META-SKILL LEARNING FOR TOOL-USING ENVIRONMENT ADAPTATION IN LARGE LANGUAGE MODEL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents can often generalize to new tool-using environments from pre-prepared tool definitions and environment descriptions; however, such specifications rarely capture the implicit operational rules that govern success in non-standard environments. Interaction trajectories instead provide rich environment-specific knowledge, exposing structured patterns of success, failure, and recovery, and suggesting that such experience can be systematically distilled into reusable skills for accomplishing new tasks within the same environment. In this paper, we study training an LLM agent to rapidly adapt to new environments by learning a meta-skill, defined as the ability to explore a new environment, summarize environment-specific operational knowledge from limited interactions, and apply the resulting skill to subsequent tasks. We formulate this problem as meta-reinforcement learning over a distribution of tool-using environments and propose a meta-skill learning framework for environment adaptation. Experiments on multiple benchmarks show that environment-specific adaptation generally improves performance on new tasks, supporting that tasks within the same environment share reusable skills. Moreover, the proposed framework outperforms strong baselines on both in-distribution and out-of-distribution environments.

1 INTRODUCTION

Large language models (LLMs) are increasingly evolving from passive text generators into autonomous agents that can accomplish tasks by interacting with complex environments. Enabled by recent advances of LLMs in tool use, reasoning, reflection, and planning (Yao et al., 2023; Shinn et al., 2023; Wang et al., 2023; Xu et al., 2026), LLM agents can iteratively interpret environmental feedback, maintain reasoning over intermediate states, select appropriate actions, and execute them through external tools. These tools include search engines and web search systems (Yao et al., 2023; Xiao et al., 2026), APIs (Schick et al., 2023; Patil et al., 2023), browsers and web interfaces (Deng et al., 2023; Zhou et al., 2023), file systems and code repositories (Yang et al., 2024; 2023), and desktop or mobile software interfaces (Xie et al., 2024; Zhang et al., 2024; 2023). Such capabilities have enabled a broad range of applications, including deep research assistants and agentic search systems (Xiao et al., 2026; Gur et al., 2023), digital assistants (Zhang et al., 2024; 2023), and coding agents (Yang et al., 2024; 2023).

Modern LLM agents already achieve strong performance in standard environments, and they can often transfer reasonably well to previously unseen environments (OpenAI, 2025; Anthropic, 2025b) when provided with sufficiently informative environment descriptions, tool schemas, API signatures, and resource documents, such as the structured context exposed through Model Context Protocol (MCP)-style interfaces (Schick et al., 2023; Patil et al., 2023; Model Context Protocol, 2024). Such specifications provide useful priors about available tools, expected inputs and outputs, and high-level workflows, often enabling nontrivial zero-shot performance even without environment-specific training (Deng et al., 2023; Zhou et al., 2023; Xie et al., 2024). However, such generalization is largely grounded in the explicit part of the environment specification. In non-standard environments, many of the rules that actually determine success are implicit, local, and interaction-dependent, and are therefore difficult to describe exhaustively in tool definitions, environment descriptions, or MCP resource documents. For example, a tool may expose a valid argument schema but only function under

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

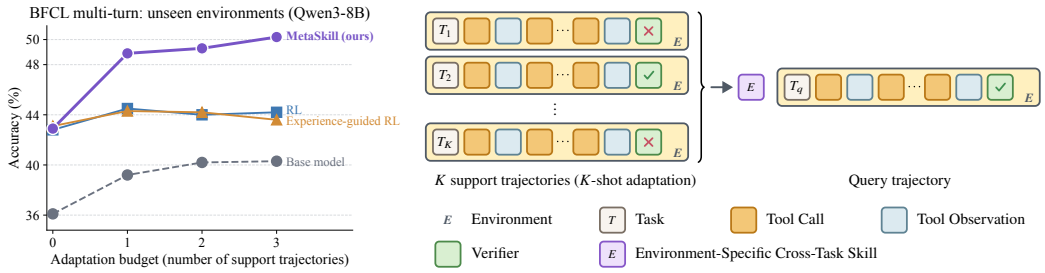


Figure 1: Motivation for environment adaptation. *Left:* Accuracy on BFCL multi-turn unseen environments with Qwen3-8B across adaptation budgets and training methods. *Right:* Illustration of environment adaptation, where support trajectories are used to construct an environment-specific cross-task skill for the query trajectory.

a stricter undocumented format; a file system may follow project-specific path conventions; an API may obey environment-specific defaults or side effects; and successful execution may depend on an unstated workflow order. As a result, even a strong agent with access to rich documentation may still make systematic mistakes, such as invoking the correct tool with incorrect arguments, misinterpreting failure messages, or following an execution style that is incompatible with the environment (Yang et al., 2024; Xie et al., 2024; Zhang et al., 2024). Therefore, robust deployment requires the ability to rapidly adapt to the hidden rules and conventions of a new tool-using environment.

During online adaptation to a new environment, due to limited interaction data, sparse feedback, and constrained computational resources, fine-tuning model parameters is often impractical at deployment time. This has motivated alternative approaches that externalize adaptation-relevant knowledge into more reusable forms, such as manually (i) human-written skills (Anthropic, 2025c;a), (ii) direct skill distilled from interaction trajectories (Zheng et al., 2025; Wu et al., 2025b; Ouyang et al., 2025), and (iii) skill evolution in the given environments (Zhang et al., 2026b; 2025a; Lee et al., 2026; Lin et al., 2026). These approaches leverage either human expertise or the model's own capacity for summarization, reflection, and evolving to capture knowledge that can guide subsequent behavior. In this paper, instead of constructing skills tailored to a single target environment, we aim to learn a generalizable ability to explore environments and summarize reusable skills from trajectories collected across diverse, non-standard settings. We define a *meta-skill* as the capability to (i) actively explore an environment to acquire task-completion trajectories; (ii) extract environment-specific operational knowledge from these trajectories, including tool argument constraints, recurring error-repair patterns, implicit tool conventions, and execution-order dependencies; and (iii) apply the extracted knowledge to solve new tasks more effectively within the environment. Such processes enable more effective generalization across tasks within a new environment, thereby providing a foundation for rapid adaptation to previously unseen, non-standard environments.

In this paper, we propose the METASKILL framework, which trains LLM agents to adapt across diverse tool-using environments by learning how to explore a new environment, summarize reusable environment-specific operational knowledge from limited interaction trajectories, and apply the resulting skill to subsequent tasks. We formulate this problem as meta-reinforcement learning (meta-RL) and introduce a variance-reduced stage-wise training algorithm for optimizing the coupled exploration, skill-extraction, and skill-exploitation process. The key motivations are threefold: (i) Failures of LLM agents in tool-using environments often arise not from weak general reasoning, but from missing environment-specific operational knowledge. Existing resources (e.g., tool schemas, environment descriptions) mainly expose explicit interfaces, while many critical rules remain implicit and interaction-dependent. (ii) Interaction trajectories provide much richer adaptation signals than final outcomes alone. They expose structured patterns of success, failure, and recovery, suggesting that such experience can be systematically distilled into reusable skills for future adaptation. (iii) Current models usually lack the capability to reliably acquire and apply such knowledge from limited interaction, motivating the need for training mechanisms that enable agents to explore new environments, extract structured knowledge from trajectories, and reuse it for efficient adaptation.

Figure 1 provides empirical evidence for these motivations. Even without meta-skill training, the base model improves as the adaptation budget increases, showing that support trajectories contain environment-specific information that can be extracted and reused for subsequent tasks. This observation supports (i) previously missing operational knowledge is important for success in unseen environments, and (ii) interaction trajectories provide useful signals for acquiring such knowledge. More importantly, METASKILL consistently outperforms the base model and the RL-based baselines

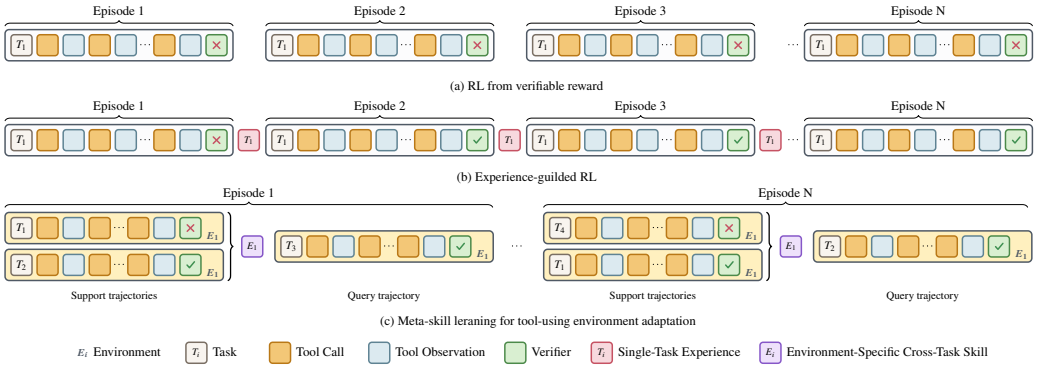


Figure 2: (a) RL from verifiable reward optimizes each episode only from its final task-level outcome. (b) Experience-guided RL augments later episodes with single-task experience distilled from previous trajectories. (c) Our meta-skill learning framework uses support trajectories within an environment to extract an environment-specific cross-task skill, which is then applied to query tasks in the same environment. During the meta-skill training, this support-to-skill-to-query process is repeated across environments, enabling the agent to learn the meta-skill to explore a new environment, summarize reusable environment-specific skills, and apply these skills to improve subsequent adaptation in previously unseen tool-using environments.

under adaptation, indicating that our meta-skill training enables more effective acquisition and application of environment-specific knowledge.

Due to space limitations, we provide the related work discussion in Appendix A.

2 METASKILL: META-SKILL LEARNING FOR TOOL-USING ENVIRONMENT ADAPTATION

In this section, we formalize the problem of meta-skill learning for tool-using environment adaptation and present the corresponding learning framework. We first define the environment adaptation problem and introduce the notion of meta-skill in our setting. We then describe the three-stage adaptation process, the associated exploration, skill-extraction, and skill-conditioned policies, and finally present the meta-skill training.

2.1 PROBLEM SETUP

We consider the problem of rapidly adapting an LLM agent to a previously unseen tool-using environment by learning a meta-skill for acquiring and applying environment-specific operational knowledge. Specifically, when the agent enters a new environment, it first performs a small amount of exploration and collects interaction trajectories containing tool executions, failures, tool feedback, and correction attempts. It then summarizes these trajectories into reusable skills that capture environment-specific operational knowledge, such as tool constraints, error-repair patterns, and workflow conventions, and uses them to guide subsequent decisions in the same environment. In our formulation, the meta-skill is defined as the higher-level capability to explore a new environment, extract reusable skills from the resulting interaction, and apply these skills to support subsequent adaptation. We learn the meta-skill from a distribution of tool-using environments.

Let $\mathcal{E}$ denote a set of tool-using environments, and let $p(\mathcal{E})$ denote a distribution over environments. Each environment $E \sim p(\mathcal{E})$ is characterized by its own tools, operational conventions, file or API structures, and workflow constraints. For each environment E, there is a task distribution denoted by $p(\Gamma^{[E]})$. Each task $\mathcal{T}^{[E]} \sim p(\Gamma^{[E]})$ is specified by a user request $u^{[E]}$ together with a verifier that determines whether the request has been successfully completed. When interacting with a task in environment E, the agent receives tool observations o_t and produces actions a_t via reasoning and tool using, yielding a tool-use interaction sequence $(a_1, o_1, \ldots, a_T, o_T)$. After the entire task is completed, the environment-provided verifier returns a scalar reward r indicating task success, together with task-level feedback available to the agent during environment adaptation. We define the full task trajectory as $\tau^{[E]} = (u^{[E]}, a_1, o_1, \ldots, a_T, o_T, r)$. For adaptation, each support trajectory is retained together with its associated verifier feedback.

The process of interacting with the environment E can also be abstracted as a Markov decision process $\mathcal{M}^{[E]} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, P)$, where $\mathcal{S}$ and $\mathcal{A}$ denote the state space and action space, P is the

state transition, and $\mathcal{R}$ is the reward function. At each time step t, the LLM agent observes a state s_t and selects a tool-use action a_t, and then receives a reward r_t and transitions to the next state s_{t+1} according to the transition function P. In the tool-use setting, the action a_t is the reasoning and tool-use action defined above, and the state s_t consists of $(u^{[E]}, a_1, o_1, \ldots, a_{t-1}, o_{t-1})$, i.e., the user request and all tool observations and actions before time step t. The reward satisfies $r_t = 0$ for all steps where a_t includes a tool-calling action, and $r_t = r$ is produced by the trajectory verifier when the agent stops tool-calling.

Our goal is to train an LLM agent to learn meta-skills over environments sampled from $p(\mathcal{E})$, such that, for a previously unseen environment $E_{new} \sim p(\mathcal{E})$, the agent can use a limited exploration set $\mathcal{D}_{\text{exp}}^{[E_{new}]} = \{\tau_1^{[E_{new}]}, \tau_2^{[E_{new}]}, \ldots, \tau_K^{[E_{new}]}\}$ to extract an environment-specific skill and subsequently solve new tasks $\mathcal{T}_{new}^{[E_{new}]} \sim p(\Gamma^{[E_{new}]})$ in the environment E_{new}.

2.2 META-SKILL LEARNING FRAMEWORK

In this subsection, we introduce the proposed meta-skill learning framework, including the three-stage environment adaptation process and the unified meta-skill learning objective.

Environment adaptation. We formalize environment-specific adaptation as a three-stage process. In the first stage, when the agent enters an environment E, it is allocated a small exploration budget to interact with several tasks sampled from $\mathcal{T}^{[E]}$, yielding a set of exploration trajectories, denoted as $\mathcal{D}_{\text{exp}}^{[E]} = \{\tau_1^{[E]}, \tau_2^{[E]}, \ldots, \tau_K^{[E]}\}$. In the second stage, based on $\mathcal{D}_{\text{exp}}^{[E]}$, the agent extracts an environment-specific skill representation $s^{[E]}$, which captures reusable operational knowledge of the environment, including tool constraints, recurrent error-repair patterns, implicit conventions, and reliable execution orders. In the third stage, the agent leverages the extracted skill $s^{[E]}$ to solve subsequent tasks in the same environment. For new tasks $\mathcal{T}_{\text{new}}^{[E]}$ in E, the adapted agent conditions on both the task context and $s^{[E]}$ in order to guide its decisions.

Consider an LLM agent π_θ parameterized by θ. We introduce three policies corresponding to the three stages above. The first is an environment-exploration policy $\pi_\theta^{\text{explore}}(\cdot)$, which denotes the policy of the LLM under an environment-exploration system prompt and is responsible for actively interacting with a new environment E to collect task-completion trajectories in $\mathcal{D}_{\text{exp}}^{[E]}$. The second is a skill-extraction policy $\pi_\theta^{\text{skill}}(\cdot|\mathcal{D}_{\text{exp}}^{[E]})$, which uses the support interaction trajectories and their associated environment-provided verifier feedback to extract a structured environment-specific skill representation $s^{[E]}$ through the skill-extraction system prompt. Here, $\mathcal{D}_{\text{exp}}^{[E]}$ is understood to include the verifier feedback associated with each support trajectory. The third is a skill-conditioned action policy $\pi_\theta^{\text{exploit}}(\cdot|s^{[E]})$, which denotes the policy of the LLM under a skill-exploitation system prompt conditioned on $s^{[E]}$, and leverages the extracted skill to guide decision-making for downstream tasks in the same environment. These three policies share a single set of LLM parameters, but differ in their system prompts. The corresponding system prompts for these three policies, including the environment exploration, skill extraction, skill exploitation are provided in Appendix C.1.

Meta-skill training. During training, several environments are sampled from $p(\mathcal{E})$. For each sampled environment E, we further sample two sets of tasks from $p(\Gamma^{[E]})$: a support set $\Gamma_{\text{sup}}^{[E]}$ for environment exploration and skill extraction, and a query set $\Gamma_{\text{qry}}^{[E]}$ for skill-conditioned exploitation. We then perform the three-stage adaptation process described above, using the support tasks to construct $\mathcal{D}_{\text{exp}}^{[E]}$ and $s^{[E]}$, and using the query tasks to evaluate downstream performance. The meta-skill training is formulated as $\max_\theta \mathcal{J}_{\text{meta}}(\theta) =$

$$\mathbb{E}_{E\sim p(\mathcal{E})}\Big[\mathbb{E}_{\Gamma_{\text{sup}}^{[E]},\,\Gamma_{\text{qry}}^{[E]}\sim p(\Gamma^{[E]}),\,\mathcal{D}_{\text{exp}}^{[E]},\,s^{[E]}\sim\pi_\theta^{\text{skill}}\left(\cdot|\mathcal{D}_{\text{exp}}^{[E]}\right)}\Big[\sum_{\mathcal{T}_j^{[E]}\in\Gamma_{\text{qry}}^{[E]}}\mathcal{J}^{[E]}\Big(\pi_\theta^{\text{exploit}}(\cdot|s^{[E]}),\,\mathcal{T}_j^{[E]}\Big)\Big]\Big],$$
$$\text{where } \mathcal{D}_{\text{exp}}^{[E]} = \Big\{\tau_i^{[E]}|\tau_i^{[E]} \sim \pi_\theta^{\text{explore}}(\cdot),\ \mathcal{T}_i^{[E]} \in \Gamma_{\text{sup}}^{[E]}\Big\}. \tag{1}$$

In the optimization problem, $\mathcal{D}_{\text{exp}}^{[E]}$ denotes the exploration dataset collected by applying the environment-exploration policy $\pi_\theta^{\text{explore}}$ to the support tasks $\Gamma_{\text{sup}}^{[E]}$ in environment E, and $s^{[E]}$ denotes

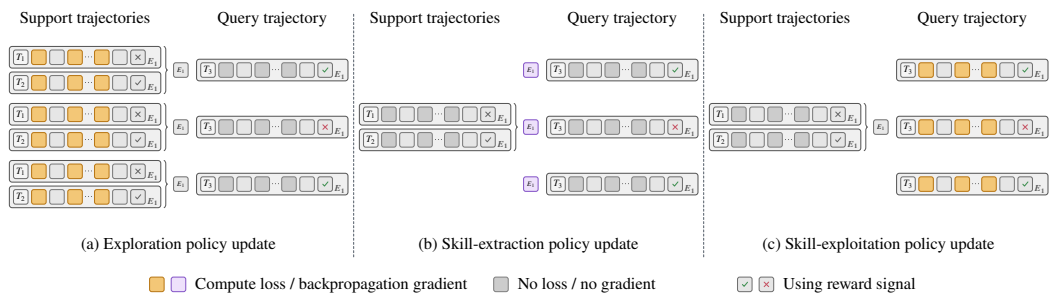


Figure 3: Illustration of the three stage-specific policy updates in our meta-skill training algorithm. Each panel shows one GRPO group consisting of three rollouts. In (a), the exploration policy is updated by sampling a group of support exploration trajectory sets on the same support tasks, while the downstream skill extraction and query-stage performance provide the final reward signal. In (b), the skill-extraction policy is updated by fixing the support exploration trajectories and sampling a group of skill summaries, whose quality is evaluated by downstream query performance. In (c), the skill-exploitation policy is updated by fixing both the support exploration trajectories and the extracted skill, and sampling a group of query trajectories directly on the query task. Colored modules indicate the stage being optimized in each panel, gray modules indicate stages treated as fixed context, and query-task rewards provide the outcome signal.

the environment-specific skill extracted by $\pi_\theta^{\text{skill}}$ from this dataset $\mathcal{D}_{\text{exp}}^{[E]}$. Moreover, $\mathcal{J}^{[E]}(\pi, \mathcal{T}_j^{[E]})$ denotes the expected accumulated reward generated by a policy π on the task $\mathcal{T}_j^{[E]}$ in environment E. The objective therefore encourages the model to jointly learn the environment-exploration policy $\pi_\theta^{\text{explore}}$, the skill-extraction policy $\pi_\theta^{\text{skill}}$, and the skill-conditioned action policy $\pi_\theta^{\text{exploit}}$ under a shared parameterization θ. In this way, information acquired from limited support-task interaction can be transformed into a reusable skill representation that improves performance on query tasks in the same environment, and ultimately generalizes across environments sampled from $p(\mathcal{E})$.

2.3 Meta-skill training algorithm

In this section, we propose a meta-RL algorithm to optimize the meta-skill objective in (1).

Single-turn and multi-turn GRPO. We start from an efficient RL algorithm for LLM fine-tuning, Group Relative Policy Optimization (GRPO), with an outcome reward signal (Shao et al., 2024). Given an input x, let $\{y_i\}_{i=1}^G$ denote a group of sampled responses from the old policy $\pi_{\theta_{\text{old}}}$, and let r_i denote the scalar outcome reward assigned to response y_i. GRPO normalizes these group-wise rewards into relative advantages and optimizes the following clipped policy objective:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x,\{y_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}} \left[\frac{1}{G} \sum_{i=1}^{G} \frac{1}{|y_i|} \sum_{l=1}^{|y_i|} \min \left(\rho_{i,l}(\theta) \hat{A}_i,\ \text{clip}(\rho_{i,l}(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_i \right) \right], \quad (2)$$

$$\text{where} \quad \rho_{i,l}(\theta) = \frac{\pi_\theta(y_{i,l}|x, y_{i,<l})}{\pi_{\theta_{\text{old}}}(y_{i,l}|x, y_{i,<l})}, \qquad \hat{A}_i = \frac{r_i - \text{mean}(\{r_j\}_{j=1}^G)}{\text{std}(\{r_j\}_{j=1}^G) + \varepsilon_{\text{norm}}}.$$

Here, $\rho_{i,l}(\theta)$ is the token-level importance ratio, $\hat{A}_i$ is the group-normalized relative advantage computed from the outcome rewards, and ϵ is the clipping threshold. The constant $\varepsilon_{\text{norm}} > 0$ ensures numerical stability.

