Long-Horizon Plan Execution in Large Tool Spaces through Entropy-Guided Branching

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have significantly advanced tool-augmented agents, enabling autonomous reasoning via API interactions. However, executing multi-step tasks within massive tool libraries remains challenging due to two critical bottlenecks: (1) the absence of rigorous, plan-level evaluation frameworks and (2) the computational demand of exploring vast decision spaces, stemming from large toolsets and long-horizon planning. To bridge these gaps, we first introduce SLATE (**S**ynthetic **L**arge-scale **A**PI **T**oolkit for **E**-commerce), a large-scale context-aware benchmark designed for the automated assessment of tool-integrated agents. Unlike static metrics, SLATE evaluates grounded execution trajectories and reveals that current agents struggle with self-correction and search efficiency. Motivated by these findings, we next propose Entropy-Guided Branching (EGB), an uncertainty-aware search algorithm that dynamically expands decision branches where predictive entropy is high. EGB improves task success by allocating search to uncertain tool-selection steps rather than exploring uniformly across the full decision space. Extensive experiments on SLATE demonstrate that our dual contribution is a step toward more reliable and scalable LLM agents in tool-rich environments.[1]

1 Introduction

Large Language Models (LLMs) have enabled agents that can reason, interact with external tools, and automate domain-specific workflows (Wang et al., 2024; Guo et al., 2024; Xi et al., 2025; Yao et al., 2023b; Shinn et al., 2023). These tool-augmented LLM agents demonstrate significant potential for automating complex workflows across diverse domains, including e-commerce (Yao et al., 2022; Yang et al., 2024), software development (Hong et al., 2023), scientific discovery (Boiko et al., 2023), and embodied AI (Ahn et al., 2024). In many real-world deployments, however, the central challenge is not to generate an entire plan from scratch. Business and operational domains often already provide standard operating procedures (SOPs) that specify the intended sequence of actions for human operators to follow. We therefore study a plan-given execution setting: an agent is provided with a procedure-level plan, but must decide which concrete tool or API to invoke at each step in order to complete the task.

This setting is practically important because executing a known plan can still be difficult in realistic tool environments. A business entity may expose a large private library of domain-specific APIs, while a single case may require a long sequence of dependent operations. Even when the next procedural step is visible, there may be no explicit annotation of which API realizes that step, what arguments should be passed, or how earlier tool outputs should condition later decisions. The agent must therefore explore a large action space across a long horizon, where a local tool-selection error can cascade into downstream failure.

The feedback structure further distinguishes this problem from many existing tool-use evaluations. In realistic workflow automation, success is often verified only after the entire procedure has finished, based on the final state of the case or execution result. A human or system is unlikely to inspect every intermediate action and assign dense step-level rewards; instead, the outcome is judged by whether the final state satisfies deterministic business or task constraints. This motivates evaluation based on verifiable final-state execution rather than subjective judgments of whether a trajectory "looks" correct.

Existing benchmarks do not fully capture this combination of plan-given execution, large tool spaces, long horizons, and sparse final rewards. Many benchmarks use limited toolsets that do

[1] All code is released in the supplementary materials.

not reflect the scale or diversity of real applications (Yao et al., 2022, 2024a; Shridhar et al., 2020). Others include large tool libraries but focus primarily on single-step tool invocation, omitting the sequential dependencies needed for procedure execution (Tang et al., 2023; Patil et al., 2024; Li et al., 2023a). Attempts at plan-level evaluation often rely on LLM-as-a-judge assessments (Qin et al., 2023; Huang et al., 2024), which can penalize valid alternative trajectories, such as redundant but harmless calls or different execution orders that reach the same final state. These limitations leave a gap for objective, plan-level benchmarks with deterministic final-state verification.

The same setting also raises an algorithmic question: how should an agent efficiently explore tool choices when only final execution success is observed? We evaluate representative ReAct, reflection, and search-based methods under this sparse-reward setting. Our experiments show that Reflexion-style methods improve over a single execution path but remain inefficient at attributing a final failure to a specific earlier step, while broad tree-search methods such as LATS/MCTS become expensive as both the number of candidate tools and the number of procedure steps grow (Yao et al., 2023b; Shinn et al., 2023; Zhou et al., 2023; Murthy et al., 2023; Wei et al., 2022; Yao et al., 2023a; Hao et al., 2023). These observations motivate a more targeted exploration strategy that identifies which steps are most likely to require reconsideration.

