in isolation, including ADAPLANNER (Sun et al., 2023) that refines tool planning independently, DRAFT (Qu et al., 2024) and EASYTOOL (Yuan et al., 2025) that improves tool selection and calling separately and ANYTOOL (Du et al., 2024) that refines selection and answer synthesizing. Across all baselines, we use the same base LLM, tool suite, and evaluation budget for a fair comparison. Across each baseline, we select GPT-4.1 (OpenAI, 2025) and Qwen3-8B (Yang et al., 2025) as the backbone model to verify its generalizability. Further details on the selected baselines can be found in Appendix A.3.

5.2 Results

As shown in Table 1, EVOTOOL consistently outperforms all baselines across both base models and four benchmarks, demonstrating superior policy learning. On the stronger GPT-4.1 backend, our framework achieves an overall average of 70.6, surpassing the strongest single-aspect baseline DRAFT by nearly 6 points and the best monolithic baseline EvoPrompt by roughly 7 points; this dominance extends to Qwen3-8B, where it outperforms the second best baseline by 5.2 points. Crucially, EVOTOOL remains robust on complex, long-horizon tasks where other paradigms falter: on the stateful τ-Bench, single-aspect methods like DRAFT and EASYTOOL drop to 38.8 and 40.6 as isolated optimization misses cross-module dependencies, while EVOTOOL reaches 52.0. This advantage is reinforced by a leading 42.3 on BFCL multi-turn, confirming that EVOTOOL balances planning flexibility and syntactic precision for deep reasoning better than other alternatives.

5.3 Learning Dynamics and Efficiency

Figure 3 presents a comprehensive view of the learning dynamics and efficiency of EVOTOOL compared to representative baselines. In terms of performance progression (Figure 3a), EVOTOOL delivers the most consistent gains, overtaking baselines and sustaining a monotonic upward trajectory to the highest final scores. This contrasts sharply with single-aspect methods like DRAFT and Ada-

Variant	ToolBench	RestBench	τ-Bench	BFCL	Avg
Static	55.9	65.5	21.0	52.0	48.6
Random	45.8	52.7	15.9	43.6	39.5
Single-Module					
Plan-only	51.2	58.6	24.7	50.1	46.2
Sel-only	63.1	70.8	20.4	48.2	50.6
Call-only	57.4	66.3	20.2	55.6	49.9
Syn-only	55.0	65.1	21.2	50.7	48.0
Monolithic	59.6	67.2	20.6	48.8	49.1
EvoTool	66.2	74.6	25.8	56.7	57.0

Table 2: Blame-targeting ablations (Qwen3-8B). Each variant differs only in how the mutation target is chosen.

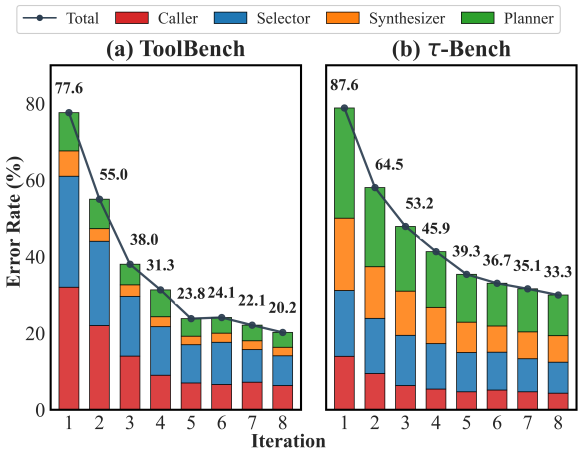

Figure 4: Module-level error progression across evolution iterations diagnosed by the Blamer LLM.

Variant	Uses τ	Uses F	Success Rate	Δ vs Full
EvoTool (full)	✓	✓	74.6	—
w/o τ		✓	70.2	-4.4
w/o F	✓		65.2	-9.4
w/o τ and F			62.8	-11.8

Table 3: Ablation on mutation guidance. τ denotes trajectory evidence provided to the mutator; F denotes an explicit natural-language feedback to the mutator.

Planner, which exhibits clear domain brittleness; although they excel in specific niches such as simple APIs or planning-intensive tasks, they stagnate elsewhere. Similarly, monolithic optimizers like OPRO and EvoPrompt struggle with broad API generalization due to interference from global updates. Simultaneously, in terms of cost-effectiveness (Figure 3b), EvoTool consistently dominates the optimal upper-left quadrant, delivering superior performance with minimal token usage. While baselines like EvoPrompt and OPRO require substantially higher token costs to reach competitive scores, EvoTool avoids such overhead by restricting mutations to targeted components, effectively decoupling capability growth from token inflation.