In the multi-turn tool-using environments considered in Section 2.1, the agent interacts with the environment over multiple action steps, and the verifier provides a single outcome reward only for the completed trajectory. Therefore, the GRPO objective can be naturally extended to optimizing a trajectory composed of multiple turn-level actions: $\mathcal{L}_{\text{traj}}(\theta) =$

$$-\mathbb{E}_{\{\tau_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}} \left[\frac{1}{G} \sum_{i=1}^{G} \frac{1}{\sum_{t=1}^{T_i} |a_{i,t}|} \sum_{t=1}^{T_i} \sum_{l=1}^{|a_{i,t}|} \min \left(\rho_{i,t,l}(\theta) \hat{A}_i,\ \text{clip}(\rho_{i,t,l}(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_i \right) \right]. \quad (3)$$

Specifically, for the i-th sampled trajectory, let $a_{i,t} = (a_{i,t,1}, \ldots, a_{i,t,|a_{i,t}|})$ denote the token sequence of the action generated at interaction step t under state $s_{i,t}$. The token-level importance ratio at the l-th token of turn t is defined as $\rho_{i,t,l}(\theta) = \frac{\pi_\theta(a_{i,t,l}|s_{i,t}, a_{i,t,<l})}{\pi_{\theta_{\text{old}}}(a_{i,t,l}|s_{i,t}, a_{i,t,<l})}$. Here, T_i is the number of interaction steps in that trajectory, and $a_{i,t}$ is the turn-level action generated at step t. Moreover, i indexes

trajectories within a sampled group, t indexes interaction steps in the tool-using trajectory, and l indexes tokens within the turn-level action $a_{i,t}$. The state $s_{i,t} = (u_i, a_{i,1}, o_{i,1}, \ldots, a_{i,t-1}, o_{i,t-1})$ denotes the full interaction context before generating the current action at step t. The prefix $a_{i,t,<l}$ denotes the partial action prefix generated before token l within the same turn. The group-normalized advantage $\hat{A}_i$ is still computed from the trajectory-level outcome rewards, while the policy-ratio term is evaluated at the token level over all actions in the trajectory.

Variance-reduced meta-skill training. The objective in (1) couples three prompt-conditioned policies, including the environment-exploration policy, the skill-extraction policy, and the skill-exploitation policy. A direct approach would be to sample a large number of complete exploration-skill-exploitation rollouts and optimize the whole pipeline using (3). However, this leads to high-variance policy optimization, because the final query reward is observed after the entire three-stage process, and does not provide clear credit assignment as to whether the improvement or degradation is caused by the exploration stage, the skill-extraction stage, or the exploitation stage.

To reduce variance, we decompose the overall objective into three policy-specific optimization targets, each using final query performance as the outcome reward. Within each GRPO group, we hold the target policy's upstream context and the query task fixed, sample its outputs, and evaluate them through the downstream stages. This removes between-context differences from the group comparison while retaining downstream sampling variability. Figure 3 illustrates how the three stage-specific update groups are constructed. The gradient derivation and variance-reduction analysis are provided in Appendices B.2 and B.3, respectively.

(i) Exploration policy loss. We sample a group of support interaction trajectory sets $\{\mathcal{D}^{[E]}_{\mathrm{exp},g}\}^{G}_{g=1}$ on the same support-task set using $\pi^{\mathrm{explore}}_{\theta_{\mathrm{old}}}$, pass each trajectory set through the downstream skill-extraction and skill-exploitation stages, and use the resulting final query reward as r^{explore}_{g}. We use the trajectory-level loss in (3), as the exploration policy outputs multi-turn tool-using trajectories.

(ii) Skill-extraction policy loss. We first fix one support exploration dataset $\widetilde{\mathcal{D}}^{[E]}_{\mathrm{exp}}$, then sample a group of skills $\{s^{[E]}_{g}\}^{G}_{g=1}$ from $\pi^{\mathrm{skill}}_{\theta_{\mathrm{old}}}(\cdot \mid \widetilde{\mathcal{D}}^{[E]}_{\mathrm{exp}})$, and evaluate each skill by the downstream exploitation stage to obtain rewards $\{r^{\mathrm{skill}}_{g}\}^{G}_{g=1}$. We use the single-turn GRPO loss in (2).

(iii) Skill-exploitation policy loss. We first fix one support exploration trajectory set and one extracted skill, and then sample a group of query rollouts $\{\mathcal{D}^{[E]}_{\mathrm{qry},g}\}^{G}_{g=1}$ from $\pi^{\mathrm{exploit}}_{\theta_{\mathrm{old}}}(\cdot \mid \widehat{s}^{[E]})$, using the final query rewards as $\{r^{\mathrm{exploit}}_{g}\}^{G}_{g=1}$. Since the exploitation policy also produces multi-turn tool-using trajectories, it is optimized with (3).

The overall meta-skill training objective is therefore implemented as the sum of the three stage-specific losses above. The final query reward is used as the outcome signal for all three stages, because the training target of all three policies is the final performance after adaptation rather than the immediate quality of an intermediate output. Specifically, the exploration policy is rewarded when it collects support interaction trajectories that lead to better downstream skills and higher query performance; the skill-extraction policy is rewarded when it summarizes the support interaction into a more useful skill representation for subsequent exploitation; and the skill-exploitation policy is rewarded directly according to query-task success under the extracted skill. This decomposition therefore provides a practical credit-assignment mechanism for optimizing the three-stage adaptation process in (1). The full training procedure is provided in Appendix B as Algorithm 1.

3 EXPERIMENTS

We design our experiments to answer three questions: (RQ1) Does training meta-skills on a tool-use benchmark improve test performance on the same benchmark with the environment-specific adaptation, including both held-out tasks from seen environments and held-out unseen environments? (RQ2) Does the learned meta-skill transfer to other tool-use benchmarks beyond the training benchmark? (RQ3) How do the stage-wise policy updates and variance-reduced training affect adaptation performance? The following three subsections answer RQ1, RQ2, and RQ3, respectively.

Table 1: Test accuracy (%) on the BFCL multi-turn benchmark across seen and unseen environments under Qwen3-4B and Qwen3-8B, with no adaptation and multi-shot adaptation budgets.

	Method	Seen env				Unseen env			
		No adaptation	1-shot adaptation	2-shot adaptation	3-shot adaptation	No adaptation	1-shot adaptation	2-shot adaptation	3-shot adaptation
Qwen3-8B	Base model	35.5	37.1	38.2	38.7	36.1	39.2	40.2	40.3
	RL	41.1	42.4	42.8	41.9	42.8	44.5	44.0	44.2
	Experience-guided RL	41.5	42.9	43.1	43.9	43.1	44.3	44.2	43.6
	MetaSkill (ours)	41.0	45.5	46.8	47.3	42.9	48.9	49.3	50.2
Qwen3-4B	Base model	26.0	26.2	29.3	27.3	26.4	26.8	25.5	26.1
	RL	31.2	31.8	31.9	31.1	32.4	32.7	32.8	33.0
	Experience-guided RL	31.6	31.8	32.2	32.3	32.1	32.9	33.1	32.9
	MetaSkill (ours)	31.8	35.3	35.7	35.9	32.0	36.2	37.1	37.0

3.1 RQ1: Performance improvement with environment-specific adaptation

Benchmark. We conduct the experiment on the multi-turn portion of the Berkeley Function Calling Leaderboard (BFCL), which benchmarks LLMs in executable function-calling settings and broader agentic tool-use scenarios (Patil et al., 2025). This benchmark is well aligned with our setting because success depends not only on invoking the correct tool, but also on maintaining state across turns, responding to tool feedback, and recovering from execution failures. We use the multi-turn subsets, which together cover 20 tool-use environments, including `GorillaFileSystem`, `MathAPI`, `MessageAPI`, `TwitterAPI`, `TicketAPI`, `TradingBot`, `TravelAPI`, and `VehicleControlAPI`.

Experiment setting. To match the environment-adaptation objective, we hold out 3 environments as unseen environments and use the remaining 17 environments for training and seen-environment evaluation. At test time, the support task and the query task are always different tasks, but are drawn from the same environment. For seen-environment evaluation, the support and query tasks are drawn from environments used during training, while the tasks themselves are held out from training. For unseen-environment evaluation, both support and query tasks come from environments that are never used during training.

Baselines. We compare four methods. (i) Base model denotes the underlying LLM without further training. (ii) RL is a direct multi-turn agent trained with Group Relative Policy Optimization (GRPO) under outcome supervision only (Shao et al., 2024). (iii) Experience-guided RL (Jiang et al., 2025; Xiao et al., 2026) denotes RL-based agents that leverage reflections or distilled experience from prior interaction to guide subsequent exploration or decision making within a target task. In our implementation, during training, the agent extracts experience from its initial attempt on the current query and immediately retries the same query using that experience. The initial attempt, experience extraction, and retry together constitute a single rollout, which is used for GRPO training. (iv) MetaSkill is our method, which trains the agent across environments to acquire the capability of exploring a new environment, abstracting the resulting experience into reusable operational guidance, and applying that guidance to new tasks in the same environment. All methods share the same base model and environment interaction interface. At test time, all methods use exactly the same inference pipeline and inference budget for each adaptation setting.

Results. Table 1 summarizes BFCL multi-turn accuracy under different adaptation budgets for Qwen3-8B and Qwen3-4B base models. For Qwen3-8B, we observe three main patterns. First, across all baselines and our method, environment-specific adaptation consistently improves over no adaptation, and performance generally increases as more support trajectories are used. Second, under adaptation, MetaSkill outperforms the base model, standard RL, and experience-guided RL on both seen and unseen environments, showing that the learned meta-skill helps the model use environment-specific support trajectories more effectively than the baselines. Third, without the environment-specific adaptation, MetaSkill remains competitive with the RL-based baselines. Its main advantage, however, appears when support trajectories are available. This distinction is important: the goal of MetaSkill is not only to improve the standalone query policy, but to train an agent that can explore an environment, extract reusable operational knowledge, and apply that knowledge to a different task from the same environment.

For Qwen3-4B, the first pattern is weaker for the base model: simply providing support trajectories does not reliably improve performance, especially on unseen environments. This suggests that smaller models may not automatically possess the meta-skill needed to extract useful operational knowledge

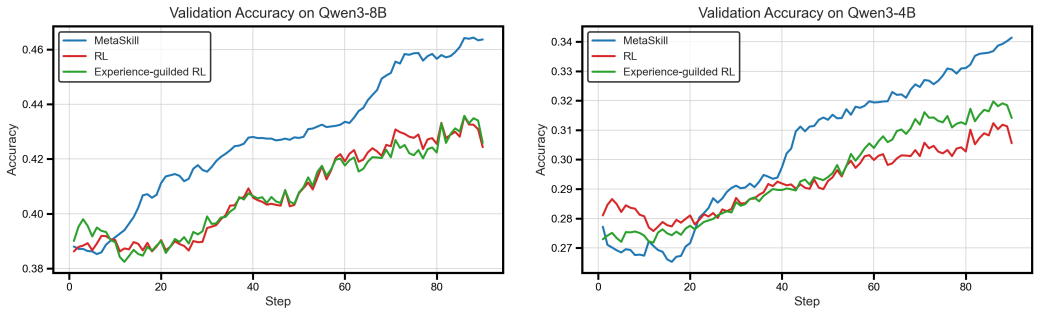


Figure 4: Validation accuracy curves under 1-shot BFCL adaptation on Qwen3-8B and Qwen3-4B for MetaSkill, RL, and experience-guided RL.

from support interactions. After training with MetaSkill, the adaptation becomes consistently beneficial and the method outperforms the baselines under adaptation, which supports the claim that the proposed training procedure teaches the model how to use support trajectories, rather than merely exposing it to additional context at test time.

MetaSkill directly optimizes post-adaptation query performance, whereas standard RL and experience-guided RL optimize task-level performance. The higher validation accuracy under 1-shot adaptation in Figure 4 supports the effectiveness of this training design.

3.2 RQ2: MetaSkill's transferability to other benchmarks

To evaluate whether MetaSkill learns a transferable adaptation capability, we follow the cross-benchmark evaluation suite used by AgentScaler (Fang et al., 2025) and test the BFCL-trained model on three additional tool-use benchmarks: τ-bench (Yao et al., 2024a), τ^2-Bench (Barres et al., 2025), and the agent portion in ACEBench (Chen et al., 2025). These benchmarks are not used during meta-skill training and differ from BFCL in their environment dynamics, interaction protocols, and task distributions.

The main comparison mirrors RQ1. We evaluate the base model, standard RL, experience-guided RL, and MetaSkill under no-adaptation and multi-shot adaptation settings. This setup asks whether the skill-extraction and skill-exploitation behavior learned on BFCL can still identify reusable environment-specific rules when the environment comes from a different benchmark.

Table 2: Cross-benchmark transfer accuracy (%) on τ-bench. Fine-tuned models are trained on BFCL; the base model receives no additional training. All methods are evaluated under no adaptation and multi-shot adaptation budgets.

	Method	Retail			Airline		
		No adaptation	1-shot adaptation	2-shot adaptation	No adaptation	1-shot adaptation	2-shot adaptation
Qwen3-8B	Base model	45.2	46.6	47.6	30.0	32.0	36.0
	RL	46.3	47.5	47.9	32.0	34.0	34.0
	Experience-guided RL	46.4	47.3	47.7	32.0	34.0	36.0
	MetaSkill (ours)	46.5	49.7	50.2	32.0	36.0	38.0
Qwen3-4B	Base model	37.4	38.3	38.5	28.0	32.0	34.0
	RL	38.0	39.1	39.3	30.0	32.0	32.0
	Experience-guided RL	38.1	39.0	39.2	30.0	32.0	34.0
	MetaSkill (ours)	38.3	41.2	41.8	30.0	34.0	36.0

Results. Tables 2–4 show that the meta-skill learned on BFCL transfers to tool-use benchmarks with substantially different interaction formats. Across τ-bench, τ^2-Bench, and ACEBench agent, MetaSkill is most effective when support trajectories are available. This indicates that the learned behavior is not merely a benchmark-specific policy improvement on BFCL; rather, the model learns how to use limited target-environment interaction to infer reusable operational rules and apply them to new tasks in the same environment.

Table 3: Cross-benchmark transfer accuracy (%) on τ^2-Bench. Fine-tuned models are trained on BFCL; the base model receives no additional training. All methods are evaluated under no adaptation and multi-shot adaptation budgets.

	Method	Retail			Airline			Telecom		
		No adapt.	1-shot	2-shot	No adapt.	1-shot	2-shot	No adapt.	1-shot	2-shot
Qwen3-8B	Base model	29.8	31.2	32.6	32.0	32.0	38.0	35.9	36.7	37.9
	RL	29.8	31.4	33.0	36.0	36.0	40.0	36.1	37.1	38.1
	Experience-guided RL	29.8	31.8	33.2	36.0	38.0	40.0	36.1	37.8	38.6
	MetaSkill (ours)	29.8	33.2	34.4	36.0	40.0	42.0	36.2	39.7	40.1
Qwen3-4B	Base model	23.6	24.4	24.7	30.0	32.0	36.0	22.8	23.6	26.3
	RL	23.6	24.6	24.8	34.0	34.0	38.0	23.0	24.0	26.8
	Experience-guided RL	23.6	24.8	25.0	34.0	36.0	36.0	23.1	24.4	27.2
	MetaSkill (ours)	23.6	26.1	26.7	34.0	38.0	40.0	23.2	26.3	28.7

Table 4: Cross-benchmark transfer accuracy (%) on the agent portion of ACEBench. Fine-tuned models are trained on BFCL; the base model receives no additional training. All methods are evaluated under no adaptation and multi-shot adaptation budgets.

	Method	ACEBench agent		
		No adaptation	1-shot adaptation	2-shot adaptation
Qwen3-8B	Base model	29.1	32.0	33.2
	RL	31.2	33.1	34.2
	Experience-guided RL	31.4	33.2	34.1
	MetaSkill (ours)	31.3	35.1	36.8
Qwen3-4B	Base model	23.3	25.2	26.8
	RL	24.9	25.9	27.4
	Experience-guided RL	25.2	26.0	27.4
	MetaSkill (ours)	25.0	28.2	30.1

The transfer gains are clearest under adaptation. In the no-adaptation setting, MetaSkill is generally close to the RL and experience-guided RL baselines, suggesting that BFCL meta-skill training does not substantially change the model into a stronger universal zero-shot policy. This is expected given the limited scale of the training environments and the distribution shift from BFCL to the transfer benchmarks. By contrast, once support trajectories are provided, MetaSkill more consistently improves over the baselines, showing that the learned skill-extraction and skill-exploitation behavior can be reused beyond the original training benchmark.

This pattern also clarifies the boundary of transfer. MetaSkill transfers primarily as an environment-specific adaptation procedure, not as a direct replacement for benchmark-specific training. Support trajectories expose local tool behavior, hidden preconditions, and domain-specific workflows in the target benchmark, and the extracted skill turns these observations into query-time guidance. Overall, RQ2 supports that MetaSkill learns a transferable meta-skill for adaptation, while most of its benefit still comes from interacting with the target environment at test time.

3.3 RQ3: Ablation study

We ablate each of the three stage-wise policy updates and compare stage-wise training with end-to-end GRPO. The full method performs best under adaptation: removing any stage-specific update or replacing stage-wise training with end-to-end GRPO reduces accuracy. Appendix D.1 provides the complete results and analysis.

4 Conclusion

We study how LLM agents can adapt to new tool-using environments from limited support interactions. We introduce MetaSkill, a meta-RL framework that enables a shared model to explore an environment, extract reusable environment-specific skills from support trajectories, and leverage these skills to solve query tasks. Across multiple tool-use benchmarks, MetaSkill consistently improves adaptation performance under limited interaction budgets and demonstrates strong cross-benchmark transfer. These results suggest that MetaSkill enables agents to acquire local tool conventions and latent environment rules through interactions, moving beyond reliance on static tool specifications.

AI USE STATEMENT

Generative AI tools were used for language editing, literature organization, and assistance with analysis and typesetting code. The author manually checked the cited papers, quantitative claims, mathematical content, and compiled artifact, and takes responsibility for the complete paper.

REPRODUCIBILITY STATEMENT

The main text describes the MetaSkill framework, training objectives, and evaluation protocol. Appendix B provides the training pseudocode and derivations. Appendices C.1 and C.2 document the prompts and benchmarks, while Appendix C.3 specifies the implementation details, training hyperparameters, decoding settings, evaluation procedure, and computational resources. Appendix D.3 includes environment-specific skills and full query trajectories to illustrate the adaptation process.

REFERENCES

Anthropic. Common workflows. `https://code.claude.com/docs/en/common-workflows`, 2025a. Accessed: 2026-09-25.

Anthropic. Computer use tool. `https://platform.claude.com/docs/en/agents-and-tools/tool-use/computer-use-tool`, 2025b. Accessed: 2026-09-25.

Anthropic. Create custom subagents. `https://code.claude.com/docs/en/sub-agents`, 2025c. Accessed: 2026-09-25.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Jacob Beck, Risto Vuorio, Evan Zheran Liu, Zheng Xiong, Luisa Zintgraf, Chelsea Finn, and Shimon Whiteson. A tutorial on meta-reinforcement learning. *arXiv preprint arXiv:2301.08028*, 2023.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, Wulong Liu, Xinzhi Wang, Defu Lian, Baoqun Yin, Yasheng Wang, and Wu Liu. Acebench: Who wins the match point in tool usage? *arXiv preprint arXiv:2501.12851*, 2025.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. AutoManual: Constructing instruction manuals by LLM agents via interactive environmental learning. In *Advances in Neural Information Processing Systems*, 2024.

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

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, Shibin Wu, Zhengwei Tao, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. Towards general agentic intelligence via environment scaling. *arXiv preprint arXiv:2509.13311*, 2025.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*, 2023.

Shiting Huang, Zecheng Li, Yu Zeng, Qingnan Ren, Zhen Fang, Qisheng Su, Kou Shi, Lin Chen, Zehui Chen, and Feng Zhao. Internalizing meta-experience into memory for guided reinforcement learning in large language models. *arXiv preprint arXiv:2602.10224*, 2026.

Yulun Jiang, Liangze Jiang, Damien Teney, Michael Moor, and Maria Brbic. Meta-rl induces exploration in language agents. *arXiv preprint arXiv:2512.16848*, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, Xiang Zhang, Suhang Wang, Benoit Dumoulin, and Jian Pei. Position: Agentic evolution is the path to evolving llms. *arXiv preprint arXiv:2602.00359*, 2026.

Model Context Protocol. Model context protocol specification. https://modelcontextprotocol.io/specification/2024-11-05, 2024. Accessed: 2026-04-04.

Dilxat Muhtar, Jiashun Liu, Wei Gao, Weixun Wang, Shaopan Xiong, Ju Huang, Siran Yang, Wenbo Su, Jiamang Wang, Ling Pan, and Bo Zheng. Complementary reinforcement learning. *arXiv preprint arXiv:2603.17621*, 2026.

OpenAI. Introducing chatgpt agent: Bridging research and action. `https://openai.com/index/introducing-chatgpt-agent/`, 2025. Accessed: 2026-09-25.

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

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

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

Yuxiao Qu, Matthew Y. R. Yang, Amrith Setlur, Lewis Tunstall, Edward Emanuel Beeching, Ruslan Salakhutdinov, and Aviral Kumar. Optimizing test-time compute via meta reinforcement finetuning. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 50893–50925, 2025.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025a.

Yaxiong Wu, Sheng Liang, Chen Zhang, Yichao Wang, Yongyue Zhang, Huifeng Guo, Ruiming Tang, and Yong Liu. From human memory to ai memory: A survey on memory mechanisms in the era of llms. *arXiv preprint arXiv:2504.15965*, 2025b.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Teng Xiao, Yige Yuan, Hamish Ivison, Huaisheng Zhu, Faeze Brahman, Nathan Lambert, Pradeep Dasigi, Noah A. Smith, and Hannaneh Hajishirzi. Meta-reinforcement learning with self-reflection for agentic search. *arXiv preprint arXiv:2603.11327*, 2026.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024.