We address these gaps from both the evaluation and methodology perspectives. First, we present SLATE (**S**ynthetic **L**arge-scale **A**PI **T**oolkit for **E**-commerce), a large-scale, context-aware benchmark for SOP-like, plan-given tool execution. Table 1 outlines how SLATE fills the aforementioned gaps in existing benchmarks. Each SLATE instance consists of a user query, an executable plan, and an associated toolset, supported by context-grounded simulation outputs for valid invocations and explicit default outputs for invalid ones. This design enables automated end-to-end evaluation with deterministic final-state verification while still accommodating diverse valid trajectories. Leveraging SLATE, we conduct a systematic study of representative agent architectures, revealing three key findings: (1) reflection-based methods improve over single-pass execution but remain limited at localizing and correcting failure-inducing steps under sparse plan-level feedback; (2) predictive entropy is strongly correlated with tool-selection errors, making it an effective signal for prioritizing where to branch; and (3) broad tree-search methods such as LATS exhibit an unfavorable utility-to-computation trade-off under matched computational budgets in long-horizon, large-tool-space settings. Motivated by this empirical diagnosis, we propose Entropy-Guided Branching (EGB, see Fig. 1), an uncertainty-aware exploration algorithm that branches at high-entropy tool-selection steps while following a greedy path when the model is confident. By allocating computation to likely failure points rather than searching uniformly across all steps, EGB improves end-to-end success while reducing unnecessary exploration overhead. Additional related work on tool-use benchmarks and reasoning/search methods is discussed in Appendix D.

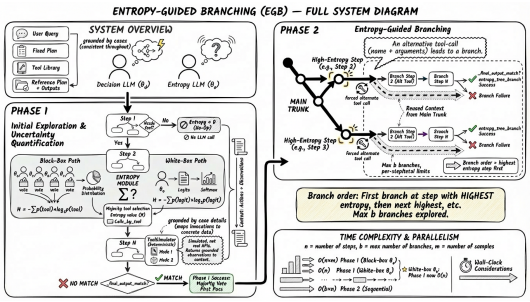

Figure 1: **Overview of the Entropy-Guided Branching (EGB) framework.** Phase 1 iterates over a fixed plan, using an entropy estimation module (black-box sampling or white-box logit-based) to quantify tool selection uncertainty at each step and execute the majority-voted tool. If the final output does not match the reference, Phase 2 branches at high-entropy steps in descending order, substituting alternative tools from the Phase 1 vote distribution and re-executing downstream steps with updated observations. Tool execution results are produced by a deterministic simulator to enable exact-match success evaluation.

2 SLATE Dataset Construction

SLATE is designed to support rigorous evaluation of tool-augmented agents in settings that combine large tool libraries with long-horizon execution. Concretely, we construct the dataset to satisfy four properties simultaneously: large tool spaces, long-horizon execution, step-wise deterministic outcomes, and grounded plan-level evaluation. We instantiate this design in e-commerce, a domain where realistic tasks naturally require multi-stage reasoning over broad API ecosystems spanning retrieval, inventory, pricing, logistics, and customer support (Yao et al., 2022; Shridhar et al., 2020; Qin

Table 1: Comprehensive Comparison of Agent Tool-Call Datasets.

Benchmark	Large Tool Spaces (1,000+ Tools)	Long-Horizon Execution (10+ Steps)	Step-wise Deterministic Outcomes	Plan-Level Grounded Evaluation
SLATE (Ours)	✓	✓	✓	✓
TRAJECT-Bench (He et al., 2025)	✓	✓	✓	×
Toolathlon (Li et al., 2026)	×	✓	✓	✓
ToolLLM (Qin et al., 2023)	✓	×	✓	×
Gorilla (Patil et al., 2024)	✓	×	✓	×
UltraTool (Huang et al., 2024)	✓	×	×	×
τ-bench (Yao et al., 2024b)	×	✓	×	✓
τ^2-Bench (Barres et al., 2025)	×	✓	✓	✓
ACEBench (Chen et al., 2025)	✓	×	✓	×
HammerBench (Wang et al., 2025)	✓	×	✓	✓
ToolSandbox (Lu et al., 2025)	×	✓	×	✓
MetaTool (Huang et al., 2023)	×	×	×	×
API-Bank (Li et al., 2023b)	✓	×	✓	✓