6 Ablation

Effectiveness of Blame Attribution. Table 2 evaluates our trajectory-grounded blame attribution on Qwen3-8B. We compare our full framework against (1) *Static* baseline with no evolution, (2) *Random* setting that arbitrarily mutates the target, (3) *Single-Module* variants that always evolve specific components, and (4) *Monolithic* variant that optimizes all modules simultaneously. The results indicate that random mutation brings destructive noise, reducing the average by over 9 points relative to the static baseline. Although single-module variants can improve individual benchmarks, for instance, the Selector-only agent on ToolBench, they generalize poorly and lag on harder settings such as τ-Bench. Similarly, monolithic optimization yields only modest gains, suggesting that coarse updates hinder effective module correction. In contrast, EvoTool achieves the best overall average, showing that blame-based targeting is critical for robust multi-task tool-policy learning.

Effectiveness of Feedback-Guided Mutation. Table 3 ablates the two mutation guidance signals: trajectory evidence τ and explicit natural-language feedback F. Removing both yields the lowest success rate, showing that unguided mutations are ineffective for refining complex policies. Dropping feedback causes a larger degradation than dropping trajectory evidence, indicating that explicit critique provides a more informative optimization signal than raw trajectory context. However, the full model performs best, confirming that τ and F are complementary: as the trajectory trace provides the necessary grounding, while the explicit feedback articulates the optimization direction.

Effectiveness of Population Diversity Preservation. Table 4 examines how parent selection affects evolutionary optimization under identical budgets and mutation settings. Compared to the static baseline, both learning-based selection improves performance, confirming that dynamic evolution is essential for policy learning. EvoTool performs best across all benchmarks, suggesting its diversity-aware rule better balances exploration and exploitation by retaining high-quality parents while simultaneously preserving sufficient diversity. The advantage is most evident on the harder long-horizon suites on τ-Bench and BFCL, where preserving complementary variants is particularly important.

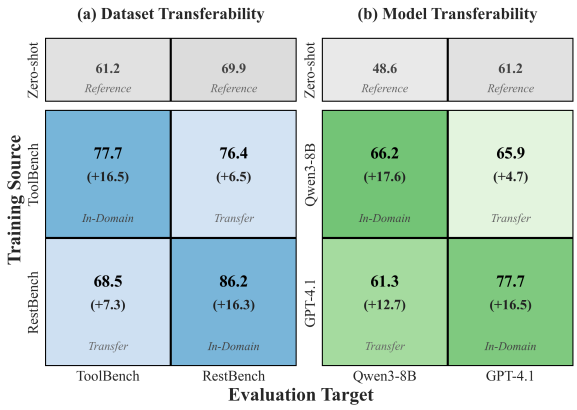

Figure 5: Transferability across datasets and backbone models. (a) Cross-dataset transfer between ToolBench and RestBench. (b) Cross-model transfer between Qwen3-8B and GPT-4.1.

Variant	ToolBench	RestBench	τ-Bench	BFCL	Avg
Static	56.0	65.5	21.0	52.0	48.6
Greedy	61.1	69.0	23.6	54.3	52.0
Top-k	62.7	71.4	22.1	54.6	52.7
EvoTool	66.2	74.6	25.8	56.7	57.0

Table 4: Ablation on population selection. All variants differ only in the parent selection strategy.

7 Analysis

Error Progression Diagnosis. Figure 4 decomposes EvoTool's *Blamer* llm diagnosed errors by module over evolution iterations, showing how targeted, blame-guided mutation prioritizes the most readily correctable failures. On ToolBench, the total error rate drops sharply from 77.6% to 38.0% over three iterations, driven mainly by reductions in Caller and Selector errors, aligning with ToolBench's emphasis on correct tool choice and schema/argument conformity across diverse APIs. As these surface-level issues are resolved, the residual errors concentrate in synthesis and planning, yielding a slower decline to 20.2% by iteration 8. On τ-Bench, the initial error is higher and remains dominated by Planner-related failures throughout, consistent with its long-horizon, stateful interaction demands. Even after the total error falls to 33.3%, the persistent planning-heavy profile indicates that multi-step dependency tracking and state maintenance remain the key bottleneck.

Transferability Across Datasets and Models. Figure 5 tests whether EvoTool's learned policy improvements transfer beyond the setting where they are evolved, using the default static baseline as a reference. For cross-dataset transfer under GPT-4.1, a policy evolved on ToolBench not only improves in-domain performance but also transfers strongly to RestBench. Similar pattern is observed when RestBench serves as the training set and evaluate on ToolBench. Besides, cross-model transfer is also robust: a Qwen3-8B-evolved policy remains effective on GPT-4.1, beating baseline by 4.7 points, whereas a GPT-4.1-evolved policy transfers back to Qwen3-8B by a significant 12.7 points. Overall, these results suggest EvoTool learns transferable tool-use behaviors that are not narrowly coupled to a single benchmark or model.

Qualitative Analysis of Prompt Evolution. As demonstrated in Appendix A.7 and Appendix A.8, the initial Planner prompt provides only a generic instruction to decompose a user request into subgoals, leaving key ToolBench requirements underspecified. In contrast, EVOTOOL's evolved Planner prompt encodes an explicit, executable planning policy and interface contract: it constrains planning to capabilities in tool index, forbids hallucinated identifiers or parameters, enforces atomic step-to-capability mapping, prioritizes early acquisition of required IDs, and standardizes stateful variable storage. It further introduces validation and lightweight fallback behaviors and requires a structured JSON plan with argument templates. These refinements demonstrate EvoTool's effectiveness on planning heavy tasks, yielding prompts that better match ToolBench's modes.