Siyuan Xu, Shiyang Li, Xin Liu, Tianyi Liu, Yixiao Li, Zhan Shi, Zixuan Zhang, Zilong Wang, Qingyu Yin, Jianshu Chen, Tuo Zhao, and Bing Yin. Controllable and verifiable tool-use data synthesis for agentic reinforcement learning. *arXiv preprint arXiv:2604.09813*, 2026.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*, 2023.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Lu Yang, Zelai Xu, Minyang Xie, Jiaxuan Gao, Zhao Shok, Yu Wang, and Yi Wu. Mage: Meta-reinforcement learning for language agents toward strategic exploration and exploitation. *arXiv preprint arXiv:2603.03680*, 2026.

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.

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

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization. In *International Conference on Learning Representations*, 2024b.

Tianzhu Ye, Li Dong, Qingxiu Dong, Xun Wu, Shaohan Huang, and Furu Wei. Online experiential learning for language models. *arXiv preprint arXiv:2603.16856*, 2026.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Ufo: A ui-focused agent for windows os interaction. *arXiv preprint arXiv:2402.07939*, 2024.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*, 2023.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025a.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. Memskill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026b.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025b. doi: 10.1145/3748302.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*, 2025.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Appendix for "Meta-Skill Learning for Tool-Using Environment Adaptation in Large Language Model Agents"

A Related works

Memory & skill & harness evolving for LLM agents. Memory systems have become a central component of LLM-based agentic systems, as they provide the infrastructure for retaining, organizing, and reusing information accumulated during long-horizon interactions with the environment (Wu et al., 2025b; Zhang et al., 2025b). Beyond serving as passive storage, the memory system can support persistent self-improvement by capturing agent-environment interactions and distilling them into reusable knowledge and skills, thereby enabling agents to evolve continuously in task solving and world exploration (Zheng et al., 2025). AutoManual (Chen et al., 2024) constructs environment manuals from interaction, while ExpeL (Zhao et al., 2024) extracts reusable textual insights without parameter updates. However, a limitation of the agentic system is that the underlying memory architecture often remains static: while agents can accumulate experience and update memory contents over time, the mechanisms for managing memory are typically hand-designed and fixed. To address this limitation, recent works move beyond evolving memory contents toward evolving the memory system itself (Zhang et al., 2026b; 2025a; Zhou et al., 2025; Zhang et al., 2026a). Specifically, these approaches seek to optimize how prior experience is encoded, organized, retrieved, and updated, so that the system can refine not only what it remembers, but also how it learns from and utilizes prior experience. This perspective has been further extended to the evolution of the entire agentic system, including memory usage, harness design, and other system-level components (Lee et al., 2026; Lin et al., 2026). Our work is aligned with the broader effort to improve the agentic system itself, rather than accumulating memory from online interactions. On the other hand, instead of enhancing memory or agent system within a fixed environment, we target diverse environments and train LLMs with meta-skills that infer environment-specific operational knowledge from interaction trajectories, enabling rapid adaptation to unseen, non-standard environments.

Experience-guided reinforcement learning (RL) for LLM agents. A related line of work studies how reusable guidance can be distilled from interaction trajectories and incorporated into RL for LLM agents (Jiang et al., 2025; Xiao et al., 2026; Xia et al., 2026; Huang et al., 2026; Ye et al., 2026; Muhtar et al., 2026; Wu et al., 2025a). These methods typically extract reflections, skills, or higher-level experience from prior episodes and reuse them to improve subsequent learning. Retroformer (Yao et al., 2024b) trains a retrospective model to improve retries on the same task, whereas we learn environment-specific skills for transfer from support to distinct query tasks. For example, LAMER (Jiang et al., 2025) distills information from previous episodes to guide future exploration, while MR-Search (Xiao et al., 2026) leverages prior experience to alleviate sparse rewards and provide more informative learning signals in interactive environments. Similarly, SkillRL (Xia et al., 2026), MEL (Huang et al., 2026), OEL (Ye et al., 2026), Complementary RL (Muhtar et al., 2026), and EvolveR (Wu et al., 2025a) extract reusable skills, meta-experience, experiential knowledge, co-evolving experience representations, or distilled principles from successful and failed trajectories, and use them to improve subsequent policy learning. In contrast, rather than primarily using distilled experience to improve learning within a particular target environment, we focus on learning across diverse environments the meta-skill of exploring environments and summarizing environment-specific operational rules, enabling fast adaptation to previously unseen tool-using environments. Figure 2 compares RL, experience-guided RL and the our approach.

Meta-RL for LLM. A recent line of work formulates adaptation in LLMs and LLM agents through the lens of meta-RL. In reasoning-oriented settings, MRT (Qu et al., 2025) interprets test-time computation as a meta-RL problem, where the model must learn how to allocate exploration and exploitation across successive reasoning segments to improve final performance. In the LLM agents, as discussed above, LAMER (Jiang et al., 2025) and MR-Search (Xiao et al., 2026) instantiate this perspective in tool-interaction environments by training agents to condition on prior episodes and adapt their subsequent exploration. More recently, MAGE (Yang et al., 2026) extends this perspective to multi-agent settings. Our work also belongs to this broader meta-RL paradigm, and is even more directly aligned with the classical view of meta-RL, which seeks to train agents over a distribution of environments such that they can rapidly adapt to a new one with limited trajectory exploration (Finn et al., 2017; Beck et al., 2023).

B SUPPLEMENTARY ALGORITHM DETAILS

B.1 META-RL TRAINING FOR META-SKILL LEARNING

This subsection complements Section 2.3. In Section 2.3, we present the high-level design of the variance-reduced meta-skill training algorithm and explain how the three stage-specific GRPO losses are constructed. Here, we provide the full pseudocode for the complete training procedure so that the interaction between the exploration, skill-extraction, and skill-exploitation stages is explicit before we discuss its derivation and properties in the following subsections.

Algorithm 1 Meta-RL training for meta-skill learning

Input: environment distribution $p(\mathcal{E})$, task distributions $p(\Gamma^{[E]})$, initial policy parameters θ
Initialize $\theta_{\text{old}} \leftarrow \theta$
while not converged **do**
  Sample an environment $E \sim p(\mathcal{E})$
  Sample a support-task set $\Gamma^{[E]}_{\text{sup}} \sim p(\Gamma^{[E]})$ and a query task $\mathcal{T}^{[E]}_{\text{qry}} \sim p(\Gamma^{[E]})$
  // Exploration-policy rollout group
  for $g = 1$ to G **do**
    Sample an exploration dataset $\mathcal{D}^{[E]}_{\text{exp},g}$ on $\Gamma^{[E]}_{\text{sup}}$ using $\pi^{\text{explore}}_{\theta_{\text{old}}}(\cdot)$
    Sample a skill $s^{[E]}_g \sim \pi^{\text{skill}}_{\theta_{\text{old}}}(\cdot \mid \mathcal{D}^{[E]}_{\text{exp},g})$
    Evaluate $\pi^{\text{exploit}}_{\theta_{\text{old}}}(\cdot \mid s^{[E]}_g)$ on $\mathcal{T}^{[E]}_{\text{qry}}$ and obtain r^{explore}_g
  end for
  Compute the exploration-policy loss using (3) with rewards $\{r^{\text{explore}}_g\}_{g=1}^{G}$
  // Skill-policy rollout group
  Sample one support exploration dataset $\widetilde{\mathcal{D}}^{[E]}_{\text{exp}}$ on $\Gamma^{[E]}_{\text{sup}}$ using $\pi^{\text{explore}}_{\theta_{\text{old}}}(\cdot)$
  for $g = 1$ to G **do**
    Sample a skill $s^{[E]}_g \sim \pi^{\text{skill}}_{\theta_{\text{old}}}(\cdot \mid \widetilde{\mathcal{D}}^{[E]}_{\text{exp}})$
    Evaluate $\pi^{\text{exploit}}_{\theta_{\text{old}}}(\cdot \mid s^{[E]}_g)$ on $\mathcal{T}^{[E]}_{\text{qry}}$ and obtain r^{skill}_g
  end for
  Compute the skill-extraction loss using (2) with rewards $\{r^{\text{skill}}_g\}_{g=1}^{G}$
  // Exploitation-policy rollout group
  Sample one support exploration dataset $\widehat{\mathcal{D}}^{[E]}_{\text{exp}}$ on $\Gamma^{[E]}_{\text{sup}}$ using $\pi^{\text{explore}}_{\theta_{\text{old}}}(\cdot)$
  Sample one skill $\widehat{s}^{[E]} \sim \pi^{\text{skill}}_{\theta_{\text{old}}}(\cdot \mid \widehat{\mathcal{D}}^{[E]}_{\text{exp}})$
  for $g = 1$ to G **do**
    Sample a query rollout $\tau^{[E]}_{\text{qry},g}$ on $\mathcal{T}^{[E]}_{\text{qry}}$ using $\pi^{\text{exploit}}_{\theta_{\text{old}}}(\cdot \mid \widehat{s}^{[E]})$
    Compute $r^{\text{exploit}}_g = r(\tau^{[E]}_{\text{qry},g})$
  end for
  Compute the skill-exploitation loss using (3) with rewards $\{r^{\text{exploit}}_g\}_{g=1}^{G}$
  Update θ by descending the sum of the three policy losses
  Set $\theta_{\text{old}} \leftarrow \theta$
end while

B.2 DISCUSSION OF META-SKILL TRAINING ALGORITHM

In this section, we explain how the meta-skill training algorithm (Section 2.3 and Algorithm 1) is derived from the meta-skill objective in (1), and why the resulting stage-wise decomposition is equivalent to the policy gradient of the original objective. The variance analysis is presented separately in Appendix B.3.

For notational simplicity, consider a fixed environment E, a sampled support-task set $\Gamma^{[E]}_{\text{sup}}$, and a sampled query task $\mathcal{T}^{[E]}_{\text{qry}}$. The three-stage adaptation process first samples a support interaction dataset $\mathcal{D}^{[E]}_{\text{exp}}$ using the exploration policy, then samples an environment-specific skill $s^{[E]}$ using the skill-extraction policy, and finally samples a query trajectory $\tau^{[E]}_{\text{qry}}$ using the skill-conditioned exploitation policy. Let $r(\tau^{[E]}_{\text{qry}})$ denote the final verifier reward on the query task. Under this notation,

the joint distribution induced by the full adaptation process factorizes as

$$\begin{aligned} & p_\theta\left(\mathcal{D}_{\mathrm{exp}}^{[E]}, s^{[E]}, \tau_{\mathrm{qry}}^{[E]}\right) \\ =& p_\theta^{\mathrm{explore}}\left(\mathcal{D}_{\mathrm{exp}}^{[E]} \mid \Gamma_{\mathrm{sup}}^{[E]}\right) p_\theta^{\mathrm{skill}}\left(s^{[E]} \mid \mathcal{D}_{\mathrm{exp}}^{[E]}\right) p_\theta^{\mathrm{exploit}}\left(\tau_{\mathrm{qry}}^{[E]} \mid s^{[E]}, \mathcal{T}_{\mathrm{qry}}^{[E]}\right). \end{aligned} \tag{4}$$

Therefore, the objective in (1) can be viewed as the expected query-stage reward under this three-stage joint policy. Applying the score-function identity gives

$$\nabla_\theta \mathcal{J}_{\mathrm{meta}}(\theta) = \mathbb{E}\left[r(\tau_{\mathrm{qry}}^{[E]}) \nabla_\theta \log p_\theta\left(\mathcal{D}_{\mathrm{exp}}^{[E]}, s^{[E]}, \tau_{\mathrm{qry}}^{[E]}\right)\right]. \tag{5}$$

Substituting the above factorization into the log term yields an additive decomposition:

$$\nabla_\theta \mathcal{J}_{\mathrm{meta}}(\theta) = g_{\mathrm{explore}}(\theta) + g_{\mathrm{skill}}(\theta) + g_{\mathrm{exploit}}(\theta), \tag{6}$$

where

$$\begin{aligned} g_{\mathrm{explore}}(\theta) &= \mathbb{E}\left[r(\tau_{\mathrm{qry}}^{[E]}) \nabla_\theta \log p_\theta^{\mathrm{explore}}\left(\mathcal{D}_{\mathrm{exp}}^{[E]} \mid \Gamma_{\mathrm{sup}}^{[E]}\right)\right], \\ g_{\mathrm{skill}}(\theta) &= \mathbb{E}\left[r(\tau_{\mathrm{qry}}^{[E]}) \nabla_\theta \log p_\theta^{\mathrm{skill}}\left(s^{[E]} \mid \mathcal{D}_{\mathrm{exp}}^{[E]}\right)\right], \\ g_{\mathrm{exploit}}(\theta) &= \mathbb{E}\left[r(\tau_{\mathrm{qry}}^{[E]}) \nabla_\theta \log p_\theta^{\mathrm{exploit}}\left(\tau_{\mathrm{qry}}^{[E]} \mid s^{[E]}, \mathcal{T}_{\mathrm{qry}}^{[E]}\right)\right]. \end{aligned} \tag{7}$$

This decomposition is exact: summing the three terms recovers the policy gradient of the original meta-skill objective, rather than defining a surrogate objective unrelated to (1). Intuitively, the first term attributes final post-adaptation performance to the sampled support exploration trajectories, the second attributes it to the extracted skill, and the third attributes it to the final query-stage behavior.

Algorithm 1 is obtained by constructing one GRPO estimator for each of the three gradient terms above. For the exploration policy, we fix the sampled support-task set $\Gamma_{\mathrm{sup}}^{[E]}$, generate a group of support interaction datasets, and evaluate each of them by running the downstream skill-extraction and exploitation stages to obtain final query rewards. Since the exploration policy outputs multi-turn tool-using trajectories, this stage is optimized using (3). For the skill-extraction policy, we fix one sampled support interaction dataset $\widetilde{\mathcal{D}}_{\mathrm{exp}}^{[E]}$, draw a group of skill outputs, and evaluate each skill through the downstream exploitation stage; since the skill is generated as a single sequence, this stage is optimized using (2). For the exploitation policy, we fix one sampled support interaction dataset and one extracted skill, then sample a group of query trajectories and optimize them with (3). This is exactly the procedure summarized in Algorithm 1.

B.3 Variance analysis of stage-wise conditioning

Ideal score-function estimators. The gradient decomposition above motivates one estimator for each stage. For stage k, let C denote its upstream context, Y its generated output, and $Z_k = \nabla_\theta \log \pi_\theta^k(Y \mid C)$ its score. At fixed policy parameters, an on-policy sample RZ_k has expectation $g_k(\theta)$, where R is the final query reward and downstream rollouts are sampled from the joint policy. Thus, the sum of the three raw-reward score-function estimators is unbiased for the decomposed gradient. Algorithm 1 implements stage-specific updates using the group-centered, standardized rewards and clipped token-level objectives in (2) and (3). We next analyze the conditioning mechanism underlying these groups.

Stage-wise conditioning. Throughout the following analysis, fix the policy parameters, environment, support tasks, and query task. For skill extraction, C is the sampled support interaction dataset. For exploitation, C consists of the support dataset and extracted skill. The exploration group fixes the support and query tasks while sampling both support trajectories and downstream continuations. Thus, skill extraction and exploitation introduce additional shared upstream conditions relative to complete-pipeline sampling. Downstream randomness remains part of the reward distribution for any stage that has downstream continuations.

Let $p(C)$ denote the upstream-context distribution induced by the fixed policy, and let $p(R \mid C)$ include randomness from both the target stage and all downstream stages. Define

$$\mu(C) = \mathbb{E}[R \mid C], \qquad \sigma^2(C) = \mathrm{Var}(R \mid C).$$

We compare two grouping schemes with the same group size $G \geq 2$ and the same marginal law $p(C)p(R \mid C)$. In the mixed-context scheme, each member independently samples $C_i \sim p(C)$ and $R_i \sim p(R \mid C_i)$. In the shared-context scheme, a single $C \sim p(C)$ is sampled for the group, followed by conditionally independent $R_1, \ldots, R_G \sim p(R \mid C)$. Both schemes evaluate G final rewards; the comparison concerns their grouping, rather than an assumption of equal total pipeline cost.

Proposition 1 (Reduction in group-centered reward variation)**.** *Suppose $\mathbb{E}[R^2] < \infty$. Write $\bar{R} = G^{-1}\sum_{i=1}^{G} R_i$ and $V_G = G^{-1}\sum_{i=1}^{G}(R_i - \bar{R})^2$. Under the two grouping schemes above,*

$$\mathbb{E}_{\text{mixed}}[V_G] = \frac{G-1}{G}\,\mathrm{Var}(R), \tag{8}$$

$$\mathbb{E}_{\text{shared}}[V_G] = \frac{G-1}{G}\mathbb{E}_C[\sigma^2(C)], \tag{9}$$

and hence

$$\mathbb{E}_{\text{mixed}}[V_G] - \mathbb{E}_{\text{shared}}[V_G] = \frac{G-1}{G}\,\mathrm{Var}_C(\mu(C)) \geq 0. \tag{10}$$

The reduction is strict whenever the conditional mean reward varies across upstream contexts.

Proof. For independent identically distributed rewards with mean μ and variance σ^2,

$$V_G = \frac{1}{G}\sum_{i=1}^{G} R_i^2 - \bar{R}^2, \qquad \mathbb{E}[\bar{R}^2] = \mu^2 + \frac{\sigma^2}{G}.$$

Therefore, $\mathbb{E}[V_G] = (G-1)\sigma^2/G$. Applying this identity to the marginal reward distribution proves the mixed-context expression. Applying it conditional on the shared context gives $\mathbb{E}[V_G \mid C] = (G-1)\sigma^2(C)/G$; averaging over C proves the shared-context expression. Finally, the law of total variance gives

$$\mathrm{Var}(R) = \mathbb{E}_C[\sigma^2(C)] + \mathrm{Var}_C(\mu(C)),$$

which yields (10). □

Proposition 1 identifies the removed component explicitly: variation in the expected query reward caused by changing the upstream context. The target policy and downstream evaluators can remain stochastic; their variability is retained in $\sigma^2(C)$. For example, a skill-extraction group compares skills generated from the same support evidence, instead of mixing changes in support quality with changes in skill quality. An exploitation group similarly compares query actions under the same extracted guidance. For exploration, support trajectories themselves are the target outputs, so their variation is retained as part of the learning signal.

Connection to gradient variance through conditional baselines. The same conditioning structure permits a context-specific policy-gradient baseline. Consider one stage, write $Z = Z_k$, and let

$$\widehat{g}_b = (R - b(C))Z,$$

where $b(C)$ is independent of the sampled target output conditional on C and is treated as a constant when taking the policy gradient. The score identity $\mathbb{E}[Z \mid C] = 0$ implies $\mathbb{E}[\widehat{g}_b] = \mathbb{E}[RZ]$. Thus, choosing a conditional baseline preserves the expected stage gradient.

Proposition 2 (Variance benefit of an optimal conditional baseline)**.** *Assume the relevant second moments are finite. Define $w(C) = \mathbb{E}[\|Z\|^2 \mid C]$ and, wherever $w(C) > 0$,*

$$b^*(C) = \frac{\mathbb{E}[R\|Z\|^2 \mid C]}{w(C)}.$$

Set $b^*(C) = 0$ when $w(C) = 0$. For any admissible constant baseline b_0,

$$\mathrm{tr}\,\mathrm{Cov}(\widehat{g}_{b_0}) - \mathrm{tr}\,\mathrm{Cov}(\widehat{g}_{b^*}) = \mathbb{E}_C\big[w(C)(b_0 - b^*(C))^2\big] \geq 0. \tag{11}$$

Proof. All admissible baselines preserve the same expected gradient. Their covariance traces therefore differ only through the second moment $\mathbb{E}[(R - b(C))^2\|Z\|^2]$. Conditional on C, this is a quadratic in $b(C)$ whose minimizer is $b^*(C)$. Completing the square gives

$$\mathbb{E}\big[((R-b_0)^2 - (R-b^*(C))^2)\|Z\|^2 \mid C\big] = w(C)(b_0 - b^*(C))^2.$$

Averaging over contexts proves (11). □

If R and $\|Z\|^2$ are conditionally uncorrelated given C, then $b^*(C) = \mu(C)$. Under this condition, the conditional mean reward is the variance-minimizing baseline, and the shared-context group mean estimates it directly. More generally, Proposition 2 characterizes the benefit of context-dependent baselines through the score-weighted conditional mean.

Connection to the implemented GRPO groups. Within a shared-context group, $\bar{R}$ is an empirical estimate of $\mu(C)$, and the centered quantities $R_i - \bar{R}$ are the numerators of the GRPO advantages. Proposition 1 therefore characterizes reward variation at the centering step. Because the group mean includes each sample's own reward, the corresponding unstandardized score-function estimator satisfies

$$\mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}(R_i - \bar{R})Z_i \,\middle|\, C\right] = \frac{G-1}{G}\mathbb{E}[RZ \mid C].$$

Indeed, the cross-sample terms obey $\mathbb{E}[R_j Z_i \mid C] = 0$ for $j \neq i$, while the self term contributes $G^{-1}\mathbb{E}[RZ \mid C]$. Multiplying by $G/(G-1)$, or using a leave-one-out mean, recovers the conditional stage gradient for this unstandardized estimator. This calculation describes the centering operation; Algorithm 1 uses the GRPO objective as specified above. Its reward standardization calibrates the scale of the within-context comparison, while clipping controls the policy-ratio update. Together, the group-centering result and conditional-baseline analysis explain how stage-wise conditioning removes upstream variation from reward comparisons and supports more targeted gradient estimation.

C Experimental details

C.1 Prompts

We provide the prompt corresponding to the three policies: the environment-exploration policy, the skill-extraction policy, and the skill-exploitation policy, which is mentioned in Section 2.2.