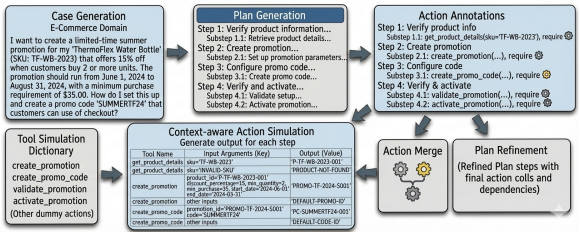

Figure 2: **Illustration of the** SLATE **dataset structure.**

et al., 2023; Qu et al., 2025). Appendix C provides the SLATE construction prompts and examples.

Each SLATE instance contains a user query, a hierarchical solution plan, grounded tool annotations, and simulated execution outputs for intermediate steps. This structure supports both step-wise verification and end-to-end plan-level evaluation at scale. Our construction pipeline follows four stages: query collection, hierarchical plan generation, tool grounding, and context-aware simulation, followed by manual post-processing for quality control. Fig. 2 illustrates the overall structure.

2.1 Benchmark Positioning and Evaluation Properties

Existing tool-use benchmarks typically capture only part of the challenge. Small-scale settings such as τ-BENCH (Yao et al., 2024a) and ALFWORLD (Shridhar et al., 2020) support controlled evaluation but do not reflect large tool spaces, while benchmarks with broader tool libraries often emphasize tool retrieval or step-level correctness rather than grounded end-to-end execution (Qin et al., 2023; Huang et al., 2023; Tang et al., 2023; Patil et al., 2024; Li et al., 2023a; Huang et al., 2024). Table 1 organizes prior datasets along four complementary properties: *Large Tool Spaces*, *Long-Horizon Execution*, *Step-wise Deterministic Outcomes*, and *Plan-Level Grounded Evaluation*. SLATE is designed to satisfy all four simultaneously, enabling objective evaluation even when multiple execution trajectories can lead to the same correct final state.

2.2 Query Collection and Dataset Definition

We begin by collecting complex user queries in e-commerce. To balance diversity and topical coherence, we work with domain experts to define 12 representative task categories: *Product Management, Inventory Management, Order Processing, Shipping & Fulfillment, Pricing & Promotions, Subscription Management, Customer Service, Returns & Refunds, Analytics & Reporting, Catalog Management, Review Management*, and *Miscellaneous*. For each instance, we randomly sample a category and use Claude 3.7 to generate and progressively complicate the query.

Formally, let $\mathcal{D} = (\mathcal{Q}, \mathcal{P}, \mathcal{T}, \mathcal{S})$ denote the SLATE dataset, where $\mathcal{Q}$ is a set of complex user queries, $\mathcal{P}$ is the corresponding set of structured hierarchical plans (Huang et al., 2024), $\mathcal{T}$ is a library of tools, and $\mathcal{S}$ is a context-aware simulator. Each plan consists of high-level steps and substeps, where a substep may invoke a tool or indicate `No Tool Required`. Each tool follows a function-call schema with runtime arguments. Fig. 3 shows the distribution of tools across categories.

2.3 Hierarchical Plan Generation and Tool Grounding

Given a complex query $q \in \mathcal{Q}$, we construct a hierarchical execution plan $p \in \mathcal{P}$ together with a coherent toolkit $\mathcal{T}$. We first generate multi-step plans independently of any tool library so that the decomposition follows task structure rather than tool names. This top-down formulation yields interpretable high-level goals and executable substeps while reducing tool-specific bias during planning.