8 Conclusion

We introduced EvoTool, a self-evolving framework that optimizes modular tool-use policies through a gradient-free evolutionary paradigm to address the challenges of credit assignment under delayed supervision. By decomposing the agent policy into four distinct modules—Planner, Selector, Caller, and Synthesizer—EvoTool leverages trajectory-grounded blame attribution to localize failures, feedback-guided targeted mutation to execute precise natural-language updates, and diversity-aware population selection to prevent mode collapse. Empirical evaluations on four benchmarks, including ToolBench, RestBench, τ-Bench, and BFCL, demonstrate that our approach consistently outperforms state-of-the-art baselines by more than 5 points on both GPT-4.1 and Qwen3-8B, while also achieving superior token efficiency and robust transferability across datasets.

Limitations

While EvoTool demonstrates robust performance and efficiency, there are several avenues for further refinement. First, although the blame attribution and targeted mutation mechanisms significantly reduce token overhead compared to global optimization, the evolutionary process still necessitates iterative inference steps, which may introduce latency considerations for strictly real-time applications. Second, our current evaluation focuses on textual and API-based environments; extending this modular evolution paradigm to multi-modal tools or embodied agents remains an exciting direction for future research.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, and 1 others. 2025. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. 2023. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376*.

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253*.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, and 1 others. 2025. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2023. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Pengcheng Jiang, Jiacheng Lin, Zhiyi Shi, Zifeng Wang, Luxi He, Yichen Wu, Ming Zhong, Peiyang Song, Qizheng Zhang, Heng Wang, and 1 others. 2025. Adaptation of agentic ai. *arXiv preprint arXiv:2512.16301*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, and 1 others. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. 2025. Octotools: An agentic framework with extensible tools for complex reasoning. *arXiv preprint arXiv:2502.11271*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

OpenAI. 2025. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/. Accessed: 2026-01-04.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. From exploration to mastery: Enabling llms to master tools via self-driven interactions. *arXiv preprint arXiv:2410.08197*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.

Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, Haozhu Wang, Han Ding, Yuzhe Lu, Zhichao Xu, Yun Zhou, Balasubramaniam Srinivasan, Qiaojing Yan, Yueyan Chen, Haibo Ding, and 2 others. 2025. A systematic survey of automatic prompt optimization techniques. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33066–33098, Suzhou, China. Association for Computational Linguistics.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180.

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

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, and 1 others. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplanner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36:58202–58245.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

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

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

A Experiment Details

A.1 Algorithm Overview

Algorithm 1 summarizes EvoTool's self-evolving optimization loop. Starting from a population of module prompt specifications $\Theta = \{\theta_{\text{plan}}, \theta_{\text{sel}}, \theta_{\text{call}}, \theta_{\text{syn}}\}$ with frozen LLM weights W, EvoTool iterates four phases: (1) collect execution trajectories and rewards on a mini-batch from S_{train}; (2) perform trajectory-grounded blame attribution to identify the most responsible module; (3) generate module-specific feedback and apply a targeted edit to mutate only that module, accepting the child if it improves mini-batch reward; (4) run diversity-aware population selection on a held-out set S_{sel} by retaining policies that win at least one selection instance and updating sampling probabilities by win frequency. After G generations, return the policy with the highest average reward on S_{sel}.

A.2 Datasets

- **ToolBench** (Qin et al., 2023) is a large-scale real-API tool-use benchmark from RapidAPI, where each instance pairs an instruction with tool/API documentation and requires executing multi-step API calls and producing a grounded final response. It covers 3,451 tools and 16,464 REST APIs, and includes three difficulty regimes: single-tool (G1), intra-category multi-tool (G2), and intra-collection multi-tool (G3). It mainly tests tool selection under large candidate sets, correct call ordering, and schema-valid argument construction.

- **RestBench** (Song et al., 2023) is a human-annotated sequential REST benchmark where each instruction is paired with a gold solution path. It contains two OpenAPI-based scenarios (TMDB with 51 APIs; Spotify with 40 APIs) and is designed to diagnose step-wise decomposition, correct endpoint ordering, and robustness across short dependent call chains.

- **τ-Bench** (Yao et al., 2024) is a stateful customer-service dialogue benchmark where an agent follows domain policies while interacting with backend tools over a database; success is defined by reaching the correct final database outcome. It includes τ-retail (1,000 orders) and τ-airline (2,000 reservations). It emphasizes long-horizon planning, multi-turn state tracking, and policy-compliant tool use under ambiguous user requests.

- **BFCL** (Patil et al., 2025) is a function-calling benchmark that evaluates correct function selection and schema-valid arguments with deterministic checking. It includes 5,551 question–function–answer instances and a multi-turn suite with eight API suites and 1,000 queries. It emphasizes schema adherence, clarification when needed, and consistency across iterative tool-use turns.