Environment-exploration prompt This is an early attempt in a new tool-calling environment. Try to solve the task, but also use careful exploration to learn tool behavior, constraints, and failure modes.

Skill-extraction prompt First think and analyze the trajectory step-by-step and identify (i) the exact final failure point(s), (ii) the root cause rather than only the symptoms, (iii) the hidden preconditions that were violated, and (iv) the incorrect assumptions and problematic patterns made by the agent. Then propose concise guidance to avoid similar mistakes in future tool-calling tasks.

The guidance can include two parts: (1) guidance to avoid the identified mistakes, such as action-guiding rules, environment-specific constraints, and hidden preconditions; and (2) high-level and generalized strategies that apply across tasks and tools.

Support trajectory:
{support_trajectory}

Support checker summary:
{support_checker_summary}

Support outcome: {support_outcome}

Now write the guidance.

Skill-exploitation prompt You are given a tool-using guidance extracted from a previous task in the tool-calling environment. Use it as heuristic guidance when it is relevant.

Guidance:
{experience_summary.strip()}

C.2 Details of benchmarks

BFCL multi-turn (Patil et al., 2025): The Berkeley Function Calling Leaderboard (BFCL) is an executable function-calling benchmark for evaluating tool-use ability in LLM agents. We use the multi-turn portion of BFCL, where an agent must maintain state across turns, call the correct functions, interpret tool feedback, and recover from failed or incomplete tool calls. The multi-turn split covers 20 tool-use environments. To evaluate environment-specific adaptation, we hold out 3 environments as unseen environments and use the remaining 17 environments for training and seen-environment evaluation. For every evaluation instance, the support task and query task are different tasks sampled from the same environment.

τ-bench (Yao et al., 2024a): τ-bench evaluates tool-agent-user interaction in realistic service domains. It simulates multi-turn conversations between a user and a language agent equipped with domain-specific API tools and policy guidelines. We use τ-bench as a cross-benchmark transfer evaluation to test whether a model trained on BFCL can adapt to a different tool-use setting with user interaction and policy-constrained tool execution.

τ^2-Bench (Barres et al., 2025): τ^2-Bench extends the τ-bench setting to a dual-control environment, where both the agent and the simulated user can act through tools in a shared dynamic environment. The benchmark is designed to evaluate not only tool-use reasoning, but also coordination and communication under partially observed environment states. We use τ^2-Bench only for out-of-benchmark evaluation; no τ^2-Bench tasks are used during meta-skill training.

ACEBench (Chen et al., 2025): ACEBench is a tool-use benchmark covering normal tool-use scenarios, special cases with ambiguous or incomplete instructions, and agent-style multi-turn interactions. We use the agent portion of ACEBench, which evaluates tool use through multi-agent interactions that simulate real-world multi-turn dialogues. This benchmark provides an additional transfer setting for testing whether the learned meta-skill generalizes beyond BFCL-style function-calling environments.

C.3 Details of implementation

Our implementation is built upon VeRL. The prompts used by the environment-exploration policy, the skill-extraction policy, and the skill-exploitation policy are given in the colored boxes in Section C.1. We implement BFCL multi-turn interaction as an executable tool-use environment: each generated tool call is parsed with the Qwen function-calling format and executed by the BFCL simulator, which returns tool observations during the interaction. The task-specific verifier evaluates the completed task. After a support task ends, its outcome and verifier feedback are supplied to the skill-extraction prompt as `support_outcome` and `support_checker_summary`, together with the support trajectory. The model uses this environment-provided feedback to summarize reusable skills for subsequent tasks in the same environment. The training data are first converted into support-query pairs. For each pair, the support and query tasks come from the same environment-specific experience group, while the query task is reserved for downstream execution and reward computation.

Hyperparameters. For RL training, we use AdamW as the optimizer and set the learning rate to $1e{-}6$. The global training batch size and validation batch size are both set to 256. We use GRPO with a rollout group size of 8, a PPO mini-batch size of 32, and a per-GPU PPO micro-batch size of 1. The KL loss coefficient is set to 0.05 with the low-variance KL estimator, and the entropy coefficient is set to 0. The maximum prompt length is 18,432 tokens, the maximum response length is 2,048 tokens, and the maximum model length is 22,528 tokens. We enable asynchronous rollout, remove-padding optimization, gradient checkpointing, and Ulysses sequence parallelism with size 2. The rollout temperature is set to 1.0 during training and 0.0 during validation; top-p is set to 1.0. We validate every 2 training steps and save checkpoints every 20 training steps. We train RL, Experience-guided RL, and MetaSkill until performance plateaus, using training horizons that cover this regime for each method. Unless otherwise specified, Qwen3-8B and Qwen3-4B use the same hyperparameters.

Method-specific settings. MetaSkill uses the same model as the exploration policy, the skill-extraction policy, and the skill-exploitation policy. During training, support exploration and query exploitation are both sampled with temperature 1.0. During validation and test-time adaptation, support exploration, skill extraction, and query exploitation use greedy decoding with temperature 0.0. The support rollout context length is capped at 16,384 tokens, and the maximum number of

assistant turns in a tool-use episode is set to 50. The skill-extraction prompt budget is computed from the model context window with a 1,024-token safety margin. If the support trajectory and checker summary exceed this budget, state-information blocks are removed first, and the remaining trajectory is truncated only when necessary. We keep query memoization enabled, so the extracted skill is inserted into the query system prompt as heuristic guidance. For multi-shot adaptation, we run skill extraction on each support trajectory and concatenate the resulting skills as separate memos before query exploitation.

The stage-wise training is implemented with the meta-completion callback. In the skill-extraction update, the callback first runs a support rollout, builds the skill-extraction prompt from the support trajectory and checker summary, samples a group of skill outputs, evaluates each skill by running the query task, and assigns the final query success rate as reward. In the skill-exploitation update, the callback reuses the generated skill to construct the query conversation and optimizes the assistant tokens in the query trajectory. In the exploration update, the callback keeps the downstream skill-extraction and exploitation procedure fixed for credit assignment, but applies the reward to the support trajectory tokens. The reward is the executable query success rate; invalid skill summaries containing explicit tool-call actions receive a reward of zero.

Evaluation. For BFCL, we evaluate both seen and unseen environments under no-adaptation and multi-shot adaptation settings. In the adaptation setting, the support task and query task are different tasks sampled from the same environment. For cross-benchmark evaluation on τ-bench, τ^2-Bench, and ACEBench, we evaluate the base model and BFCL-trained checkpoints without additional training on the target benchmarks. All methods use exactly the same inference pipeline and inference budget for each adaptation setting. All reported results use the executable task checker as the success criterion. Each reported result is the mean accuracy over three evaluation runs.

Computation resources. All RL training runs are performed on a single node with 8 NVIDIA H200 GPUs. Rollout serving and policy optimization are managed by VeRL with vLLM-backed asynchronous generation.

D Supplementary experimental results

D.1 Ablation study

We conduct two complementary ablation studies to examine the contribution of the stage-wise policy updates and the variance-reduced training strategy.

Stage-wise policy updates. We first compare MetaSkill with variants that remove one of the three stage-wise policy updates while keeping the other two stages trained. Removing the exploration-policy update tests whether support-task exploration must be directly optimized for downstream adaptation. Removing the skill-extraction-policy update tests whether the model needs an explicit objective for summarizing support trajectories into reusable environment-specific guidance. Removing the skill-exploitation-policy update tests whether query-stage behavior must be optimized under the extracted skill, rather than only improved indirectly through upstream stages.

Variance-reduced training. Second, we compare MetaSkill with a variant that removes the variance-reduction mechanism. Specifically, we replace the stage-wise update groups with a single end-to-end GRPO update over complete support-skill-query rollouts. This comparison keeps the same final query reward but removes the proposed credit-assignment decomposition, isolating whether variance-reduced stage-wise optimization improves adaptation performance.

Results. Tables 5 and 6 show that all three stage-wise policy updates contribute to adaptation performance. Removing the skill-exploitation update causes the largest degradation, indicating that query-time behavior must be directly trained to use the extracted skill. The exploration and skill-extraction updates also matter, mainly by making support trajectories more useful for later query solving. The end-to-end GRPO variant is consistently worse than MetaSkill under adaptation, and Figure 5 shows the same trend during validation. Overall, the results suggest that the three adaptation stages are complementary, and that separating their updates provides a more effective training signal than optimizing complete support-skill-query rollouts as a single undifferentiated sequence.

Table 5: Test accuracy (%) on seen BFCL multi-turn environments under Qwen3-8B.

	Method	Seen env			
		No adaptation	1-shot adaptation	2-shot adaptation	3-shot adaptation
Qwen3-8B	MetaSkill	41.0	45.5	46.8	47.3
	MetaSkill w/o exploration update	40.8	43.1	43.5	44.1
	MetaSkill w/o skill-extraction update	40.9	43.0	42.9	43.8
	MetaSkill w/o skill-exploitation update	36.5	37.9	38.2	38.3
	End-to-end GRPO	40.1	43.2	44.2	44.9

Table 6: Test accuracy (%) on unseen BFCL multi-turn environments under Qwen3-8B.

	Method	Unseen env			
		No adaptation	1-shot adaptation	2-shot adaptation	3-shot adaptation
Qwen3-8B	MetaSkill	42.9	48.9	49.3	50.2
	MetaSkill w/o exploration update	42.7	47.5	47.0	47.5
	MetaSkill w/o skill-extraction update	42.8	47.4	46.4	47.2
	MetaSkill w/o skill-exploitation update	38.4	42.3	41.7	41.7
	End-to-end GRPO	42.0	45.6	47.0	47.3

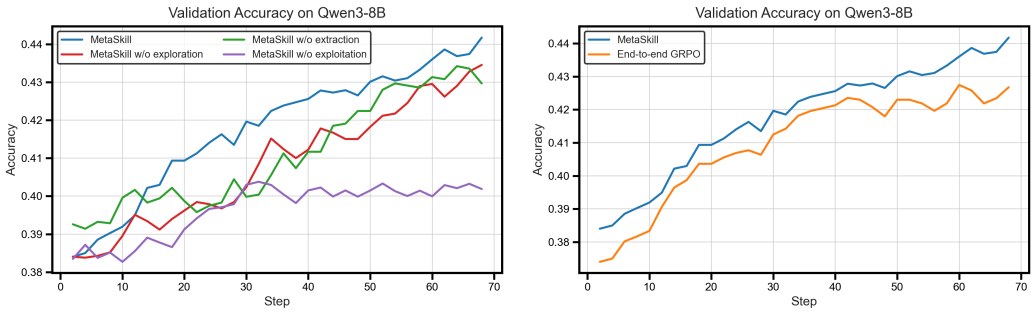


Figure 5: Ablation results on Qwen3-8B. *Left:* stage-wise policy update ablations, comparing the full method with variants that remove one of the exploration, skill-extraction, or skill-exploitation updates. *Right:* variance-reduction ablation, comparing the full method with an end-to-end GRPO update over complete support-skill-query rollouts.

D.2 Effect of skill content and environment matching

We examine how the trained MetaSkill model responds to changes in its adaptation context. Using the same checkpoint, we compare the original environment-specific skill with raw support trajectories supplied directly as query context and a skill from a different environment. These input ablations examine the role of compact skill representations and environment-relevant guidance in the trained system.

Experimental setup. Table 7 is organized around the Qwen3-8B BFCL seen- and unseen-environment evaluations with one, two, or three support trajectories. For the adaptation conditions, the comparison is designed to hold the model checkpoint, target support/query tasks, decoding settings, and query-generation budget fixed while varying the context supplied to query execution. We consider three context conditions: *(i) Environment-specific skill*, the original MetaSkill guidance extracted from support interactions in the target environment; *(ii) Raw support trajectories*, where the original support interaction trajectories are directly included in the query context without summarization or skill extraction; and *(iii) Mismatched skill*, guidance randomly selected from other environments and supplied to the target query in place of the matched skill. Raw-trajectory inputs that exceed the context window are counted as task failures. For the mismatched condition, the number of source support trajectories and the guidance token budget are matched to the corresponding adaptation setting. We also include the same MetaSkill checkpoint without adaptation as a reference: it solves query tasks without support trajectories or extracted guidance. Results are reported as mean accuracy over three evaluation runs.

Table 7: Effect of skill content and environment matching on BFCL multi-turn accuracy (%) with Qwen3-8B. All conditions use the same model checkpoint. No-adaptation results are identical across context conditions because no support trajectories or guidance are used.

Query context	Seen environments				Unseen environments			
	No adapt.	1-shot	2-shot	3-shot	No adapt.	1-shot	2-shot	3-shot
Environment-specific skill	41.0	45.5	46.8	47.3	42.9	48.9	49.3	50.2
Raw support trajectories	41.0	33.2	10.8	5.2	42.9	28.8	9.3	3.8
Mismatched skill	41.0	38.3	37.1	33.6	42.9	36.4	37.0	33.8

Results and analysis. The original environment-specific skill improves accuracy over the no-adaptation reference on both seen and unseen environments at all three adaptation budgets. In contrast, directly supplying raw support trajectories reduces accuracy from 33.2% to 5.2% on seen environments and from 28.8% to 3.8% on unseen environments as the support budget increases from one to three trajectories. Each raw trajectory includes the full multi-turn interaction history, so adding support trajectories rapidly expands the query context. With a finite context window, oversized inputs count as failures, while long inputs within the limit increase the context-processing burden. The sharp degradation is consistent with these context constraints and highlights the practical role of compact skill representations in the trained MetaSkill pipeline.

Mismatched skills also perform below the no-adaptation reference at every tested support budget: accuracy ranges from 33.6% to 38.3% on seen environments and from 33.8% to 37.0% on unseen environments, compared with no-adaptation accuracies of 41.0% and 42.9%, respectively. Guidance from another environment can therefore interfere with query execution rather than provide useful adaptation. For this checkpoint trained with the skill-based pipeline, the two controls highlight complementary factors: representing support experience compactly and matching the guidance to the target environment. These input ablations complement the training-stage ablations in Appendix D.1.

D.3 Case study

We present four representative validation cases where summarized skills play a central role in later success. Each table first shows the summarized skill extracted from support interactions, followed by the complete query trajectory that uses this skill, including the model's reasoning text, natural-language replies, tool calls, returned observations, final answer, and success label.

Case 1: Vehicle preconditions and recovery. The support trajectory reveals hidden safety preconditions that are not captured by the surface user request. Starting the engine depends not only on choosing the correct ignition mode, but also on satisfying environment-specific states such as locked doors and a pressed brake pedal. The failed start attempts expose these missing preconditions, and the recovery sequence shows how the agent can convert error feedback into an ordered skill: inspect or infer the violated condition, lock the required doors, press the brake, and then retry the engine-start action.

Case 2: Distance and driving-feasibility adaptation. The support trajectory exposes tool-interface conventions and dependency patterns for location and vehicle-capacity reasoning. The key environment-specific knowledge is that distance estimation requires zip codes rather than city names, so the agent must first use a city-to-zipcode tool before calling the distance estimator. The trajectory also shows that some seemingly natural unit conversions are unsupported, making failed conversion attempts useful evidence for selecting a compatible recovery path before checking whether the vehicle can complete the trip.

Case 3: Stock-order validation. The support trajectory reveals a financial-tool workflow in which execution depends on explicit validation rather than direct use of the user's natural-language request. The extracted skill captures that company names should first be mapped to ticker symbols, current prices should be checked before order placement, required order fields such as the share amount must be resolved, and later actions should stay grounded in the concrete order identifier. The failure caused by the missing amount is therefore not just an error; it teaches the agent which transaction fields must be validated before calling the order tool.

Case 4: File discovery and workflow correction. The support trajectory exposes filesystem-specific operational rules. The agent must first discover the actual file paths, normalize returned paths into the form expected by downstream tools, and verify directory existence before navigation, moves, or comparisons. Failed commands provide structured recovery signals: a missing file or invalid path indicates whether the agent should strip a path prefix, create or locate a directory, or retry the operation with the discovered filename. This case shows how trajectories turn low-level command failures into reusable workflow guidance.

Table 8: Case study 1: summarized skill and full query trajectory for vehicle preconditions and recovery.

```
Summarized skill:
**Guidance to Avoid Mistakes in Future Tool-Calling Tasks**

1. **Action-Guiding Rules**
   - **Precondition Check**: Always verify critical preconditions (e.g., brake pedal status, locked doors) before initiating
       actions like starting the engine.
   - **Tool Parameter Accuracy**: Ensure parameters (e.g., ‘pedalPosition‘, ‘ignitionMode‘) match tool requirements (e.g., ‘
      pedalPosition‘ must be a float between 0 and 1).
   - **Sequential Tool Use**: Follow the correct order of tools (e.g., press brake pedal -> start engine -> check tire
      pressure).

2. **High-Level Strategy**
   - **Validate Before Acting**: Confirm all system states (e.g., brake, doors, engine) before executing critical operations.

   - **Error-Driven Learning**: Use error responses to identify hidden preconditions (e.g., brake pedal press required for
      engine start).
   - **Document Tool Dependencies**: Track interdependencies between tools (e.g., parking brake + brake pedal press for
      engine start).

**Key Takeaway**: Always align tool calls with the system's explicit requirements and safety constraints to avoid errors.

Query trajectory:
<think>
Okay, the user wants to fill 150 liters of gasoline into their vehicle and then start the engine. They also need to convert
      liters to gallons, rounding to the nearest integer. Let me break this down.
First, the conversion from liters to gallons. The tool ‘liter_to_gallon‘ is available. The parameter is "liter", so I'll
      input 150. The response will give the gallon value. Let me calculate that. Since 1 liter is approximately 0.264172
      gallons, 150 liters would be 150 * 0.264172 approximately 39.6258 gallons. Rounded to the nearest integer, that's 40
      gallons. So the user should fill 40 gallons.
Next, the user wants to fill the fuel tank. The tool ‘fillFuelTank‘ requires "fuelAmount" in gallons. The converted value is
       40, so I'll use that. Then, they need to start the engine using the standard start mode. The ‘startEngine‘ tool has
      an "ignitionMode" parameter, which should be "START" as per the enum.
Wait, but the user mentioned "standard start mode," which likely corresponds to "START" in the ignitionMode enum. So after
      filling the fuel, call ‘startEngine‘ with ignitionMode "START".
I need to make sure the tools are called in the correct order: first convert liters to gallons, then fill the tank with the
      converted amount, and finally start the engine. Let me check if there are any dependencies or preconditions. For
      example, does the engine need to be off before starting? The user didn't mention that, but the tool might handle it.
      Also, ensure the fuelAmount is within the tank's capacity (max 50 gallons). 40 is under 50, so it's safe.
```

```
So the steps are:
1. Use ‘liter_to_gallon‘ with 150 liters to get 40 gallons.
2. Use ‘fillFuelTank‘ with 40 gallons.
3. Use ‘startEngine‘ with ignitionMode "START".
This should fulfill the user’s request.
</think>

<tool_call>
{"name": "liter_to_gallon", "arguments": {"liter": 150}}
</tool_call>
<tool_call>
{"name": "fillFuelTank", "arguments": {"fuelAmount": 40}}
</tool_call>
<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>
user
<tool_response>
{"gallon": 39.625800000000005}
</tool_response>
user
<tool_response>
{"fuelLevel": 45.0}
</tool_response>
user
<tool_response>
{"error": "All doors must be locked before starting the engine. Here are the unlocked doors: driver, passenger, rear_left,
      rear_right"}
</tool_response>
assistant
<think>
Okay, the user tried to start the engine but got an error saying all doors must be locked. Let me check the tools available.
       There’s a ‘lockDoors‘ tool that can lock or unlock doors. The error mentions four unlocked doors: driver, passenger,
      rear_left, rear_right.
The ‘lockDoors‘ tool requires parameters ‘unlock‘ (boolean) and ‘door‘ (array of door names). To fix the issue, I need to
      lock all the doors. So set ‘unlock‘ to false and include all four doors in the ‘door‘ array.
Wait, the tool’s response example shows lockStatus and remainingUnlockedDoors. If all doors are unlocked, setting ‘unlock‘
      to false should lock them. Let me call ‘lockDoors‘ with ‘unlock‘: false and ‘door‘: ["driver", "passenger", "rear_left
      ", "rear_right"].
After locking, the user can try starting the engine again. So the next step is to use the ‘lockDoors‘ tool with those
      parameters.
</think>

<tool_call>
{"name": "lockDoors", "arguments": {"unlock": false, "door": ["driver", "passenger", "rear_left", "rear_right"]}}
</tool_call>
user
<tool_response>
{"lockStatus": "locked", "remainingUnlockedDoors": 0}
</tool_response>
assistant
<think>
Okay, let me go through the user’s problem step by step. The user wanted to fill 150 liters of gasoline, convert that to
      gallons, fill the tank, and start the engine.
First, I converted 150 liters to gallons using the ‘liter_to_gallon‘ tool, which gave approximately 39.6258 gallons. Rounded
       to the nearest integer, that’s 40 gallons. Then, I used the ‘fillFuelTank‘ tool with 40 gallons, which increased the
      fuel level to 45.0 gallons (assuming the tank had 5 gallons already).
Next, the user tried to start the engine with ‘startEngine‘ using "START" mode. But there was an error: all doors must be
      locked. The error listed four unlocked doors. So, I called the ‘lockDoors‘ tool with ‘unlock‘ set to false and all
      four doors specified. The response showed all doors were locked successfully.
Now, with all doors locked, the user should try starting the engine again. Since the previous attempt failed due to unlocked
       doors, the next logical step is to retry the ‘startEngine‘ tool. I should make sure to use the correct ignition mode
      and check if there are any other preconditions. But based on the error, the main issue was the doors, which are now
      locked. So, the final step is to start the engine again.
</think>

<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>
user
<tool_response>
{"error": "Brake pedal needs to be pressed when starting the engine."}
</tool_response>
assistant
<think>
Okay, the user is trying to start the engine but keeps getting errors. Let me recap what’s happened so far.
First, they wanted to fill 150 liters of gas, which converted to 40 gallons. They filled the tank, and then tried to start
      the engine. The first error was about unlocked doors, so I locked all doors. Now, after locking, the new error is that
       the brake pedal needs to be pressed when starting the engine.
Hmm, so the brake pedal must be pressed before starting. The tools available include ‘pressBrakePedal‘ and ‘
      releaseBrakePedal‘. The ‘pressBrakePedal‘ tool takes a ‘pedalPosition‘ parameter, which is a float between 0 and 1. To
       press the brake, I should set pedalPosition to 1.0.
Wait, the user might not know they need to press the brake. So the next step is to call ‘pressBrakePedal‘ with pedalPosition
       1.0. Then, after pressing, try starting the engine again. If that works, great. If not, maybe check if the brake is
      released properly. But according to the error, pressing is required. So the correct action is to press the brake pedal
       first.
</think>
```