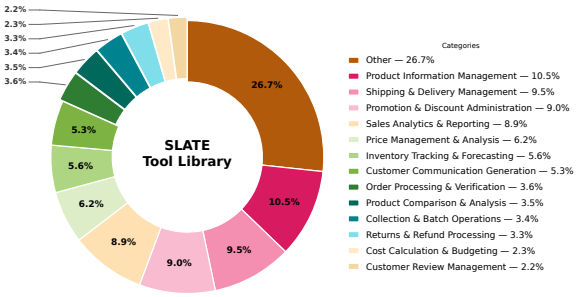

Figure 3: **SLATE tool-library category distribution.**

For tool-requiring substeps, we define normalized tool schemas with names, arguments, functionality, and output formats. Tool grounding follows three principles. First, tools should not trivially mirror substep descriptions, since that would reduce the difficulty of tool selection. Second, later tool arguments should be derivable from earlier outputs or the original query context to preserve execution dependencies. Third, to promote reuse and reduce redundancy, we embed generated tool names and descriptions using ModernBERT (Warner et al., 2025), retrieve the top-50 semantically related existing tools for each new query, and encourage the LLM to reuse them when possible. We then normalize semantically similar tools within functional categories and update plans accordingly.

2.4 Context-Aware Tool Simulation

Implementing every tool directly in a large interactive environment is impractical, so SLATE uses a context-aware simulator to provide execution feedback. During dataset generation, each plan is annotated with grounded tool calls and their argument dependencies. The simulator stores coherent outcomes in a lookup table of `(tool_name, arguments)` $\rightarrow$ `outcome`, together with default responses for mismatched invocations. To handle runtime variation, a semantic equivalence module matches agent-generated calls to stored entries beyond simple string equality.

At evaluation time, when an invocation matches the grounded resolution, the simulator returns a contextually coherent outcome; otherwise, it emits a failure signal. This design enables automated end-to-end plan evaluation while preserving the closed-loop interaction required by acting-based agents. We remark that SLATE and its simulation framework may be of independent interest to the community for research on long-horizon planning and the development of more resilient tool-augmented agents.

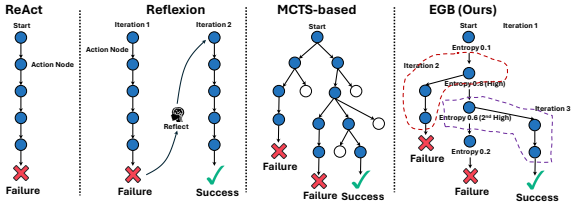

Figure 4: Comparison of different paradigms.

3 Method

We propose *Entropy-Guided Branching (EGB)* (illustrated in Fig. 4), an uncertainty-aware acting strategy designed to navigate expansive tool spaces through iterative trajectory refinement. Unlike pure reasoning-based approaches, EGB operates within a MDP framework, where the agent's policy $\pi(a_{i,j}|H_{i,j})$ is continuously informed by environmental feedback $o_{i,j}$ from a tool simulator $\mathcal{S}$. Detailed pseudocode and complexity analysis for EGB are provided in Appendix B.

3.1 Problem Definition and Agent Tool-use Paradigms

We formulate plan-guided tool utilization as a Markov Decision Process (MDP) (Bellman and Dreyfus, 2015). Given a query $q \in \mathcal{Q}$, its corresponding plan p, and a tool library $\mathcal{T}$, an algorithm $\mathcal{A}$ sequentially processes each substep $ss_{i,j}$ in p. At each substep, the policy $\pi(a_{i,j} \mid H_{i,j})$ selects an action $a_{i,j}$, consisting of a tool choice $t_{i,j} \in \mathcal{T} \cup \{\text{NO_OP}\}$ and its arguments, conditioned on the execution history $H_{i,j} = \langle (hs_r, ss_{r,s}, a_{r,s}, o_{r,s}) \rangle_{(r,s) \prec (i,j)}$. Each observation $o_{r,s}$ is returned by the simulator $\mathcal{S}$ after executing $a_{r,s}$.