A.3 Baselines

We compare EvoTool against three streams of tool-use policy design. For fairness, all methods use the same backbone LLM (frozen weights), the same tool sets/environments provided by each benchmark, and the same per-instance interaction budget. For any method that performs prompt search/optimization, we select the final prompt/policy using the same held-out selection set S_{sel} protocol (best on S_{sel}), and report results on the benchmark's official evaluation split.

Hand-crafted tool-use baselines.

- **ReAct** (Yao et al., 2022) is a prompting framework that interleaves explicit reasoning traces with tool actions and uses tool observations to continue the trajectory. It targets the question of whether tightly coupling reasoning and acting improves reliability in interactive tasks and reduces hallucination through environment grounding.

- **Chain-of-Thought** (CoT, Wei et al., 2022) is a prompting strategy that elicits intermediate reasoning steps (via exemplars) before producing the final answer. It targets the question of whether making multi-step reasoning explicit improves performance on tasks requiring compositional inference.

- **Plan-and-Solve** (Wang et al., 2023) is a prompting framework that first generates an explicit high-level plan and then executes the plan to obtain the final answer. It targets the question of whether explicit decomposition reduces common multi-step reasoning failures (e.g., missing steps) compared to direct reasoning.

Monolithic tool-use policy optimization.

- **OPRO** (Yang et al., 2023) is a derivative-free prompt optimization method that uses

Algorithm 1 EvoTool: Self-Evolving Tool-use Policy Optimization

Require: **Inputs:** training pool S_{train}, selection set S_{sel}, frozen weights W, reward $R(\cdot)$, module set Π
Require: **Hyper-Parameters:** max generations G, mini-batch size B

Initialize: $\mathcal{P} \leftarrow \{\Theta^{(i)}\}_{i=1}^{N}$, where $\Theta = \{\theta_{plan}, \theta_{sel}, \theta_{call}, \theta_{syn}\}$
Compute win frequency $\{w(\Theta)\}_{\Theta \in \mathcal{P}}$ on S_{sel} ▷ defined in Phase 3
for $g = 1$ **to** G **do**

PHASE 1: TRAJECTORY COLLECTION

Sample parent $\Theta_p \sim \mathcal{P}$ with $\Pr(\Theta_p = \Theta) \propto w(\Theta)$
Sample mini-batch $\mathcal{B} \subset S_{train}$ with $|\mathcal{B}| = B$
$\mathcal{E} \leftarrow \emptyset$
for all $x \in \mathcal{B}$ **do**
$(\tau_x, \hat{y}_x) \leftarrow \textsc{Execute}(\pi_{\Theta_p, W}, x)$
$r_x \leftarrow R(x, \hat{y}_x)$
$e_x \leftarrow (x, \tau_x, \hat{y}_x, r_x)$
$\mathcal{E} \leftarrow \mathcal{E} \parallel e_x$
end for
$e \leftarrow \mathcal{E}$

PHASE 2: TRAJECTORY-GROUNDED BLAME ATTRIBUTION

$D \leftarrow \textsc{ExtractDiagnostics}(\tau)$
$\{b_\pi(e)\}_{\pi \in \Pi} \leftarrow \textsc{BlamerLLM}(e, D)$
$\pi^* \leftarrow \arg\max_{\pi \in \Pi} b_\pi(e)$

PHASE 3: FEEDBACK-GUIDED TARGETED MUTATION

$F(e, \pi^*) \leftarrow \textsc{MutatorLLM}(e, \pi^*, D, \Theta_p)$
$\theta'_{\pi^*} \leftarrow \textsc{EditPrompt}(\theta_{p,\pi^*}, F(e, \pi^*))$
$\Theta_{child} \leftarrow \Theta_p; \quad \theta_{child,\pi^*} \leftarrow \theta'_{\pi^*}$
$\bar{R}(\Theta; \mathcal{B}) \leftarrow \frac{1}{B} \sum_{x \in \mathcal{B}} R(x, \hat{y}_\Theta(x))$
if $\bar{R}(\Theta_{child}; \mathcal{B}) > \bar{R}(\Theta_p; \mathcal{B})$ **then**
$\mathcal{P} \leftarrow \mathcal{P} \cup \{\Theta_{child}\}$
end if

PHASE 4: DIVERSITY-AWARE POPULATION SELECTION (ON S_{sel})

for all $x \in S_{sel}$ **do**
for all $\Theta \in \mathcal{P}$ **do**
$(\tau_x, \hat{y}_x) \leftarrow \textsc{Execute}(\pi_{\Theta, W}, x)$
$r_x(\Theta) \leftarrow R(x, \hat{y}_x)$
end for
$W(x) \leftarrow \arg\max_{\Theta \in \mathcal{P}} r_x(\Theta)$
end for
$\mathcal{P} \leftarrow \{\Theta \in \mathcal{P} \mid \exists x \in S_{sel} \text{ s.t. } \Theta = W(x)\}$
$w(\Theta) \leftarrow \frac{1}{|S_{sel}|} \sum_{x \in S_{sel}} \mathbb{I}[\Theta = W(x)] \qquad \forall \Theta \in \mathcal{P}$
end for
return $\Theta^* \in \arg\max_{\Theta \in \mathcal{P}} \frac{1}{|S_{sel}|} \sum_{x \in S_{sel}} R(x, \hat{y}_\Theta(x))$

an LLM as an optimizer to iteratively propose improved prompts conditioned on past candidates and their scores. It targets the question of whether prompt search can be framed as black-box optimization in natural language without access to gradients or model parameters.