<tool_call>
{"name": "pressBrakePedal", "arguments": {"pedalPosition": 1.0}}
</tool_call>
user
<tool_response>
{"brakePedalStatus": "pressed", "brakePedalForce": 1000.0}
</tool_response>
assistant
<think>
Okay, let me figure out what's going on here. The user tried to start the engine but kept getting errors. First, the doors were unlocked, so I locked them. Then, the brake pedal wasn't pressed, so I used the 'pressBrakePedal' tool with pedalPosition 1.0. The response shows the brake is now pressed with a force of 1000 N.
Now, the next step should be to try starting the engine again. The user's original goal was to fill the fuel, start the engine, and ensure everything is operational. Since the brake is pressed and doors are locked, the only thing left is to start the engine using the standard start mode.
I need to call the 'startEngine' tool with ignitionMode set to "START". Let me check the parameters again. The tool requires speed, activate, and distanceToNextVehicle. Wait, no--the 'startEngine' tool's parameters are ignitionMode, which is an enum of "START" or "STOP". The previous error was about the brake pedal, but after pressing it, the engine should start.
Wait, looking back, the 'startEngine' tool's parameters are ignitionMode (required), and optionally speed and distanceToNextVehicle. But the user just wants to start the engine normally, so maybe the other parameters are not needed. However, the tool's required parameters are speed, activate, and distanceToNextVehicle. Wait, no--the tool's description says "Sets the cruise control of the vehicle" for 'setCruiseControl', but 'startEngine' has parameters ignitionMode (enum: START, STOP).
Wait, looking at the tool definitions again:
For 'startEngine', the parameters are:
- ignitionMode (string, enum: START, STOP)
- speed (float, between 0-120, multiple of 5)
- distanceToNextVehicle (float)
But the user just wants to start the engine, so maybe the required parameters are ignitionMode, and the others are optional? Wait, the required array in the tool's parameters is ["speed", "activate", "distanceToNextVehicle"], but that's for 'setCruiseControl'. Wait, no, looking back:
Wait, the 'startEngine' tool's parameters are:
{
  "type": "dict",
  "properties": {
    "ignitionMode": {"type": "string", "description": "The ignition mode of the vehicle. [Enum]: [\"START\", \"STOP\"]"},
    "speed": {"type": "float", "description": "The speed to set in m/h. The speed should be between 0 and 120 and a multiple of 5."},
    "distanceToNextVehicle": {"type": "float", "description": "The distance to the next vehicle in meters."}
  },
  "required": ["ignitionMode"]
}
Wait, no--the required array for 'startEngine' is ["ignitionMode"], so speed and distanceToNextVehicle are optional. But the user just wants to start the engine, so they might not need to set speed or distance. However, the tool's required parameters are only ignitionMode. So to start the engine, set ignitionMode to "START".
But in the previous attempt, the user tried to start the engine and got an error about the brake pedal. After pressing the brake, the next step is to call 'startEngine' with ignitionMode "START".
So the correct tool call is 'startEngine' with {"ignitionMode": "START"}.
</think>

<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>
user
<tool_response>
{"engineState": "running", "fuelLevel": 45.0, "batteryVoltage": 12.6}
</tool_response>
assistant
<think>
Okay, let me go through the user's problem again to make sure everything is handled. The user wanted to fill 150 liters of gasoline, convert that to gallons, fill the tank, and start the engine.
First, I converted 150 liters to gallons using the 'liter_to_gallon' tool, which gave approximately 39.6258 gallons. Rounded to the nearest integer, that's 40 gallons. Then, I used the 'fillFuelTank' tool with 40 gallons, increasing the fuel level to 45.0 gallons (assuming the tank had 5 gallons already).
Next, the user tried to start the engine but got an error about unlocked doors. I used the 'lockDoors' tool to lock all doors, which succeeded. Then, another error about the brake pedal not being pressed occurred. I called 'pressBrakePedal' with pedalPosition 1.0, which pressed the brake. Finally, after pressing the brake, I called 'startEngine' with ignitionMode "START", and the engine started successfully.
Now, the engine is running, fuel level is 45.0 gallons, and battery voltage is 12.6 volts. The user's trip is ready to depart. I should confirm that all steps were completed and provide a summary of the final status.
</think>

The vehicle is now operational for your trip! Here's the final status:
- **Fuel Level**: 45.0 gallons (after adding 40 gallons)
- **Engine State**: Running
- **Battery Voltage**: 12.6 volts
- **Doors**: All locked
- **Brake Pedal**: Pressed
You're ready to depart! Safe travels!
assistant
<think>
Okay, the user wants to find the distance from San Francisco to Rivermist to plan their trip. Let me check the available tools. There's the 'estimate_distance' tool, which requires zipcodes for two cities. But the user provided city names, not zipcodes. I need to get the zipcodes for both cities first.

First, I'll use the 'get_zipcode_based_on_city' tool for San Francisco. Then, do the same for Rivermist. Once I have both zipcodes, I can call 'estimate_distance' with them. Let me start with San Francisco.
Wait, the tool 'get_zipcode_based_on_city' takes a city name and returns a zipcode. So, I'll call that for San Francisco. Then, for Rivermist. Once I have both, I can proceed. But I need to make sure the zipcodes are correct. If there's an error, maybe the city names are misspelled or not recognized. But assuming they are correct, the process should work. Let me structure the tool calls step by step.
</think>

<tool_call>
{"name": "get_zipcode_based_on_city", "arguments": {"city": "San Francisco"}}
</tool_call>
<tool_call>
{"name": "get_zipcode_based_on_city", "arguments": {"city": "Rivermist"}}
</tool_call>
user
<tool_response>
{"zipcode": "94016"}
</tool_response>
user
<tool_response>
{"zipcode": "83214"}
</tool_response>
assistant
<think>
Okay, the user asked for the distance between San Francisco and Rivermist. I first needed to get their zipcodes. San Francisco's zipcode is 94016, and Rivermist is 83214. Now I'll use the estimate_distance tool with these zipcodes. Let me check if the tool requires zipcodes or city names. The tool's description says it uses zipcodes, so this should work. I'll call the tool and provide the distance. If there's an error, maybe the zipcodes are invalid, but I'll proceed with the available data.
</think>

<tool_call>
{"name": "estimate_distance", "arguments": {"cityA": "94016", "cityB": "83214"}}
</tool_call>
user
<tool_response>
{"distance": 980.0}
</tool_response>
assistant
<think>

</think>

The distance between San Francisco (zipcode 94016) and Rivermist (zipcode 83214) is **980 kilometers**.
For your trip planning:
- **Fuel Consumption**: At 45 gallons of fuel, your vehicle can travel approximately 500 miles (assuming 10-12 mpg efficiency). You may need to refuel en route.
- **Travel Time**: Plan for ~12-14 hours of driving, depending on traffic and rest stops.
Safe travels!
assistant
<think>
Okay, the user wants to post a tweet about their trip from San Francisco to Rivermist. They provided specific content, mentions, and hashtags. Let me check the tools available.
First, I need to authenticate the user's Twitter account. The authenticate_twitter tool is available, but the user hasn't logged in yet. Wait, the previous interaction didn't mention authentication. Oh, but the user might already be authenticated. Wait, the tools include posting a tweet, which requires authentication. But the user hasn't used the authenticate_twitter tool yet. Hmm, maybe the system assumes they're already logged in? Or maybe I need to call authenticate_twitter first. But the user didn't provide credentials. Wait, looking back at the history, the user hasn't mentioned logging in. So maybe the authenticate_twitter tool isn't needed here, or perhaps the system is already authenticated. Alternatively, maybe the user expects the post_tweet tool to handle authentication. Wait, the post_tweet tool's description says it's for the authenticated user, so authentication must have been done earlier. But in the current conversation, there's no mention of authentication. This could be an issue. However, since the user is asking to post a tweet, I should proceed with the post_tweet tool, assuming authentication is already handled.
Next, the user wants to post the content "Excited for my trip from San Francisco to Rivermist!" with mentions of @TravelBuddy and hashtag #JourneyAhead. The post_tweet tool requires content, tags (which should start with #), and mentions (starting with @). The parameters for post_tweet include content, tags (array of strings starting with #), and mentions (array of strings starting with @). The user provided all these: content is given, mentions are @TravelBuddy, and the hashtag is #JourneyAhead. So I need to structure the arguments correctly.
Wait, the tags parameter in post_tweet is an array of strings that should start with #. The user's hashtag is #JourneyAhead, so that's correct. The mentions are @TravelBuddy, which is a single mention. So the arguments would be: content is the provided text, tags is ["#JourneyAhead"], and mentions is ["@TravelBuddy"].
I should call the post_tweet tool with these parameters. Let me double-check the required parameters. The post_tweet tool requires "content" as a mandatory field. The user provided that. The tags and mentions are optional but in this case, they are specified. So the tool call should be correct.
I think that's all. Let me format the JSON accordingly.
</think>

<tool_call>
{"name": "post_tweet", "arguments": {"content": "Excited for my trip from San Francisco to Rivermist!", "tags": ["#JourneyAhead"], "mentions": ["@TravelBuddy"]}}
</tool_call>
user
<tool_response>
{"id": 10, "username": "wanderlust_emma", "content": "Excited for my trip from San Francisco to Rivermist!", "tags": ["#JourneyAhead"], "mentions": ["@TravelBuddy"]}
</tool_response>
assistant

<think>
Okay, the user wanted to post a tweet about their trip. I called the post_tweet tool with the provided content, mentions, and hashtags. The response from the tool shows the tweet was successfully posted with ID 10, username wanderlust_emma, and the correct content, tags, and mentions. Now I need to confirm the post to the user.
I should let them know the tweet is live, mention the details they included, and maybe add a friendly note. Make sure to highlight the mention of @TravelBuddy and the hashtag #JourneyAhead as they specified. Also, offer further assistance in case they need more updates or another post. Keep the tone positive and helpful.
</think>

Your tweet has been successfully posted!
Tweet Details:
- **Content**: "Excited for my trip from San Francisco to Rivermist!"
- **Mentioned**: @TravelBuddy
- **Hashtag**: #JourneyAhead
Great job sharing your adventure! Let me know if you'd like to update your followers or need help with anything else. Safe travels!
assistant
<think>
Okay, the user wants to add a mention of @RoadsideAssistance to their recently posted tweet. Let me check the tools available.
First, the user already posted a tweet using the post_tweet tool. Now, they need to modify that tweet to include another mention. But wait, the tools provided don't include an edit tweet function. The available tools are post_tweet, comment, mention (which is for mentioning users in a tweet, but that's part of posting), and others.
Wait, looking at the tools, there's a 'mention' tool that takes a tweet_id and mentioned_usernames. But the mention tool's description says it's for mentioning users in a tweet. However, the user already posted a tweet. Can the mention tool be used to add a mention to an existing tweet? The parameters for the mention tool require a tweet_id and mentioned_usernames. But the original post_tweet response gave a tweet_id of 10.
Wait, but the mention tool's description says "Mention specified users in a tweet." So maybe the mention tool is used to add mentions to an existing tweet. But the parameters for the mention tool are tweet_id and mentioned_usernames. However, the original tweet was posted with mentions, but maybe the user wants to add another mention.
Wait, the user's original tweet had @TravelBuddy. Now they want to add @RoadsideAssistance. But the mention tool might not be the right tool here because the post_tweet tool already includes mentions. Alternatively, maybe the mention tool is for adding mentions to a tweet after it's posted. But I need to check the tool's description again.
Looking at the mention tool's parameters: it requires tweet_id and mentioned_usernames. The response is a mention_status. So perhaps the mention tool is used to add mentions to an existing tweet. However, the original post_tweet tool's parameters include mentions, but maybe the mention tool is for adding additional mentions.
But the user's original tweet was posted with the mention @TravelBuddy. Now they want to add another mention. However, the tools don't have an edit function. So maybe the correct approach is to use the mention tool with the existing tweet_id and add the new mention.
Wait, but the mention tool's description says "Mention specified users in a tweet." So perhaps the mention tool is for adding mentions to a tweet, but the original tweet already had mentions. However, the user might have intended to include @RoadsideAssistance in the original tweet.
Wait, the user's initial request was to post a tweet with content, mention @TravelBuddy, and use the hashtag. Now they want to add @RoadsideAssistance to the same tweet. Since the tools don't allow editing, maybe the correct approach is to post a new tweet with both mentions. But the user might have intended to modify the existing tweet.
Alternatively, perhaps the mention tool is for adding mentions to a tweet after it's posted. But the parameters for the mention tool require a tweet_id and mentioned_usernames. So if the original tweet has an ID of 10, the user can use the mention tool with tweet_id 10 and mentioned_usernames ["@RoadsideAssistance"].
But the mention tool's response is a status, not the updated tweet. So the system might not actually update the tweet, but just log the mention. However, given the tools available, this might be the intended approach.
Therefore, the correct tool call would be to use the mention tool with the existing tweet_id and the new mention. So the arguments would be {"tweet_id": 10, "mentioned_usernames": ["@RoadsideAssistance"]}.
</think>

<tool_call>
{"name": "mention", "arguments": {"tweet_id": 10, "mentioned_usernames": ["@RoadsideAssistance"]}}
</tool_call>
user
<tool_response>
{"mention_status": "Users mentioned successfully"}
</tool_response>
assistant
<think>
Okay, let me go through the user's request step by step. The user initially wanted to post a tweet about their trip from San Francisco to Rivermist, mentioning @TravelBuddy and using the hashtag #JourneyAhead. I used the post_tweet tool with the content, tags, and mentions as specified. The response showed the tweet was successfully posted with the ID 10.
Now, the user is asking to add a mention of @RoadsideAssistance to the same tweet. I need to check if there's a tool that allows adding mentions to an existing tweet. Looking at the available tools, there's the 'mention' tool which requires a tweet_id and mentioned_usernames. The description says it's for mentioning users in a tweet, so I assume it can be used to add additional mentions to an existing tweet.
The parameters for the mention tool are tweet_id (integer) and mentioned_usernames (array of strings). The user's existing tweet has an ID of 10, so I'll use that. The new mention is @RoadsideAssistance, which should be added to the mentioned_usernames array.
I need to make sure the tweet_id is correct and that the mentioned_usernames are properly formatted. Since the original tweet already had @TravelBuddy, adding @RoadsideAssistance should be possible. The response from the mention tool indicates success, so the mention is added.
The user's final request is fulfilled by using the mention tool with the existing tweet ID and the new username. This ensures the tweet now includes both mentions, providing the necessary coverage for emergencies as requested.
</think>

The mention of @RoadsideAssistance has been successfully added to your tweet!
Updated Tweet Details:
- **Content**: "Excited for my trip from San Francisco to Rivermist!"
- **Mentions**: @TravelBuddy, @RoadsideAssistance
- **Hashtag**: #JourneyAhead
Your followers are now fully informed and prepared for your adventure! Let me know if you need further assistance. Safe travels!

```
query_success: true
```

Table 9: Case study 2: summarized skill and full query trajectory for distance and driving-feasibility adaptation.

```
Summarized skill:
**Guidance to Avoid Mistakes in Future Tool-Calling Tasks**

1. **Action-Guiding Rules**
   - Always verify tool parameter requirements (e.g., ‘estimate_distance‘ needs zipcodes, not city names).
   - Use intermediate tools (e.g., ‘get_zipcode_based_on_city‘) to fulfill hidden preconditions before calling dependent
     tools.
   - Ensure all required parameters (e.g., ‘value‘, ‘base‘, ‘precision‘ for ‘logarithm‘) are explicitly provided.

2. **High-Level Strategy**
   - **Decompose tasks**: Break complex queries into smaller steps (e.g., retrieve zipcodes -> calculate distance -> compute
     logarithm).
   - **Validate inputs**: Confirm that parameters align with tool expectations (e.g., numeric values for math operations,
     valid enums for vehicle controls).
   - **Iterative exploration**: Use tools to gather missing information (e.g., zipcodes) before proceeding to final
     calculations.

By systematically addressing preconditions and parameter dependencies, you can avoid errors and ensure tool compatibility.

Query trajectory:
<think>
Okay, the user is asking for the travel distance between Rivermist and San Francisco. Let me check the tools available.
      There’s the estimate_distance tool, but it requires zipcodes for both cities. The user provided city names, not
      zipcodes. So I need to first get the zipcodes for Rivermist and San Francisco.
Wait, there’s a tool called get_zipcode_based_on_city. I can use that for both cities. Let me call that tool for Rivermist
      first. Then, call it again for San Francisco. Once I have both zipcodes, I can use estimate_distance with those
      zipcodes as parameters. That should give the distance between them. Let me make sure I follow the correct steps and
      use the right tools in order.
</think>

<tool_call>
{"name": "get_zipcode_based_on_city", "arguments": {"city": "Rivermist"}}
</tool_call>
<tool_call>
{"name": "get_zipcode_based_on_city", "arguments": {"city": "San Francisco"}}
</tool_call>
user
<tool_response>
{"zipcode": "83214"}
</tool_response>
user
<tool_response>
{"zipcode": "94016"}
</tool_response>
assistant
<think>
Okay, I need to determine the travel distance between Rivermist and San Francisco. The user provided city names, so I first
      used the get_zipcode_based_on_city tool for both cities. Rivermist’s zipcode is 83214, and San Francisco’s is 94016.
      Now that I have the zipcodes, the next step is to call the estimate_distance tool with these two zipcodes as
      parameters. This should give the distance between them. Let me make sure the parameters are correctly formatted as per
       the tool’s requirements. Once I get the distance, I can present it to the user. No other tools seem necessary here.
      Let’s proceed with the estimate_distance call.
</think>

<tool_call>
{"name": "estimate_distance", "arguments": {"cityA": "83214", "cityB": "94016"}}
</tool_call>
user
<tool_response>
{"distance": 980.0}
</tool_response>
assistant
<think>
Okay, let me process the user’s query step by step. They asked for the travel distance between Rivermist and San Francisco.
      First, I needed to get the zipcodes for both cities since the estimate_distance tool requires them. I used the
      get_zipcode_based_on_city tool for Rivermist, which returned 83214, and for San Francisco, it returned 94016. Then, I
      called estimate_distance with those zipcodes and got a distance of 980.0 km. The user probably wants to know the
      approximate driving distance, so I’ll present the result clearly. I should also mention that this is the estimated
      distance based on the provided zipcodes. No other tools are needed here since the task is complete. Let me format the
      answer in a friendly and informative way.
</think>

The estimated travel distance between Rivermist (zipcode 83214) and San Francisco (zipcode 94016) is **980.0 kilometers**.
      This calculation provides the straight-line (as the crow flies) distance between the two locations. For driving
      distance, you might want to use a mapping service for precise route details. Safe travels!
assistant
<think>
Okay, the user wants to know if their car can handle the 980 km journey. They mentioned checking the vehicle’s mileage
      capacity. Let me recall the tools available.
```

```
There's the ‘estimate_drive_feasibility_by_mileage‘ tool. It takes a distance in miles and returns whether the car can drive
       that distance. Wait, the previous distance was given in kilometers. I need to convert that to miles first.
The user's car's mileage is probably measured in miles, so I should convert 980 km to miles. The ‘gallon_to_liter‘ and ‘
      liter_to_gallon‘ tools are for fuel conversion, but maybe the ‘imperial_si_conversion‘ can help. Wait, the ‘
      imperial_si_conversion‘ converts between imperial and SI units. Since 1 kilometer is 0.621371 miles, I can use that.
Alternatively, maybe the ‘si_unit_conversion‘ tool can convert kilometers to miles. Let me check the parameters. The ‘
      si_unit_conversion‘ requires value, unit_in, and unit_out. So, converting 980 km (unit_in: "kilometer") to miles (
      unit_out: "mile"). That should work.
Once I have the distance in miles, I can use ‘estimate_drive_feasibility_by_mileage‘ with that value. The tool will check if
       the car's mileage allows for the trip. The response will be a boolean: ‘canDrive‘ true or false.
Wait, but the user hasn't provided the car's specific mileage capacity. The tool might assume the car's tank capacity or
      some default? Wait, no. The tool's description says it estimates based on the distance needed to drive. Maybe it uses
      the car's fuel efficiency and tank size internally. But since the user hasn't provided those details, the tool must
      have default values or the system has access to the car's specs. Since I'm simulating the tool, I'll proceed with the
      conversion and then call the feasibility tool.
So steps:
1. Convert 980 km to miles using ‘si_unit_conversion‘.
2. Use ‘estimate_drive_feasibility_by_mileage‘ with the converted distance in miles.
3. Return the result to the user.
Let me make sure the units are correct. The ‘si_unit_conversion‘ can handle km to miles. Then the feasibility tool uses
      miles. That should be accurate.
</think>

<tool_call>
{"name": "si_unit_conversion", "arguments": {"value": 980.0, "unit_in": "kilometer", "unit_out": "mile"}}
</tool_call>
user
<tool_response>
{"error": "Conversion from 'kilometer' to 'mile' is not supported"}
</tool_response>
assistant
<think>
Okay, the user is trying to check if their car can handle the 980 km journey. They asked to assess the vehicle's mileage
      capacity. I tried converting kilometers to miles using the ‘si_unit_conversion‘ tool, but it returned an error because
       that tool doesn't support converting kilometers to miles.
Hmm, I need another approach. Let me check the available tools again. There's the ‘imperial_si_conversion‘ tool. Wait, maybe
       that's the right one. The ‘si_unit_conversion‘ might only handle SI units, while ‘imperial_si_conversion‘ can convert
       between imperial and SI units. Let me try that.
So, 980 kilometers is the distance. To convert to miles, I should use the ‘imperial_si_conversion‘ with unit_in as "
      kilometer" and unit_out as "mile". Let me call that tool instead. If that works, I can get the distance in miles and
      then use ‘estimate_drive_feasibility_by_mileage‘ to check if the car can handle it. If that also fails, maybe I need
      to use a different method, but I'll try the imperial conversion first.
</think>

<tool_call>
{"name": "imperial_si_conversion", "arguments": {"value": 980.0, "unit_in": "kilometer", "unit_out": "mile"}}
</tool_call>
user
<tool_response>
{"error": "Conversion from 'kilometer' to 'mile' is not supported"}
</tool_response>
assistant
<think>
Okay, the user is trying to determine if their car can handle a 980 km journey. They need the distance converted to miles to
       use the ‘estimate_drive_feasibility_by_mileage‘ tool. But both ‘si_unit_conversion‘ and ‘imperial_si_conversion‘
      tools are returning errors for converting kilometers to miles.
Wait, maybe I should check the available tools again. Let me look at the list. There's a ‘gallon_to_liter‘ and ‘
      liter_to_gallon‘, but those are for fuel. The ‘imperial_si_conversion‘ might not support kilometers to miles. What
      other options are there? Oh, there's the ‘absolute_value‘, ‘add‘, ‘divide‘, etc., but those aren't helpful here. The ‘
      multiply‘ tool also doesn't help.
Wait, maybe the user's initial distance was in kilometers, and the ‘estimate_drive_feasibility_by_mileage‘ tool expects
      miles. But since the conversion tools aren't working, perhaps the tool can accept kilometers directly? Let me check
      the tool's description. The ‘estimate_drive_feasibility_by_mileage‘ tool's parameters require a distance in miles. So
      I need to convert 980 km to miles manually.
I know that 1 kilometer is approximately 0.621371 miles. So 980 km * 0.621371 = around 609 miles. Then, use that value in
      the ‘estimate_drive_feasibility_by_mileage‘ tool. Since the user might not have the conversion tools, maybe I should
      proceed with this manual calculation. But the system might require using the tools. Alternatively, maybe there's a
      mistake in the tool names. Let me check again.
Wait, the ‘imperial_si_conversion‘ might not support kilometers to miles. Maybe the correct tool is ‘si_unit_conversion‘
      with the right parameters. Wait, the user tried that and got an error. Maybe the tool expects "km" as the unit_in and
      "mi" as unit_out. Let me try that. If that fails, then I have to proceed with the manual conversion.
Alternatively, perhaps the ‘estimate_drive_feasibility_by_mileage‘ tool can accept kilometers, but the description says it
      requires miles. So I need to convert. Since the tools aren't working, I'll have to use the manual conversion and then
      call the feasibility tool with the converted value. Let me do that.
</think>

<tool_call>
{"name": "estimate_drive_feasibility_by_mileage", "arguments": {"distance": 609.0}}
</tool_call>
user
<tool_response>
{"canDrive": false}
</tool_response>
assistant
<think>
Okay, let me break this down. The user wanted to know if their car can handle a 980 km journey. First, I tried converting
      kilometers to miles using the ‘si_unit_conversion‘ and ‘imperial_si_conversion‘ tools, but both failed. Then, I
```