Under this formulation, agent strategies can be divided into two broad paradigms. *Reasoning-based* methods synthesize a complete action sequence in open loop without using intermediate simulator observations, following the trajectory of CoT-, ToT-, or RAP-style planning (Wei et al., 2022; Yao et al., 2023a; Hao et al., 2023). In contrast, *acting-based* methods update decisions online using execution feedback. ReAct (Yao et al., 2023b) establishes the basic Thought–Action–Observation loop, Reflexion (Shinn et al., 2023) augments it with case-level self-correction, and MCTS-style approaches such as LATS and REX (Zhou et al., 2023; Murthy et al., 2023) perform lookahead search over the state–action space. EGB belongs to the acting-based family, but uses uncertainty over tool choices to trigger selective replay

rather than constructing a full search tree.

3.2 Initial Exploration and Uncertainty Quantification

In the first iteration, EGB performs a sequential execution of the plan p inspired by the ReAct paradigm while evaluating the uncertainty of every substep. In the black-box EGB-Sampling variant, where internal model logits are inaccessible, we generate m candidate actions $\{a_{i,j}^{(1)}, a_{i,j}^{(2)}, \ldots, a_{i,j}^{(m)}\}$ through independent samplings from the policy:

$$a_{i,j}^{(1...m)} \sim \pi(a_{i,j}|H_{i,j}) \quad (1)$$

The agent executes the final action $a_{i,j}$ determined by a majority vote among these m candidates and receives the observation $o_{i,j} = \mathcal{S}(a_{i,j})$. Simultaneously, we calculate the predictive entropy $E_{i,j}$ based on the distribution of the m samples to serve as a proxy for uncertainty. Importantly, this estimation method relies solely on output diversity and does not require access to internal model logits or hidden states. When logits are available, EGB can instead use internal distributional signals, as demonstrated in our open-source model experiments. The initial execution continues until the agent either reaches a terminal state or the plan fails to achieve the goal.

3.3 Ranked Branching and Iterative Search

If the initial trajectory results in failure, EGB leverages the recorded entropy signals to optimize the exploration-exploitation trade-off. We rank all substeps $ss_{i,j}$ of the failed trace in descending order of their entropy values $\{E_{i,j}\}$. This ranking identifies decision points where the model was least confident (highest entropy), suggesting that the optimal tool might be a plausible alternative rather than the top-voted choice. In subsequent iterations, EGB initiates branching from the step with the highest remaining uncertainty. Specifically, it selects a previously unexecuted candidate action from the original sampling set $\{a_{i,j}^{(1...m)}\}$ to spawn a new trajectory. The agent then resumes execution from this new state by following the plan-conditioned history until a successful terminal state is identified or the branching budget is exhausted.

In essence, unlike MCTS-based methods that construct full search trees, EGB performs sequential replays with selective branching triggered at high-uncertainty decision points along failed trajectories. This design offers key advantages in tool-rich environments: while MCTS branches based on exploration-exploitation scores that become noisy in long-horizon tasks, EGB uses entropy as a localized diagnostic signal. Moreover, MCTS typically branches during rollouts before observing outcomes, whereas EGB explicitly leverages failed trajectory information to pinpoint and rectify the specific decisions responsible for failure, avoiding prohibitive exhaustive searches in large action spaces.

4 Experiments

We set up experiments on the SLATE synthetic dataset to evaluate EGB against representative baselines, and conduct studies on the impact of hyperparameters and computational costs.

Table 2: Evaluation of search methods on the SLATE synthetic dataset with Claude-Sonnet-4.

Method	Plan-level	Step-wise	
	Execution Success Rate	Tool Match Rate	Action Identification Accuracy
Baseline-LLM	\	65.2 ± 0.8	\
ReAct	29.3 ± 1.2	66.4 ± 0.4	85.7 ± 0.1
Reflexion	44.7 ± 2.3	61.5 ± 1.6	83.4 ± 0.5
LATS	36.5 ± 2.5	63.4 ± 1.3	87.3 ± 0.5
EGB-Sampling (Ours)	**54.0 ± 2.0**	68.5 ± 1.5	87.9 ± 0.2

Table 3: Evaluation of search methods on the SLATE synthetic dataset with Qwen2.5-7B-Instruct.