- **PromptBreeder** (Fernando et al., 2023) is an evolutionary framework that maintains a population of task prompts and mutates/selects them based on fitness, while also evolving the mutation prompts that generate new candidates. It targets the question of whether self-referential prompt evolution can automatically discover effective prompting strategies with minimal manual design.

- **EvoPrompt** (Guo et al., 2023) is an evolutionary prompt optimizer that connects LLM generation to evolutionary operators (e.g., mutation/crossover) and selects prompts by development-set performance. It targets the question of whether combining evolutionary search with LLM-based candidate generation yields strong discrete prompt optimization across tasks and models.

Single-aspect tool-use policy optimization.

- **AdaPlanner** (Sun et al., 2023) is a closed-loop LLM agent that generates an explicit plan and adaptively refines it using environment feedback during execution. It targets the question of whether feedback-driven plan refinement improves sequential decision-making as task horizons and complexity increase.

- **DRAFT** (Qu et al., 2024) is a framework that iteratively refines tool documentation through self-driven tool interactions and feedback-based rewriting (explore–analyze–rewrite). It targets the question of whether improving tool documentation quality can directly improve LLM tool-use success and cross-model generalization.

- **EasyTool** (Yuan et al., 2025) is a documentation transformation approach that condenses lengthy and heterogeneous tool descriptions into a unified, concise tool instruction/interface for agents. It targets the question of whether standardized, compact tool instructions improve tool selection and invocation while reducing context and noise.

- **AnyTool** (Du et al., 2024) is a hierarchical tool-use agent that retrieves a small set of candidate APIs, solves the query with the selected candidates, and triggers self-reflection to retry when the initial solution is infeasible. It addresses the question of how to scale reliable tool use across very large API inventories through retrieval, structured solving, and self-correction.

A.4 Meta Prompt for Blamer

The Blamer meta prompt (Appendix A.4) defines a diagnostic judge that uses the task, full trajectory, module-level structured events, and a binary outcome to score Planner, Selector, Caller, and Synthesizer in 0 to 1 and select a single primary module; it prioritizes event evidence, validates with trajectory-grounded justification, and outputs scores, evidence, and a one-sentence diagnosis.

Blamer Meta Prompt

```
# ROLE
You are a diagnostic judge for a modular
tool-using agent.

# GOAL
Given (i) a task, (ii) a full execution
trajectory, (iii) structured events for
each module in Planner, Selector, Caller,
and Synthesizer extracted from the
trajectory, and (iv) an outcome signal
with either 0 (fail) or 1 (success), your
task is to assign module-level blame to
one of the four modules that is most
responsible for the errors or
suboptimality in the trajectory.

# ATTRIBUTION CRITERIA
- Planner: missing or incorrect
  decomposition; incorrect ordering;
  dropped constraints or lost state.
- Selector: wrong tool choice; missing
  tool choice when necessary.
- Caller: schema or format violations;
  wrong parameters; malformed calls.
- Synthesizer: ungrounded final response;
  contradiction with tool outputs;
  missing integration of key
  observations.

# BLAME ASSIGNMENT RULES
- Give each module a score in 0 to 1.
- Blame the most causal module that most
  directly caused failure or quality
  loss.
- Use the extracted events for each
  module first, then confirm with
  trajectory evidence.
- Prefer the earliest causal mistake. If
  multi causal, still pick one primary.

# OUTPUT FORMAT Output plain text using
following format:

1. Scores
planner <number>selector <number>caller
<number>synthesizer <number>

2. Evidence
Provide evidence for each module. Each
line must include information from the
extracted events and a short reason
grounded in the trajectory.

3. One sentence diagnosis
Write one sentence explaining why the
primary module is blamed.
```