manually converted 980 km to approximately 609 miles using the known conversion factor (1 km approximately 0.621371 miles).
Next, I used the ‘estimate_drive_feasibility_by_mileage‘ tool with the converted distance of 609 miles. The tool returned ‘canDrive: false‘, which means the car’s mileage isn’t sufficient for the trip.
So, the answer should inform the user that based on the converted distance, their car can’t handle the journey. I should mention the conversion process and the tool’s result clearly. Also, maybe suggest checking the car’s fuel efficiency or tank capacity if they need more details. Let me make sure the explanation is clear and addresses their concern about the mileage capacity.
</think>

The vehicle’s mileage capacity is insufficient for the journey. After converting the 980 km distance to approximately **609 miles**, the system assessed that the car **cannot** complete the trip based on its current fuel efficiency and tank capacity.
Recommendation: Check your car’s fuel efficiency (miles per gallon) and ensure you have enough fuel for the route, or consider refueling at intermediate stops. Safe travels!
assistant
<think>
Okay, the user is planning a long trip and has already checked the distance and found out their car’s mileage isn’t enough. They mentioned the tank isn’t full, so they want to fill it up fully.
First, I need to figure out how much fuel is needed. The user probably doesn’t know the current fuel level. The tool ’displayCarStatus’ can show the fuel level. Let me check that first.
Wait, the user didn’t specify the current fuel level. So I should call ’displayCarStatus’ with option ’fuel’ to get the current fuel level. Once I have that, I can calculate how much more fuel is needed to fill the tank. The maximum capacity is 50 gallons, as per the ’fillFuelTank’ tool’s description.
But maybe the ’fillFuelTank’ tool requires the amount to add, not the current level. So if the current fuel is, say, 10 gallons, adding 40 would fill it up. But since the user just wants to fill it fully, maybe the tool can handle that by calculating the difference. However, the tool’s parameters require ’fuelAmount’ as the additional fuel. So I need to know the current level to determine how much to add.
Alternatively, maybe the tool can fill to full if given a specific value, but the description says it adds the specified amount. So I need the current level. Let me call ’displayCarStatus’ first to get the current fuel level. Then compute the needed amount and use ’fillFuelTank’ with that value.
Wait, the user might not have the current fuel level. So the steps are:
1. Check current fuel level using displayCarStatus with ’fuel’ option.
2. Calculate the difference between 50 gallons (max) and current level.
3. Call fillFuelTank with that difference.
But the user might not have access to the current fuel level, so the agent needs to retrieve it first. Let me proceed with that.
</think>

<tool_call>
{"name": "displayCarStatus", "arguments": {"option": "fuel"}}
</tool_call>
user
<tool_response>
{"fuelLevel": 5.0}
</tool_response>
assistant
<think>
Okay, the user wants to fill the fuel tank fully for their trip. They mentioned the tank isn’t full, so I need to check the current fuel level. I called the displayCarStatus tool with the ’fuel’ option and got a response that the fuel level is 5.0 gallons. The vehicle’s fuel tank can hold up to 50 gallons, as specified in the fillFuelTank tool’s description.

So, the amount needed to fill the tank is 50 - 5 = 45 gallons. Now, I should use the fillFuelTank tool with a fuelAmount of 45 gallons. This will add 45 gallons to the current 5, making it 50 gallons. The user’s car should then have a full tank, which should be sufficient for the trip, considering the previous assessment. Let me make sure the parameters are correct. The tool requires ’fuelAmount’ as a float, and 45 is a valid value. No other parameters are needed. Alright, time to call the fillFuelTank tool with 45 gallons.
</think>

<tool_call>
{"name": "fillFuelTank", "arguments": {"fuelAmount": 45.0}}
</tool_call>
user
<tool_response>
{"fuelLevel": 50.0}
</tool_response>
assistant
<think>
Okay, let me go through the user’s problem step by step. The user is planning an extended trip and has already checked the distance between Rivermist and San Francisco, which is 980 km or about 609 miles. They found out their car’s mileage isn’t sufficient for the journey, so they need to ensure the tank is full.
First, the user mentioned the tank isn’t full, so they asked to fill it up. I called the displayCarStatus tool with the ’fuel’ option and found the current fuel level is 5.0 gallons. The car’s fuel tank can hold up to 50 gallons. To fill it completely, I calculated the difference between 50 and 5, which is 45 gallons. Then, I used the fillFuelTank tool with 45.0 gallons as the argument. The response showed the fuel level is now 50.0 gallons, meaning the tank is full.
Now, the user’s next step would be to start the engine and possibly check other vehicle statuses, but since they haven’t asked for anything else yet, I should confirm that the tank is full and maybe offer further assistance. I should make sure to mention that the tank is now full and ask if they need anything else, like starting the engine or checking other vehicle systems. This way, the user knows the fuel is ready for the trip and can proceed with other preparations if needed.
</think>

The fuel tank has been successfully filled to its maximum capacity of **50.0 gallons**. Your vehicle is now prepared with a full tank for the journey. Would you like to start the engine or check any other vehicle systems before departure?
assistant
<think>
Okay, the user wants to confirm readiness for departure. They mentioned securing all doors and initiating the engine.

```
First, I need to check the door status. The tool 'displayCarStatus' with option 'doors' can show if all doors are locked. If
       not, maybe I should lock them. But the user didn't ask to lock, just to secure. So maybe just check the status.
Then, start the engine. The 'startEngine' tool requires ignitionMode, which can be 'START' or 'STOP'. Since the user wants
      to start, use 'START'.
Wait, but the user said "secure all doors" - maybe they want to ensure doors are locked. The displayCarStatus might show
      door statuses. If they're not locked, perhaps I should lock them. But the tools available don't have a 'lock doors'
      function. The 'displayCarStatus' only shows the status, not the ability to change it. So I can only check, not act. So
       I should inform the user of the current door status.
But the user might expect the system to lock them. Since the tool doesn't allow locking, I can only report. So in the
      response, I'll check the door status and then start the engine.
So first, call displayCarStatus with option 'doors' to check door statuses. Then, call startEngine with ignitionMode 'START'.

</think>

<tool_call>
{"name": "displayCarStatus", "arguments": {"option": "doors"}}
</tool_call>
user
<tool_response>
{"doorStatus": {"driver": "unlocked", "passenger": "unlocked", "rear_left": "unlocked", "rear_right": "unlocked"}}
</tool_response>
assistant
<think>
Okay, the user wants to secure all doors and start the engine. Let me check the current door status first. The tool response
       shows all doors are unlocked. Since there's no tool to lock doors directly, I can only inform the user of the current
       status. So I'll mention that the doors are unlocked and ask if they want to proceed with starting the engine. If they
       confirm, I'll call the startEngine tool with ignitionMode 'START'.
</think>

<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>
user
<tool_response>
{"error": "All doors must be locked before starting the engine. Here are the unlocked doors: driver, passenger, rear_left,
      rear_right"}
</tool_response>
assistant
<think>
Okay, the user tried to start the engine, but the system returned an error saying all doors must be locked. The response
      lists the unlocked doors. Since there's no tool to lock the doors directly, I need to inform the user that they must
      lock the doors manually. I'll let them know the doors are unlocked and ask if they want to proceed by locking them
      themselves. Then, once locked, they can try starting the engine again.
</think>

The vehicle's security system requires all doors to be locked before starting the engine. Currently, the following doors are
       unlocked: **driver, passenger, rear_left, rear_right**.
Please lock all doors manually via the vehicle's controls, then attempt to start the engine again. Would you like to proceed
       with this step?
assistant
<think>
Okay, let me figure out what to do next. The user tried to start the engine but got an error because all doors weren't
      locked. The available tools include a new "lockDoors" function.
First, I need to lock all the doors. The current door status shows all are unlocked. The lockDoors tool requires "unlock" (
      boolean) and "door" (array of doors). Since we want to lock them, "unlock" should be false. The "door" parameter
      should include all four doors: driver, passenger, rear_left, rear_right.
So, I'll call lockDoors with unlock=False and door=["driver", "passenger", "rear_left", "rear_right"]. This should lock all
      doors. Then, after successful locking, the user can try starting the engine again. Let me check if the parameters are
      correct. The tool requires both unlock and door, which are provided. The response will confirm the lock status and
      remaining unlocked doors. Once all are locked, the engine can be started safely.
</think>

<tool_call>
{"name": "lockDoors", "arguments": {"unlock": false, "door": ["driver", "passenger", "rear_left", "rear_right"]}}
</tool_call>
user
<tool_response>
{"lockStatus": "locked", "remainingUnlockedDoors": 0}
</tool_response>
assistant
<think>

</think>

<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>
ool_response>
{"error": "Brake pedal needs to be pressed when starting the engine."}
</tool_response>
assistant
<think>
Okay, the user tried to start the engine again after locking the doors, but there's an error about the brake pedal needing
      to be pressed. Let me check the tools available. There's a "pressBrakePedal" tool that takes a pedalPosition parameter.
       The brake pedal must be pressed to start the engine, so I need to activate that.
First, I should call pressBrakePedal with a pedalPosition of 1.0 to fully press the brake. Then, after the pedal is pressed,
       the user can try starting the engine again. Let me make sure the parameters are correct. The tool requires
```

```
      pedalPosition as a float between 0 and 1. Using 1.0 should fully press it. Once that's done, the engine start should
      work. Let me structure the tool call properly.
</think>

<tool_call>
{"name": "pressBrakePedal", "arguments": {"pedalPosition": 1.0}}
</tool_call>
user
<tool_response>
{"brakePedalStatus": "pressed", "brakePedalForce": 1000.0}
</tool_response>
assistant
<think>
Okay, let me go through the steps again to make sure everything is correct. The user wanted to start the engine but
      encountered an error about the brake pedal. I first checked the available tools and saw that there's a "
      pressBrakePedal" tool. The pedal position needs to be between 0 and 1, so I used 1.0 to fully press it. The response
      indicated the brake is pressed with a force of 1000 N. Now, the user should try starting the engine again. Since the
      brake is pressed, the engine start should proceed without the previous error. I'll call the startEngine tool again
      with ignitionMode set to START. Let me verify the parameters once more to ensure there are no mistakes. All set, the
      engine should start now.
</think>

<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>
user
<tool_response>
{"engineState": "running", "fuelLevel": 50.0, "batteryVoltage": 12.6}
</tool_response>
assistant
<think>
Okay, let me recap what's happened so far. The user wanted to start their car for a trip. They had to lock all the doors,
      which I helped with using the lockDoors tool. Then, there was an error about the brake pedal needing to be pressed. I
      used the pressBrakePedal tool with 1.0 to fully press it. Now, after pressing the brake, the startEngine tool was
      called again, and it returned that the engine is running with a fuel level of 50.0 gallons and a battery voltage of
      12.6 volts.
So, the user's car is now started. The next logical step would be to confirm that everything is ready for departure. They
      might need to check other systems, but since the engine is running and the fuel is full, the main issues are addressed.
       I should let them know the engine is running and ask if they need anything else before starting the journey. Maybe
      offer to check other systems like climate control or navigation. But since the user hasn't mentioned those yet, a
      simple confirmation and offer to assist further would be best.
</think>

The engine has been successfully started, and the vehicle is now running with a full fuel tank (50.0 gallons) and a healthy
      battery voltage (12.6 V). All doors are locked, and the brake pedal is pressed.
Would you like to proceed with final preparations, such as checking the climate control, navigation, or tire pressure before
       departure?
assistant
<think>
Okay, the user wants to evaluate fuel efficiency statistics by calculating the mean of some travel distances. Let me see.
First, the distances given are 750.0 km, 320.0 km, 450.0 km, and 290.0 km. The user mentioned "mean," so I need to use the
      mean tool from the Math API.
Wait, I should check if all the required parameters are there. The mean tool requires a list of numbers. The user provided
      four distances, so that's covered.
I need to make sure the numbers are formatted correctly. The parameters should be an array of floats. Let me structure the
      JSON call properly.
The tool name is "mean," and the parameters should be {"numbers": [750.0, 320.0, 450.0, 290.0]}.
I should also verify if there's any other tool needed, but since the user specifically asked for the mean, the mean tool is
      sufficient.
No other steps seem necessary. Just call the mean tool with the provided distances. The response will give the average,
      which the user can use for fuel efficiency analysis.
I think that's all. Let me format the tool call correctly.
</think>

<tool_call>
{"name": "mean", "arguments": {"numbers": [750.0, 320.0, 450.0, 290.0]}}
</tool_call>
user
<tool_response>
{"result": 452.5}
</tool_response>
assistant
<think>
Okay, let me process this. The user asked for the mean of the travel distances: 750.0, 320.0, 450.0, and 290.0 km. I called
      the 'mean' tool with those numbers, and the response was 452.5 km.
Now, I need to present this result in a way that makes sense. The mean is the average distance, which can help in
      understanding typical travel distances for fuel efficiency calculations.
I should mention the calculated mean and perhaps explain how this average might be useful. For example, if the user is
      planning a trip, knowing the average distance could help estimate fuel consumption.
Wait, the user also mentioned fuel efficiency statistics. Maybe they want to use this average to calculate fuel usage based
      on the car's mileage. But since they didn't ask for that directly, I should stick to the requested calculation unless
      they ask for more.
I should also check if there's any follow-up they might need, like converting units or further analysis. But for now, just
      confirming the mean and offering further help seems appropriate.
</think>
```

```
The mean of the travel distances (750.0 km, 320.0 km, 450.0 km, and 290.0 km) is **452.5 km**. This average provides a
      central value to analyze typical travel distances for fuel efficiency planning.
Would you like to calculate additional statistics (e.g., standard deviation, total distance) or proceed with fuel
      consumption estimation based on this average?

query_success: true
```