Method	Plan-level	Step-wise	
	Execution Success Rate	Tool Match Rate	Action Identification Accuracy
Baseline-LLM	\	23.2 ± 0.7	\
ReAct	29.3 ± 1.2	30.0 ± 0.5	78.2 ± 0.4
Reflexion	44.2 ± 2.1	25.2 ± 0.7	82.9 ± 2.9
EGB-Sampling (Ours)	**51.3 ± 6.0**	33.6 ± 2.4	83.0 ± 0.6
EGB-Logits (Ours)	**67.8 ± 4.5**	36.4 ± 0.8	85.1 ± 0.4

4.1 Experimental Settings

Baselines: We evaluate EGB against both reasoning-based and acting-based methods. For the former, we consider **Baseline-LLM**, which selects tools without access to execution history or trajectory-level reasoning and is therefore excluded from plan-level evaluation. For acting-based methods, we compare against: **ReAct**, which interleaves reasoning traces with tool execution in a sequential manner; **Reflexion**, which incorporates self-reflection by analyzing execution feedback to refine subsequent actions; and **LATS**, which builds an entire search tree with environment-aware rollouts.

Table 4: Comparison of computational cost per case across different search methods on SLATE synthetic dataset.

Method	Claude-Sonnet-4	Qwen2.5-7B-Instruct			
	Running Time (seconds)	Running Time * (seconds)	Input Tokens	Output Tokens	LLM Invokes
Baseline-LLM	77	138	1.83×10^5	3.8×10^3	32.0
ReAct	195	150	1.95×10^5	4.0×10^3	32.0
Reflexion	766	707	8.90×10^5	2.41×10^4	122.2
LATS	620	\	\	\	\
EGB-Logits (Ours)	\	254	$(1.61 + 4.86) \times 10^5$ †	6.4×10^3	44.6 + 149.6 †
EGB-Sampling (Ours)	718	1,029	2.19×10^6	3.17×10^4	176.4

† For EGB-Logits, input token usage consists of 1.61×10^5 from generation queries and 4.86×10^5 from logits-only forward passes (no output tokens). LLM invokes include 44.6 generation calls and 149.6 lightweight forward passes. * Parallelization across all methods is disabled for Qwen experiments.

Model: We conduct primary experiments on Claude-Sonnet-4, a proprietary black-box model with inaccessible logits, to demonstrate that EGB is applicable in real-world deployment settings. To validate generalizability, we additionally evaluate on Qwen2.5-7B-Instruct (Qwen et al., 2025), an open-source model with accessible logits. We selected Qwen2.5-7B based on preliminary experiments indicating competitive instruction-following performance and stable entropy-uncertainty correlation among models of comparable size. For cross-benchmark validation, we also use GPT-OSS-120b (OpenAI et al., 2025), an open-weight reasoning model with an efficient mixture-of-experts transformer architecture. We use AWS Bedrock to access Claude and GPT models, and 5 NVIDIA L4 GPUs for Qwen experiments.

Evaluation Metrics: We assess performance from two complementary perspectives: **plan-level** and **step-wise** evaluation.

- *Step-wise evaluation* analyzes decision-making at each individual step through two metrics: Action Identification Accuracy measures the proportion of steps where the agent correctly determines whether a tool call is needed. Tool Match Rate evaluates, among steps requiring tools, the alignment rate between the agent's tool selection and the reference tool in the plan.

- *Plan-level evaluation* assesses the correctness of the final execution result. We report Execution Success Rate, defined as the proportion of cases where the final execution results exactly match the annotated reference resolutions in the plan, regardless of intermediate missteps that do not affect the outcome. For methods with plan-level retries (Reflexion, EGB, and LATS), this is equivalent to *pass@k* as introduced in τ-bench (Yao et al., 2024b), i.e., the probability of reaching a success before exhausting k trials. For EGB, we denote this budget by b.

Configurable Hyperparameters: EGB introduces two critical hyperparameters: m, the number of samplings used to estimate the predictive entropy of tool selection at each step, and b, the branching budget that limits the number of iterations for replanning and exploration. Unless otherwise specified, we set $m = 10$ and $b = 5$ in experiments. For all experiments, we set LLM inference hyperparameters temperature = 1. These settings ensure consistent and comparable results across all baseline methods and model configurations.