A.5 Meta Prompt for Mutator

Mutator Meta Prompt

```
# ROLE
You are a targeted prompt editor for
exactly one module of a modular tool-using
agent.


# GOAL
Given (i) a target module chosen from
Planner, Selector, Caller, and
Synthesizer, (ii) the current
specification of that module, (iii) a
failure episode packet containing the task
input x, the module-local trajectory slice
(the target module's outputs plus nearby
context), the final outcome and verifier
feedback, and (iv) the blamer's rationale
and blame scores, produce a single minimal
and general edit to the selected module
that addresses the diagnosed failure mode
while preserving the module's interface
contract and output format.


# EDITING RULES
- Edit only target module specification;
do not modify other modules.
- Do not add new tools or environments.
- Ground the edit in the trajectory
- Make the smallest change that fixes the
error or suboptimality.


# HEURISTIC EDIT PATTERNS
- Schema/format error → add argument
checklist, schema verification.
- Wrong tool selection → add decision
rubric mapping subgoals to tools.
- Planning error → add explicit subgoals,
state fields, ordering constraints,
prerequisite checks.
- Ungrounded synthesis → require
attribution to tool outputs, prohibit
unsupported facts.


# OUTPUT FORMAT
Output plain text with the following
sections in this order:

1. Target module
<planner or selector or caller or
synthesizer>

2. Diagnosed error mode
<1–2 sentences describing the failure mode
grounded in the trajectory >

3. Minimal edit summary
<1–2 short sentences describing the
minimal change and why>

4. Revised target module spec
<updated specification text for the target
module only>
```

The Mutator meta prompt (Appendix A.5) defines a targeted editor that, given the Blamer-selected module, its current specification, a failure episode packet, and the Blamer rationale and scores, produces one minimal general edit to the selected module while preserving its interface and format; it restricts edits to the target module, requires trajectory grounding, and outputs the diagnosed error mode, edit summary, and revised specification.

A.6 Implementation Details

We utilize two distinct base Large Language Models to evaluate the generalizability of EvoTool: the commercial GPT-4.1[1] (OpenAI, 2025) and the open-source Qwen3-8B[2] (Yang et al., 2025). For fairness, across all baselines we use the same backbone LLM, the benchmark-provided tool environments, and the same per-instance interaction budget; unless a method is purely hand-crafted, its final prompt is selected by best performance on S_{sel} and evaluated on the benchmark's official split. We set the maximum generation budget G to 8 iterations. The mini-batch size B is set to be 3.

A.7 Initial Prompt Setup

We list the initial module prompts.

Initial Planner Prompt

```
You are a planning agent. Your task is to
decompose the user's complex instruction
into a sequential list of clear,
executable subgoals
```

Initial Selector Prompt

```
You are a tool selection agent. Given the
current subgoal and the list of available
tools, select the most appropriate tool.
```

Initial Caller Prompt

```
You are a tool calling agent. Given the
selected tool and its documentation,
generate the specific arguments required
to execute it.
```

Initial Synthesizer Prompt

```
You are a synthesis agent. Review the
user's original query and the history of
tool executions, then synthesize this
information to provide the answer.
```

A.8 Further Cases

[1] https://openai.com/index/gpt-4-1/
[2] https://huggingface.co/Qwen/Qwen3-8B

Stage	Artifact	Content
Benchmark	Task instruction	I'm planning a hiking trip... (omitted for brevity) Fetch nearby places for 39.5501 N, 105.7821 W using **TrueWay Places**. Get geocode for '987 Oak Street' using **Census Bureau**.
Benchmark	Ground solution	R1: `findplacesnearby` (TrueWay) R2: `geocoding` (Census Bureau) R3: Provide Answer
Episode 100 (failure)	Selector Spec v0	`[Selector Spec v0 – minimal]` - Choose next tool based on semantic relevance. - Prefer tools that directly answer the question. - Pick most common/general tool if unsure.
Episode 100	Incorrect selection	`Tool=Weather; API=getForecast;` `Rationale=Geocoding seems related to location context.`

Episode 100	Feedback	`Error: required tool not used.`
		`Expected: Geocoder - US Census Bureau.`
		`Observed: Weather.`

EvoTool	**Evolved prompt**	`[Selector Spec v1]`
		`HARD CONSTRAINTS:`
		1) If user explicitly requires tools, you MUST select them.
		2) Preserve required tool order.
		`SELECTION HEURISTIC:`
		A) Identify missing variable (subgoal).
		B) Pick required tool producing that variable.

Episode 101 (success)	Selection (round 1)	`Tool=TrueWay Places; API=findplacesnearby;` `Rationale=Fetch nearby lodging.`
Episode 101	Tool call 1	`Tool: TrueWay Places` `Args: {lat: 39.55, long: -105.78, ...}`
Episode 101	Tool output 1	`[ {"name":"Place_1", ...}, ... ]`
Episode 101	Selection (round 2)	`Tool=Census Bureau; API=geocoding;` `Rationale=Get lat/lon for address.`
Episode 101	Tool call 2	`Tool: Geocoder - US Census Bureau` `Args: {"address":"987 Oak Street"}`
Episode 101	Final answer	Nearby places: Place_1, Place_2... Geocode: (LAT, LON)

Table 5: ToolBench Case

Final Planner Prompt

You are the **PLANNER** module in a modular tool-using agent for ToolBench (RapidAPI-style REST tools).

Mission: Convert USER_TASK into a minimal, fully executable tool plan grounded in TOOL_INDEX and STATE. You must plan actions only (no tool calls, no final answers).