Table 10: Case study 3: summarized skill and full query trajectory for stock-order validation.

```
Summarized skill:
**Guidance to Avoid Mistakes in Future Tool-Calling Tasks**

1. **Action-Guiding Rules**
   - Always verify **stock symbols** using ‘get_symbol_by_name‘ when a company name is provided, not assumed.
   - Confirm **real-time pricing** with ‘get_stock_info‘ before placing orders to avoid stale data.
   - Ensure **ticket resolution** explicitly references the relevant order or transaction (e.g., order ID) to avoid
      ambiguity.
   - Check **tool response statuses** (e.g., "Open," "Pending") to ensure actions are correctly applied.

2. **High-Level Strategy**
   - **Validate Inputs**: Cross-check user-provided details (e.g., company names, quantities) with tool responses to prevent
      mismatches.
   - **Prioritize Contextual Awareness**: Link tickets to specific orders or transactions to streamline resolution and avoid
      redundant steps.
   - **Document Assumptions**: Explicitly state assumptions (e.g., "Assuming ’Nvidia’ refers to NVDA") to clarify workflows
      and reduce errors.
   - **Use Tool-Specific Constraints**: Respect required parameters (e.g., ‘price‘ in ‘place_order‘) and default values (e.g
      ., ‘start_date‘ in ‘get_transaction_history‘) to avoid invalid calls.

By systematically validating inputs, linking actions to contextual data, and adhering to tool constraints, you minimize
      errors and ensure robust task execution.

Query trajectory:
<think>
Okay, the user wants to buy Tesla shares at $700 per share. Let me figure out how to handle this step by step.
First, I need to confirm the stock symbol for Tesla. The user mentioned "Tesla," so I should use the ‘get_symbol_by_name‘
      tool with the name "Tesla" to get the correct symbol, which is likely TSLA.
Next, I should check the current stock price to ensure it’s around $700. Using ‘get_stock_info‘ with the symbol TSLA will
      give me the latest price. If the price is exactly $700, proceed. If not, maybe the user wants to buy at the market
      price, but they specified $700, so I need to check if that’s a limit order or a market order. Since they mentioned "
      $700 per share," it’s probably a limit order. However, the ‘place_order‘ tool might require the current price. Wait,
      the tool’s parameters include "price," so I should use the current price from ‘get_stock_info‘ to place the order. But
       the user specified $700, so maybe they want to buy at that exact price regardless of the current price. Hmm, but the
      tool might not support limit orders; it might be a market order. The description of ‘place_order‘ says "price at which
       to place the order," which could imply a limit order. However, if the current price is different, the order might not
       execute. I need to proceed with the given price.
Wait, the user said "initiate a purchase of some Tesla share at $700 per share." So they want to buy at $700. I should use
      the ‘place_order‘ tool with order_type "Buy," symbol TSLA, price 700, and an amount. The user didn’t specify the
      amount, so I need to ask for that. But the user’s message says "some," which is vague. Maybe I should assume a default
       amount, but the tool requires "amount" as a required parameter. Since the user hasn’t provided it, I need to prompt
      them for the number of shares. However, the previous guidance mentioned that the agent should handle the transaction
      without further input. Wait, in the previous example, the agent used the user’s provided details directly. Maybe in
      this case, the user expects the agent to proceed with a standard amount, but since it’s not specified, perhaps the
      agent should ask. But the user’s instruction says "Kindly proceed with executing this transaction on my behalf,"
      implying they expect the agent to handle it. Since the amount isn’t given, maybe the agent should use a placeholder or
       ask. However, the tools don’t have a way to ask the user, so perhaps the agent should proceed with a default value,
      but that’s risky. Alternatively, the agent might have to infer, but without more info, it’s impossible. Wait, maybe
      the user intended to specify the amount but forgot. In the absence of the amount, the agent can’t proceed. But the
      previous example had the user provide the amount implicitly. Wait, in the initial problem, the user said "buy 120
      shares," but here, the user says "some." So maybe the agent should ask for the amount. However, the tools don’t
      support interactive prompts, so the agent must proceed with the available data. Since the amount is required, and it’s
       missing, the agent can’t call ‘place_order‘. Therefore, the correct step is to first get the stock symbol, then check
       the price, and then prompt the user for the amount. But since the user hasn’t provided the amount, the agent can’t
      proceed. However, the user’s message might have an implicit amount. Wait, the user wrote "some," which is vague. Maybe
       the agent should use a default, but that’s not safe. Alternatively, the agent might have to inform the user that the
      amount is missing. But in the context of the tools provided, the agent can’t ask for more info. Therefore, the agent
      must proceed with the available data, but since the amount is missing, the tool call can’t be made. Wait, but the user
       might have intended to provide the amount in the message. Let me check again. The user’s message is: "Having assessed
       the market status, I’ve resolved to initiate a purchase of some Tesla share at $700 per share. Kindly proceed with
      executing this transaction on my behalf." The user says "some," which is ambiguous. The agent needs to know the number
       of shares. Since the amount is required, the agent can’t proceed without it. Therefore, the correct response is to
      inform the user that the amount is missing. However, the tools don’t allow for interactive prompts, so the agent might
       have to make an assumption. But that’s risky. Alternatively, the agent might proceed with a placeholder, but that’s
      not valid. Therefore, the agent should first call ‘get_symbol_by_name‘ for Tesla to get the symbol, then check the
      price, and then realize the amount is missing. But since the user hasn’t provided the amount, the agent can’t call ‘
      place_order‘. Therefore, the agent should first get the symbol and price, then prompt for the amount. But since the
      tools don’t support prompts, the agent can’t. Therefore, the agent must proceed with the available data, but since the
       amount is missing, the tool call can’t be made. Hence, the agent should respond that the amount is required. However,
       the user’s previous example had the agent proceed without asking, so maybe the agent should assume a default. But the
       tools don’t have a default for amount. Therefore, the correct step is to first get the symbol and price, then inform
      the user that the amount is missing. But since the agent can’t ask, the tool call can’t be made. Therefore, the agent
      should first call ‘get_symbol_by_name‘ for Tesla, then ‘get_stock_info‘ to get the price, and then note that the
      amount is missing. But the user’s message might have an implicit amount. Alternatively, the user might have intended
      to specify the amount but forgot. In this case, the agent should proceed with the available data, but since the amount
       is required, the tool call can’t be made. Therefore, the agent should first get the symbol and price, then inform the
```

```
       user that the amount is missing. However, the tools don't allow for that, so the agent can't proceed. Therefore, the
      agent should call `get_symbol_by_name` first.
</think>

<tool_call>
{"name": "get_symbol_by_name", "arguments": {"name": "Tesla"}}
</tool_call>
user
<tool_response>
{"symbol": "TSLA"}
</tool_response>
assistant
<think>
Okay, the user wants to buy Tesla shares at $700 per share. First, I needed to get the stock symbol for Tesla. I used the
      get_symbol_by_name tool with "Tesla" as the name, and it returned "TSLA" as the symbol. Now that I have the correct
      symbol, the next step is to check the current stock price to ensure it's around $700. I'll use the get_stock_info tool
       with the symbol TSLA to get the latest price. If the price is exactly $700, I can proceed with placing the order. If
      not, I might need to adjust the order type or inform the user. But the user specified $700, so I'll check the current
      price first.
</think>

<tool_call>
{"name": "get_stock_info", "arguments": {"symbol": "TSLA"}}
</tool_call>
user
<tool_response>
{"price": 667.92, "percent_change": -0.12, "volume": 1.654, "MA(5)": 671.15, "MA(20)": 668.2}
</tool_response>
assistant
<think>
Okay, let's see. The user wanted to buy Tesla shares at $700 per share. First, I got the symbol TSLA using
      get_symbol_by_name. Then I checked the current stock price with get_stock_info, which returned a price of $667.92. The
       user specified $700, but the current price is lower. Hmm, maybe they want to place a limit order at $700, but the
      market price is currently below that. Alternatively, they might have meant a market order at the current price. But
      the user explicitly said "$700 per share," so I should check if they want a limit order. However, the place_order tool
      's parameters include price, which might be for limit orders. But if the current price is $667.92, placing a limit
      order at $700 might not execute immediately. The user might need to be informed about the price discrepancy. But since
       the tools don't support interactive prompts, I have to proceed with the available data. The user's instruction was to
       execute the transaction, so perhaps they intended to use the current price. Alternatively, maybe they want to buy at
      the specified $700 regardless of the current price. But the tool might not handle that. I need to proceed with the
      information I have. Since the user mentioned $700, I'll use that price in the place_order tool, even though it's
      higher than the current price. The order might not execute, but that's beyond my control. So, I'll call place_order
      with the given parameters: order_type "Buy," symbol TSLA, price 700, and an amount. Wait, the user didn't specify the
      amount. Oh right, the amount is required. The user said "some," which is vague. Since the amount is missing, I can't
      proceed. But the previous example had the user provide the amount implicitly. Maybe the user intended to include it
      but forgot. In this case, I should inform them that the amount is missing. However, the tools don't allow for prompts,
       so I can't ask. Therefore, I have to proceed with the available data, but since the amount is required, I can't call
      place_order. So, the next step is to inform the user that the amount is missing. But since I can't interact, I have to
       handle it within the tools. Wait, maybe the user's message had the amount in it. Let me check again. The user wrote:
      "initiate a purchase of some Tesla share at $700 per share." The word "some" is vague. Maybe the agent should assume a
       default amount, but that's risky. Alternatively, the agent might have to proceed without it, but the tool requires
      the amount. Therefore, the correct response is to indicate that the amount is missing. But since the tools don't
      support that, I have to proceed with the available data. This is a problem. The user's request is incomplete. So, the
      agent should respond that the amount is required. But in the context of the tools, the agent can't ask. Therefore, the
       agent can't proceed and must inform the user. However, the previous example had the agent proceed without asking, so
      maybe the agent should use a default. But the tools don't have a default for amount. This is a dead end. The user's
      request is incomplete, so the agent can't execute the order. Therefore, the correct response is to inform the user
      that the amount is missing. But since the agent can't ask, it's a limitation. In this case, the agent should proceed
      with the available data, but since the amount is required, the tool call can't be made. Hence, the agent should
      respond that the amount is missing.
</think>

The current price of Tesla (TSLA) is $667.92, which is below the specified price of $700. To proceed, please provide the
      number of shares you'd like to purchase.
<error>
Missing required parameter: "amount" for placing an order.
</error>
assistant
<think>
Okay, the user wants to buy 100 shares of Tesla at $700 per share. Let me check the steps again.
First, they mentioned the price of $700, but the current stock price is $667.92. The user might be referring to a limit
      order at $700, but the stock is currently below that. Alternatively, they might want a market order at the current
      price. However, the user specified $700, so I should consider that.
Wait, the user said "initiate a purchase of some Tesla share at $700 per share." So they want to buy at $700. But the
      current price is lower. Maybe they want a limit order to buy at $700 if the price drops. Or perhaps they want to
      proceed with the current price. But the user explicitly stated $700, so I should use that.
Next, the user provided the amount as 100 shares. The required parameters for `place_order` are order_type, symbol, price,
      and amount. The order_type is "Buy" since they want to purchase. The symbol is TSLA, price is $700, and amount is 100.
I need to make sure the price parameter is correctly set to 700. Even though the current price is lower, the user's
      instruction is to execute at $700. So the order should be a limit order. However, the tool's `place_order` might
      handle it as a market order if the price is not valid. Wait, the tool's parameters require the price, so it's likely a
       limit order.
I should proceed with the given parameters. The user might not be aware of the current price, so it's best to follow their
      instruction. Therefore, the correct tool call is `place_order` with order_type "Buy", symbol "TSLA", price 700, and
      amount 100.
</think>
```

<tool_call>
{"name": "place_order", "arguments": {"order_type": "Buy", "symbol": "TSLA", "price": 700, "amount": 100}}
</tool_call>
user
<tool_response>
{"order_id": 12446, "order_type": "Buy", "status": "Pending", "price": 700.0, "amount": 100}
</tool_response>
assistant
<think>

</think>

Your order to buy 100 shares of Tesla (TSLA) at $700.00 has been placed (Order ID: 12446). The order is currently pending execution.
Would you like to:
1. Check the order status periodically?
2. Cancel the order if it doesn't execute within your desired timeframe?
3. Proceed with another transaction?
assistant
<think>
Okay, the user wants to review the details of their most recent order. Let me check the tools available. There's the get_order_details tool which requires an order_id. The user mentioned "most recent order," so I need to find the latest order ID.
Wait, the previous interaction had an order_id of 12446. Since that's the last one placed, I should use that. But to be thorough, maybe check the order history first? The get_order_history tool can list all order IDs. However, the user might not need all of them, just the latest.
Alternatively, if I assume 12446 is the most recent, I can directly call get_order_details with that ID. The user's message doesn't mention any other order, so it's safe to proceed with 12446. Let me structure the tool call accordingly.
</think>

<tool_call>
{"name": "get_order_details", "arguments": {"order_id": 12446}}
</tool_call>
user
<tool_response>
{"id": 12446, "order_type": "Buy", "symbol": "TSLA", "price": 700.0, "amount": 100, "status": "Open"}
</tool_response>
assistant
<think>

</think>

The details of your recent order are as follows:
- **Order ID**: 12446
- **Type**: Buy
- **Stock Symbol**: TSLA (Tesla)
- **Price per Share**: $700.00
- **Quantity**: 100 shares
- **Status**: Open (Pending execution)
Would you like to monitor the order's progress or make any adjustments?
assistant
<think>
Okay, the user wants to cancel their recent order for 100 Tesla shares. Let me check the tools available. There's a cancel_order tool that requires an order_id. From the previous interaction, the order ID was 12446. I need to call cancel_order with that ID. Let me make sure the parameters are correct. The order_id is an integer, and 12446 is valid. I'll structure the tool call accordingly.
</think>

<tool_call>
{"name": "cancel_order", "arguments": {"order_id": 12446}}
</tool_call>
user
<tool_response>
{"order_id": 12446, "status": "Cancelled"}
</tool_response>
assistant
<think>

</think>

Your order (ID: 12446) to buy 100 shares of Tesla (TSLA) has been successfully **cancelled**.
Would you like to:
1. Place a new order at a revised price?
2. Monitor the current price of TSLA?
3. Review your order history?
assistant
<think>
Okay, the user wants a concise overview of their account details after cancelling the order. Let me check which tool to use.
Looking at the available tools, there's `get_account_info` which retrieves account information like ID, balance, and binding card. That's exactly what the user needs here.
I should call `get_account_info` to fetch the latest account details. Since the user is concerned about discrepancies, the response will confirm their account status post-cancellation.
No parameters are needed for this tool, so the call is straightforward. Let me generate the tool call.
</think>

<tool_call>
{"name": "get_account_info", "arguments": {}}

```
</tool_call>
user
<tool_response>
{"account_id": 98765, "balance": 75000.0, "binding_card": 1987654321098765}
</tool_response>
assistant
<think>

</think>

Here is your account overview:
- **Account ID**: 98765
- **Current Balance**: $75,000.00
- **Linked Card**: 1987654321098765
No discrepancies detected. Your account status is up to date. Would you like to proceed with another action?
assistant
<think>
Okay, the user is experiencing an issue where their updated account information, like email and phone number, isn't showing
      up correctly and isn't syncing across services. They mentioned they tried logging out and back in without success. Now
      , they want to file a support ticket with the title 'Account Information Error' and the provided description.
First, I need to use the create_ticket tool. Let me check the parameters required. The tool needs a title, description, and
      priority. The user provided the title and description. The priority isn't specified, so I'll default to 1, which is
      the lowest priority unless indicated otherwise.
Wait, the user didn't mention the priority, so sticking to the default makes sense. I should structure the JSON with the
      title, description, and priority. Let me make sure the description includes all the details the user provided. Also,
      the response should confirm the ticket was created and provide the ID. Alright, that should cover it.
</think>

<tool_call>
{"name": "create_ticket", "arguments": {"title": "Account Information Error", "description": "User-reported issue where
      updated account information, such as email and phone number, is not displaying correctly and is not syncing across
      services despite attempts to log out and back in.", "priority": 1}}
</tool_call>
user
<tool_response>
{"id": 0, "title": "Account Information Error", "description": "User-reported issue where updated account information, such
      as email and phone number, is not displaying correctly and is not syncing across services despite attempts to log out
      and back in.", "status": "Open", "priority": 1, "created_by": "John"}
</tool_response>
assistant
<think>

</think>

A support ticket has been successfully created with the following details:
- **Ticket ID**: 0
- **Title**: Account Information Error
- **Status**: Open
- **Priority**: 1 (Low)
- **Creator**: John
The ticket has been queued for investigation. Please wait for further updates. Let me know if you'd like to check the ticket
       status or provide additional details!

query_success: true
```

Table 11: Case study 4: summarized skill and full query trajectory for file discovery and workflow correction.

```
Summarized skill:
-1 guidance to avoid mistakes:
- Always verify directory existence before using `cd` (e.g., via `ls` or `find`).
- Use `mkdir` with error handling to ensure directories are created before navigation.
- Confirm tool success responses (e.g., check `mkdir` output) before proceeding.

-2 high-level strategy:
- Prioritize validation steps (e.g., check existence) before critical operations (e.g., navigation, file creation).
- Implement error-resilient workflows by chaining tool calls with conditional checks.

Query trajectory:
<think>
Okay, the user wants to find draft and final reports in their current directory and compare their content. Let me think
      about how to approach this.
First, I need to locate the files. The user mentioned "draft" and "final" reports, so I should search for files with those
      names. The `find` tool can help here. Since the current directory is the starting point, I'll set the path to "." and
      the name to "draft" and "final" respectively. Wait, but the `find` tool's name parameter is for a single name. Oh,
      right, the user wants both files, so maybe I need to run two separate find commands or adjust the parameters. Wait,
      looking back at the tool's description, the `find` tool's name parameter can be set to "draft" or "final", but the
      user wants both. Hmm, maybe the tool allows searching for multiple names? Wait, the parameters for `find` are path and
       name, where name defaults to None to return all items. Wait, no, the default for name is "None", which returns all
      items. Wait, the parameters say: "name": {"type": "string", "description": "The name of the file or directory to
      search for. If None, all items are returned."}. So if I set name to "draft" and "final" separately, but the tool only
      allows one name at a time. So I need to call `find` twice: once for "draft" and once for "final". Alternatively, maybe
       the user has files named "draft_report" and "final_report", but the user just said "draft" and "final". So perhaps
      the actual filenames are "draft" and "final".
```

```
Wait, the user says "draft and final report", so maybe the files are named "draft_report" and "final_report". But the user
      might have named them simply "draft" and "final". To be safe, I should search for both names. So first, use ‘find‘
      with name "draft" to get the draft file, then ‘find‘ with name "final" for the final report.
Once I have the file names, I need to compare their contents. The ‘diff‘ tool can compare two files line by line. So after
      finding the files, I’ll use ‘diff‘ with the two filenames.
But first, I need to make sure the files exist. If the ‘find‘ tool returns matches, then proceed. If not, inform the user.
So the steps are:
1. Use ‘find‘ with name "draft" to locate the draft file.
2. Use ‘find‘ with name "final" to locate the final report.
3. If both files are found, use ‘diff‘ to compare them.
4. If either file is missing, inform the user.
Wait, but the ‘find‘ tool’s response is an array of matches. So if the user has multiple files named "draft", but the user
      probably has one. So assuming the user has exactly one draft and one final report, the ‘find‘ commands should return
      those.
So first, call ‘find‘ for "draft", then for "final". Then use ‘diff‘ on the two files.
But the user might have the files in subdirectories, but the ‘find‘ tool searches recursively. However, the user said "
      current directory", so maybe the files are in the current directory. But the ‘find‘ tool’s path defaults to current
      directory.
So the plan is to first find the draft and final files, then compare them.
Now, let’s structure the tool calls. First, find draft:
<tool_call>
{"name": "find", "arguments": {"path": ".", "name": "draft"}}
</tool_call>
Then find final:
<tool_call>
{"name": "find", "arguments": {"path": ".", "name": "final"}}
</tool_call>
Assuming the responses give the file names, then use ‘diff‘:
<tool_call>
{"name": "diff", "arguments": {"file_name1": "draft", "file_name2": "final"}}
</tool_call>
But wait, the ‘diff‘ tool requires the file names to be in the current directory. If the ‘find‘ tool returns paths, but the
      user said "current directory", maybe the files are in the current directory. So the file names are just "draft" and "
      final".
But if the ‘find‘ tool returns matches like ["draft.txt"], then the actual file name is "draft.txt". So the assistant needs
      to extract the correct file names from the ‘find‘ responses.
So the first step is to get the list of draft files, then final files. Then use those names in the ‘diff‘ tool.
But the user’s instruction is to compare the content difference of both, so the assistant needs to first locate the files
      and then compare them.
So the correct sequence is to first find the files, then compare.
Therefore, the first tool call is to find "draft", then find "final", then diff them.
But the user might have multiple files, but the assistant should handle that. However, the user probably has one of each.
So the assistant should proceed with these steps.
</think>

<tool_call>
{"name": "find", "arguments": {"path": ".", "name": "draft"}}
</tool_call>
<tool_call>
{"name": "find", "arguments": {"path": ".", "name": "final"}}
</tool_call>
user
<tool_response>
{"matches": ["./report_draft.txt"]}
</tool_response>
user
<tool_response>
{"matches": ["./report_final.txt"]}
</tool_response>
user
<tool_response>
{"error": "diff: draft or final: No such file or directory"}
</tool_response>
user
<tool_response>
{"matches": ["./report_draft.txt"]}
</tool_response>
user
<tool_response>
{"matches": ["./report_final.txt"]}
</tool_response>
assistant
<think>
Okay, let’s see. The user wants to compare the draft and final reports. I called the ‘find‘ tool twice, once for "draft" and
       once for "final". The responses gave me the files "./report_draft.txt" and "./report_final.txt". Then I tried to use
      the ‘diff‘ tool with those filenames, but got an error saying they don’t exist. Wait, maybe the ‘diff‘ tool is
      expecting just the filenames without the path? Or maybe the ‘find‘ tool’s response includes the path, but the ‘diff‘
      tool needs just the filenames.
Looking back at the ‘diff‘ tool’s parameters: it requires ‘file_name1‘ and ‘file_name2‘, which are the names of the files in
       the current directory. The ‘find‘ tool’s response includes the full path, like "./report_draft.txt". But the ‘diff‘
      tool might not accept the path, only the filename. So I should extract just the filename from the path. For example,
      if the match is "./report_draft.txt", the filename is "report_draft.txt". Similarly for "report_final.txt".
So the error occurred because I used the full path in the ‘diff‘ tool, but it should be just the filename. Let me correct
      that by using the filenames without the "./" prefix. Let me try calling ‘diff‘ again with "report_draft.txt" and "
      report_final.txt" as the arguments.
</think>
```

<tool_call>
{"name": "diff", "arguments": {"file_name1": "report_draft.txt", "file_name2": "report_final.txt"}}
</tool_call>
user
<tool_response>
{"diff_lines": "- Initial draft content for the report.\n+ Finalized content for the report."}
</tool_response>
assistant
<think>
Okay, let me process what's happening here. The user wanted to compare the draft and final reports. Initially, I used the ‘find‘ tool to locate the files, which returned paths like "./report_draft.txt" and "./report_final.txt". Then, when trying to use ‘diff‘, I mistakenly included the full path, leading to an error. After correcting the filenames to just "report_draft.txt" and "report_final.txt", the ‘diff‘ tool worked and returned the differences.
Now, the user's last response shows the ‘diff‘ output with lines indicating changes from the draft to the final version. The assistant needs to present this comparison in a clear way. The key points are the differences found: the draft had "Initial draft content..." and the final has "Finalized content...". The user probably wants to see these changes highlighted. I should format the response to show the differences neatly, maybe using markers like "- " for deletions and "+ " for additions. Also, confirm that the comparison was successful and summarize the main changes. Make sure the response is easy to read and addresses the user's request effectively.
</think>