4.2 Experimental Results

Results on a Proprietary Model: We first conduct comprehensive experiments on Claude-Sonnet-4 to evaluate EGB against baseline methods, with results presented in Table 2. At the plan level, EGB achieves substantial improvements in execution success rate, outperforming ReAct by 24.7% and Reflexion by 9.3%. Notably, EGB also surpasses LATS by 17.5% despite LATS being constrained to 25 global search node visits per case—an empirical limit chosen to match EGB's average computational budget. This significant plan-level advantage, coupled with only marginal improvements in step-wise metrics (2.0%–7.0% on Tool Match Rate), reveals a critical limitation of baseline methods: without access to step-level correctness feedback from the simulator, they struggle to identify error-prone decisions under constrained computation. Unlike EGB, which leverages model uncertainty (entropy) to prioritize likely failure points, baseline methods rely solely on end-of-trajectory feedback. In long-horizon tasks, this leaves them unable to effectively localize which steps in the Markov decision process are responsible for failure. This effect is particularly evident for Reflexion and

Table 5: Entropy-error relationship on SLATE using EGB-Logits.

Entropy Range	#Steps	#Errors	Error Rate
$[0.000, 0.640)$	365	184	50.7%
$[0.640, 1.281)$	116	79	68.1%
$[1.281, 1.921)$	67	52	77.6%
$[1.921, 2.561)$	21	20	95.2%
$[2.561, 3.202]$	15	14	93.3%

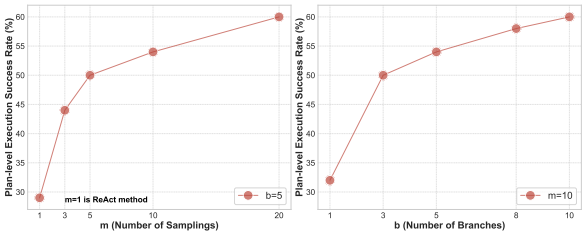

Figure 5: Evaluation of EGB search with Claude-Sonnet-4 on the SLATE synthetic dataset with varied configurations: (a) number of samplings, m, used to estimate entropy, and (b) budget, b, as the maximum number of branch trials.

LATS, which exhibit *decreased* Tool Match Rates compared to Baseline-LLM and ReAct, suggesting they may modify previously correct tool selections into incorrect ones during search. Their plan-level improvements thus appear to stem more from exploring additional execution paths than from precisely targeting problematic decisions.

Computation Cost: Table 4 presents the computational cost comparison. For Qwen experiments, we disable parallelization across all methods to ensure fair comparison on a single GPU, resulting in EGB-Sampling being slower than Reflexion (1,029s vs. 707s). However, the m independent samples in EGB-Sampling can be trivially parallelized—as demonstrated in Claude experiments with 2 concurrent workers, where EGB-Sampling achieves 718 seconds (**6.3%** faster than Reflexion). Compared with LATS (620s on Claude), EGB-Sampling incurs moderate extra runtime but delivers substantially higher execution success in Table 2. EGB-Logits demonstrates the strongest efficiency, requiring only 254 seconds (**2.8**$\times$ faster than Reflexion) by leveraging lightweight forward passes without autoregressive generation. Notably, EGB-Logits reduces output tokens by **73%** compared to Reflexion (6.4×10^3 vs. 2.41×10^4). These results indicate that entropy-guided branching improves success with different cost profiles: sampling is broadly applicable but latency-sensitive, whereas logits-based entropy is more efficient when white-box access is available.

Impact of Hyperparameters: We systematically investigate EGB's two critical hyperparameters, as illustrated in Fig. 5. The number of samplings m controls entropy estimation quality: performance increases from 29% at $m = 1$ to 60% at $m = 20$. Similarly, the branching budget b governs the extent of error correction: success rate improves from 32% at $b = 1$ to 60% at $b = 10$. Both parameters exhibit diminishing returns beyond certain thresholds; thus, we adopt $m = 10$ and $b = 5$ as our default configuration to balance performance gains against computational cost. Notably, when $m = 1$, EGB degenerates to the ReAct method without leveraging uncertainty signals. When $b = 1$, only a minimal branch opportunity is available, so performance remains close to the initial trajectory quality. The key takeaway is that performance improves with both more accurate entropy estimation and increased branching opportunities for error correction.