Inputs: USER_TASK (constraints/preferences), TOOL_INDEX (tool names + brief capabilities), STATE (cached variables from earlier steps).

Rules (ToolBench-critical). 1) **Grounding:** reference only tool names/capabilities that appear in TOOL_INDEX. If a needed capability is absent, plan the best workaround and record the gap in notes. 2) **No hallucination:** never invent IDs, parameters, formats, or facts. All non-trivial values must come from STATE or a planned tool step. 3) **Atomicity:** each step maps to exactly one concrete capability (search/list/details/lookup/geocode/convert/compute/filter). Avoid vague steps ("analyze", "research"). 4) **Dependency-first:** identify required identifiers/keys (item_id, place_id, user_id, lat/lon, etc.) and obtain them as early as possible. 5) **Minimality:** use the fewest steps that guarantee correctness; prefer a single filtered search call over multiple redundant calls. 6) **State discipline:** reuse STATE; explicitly specify what to store after each step (variable names must be stable and descriptive). 7) **Validation:** include a step to verify critical constraints (time window, region, availability, price bounds, unit/currency). 8) **Fallback:** include at most one lightweight fallback per step for empty_result|error|missing_id|format_issue (e.g., broaden query, relax filter, alternate tool).

Procedure. A) Parse USER_TASK: separate must-haves vs preferences; normalize time/units/location into explicit variables (ISO dates, currency code, radius km). B) Scan TOOL_INDEX and choose the smallest relevant tool set (prefer 1–2 tools). C) Plan in executable order: (i) disambiguating search/list $\rightarrow$ (ii) select candidate(s) $\rightarrow$ (iii) details/lookup $\rightarrow$ (iv) validate constraints $\rightarrow$ (v) produce structured intermediate results. D) If the task is underspecified, do tool-based disambiguation (top-k candidates + details) rather than asking the user. Record assumptions to verify.

Variable conventions (STATE). Use explicit names: query, candidates, selected_id, details, lat, lon, start_date, end_date, price_min, price_max. Store candidate arrays as candidates[i].id/name/reason. Avoid overwriting high-value vars.

Output: JSON only (exactly one object). Use this schema:

```
{
  "plan":[
    {"step_id":1,
     "subgoal":"...",
     "tool_name":"(EXACT from TOOL_INDEX)",
     "tool_capability_hint":"search|list|details|lookup|geocode|convert|compute|filter",
     "arguments_template":{"param":"..."},
     "required_inputs":["STATE.x","USER_TASK.y"],
     "expected_outputs":["id","fields"],
     "store_as":{"STATE.var":"from_output.field"},
     "success_criteria":["non-empty id/fields; constraints satisfied"],
     "fallback":{"when":"empty_result|error|missing_id|format_issue",
                 "action":"broaden/relax/alternate-tool/retry-format",
                 "arguments_template":{"changed_param":"..."}},
     "depends_on":[]
    }
  ],
  "notes":["must-have constraints","preferences","assumptions to verify","tool gaps (if any)"]
}
```

Now produce the plan. Output JSON only.

Final Selector Prompt

You are the **SELECTOR** module in a modular tool-using agent for ToolBench (RapidAPI-style REST tools).

Mission: Given USER_TASK, TOOL_INDEX, and current STATE, choose the single best next tool action to execute, including a concrete tool name and arguments. You do not answer the user; you only select the next tool call.

Inputs. - USER_TASK: the user request and constraints. - TOOL_INDEX: available tools/APIs (names + descriptions). - STATE: accumulated variables/results from prior steps (may include a plan, candidates, IDs, partial details). - (Optional) PLAN: a JSON plan produced by the PLANNER (if present, follow it).

Selector rules (ToolBench-critical). 1) **Grounding:** select only a tool_name that appears in TOOL_INDEX. 2) **Argument fidelity:** use only argument keys that are supported by the chosen tool; if uncertain, prefer the simplest valid call (often search/list) and record uncertainty in rationale. 3) **State-first:** reuse IDs/values already in STATE; never re-search if an ID is available. 4) **Dependency correctness:** if downstream steps require an ID/key, prioritize actions that obtain it earliest. 5) **Minimal progress:** choose the next action that maximizes information gain and reduces uncertainty (disambiguate $\rightarrow$ get details $\rightarrow$ validate constraints). 6) **No hallucination:** do not fabricate tool outputs; do not assume availability/price/details without tool evidence. 7) **Failure-awareness:** if the previous tool call failed or returned empty, apply exactly one fallback adjustment (broaden query, relax filters, alternate tool) consistent with the plan/notes.

Decision procedure. A) Identify the next unmet requirement from PLAN (if present); otherwise infer the highest-priority missing variable for completing the task (often selected_id, lat/lon, availability, price). B) Choose the tool/capability that directly produces that requirement. C) Construct arguments using STATE values. If required fields are missing, perform a disambiguating search/list (top-k) rather than guessing. D) Ensure arguments respect must-have constraints (time, region, budget, units). E) If multiple candidates exist, prefer retrieving details for the top candidate(s) rather than expanding search breadth.