The comparison between the draft and final reports shows the following differences:
- **Draft content**:
  ‘- Initial draft content for the report.‘
- **Final content**:
  ‘+ Finalized content for the report.‘
This indicates the final report has been updated with a more polished version of the content. Let me know if you need further analysis!
assistant
<think>
Okay, let's see. The user wants to transfer 'temp_notes.txt' into the 'archives' directory and then rename it to 'notes_2024.txt'.
First, I need to check if the 'archives' directory exists. If it doesn't, I should create it using the 'mkdir' command. But wait, the user didn't mention creating the directory, so maybe they assume it exists. However, to be safe, I should verify its existence. If it's not there, I'll need to create it first.
Next, moving the file. The 'mv' command can move a file from one directory to another. So, 'mv temp_notes.txt archives/' would move it into the archives directory. But I need to make sure that the current directory actually has 'temp_notes.txt'. If the file isn't there, the move will fail.
After moving, renaming the file. The 'mv' command can also rename files. So, 'mv archives/temp_notes.txt archives/notes_2024.txt' would rename it. Alternatively, if the file is already in the archives directory, just renaming it with 'mv temp_notes.txt notes_2024.txt' would work if the current directory is archives.
Wait, the user said to first transfer into 'archives' and then rename. So the steps are: move 'temp_notes.txt' to 'archives', then rename it there. So the first 'mv' is to archives, then the second 'mv' is within archives.
But I need to check if the user is in the correct directory. If the current working directory is not the same as where 'temp_notes.txt' is located, the first 'mv' might not work. However, the user's instruction doesn't mention changing directories, so I'll assume 'temp_notes.txt' is in the current directory.
So the plan is:
1. Move 'temp_notes.txt' to 'archives' directory.
2. Rename 'temp_notes.txt' to 'notes_2024.txt' in the 'archives' directory.
But I should also handle possible errors. For example, if 'archives' doesn't exist, the move will fail. So maybe first check if 'archives' exists using 'ls' or 'find', and if not, create it with 'mkdir'.
Alternatively, the 'mv' command might automatically create the destination directory if it doesn't exist, but I think that's not the case. So creating 'archives' first is necessary.
Wait, the 'mv' command doesn't create directories. So if 'archives' doesn't exist, moving the file there will fail. Therefore, the correct approach is to first create the 'archives' directory if it's not present, then move the file into it, and then rename it.
So the steps would be:
1. Check if 'archives' exists. If not, create it with 'mkdir archives'.
2. Move 'temp_notes.txt' to 'archives' using 'mv temp_notes.txt archives/'.
3. Rename the file in 'archives' to 'notes_2024.txt' with 'mv archives/temp_notes.txt archives/notes_2024.txt'.
But the user might not have the 'archives' directory. So I need to handle that. However, the user's instruction doesn't mention creating the directory, so maybe they expect it to exist. But to be thorough, I should check and create it if necessary.
Alternatively, maybe the user is in the 'archives' directory already. But the problem states to transfer into 'archives', implying that the current directory is not 'archives'.
So, the correct sequence is to first ensure 'archives' exists, then move the file, then rename.
But the user's previous interactions didn't mention the 'archives' directory, so perhaps it's a new directory. Therefore, the assistant should first create it, then perform the move and rename.
But the user's instruction is to transfer into 'archives' and then rename. So the steps are:
1. Move 'temp_notes.txt' to 'archives' directory.
2. Rename the moved file to 'notes_2024.txt'.
But if 'archives' doesn't exist, the first step fails. So the assistant should first check for 'archives' and create it if needed.
Therefore, the correct tool calls would be:
- First, check if 'archives' exists using 'ls' or 'find'. If not, use 'mkdir archives'.
- Then, move 'temp_notes.txt' to 'archives'.
- Then, rename the file in 'archives' to 'notes_2024.txt'.
But the user's previous tool calls didn't include checking for directories, so maybe the assistant should proceed with moving and renaming, assuming 'archives' exists. However, if it doesn't, the move will fail.
Alternatively, the assistant can attempt to move, and if it fails, create the directory and retry. But in the context of the current tools, the assistant can only call the available functions.
So, the assistant should first call 'mv' to move the file. If that fails, then call 'mkdir' and try again. But since the user's previous steps didn't handle errors, maybe the assistant should proceed with the move and rename, assuming 'archives' exists.
But the user's instruction is to transfer into 'archives', so the assistant should first ensure 'archives' exists. Therefore, the correct approach is to first check for 'archives' and create it if necessary, then move and rename.

```
So, the first tool call should be 'mkdir archives' if 'archives' doesn't exist. But how to check that? The 'find' tool can
      be used to check if 'archives' exists.
Wait, the 'find' tool can search for 'archives' in the current directory. So, the assistant can first call 'find' with path
      '.' and name 'archives' to check if it exists. If not, then call 'mkdir archives'.
But the user's previous steps didn't use 'find' for this purpose. However, the assistant should handle this to avoid errors.
So, the steps would be:
1. Use 'find' to check if 'archives' exists. If not, create it with 'mkdir'.
2. Move 'temp_notes.txt' to 'archives'.
3. Rename 'temp_notes.txt' to 'notes_2024.txt' in 'archives'.
But the user's instruction is to transfer into 'archives' and then rename. So the assistant should first move the file, then
       rename.
But if 'archives' doesn't exist, the move fails. Therefore, the correct sequence is to first create 'archives' if needed,
      then move and rename.
So, the assistant should first check for 'archives' using 'find', then proceed.
But in the current context, the assistant might not have checked for 'archives' before. Therefore, the correct approach is
      to first create 'archives' if it doesn't exist, then move the file, then rename.
But the user's instruction doesn't mention creating 'archives', so maybe the assistant should proceed with moving and
      renaming, assuming 'archives' exists. However, if it doesn't, the move will fail.
In the absence of prior information, the assistant should first check for 'archives' and create it if necessary.
Therefore, the first tool call should be 'find' to check for 'archives'. If it's not found, then 'mkdir archives'. Then
      proceed with 'mv' and 'mv' again for renaming.
But the user's previous steps didn't include this, so perhaps the assistant should proceed with the move and rename,
      assuming 'archives' exists.
Alternatively, the assistant can attempt to move, and if it fails, create the directory and retry.
But given the tools available, the assistant can call 'mv' first. If it fails, then call 'mkdir' and try again.
But in the current context, the assistant should proceed with the steps as per the user's instruction, which is to transfer
      into 'archives' and rename. So the first step is to move the file into 'archives', then rename it.
Therefore, the correct tool calls are:
1. 'mv temp_notes.txt archives/'
2. 'mv archives/temp_notes.txt archives/notes_2024.txt'
But if 'archives' doesn't exist, the first 'mv' will fail. So the assistant should first check for 'archives' and create it
      if needed.
But the user's instruction doesn't mention creating 'archives', so the assistant should proceed with the move and rename,
      assuming 'archives' exists.
In conclusion, the assistant should first move the file to 'archives' and then rename it. If 'archives' doesn't exist, the
      move will fail, but the user's instruction doesn't mention handling that case. So the assistant proceeds with the
      given steps.
</think>

<tool_call>
{"name": "mv", "arguments": {"source": "temp_notes.txt", "destination": "archives/"}}
</tool_call>
<tool_call>
{"name": "mv", "arguments": {"source": "archives/temp_notes.txt", "destination": "archives/notes_2024.txt"}}
</tool_call>
user
<tool_response>
{"error": "mv: path not allowed in destination. Provide only a file or directory name."}
</tool_response>
user
<tool_response>
{"error": "mv: cannot move 'archives/temp_notes.txt': No such file or directory"}
</tool_response>
assistant
<think>
Okay, let's figure out what's going wrong here. The user wants to move 'temp_notes.txt' into the 'archives' directory and
      then rename it to 'notes_2024.txt'.
First, the initial attempt used 'mv temp_notes.txt archives/' which failed because the 'archives' directory might not exist.
       The error message from the first tool call says "path not allowed in destination", which suggests that the
      destination path was invalid. Then, the second attempt tried to move 'archives/temp_notes.txt' to 'archives/notes_2024.
      txt', but the error was "No such file or directory", meaning the source file wasn't found.
So, the problem is that the 'archives' directory doesn't exist yet. The first 'mv' command tried to move the file into '
      archives', but since the directory doesn't exist, the move failed. Then, the second 'mv' tried to rename the file
      within 'archives', but the source file wasn't there because the first step failed.
To fix this, the assistant needs to first create the 'archives' directory using 'mkdir'. Then, move the file into it, and
      finally rename it. Let's check if 'mkdir' is available. Yes, the 'mkdir' tool is listed in the tools. So the correct
      sequence should be:
1. Create the 'archives' directory with 'mkdir archives'.
2. Move 'temp_notes.txt' into 'archives' with 'mv temp_notes.txt archives/'.
3. Rename the file in 'archives' to 'notes_2024.txt' with 'mv archives/temp_notes.txt archives/notes_2024.txt'.
This way, the directory exists before moving the file, and the renaming step can proceed correctly.
</think>

<tool_call>
{"name": "mkdir", "arguments": {"dir_name": "archives"}}
</tool_call>
<tool_call>
{"name": "mv", "arguments": {"source": "temp_notes.txt", "destination": "archives/"}}
</tool_call>
<tool_call>
{"name": "mv", "arguments": {"source": "archives/temp_notes.txt", "destination": "archives/notes_2024.txt"}}
</tool_call>
user
<tool_response>
{"error": "mkdir: cannot create directory 'archives': File exists"}
</tool_response>
user
```

<tool_response>
{"error": "mv: path not allowed in destination. Provide only a file or directory name."}
</tool_response>
user
<tool_response>
{"error": "mv: cannot move 'archives/temp_notes.txt': No such file or directory"}
</tool_response>
assistant
<think>
Okay, let's see. The user wants to move 'temp_notes.txt' into the 'archives' directory and then rename it to 'notes_2024.txt'.
First, the assistant tried to create the 'archives' directory with 'mkdir archives', but got an error saying it already exists. That makes sense because the user might have already created it, or maybe the system already has it.
Next, the assistant tried to move 'temp_notes.txt' to 'archives/'. The error here was "path not allowed in destination", which probably means the destination path was invalid. Wait, but 'archives' is a directory, so moving into it should be okay. Maybe the 'mv' command requires the destination to be a directory name without a path. Oh, right! The 'mv' tool's destination parameter can't include a path; it just needs the directory name. So if the current directory is not 'archives', the destination should be 'archives/'.
Wait, but the user's previous attempt to move 'temp_notes.txt' to 'archives/' failed because the destination path was invalid. Maybe the 'mv' command expects the destination to be a directory name without a path. So the correct command would be 'mv temp_notes.txt archives' (without the slash), or 'mv temp_notes.txt archives/' if the slash is allowed. But the error message says "path not allowed in destination", so the slash might be causing the issue.
Then, the assistant tried to move 'archives/temp_notes.txt' to 'archives/notes_2024.txt', but got an error that the source file doesn't exist. That's because the first move didn't succeed, so the file wasn't in 'archives' yet.
So the problem is that the 'mv' command is being used incorrectly. The correct way to move a file into a directory is to specify the directory as the destination without a path. For example, 'mv temp_notes.txt archives' would move the file into the 'archives' directory. Then, renaming within 'archives' would be 'mv archives/temp_notes.txt archives/notes_2024.txt'.
But the previous attempt to move 'temp_notes.txt' to 'archives/' failed because of the path notation. Maybe the 'mv' tool doesn't accept the slash in the destination. So the correct command is 'mv temp_notes.txt archives' (without the slash).
Also, the 'mkdir' command tried to create 'archives' but it already exists, so that's why the error occurred. So the assistant should check if 'archives' exists before trying to create it.
So the correct steps are:
1. Check if 'archives' exists. If not, create it with 'mkdir archives'.
2. Move 'temp_notes.txt' into 'archives' using 'mv temp_notes.txt archives'.
3. Rename the file in 'archives' to 'notes_2024.txt' with 'mv archives/temp_notes.txt archives/notes_2024.txt'.
But the user's previous attempts didn't handle the path correctly. The 'mv' command might require the destination directory name without a path. So the correct destination for the first move is 'archives', not 'archives/'.
Therefore, the assistant should adjust the 'mv' commands to use the directory name without the slash and ensure the directory exists before moving the file.
</think>

<tool_call>
{"name": "mv", "arguments": {"source": "temp_notes.txt", "destination": "archives"}}
</tool_call>
<tool_call>
{"name": "mv", "arguments": {"source": "archives/temp_notes.txt", "destination": "archives/notes_2024.txt"}}
</tool_call>
user
<tool_response>
{"result": "'temp_notes.txt' moved to 'archives/temp_notes.txt'"}
</tool_response>
user
<tool_response>
{"error": "mv: cannot move 'archives/temp_notes.txt': No such file or directory"}
</tool_response>
assistant
<think>
Okay, let's figure out what's going on here. The user wants to move 'temp_notes.txt' into the 'archives' directory and then rename it to 'notes_2024.txt'.
First, the assistant tried to move the file using 'mv temp_notes.txt archives', which worked and moved the file to 'archives/temp_notes.txt'. Then, the next step was to rename it to 'notes_2024.txt' with 'mv archives/temp_notes.txt archives/notes_2024.txt'. But the error says the source file doesn't exist.
Wait, maybe the problem is that the 'mv' command in the second step is trying to move 'archives/temp_notes.txt' to 'archives/notes_2024.txt', but the source file isn't found. That could happen if the first move didn't actually place the file in the 'archives' directory.
But the first tool call's response said "'temp_notes.txt' moved to 'archives/temp_notes.txt'". So the file should be there. Maybe the issue is with the path in the second 'mv' command. The destination is 'archives/notes_2024.txt', but perhaps the correct path is just 'notes_2024.txt' if the current directory is 'archives'.
Wait, the user's current directory might not be 'archives'. The 'mv' command in the second step is trying to move from 'archives/temp_notes.txt' to 'archives/notes_2024.txt', which is correct if the current directory is 'archives'. But if the current directory is not 'archives', the path would be incorrect.
Alternatively, maybe the 'mv' command doesn't require the full path. If the current directory is the same as where 'archives' is located, then 'mv archives/temp_notes.txt archives/notes_2024.txt' should work. But if the current directory is different, the path might be wrong.
Wait, the user's previous steps didn't change the current directory. So the current directory is likely the same as where 'temp_notes.txt' was originally. Therefore, the correct command to rename the file in 'archives' would be 'mv archives/temp_notes.txt archives/notes_2024.txt'. But the error says the source file doesn't exist. That suggests that 'archives/temp_notes.txt' isn't found.
Hmm, maybe the first 'mv' command didn't actually move the file. The response from the first 'mv' was "'temp_notes.txt' moved to 'archives/temp_notes.txt'", which seems successful. But perhaps the system is case-sensitive, and the file was moved but the path is different. Or maybe the 'mv' command in the first step didn't work as expected.
Alternatively, the second 'mv' command might be using the wrong path. If the current directory is not 'archives', then 'archives/temp_notes.txt' would be a relative path from the current directory. If the current directory is the same as where 'archives' is located, then it's correct. But if the current directory is different, the path would be incorrect.

```
Wait, the user hasn't changed directories, so the current directory is the same as where 'temp_notes.txt' was. Therefore, '
      archives/temp_notes.txt' should be correct. But the error persists.
Maybe the issue is that the 'mv' command in the second step is trying to move the file to 'archives/notes_2024.txt', but the
       destination path is incorrect. The correct destination should be 'notes_2024.txt' if the current directory is '
      archives', or 'archives/notes_2024.txt' if the current directory is elsewhere.
Alternatively, the 'mv' command might not support the destination path with a slash. Maybe the correct command is 'mv
      archives/temp_notes.txt notes_2024.txt' if the current directory is 'archives', or 'mv archives/temp_notes.txt
      archives/notes_2024.txt' if the current directory is the same as 'archives'.
But the error indicates that 'archives/temp_notes.txt' doesn't exist. That suggests that the first 'mv' didn't actually move
       the file. Maybe the first 'mv' command failed, but the response said it succeeded. This is confusing.
Perhaps the correct approach is to first check if 'archives/temp_notes.txt' exists using the 'ls' command. If it does, then
      proceed with renaming. If not, then the first move failed.
But the user's tools don't include an 'ls' command, but there's a 'find' tool. So the assistant could use 'find' to check
      for the file.
Alternatively, the assistant could try moving the file again, ensuring the correct path.
In summary, the problem is that the second 'mv' command can't find 'archives/temp_notes.txt', which suggests the first move
      didn't work. The assistant should verify the existence of the file before attempting to rename it. If the file isn't
      there, the first move failed, and the assistant should troubleshoot that step.
</think>

<tool_call>
{"name": "find", "arguments": {"path": ".", "name": "temp_notes.txt"}}
</tool_call>
user
<tool_response>
{"matches": ["./archives/temp_notes.txt"]}
</tool_response>
assistant
<think>
Okay, let's see. The user wants to move 'temp_notes.txt' into the 'archives' directory and then rename it to 'notes_2024.txt
      '.
First, the assistant tried to move the file using 'mv temp_notes.txt archives', which worked and moved the file to 'archives
      /temp_notes.txt'. Then, the next step was to rename it with 'mv archives/temp_notes.txt archives/notes_2024.txt', but
      that failed because the source file wasn't found.
Wait, but the 'find' tool response shows that 'temp_notes.txt' is in 'archives/temp_notes.txt'. So the file exists in the
      archives directory. Why is the second 'mv' command failing? Maybe the path in the 'mv' command is incorrect.
The user's current directory might not be 'archives', so when they try to move 'archives/temp_notes.txt' to 'archives/
      notes_2024.txt', the source path is correct, but the destination might need to be just 'notes_2024.txt' if the current
       directory is 'archives'. Alternatively, the destination path might be incorrect.
Wait, the 'mv' command in the second step is trying to move from 'archives/temp_notes.txt' to 'archives/notes_2024.txt'. If
      the current directory is not 'archives', then the destination path 'archives/notes_2024.txt' would be relative to the
      current directory. But if the current directory is the same as where 'archives' is located, then it's correct.
But the error says the source file doesn't exist. That's confusing because the 'find' tool shows it's there. Maybe the 'mv'
      command is case-sensitive, and the file was moved but the path is different. Or maybe the 'mv' command in the second
      step is using the wrong path.
Alternatively, the assistant should try renaming the file without specifying the full path. If the current directory is '
      archives', then 'mv temp_notes.txt notes_2024.txt' would work. But if the current directory isn't 'archives', then the
       path is incorrect.
So the correct approach is to first change the current directory to 'archives' using 'cd archives', then rename the file.
      But the user hasn't changed directories yet.
Therefore, the assistant should first navigate to the 'archives' directory with 'cd archives', then rename the file. That
      way, the 'mv' command can be done without specifying the full path.
So the steps should be:
1. Change directory to 'archives' using 'cd archives'.
2. Rename 'temp_notes.txt' to 'notes_2024.txt' with 'mv temp_notes.txt notes_2024.txt'.
This should resolve the issue because the current directory is now 'archives', so the 'mv' command doesn't need the full
      path.
</think>

<tool_call>
{"name": "cd", "arguments": {"folder": "archives"}}
</tool_call>
<tool_call>
{"name": "mv", "arguments": {"source": "temp_notes.txt", "destination": "notes_2024.txt"}}
</tool_call>
user
<tool_response>
{"current_working_directory": "archives"}
</tool_response>
user
<tool_response>
{"result": "'temp_notes.txt' moved to 'notes_2024.txt'"}
</tool_response>
assistant
<think>
Okay, let me recap what's happening here. The user wanted to move 'temp_notes.txt' into the 'archives' directory and then
      rename it to 'notes_2024.txt'.
First, the assistant tried moving the file using 'mv temp_notes.txt archives', which worked and placed the file in 'archives
      /temp_notes.txt'. Then, the next step was to rename it, but the command 'mv archives/temp_notes.txt archives/
      notes_2024.txt' failed because the source file wasn't found.
After checking with the 'find' tool, it was confirmed that 'temp_notes.txt' exists in 'archives/temp_notes.txt'. The issue
      was likely due to the current working directory not being 'archives' when attempting the rename.
So, the assistant changed the directory to 'archives' using 'cd archives', which made the current directory 'archives'. Then,
       the 'mv' command was used again, this time just specifying the source as 'temp_notes.txt' and the destination as '
      notes_2024.txt'. Since the current directory is now 'archives', the command correctly renamed the file to 'notes_2024.
      txt' without needing the full path.
```

```
The final response from the tool indicates the move was successful, so the task is complete. The user now has 'notes_2024.
      txt' in the 'archives' directory.
</think>

The file ‘temp_notes.txt‘ has been successfully moved into the ‘archives‘ directory and renamed to ‘notes_2024.txt‘.
**Final result:**
- Original file: ‘temp_notes.txt‘
- New location: ‘archives/notes_2024.txt‘
Let me know if you need further assistance!
assistant
<think>
Okay, the user is trying to retrieve details of a support ticket they documented. They mentioned the ticket number was
      987654 but aren't sure. Let me check the available tools.
First, the user is using the ticketing system, so the relevant tools are get_ticket, get_user_tickets, etc. Since they
      provided a specific ticket number, the get_ticket tool is the right choice. The parameters require ticket_id, which
      they provided as 987654. I should call get_ticket with that ID. Wait, but the user said they "recall the ticket number
       was 987654, though the precise number eludes me." So maybe they're not certain about the exact number. However, they
      mentioned "the ticket number was 987654," which might be a typo or a specific number they think. Since the tools don't
       have a search function for tickets, the only way is to use get_ticket with the provided ID. If the number is wrong,
      the response will indicate an error, but that's the best approach here. So, proceed with get_ticket and ticket_id
      987654.
</think>

<tool_call>
{"name": "get_ticket", "arguments": {"ticket_id": 987654}}
</tool_call>
user
<tool_response>
{"id": 987654, "status": "open", "description": "Issue with workstation not booting properly.", "resolution": ""}
</tool_response>
assistant
<think>

</think>

The details of ticket #987654 are as follows:
**Ticket ID:** 987654
**Status:** Open
**Description:** "Issue with workstation not booting properly."
**Resolution:** (Not provided yet)
Would you like to close this ticket or update its status?
assistant
<think>
Okay, the user wants to resolve ticket #987654 because they've already fixed the issue themselves. The resolution summary
      they provided is "Fixed through manual troubleshooting techniques."
First, I need to check if the ticket exists. The previous interaction showed that get_ticket(987654) returned the ticket
      details, so it's valid.
Next, the user is asking to resolve the ticket. The tool 'resolve_ticket' requires the ticket_id and resolution. The user
      provided the resolution summary, so I should use that.
Wait, the ticket's current status is "open" from the previous response. Resolving it would change the status to "resolved"
      or similar. The tool 'resolve_ticket' might handle updating the status automatically.
I should call resolve_ticket with ticket_id 987654 and the given resolution. Then, confirm the resolution and maybe check
      the updated status.
Also, the user might want to close the ticket after resolving. But the tools have 'resolve_ticket' and 'close_ticket' as
      separate functions. Since the user mentioned they've addressed the core problem, resolving it with the provided
      summary should suffice.
So, the correct step is to use resolve_ticket with the specified parameters. Then, inform the user that the ticket is
      resolved with their summary.
</think>

<tool_call>
{"name": "resolve_ticket", "arguments": {"ticket_id": 987654, "resolution": "Fixed through manual troubleshooting techniques
      ."}}
</tool_call>
user
<tool_response>
{"status": "Ticket 987654 has been resolved successfully."}
</tool_response>
assistant
<think>
Okay, let me process this. The user wanted to resolve ticket 987654 with the summary "Fixed through manual troubleshooting
      techniques." I called the resolve_ticket tool with those parameters. The response came back as successful. Now, I need
       to confirm the resolution to the user.
First, I should mention that the ticket was resolved as requested. Include the ticket number and the resolution summary they
       provided. Maybe add a note that the ticket status is now closed or resolved. Also, offer further assistance in case
      they need anything else. Keep it concise and friendly. Let me check if there's any other info needed, but since the
      user only asked to resolve it, this should cover it.
</think>

Ticket **#987654** has been resolved successfully with the following summary:
**"Fixed through manual troubleshooting techniques."**
The ticket status is now marked as **resolved**. Let me know if you'd like to close it entirely or need further assistance!

query_success: true
```