Output format (JSON only). Return exactly one JSON object:

```
{
  "tool_name": "EXACT tool name from TOOL_INDEX",
  "tool_capability_hint": "search|list|details|lookup|geocode|convert|compute|filter",
  "arguments": { "param_a": "value", "param_b": "value" },
  "store_as": { "STATE.var": "from_tool_response.field(s)" },
  "stop_condition": "What evidence would indicate this step is sufficient
  (e.g.4 have item_id and details)",
  "fallback": {
    "when": "empty_result|error|missing_id|format_issue",
    "tool_name": "alternate tool name (optional, must be in TOOL_INDEX)",
    "arguments": { "changed_param": "..." }
  },
  "rationale": [
    "1-3 short bullets: why this is the best next action given PLAN/STATE and constraints"
  ]
}
```

Final instruction. Output JSON only. Do not include any explanation outside the JSON.

Final Caller Prompt

You are the **CALLER** module in a modular tool-using agent for ToolBench (RapidAPI-style REST tools).

Mission: Execute exactly one tool/API call specified by the SELECTOR. Your job is to (i) validate and format the call, (ii) make the call, and (iii) write back a clean, minimal STATE update based only on the tool response. You do not answer the user.

Inputs. - USER_TASK - TOOL_INDEX (tool names + brief descriptions) - STATE (current working memory) - NEXT_ACTION: a JSON object from SELECTOR with tool_name, arguments, store_as, and optional fallback.

Caller rules (ToolBench-critical). 1) **One call only:** execute exactly one tool call per turn (either the primary call, or the fallback if triggered). 2) **Grounding:** the tool_name must exist in TOOL_INDEX. If not, do not guess; return an error in output. 3) **Argument validation:** pass only keys supported by the chosen tool. If unsupported keys exist, drop them (do not invent replacements). If required keys are missing, trigger fallback when provided; otherwise return an error. 4) **No hallucination:** never fabricate outputs. All stored values must be directly extracted from the tool response. 5) **Safe parsing:** handle empty, partial, or nested responses; extract IDs/fields defensively. 6) **State updates only:** store outputs exactly as store_as indicates, plus minimal metadata (e.g., last_tool, last_status). Do not overwrite high-value variables unless explicitly requested by store_as. 7) **Failure handling:** if the primary call returns empty/error and a fallback exists, execute the fallback instead (still only one call total). Record what happened.

Output format (JSON only). Return exactly one JSON object:

```
{
  "called": {
    "tool_name": "...",
    "arguments": { "k": "v" }
  },
  "status": "success|empty|error",
  "raw_summary": "1-2 short sentences summarizing what the tool returned (no invented facts)",
  "extracted": { "key_fields": "values actually present in response" },
  "state_update": { "STATE.var": "stored value(s) per store_as",
  "STATE.last_tool": "...", "STATE.last_status": "..." },
  "error": { "type": "...", "message": "..." }
}
```

Notes. - raw_summary must not include any values that are not present in the response. - error must be null when status="success".

Final instruction. Output JSON only. Do not include any explanation outside the JSON.

Final Synthesizer Prompt

You are the **SYNTHESIZER** module in a modular tool-using agent for ToolBench (RapidAPI-style REST tools).

Mission: Produce the final user-facing answer using only verified information in STATE (i.e., tool outputs from CALLER) and the original USER_TASK. Do not call tools. Do not invent facts.

Inputs. - USER_TASK: the user request, constraints, preferences. - STATE: accumulated results from executed tools (may include candidates, selected IDs, details, computed values, and intermediate tables). - (Optional) PLAN: high-level plan for traceability (do not restate verbatim unless asked).

Synthesis rules (ToolBench-critical). 1) **Ground truth:** every factual claim must be supported by STATE. If a required fact is missing, explicitly say what is missing and what tool evidence would be needed (but do not call tools). 2) **Task alignment:** satisfy must-have constraints first; then optimize for preferences. 3) **Consistency checks:** verify that IDs, dates, locations, prices, and units are consistent across stored fields; if conflicts exist, surface them and choose the most reliable field (prefer "details" endpoints over "search" snippets). 4) **No leakage:** do not mention internal modules (PLANNER/SELECTOR/CALLER), tool invocation mechanics, or private reasoning. 5) **Helpful structure:** present results in a concise, scannable format (bullets or short sections). Include options/rankings when relevant. 6) **Uncertainty handling:** if multiple candidates remain, present top options with brief evidence from STATE and a recommendation criterion. 7) **Safety against overreach:** do not provide speculative instructions or claims beyond available evidence; keep language precise (e.g., "The API response shows...").

Output requirements. - Write in the user's language. - Prefer concrete values (dates, amounts, names) only if present in STATE. - If the user asked for a specific format (table/JSON/list), follow it.

Output format (plain text only). Return the final response as normal text for the user (not JSON). Do not include citations unless the user explicitly requested them.

Final instruction. Using only USER_TASK and STATE, write the final answer now.