Results on an Open-source Model: To further validate our approach, we conduct experiments on Qwen2.5-7B-Instruct, a white-box model that enables direct computation of entropy from output logits rather than sampling-based estimation. As shown in Table 3, the findings mirror those observed with Claude-Sonnet-4: EGB substantially outperforms baseline methods at the plan level while achieving marginally better step-wise performance. We omit LATS from this evaluation due to the prohibitive computational cost of hosting open-source models for extensive tree search. Notably, EGB-Logits achieves 67.8% execution success rate, surpassing EGB-Sampling by 16.5%. This observation supports entropy as a useful relative signal for prioritizing error-prone decisions. Moreover, accurate entropy computation from logits proves more effective than sampling-based estimation.

Error Rate and Entropy Correlation on SLATE: To assess whether entropy provides a useful branching signal, we analyze EGB-Logits at the step level, where error is defined as a first-pass tool mismatch against the reference trajectory. As shown in Table 5, correct steps have lower entropy than error steps (mean $= 0.052$ vs. 0.119), and error rate rises from 50.7% to 93.3% across entropy bins (with a small fluctuation in the highest-entropy bins). These results indicate that entropy is an effective relative prioritization signal for branch selection rather than a strict binary error detector. In successful branch corrections, errors are reduced through direct tool fixes, cascade fixes via corrected execution context, and functionally equivalent al-

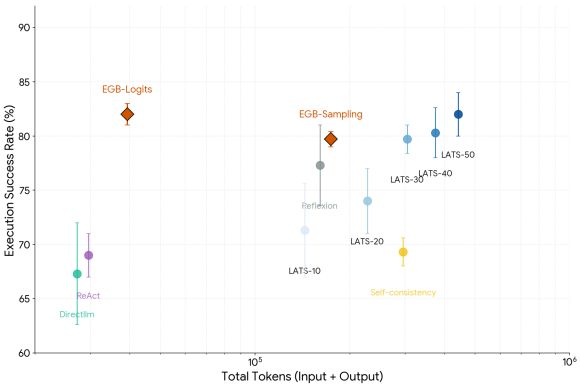

Figure 6: Cross-benchmark validation on adapted UltraTool with GPT-OSS-120b.

ternative tools; detailed breakdowns are provided in Appendix Sec. A.2.

Cross-Benchmark Validation on Adapted UltraTool: Because SLATE is synthesized with Claude-family models, we additionally evaluate on adapted UltraTool to reduce potential model-family bias and test whether our conclusions transfer beyond SLATE. We choose UltraTool because its plans are shorter-horizon, enabling practical comparison against higher-budget tree-search baselines that are too expensive on SLATE. In Figure 6, **Self-Consistency** samples multiple ($m = 10$) tool choices per step and executes the majority-voted action, while **LATS-k** denotes LATS with an upper limit of $[10, 50]$ distinct search-tree nodes visited. EGB remains consistently strong: EGB-Logits matches the best reported success rate (LATS-50) while using only a fraction of the token cost, and EGB-Sampling also achieves high performance comparable to LATS-30; meanwhile, LATS shows the expected budget–accuracy trade-off, with success improving as node budget increases but with diminishing returns. We also studied EGB's robustness to inaccurate plan specifications in Appendix Sec. A.3.

5 Conclusion

In this work, we introduced SLATE, a large scale benchmark for assessing tool augmented agents in long horizon e-commerce tasks. Our evaluation revealed that current methods struggle with efficiency and self-correction in vast action spaces. To address this, we proposed Entropy Guided Branching (EGB), an uncertainty aware search algorithm that dynamically allocates exploration effort where predictive entropy is high. Experimental results demonstrate that EGB significantly improves task success rates and computational efficiency. Together, SLATE and EGB provide a robust foundation for building reliable LLM agents for complex environments.

6 Limitations

Despite the contribution of SLATE and EGB, several limitations warrant further investigation.

Limitations of SLATE. While SLATE incorporates intricate sequential dependencies and context-aware responses, it remains a synthetic benchmark. Its simulator returns deterministic outcomes for matched tool invocations to enable controlled plan-level evaluation, which does not fully capture operational behaviors of real APIs such as latency spikes, retry policies, transient invocation failures, or external service instability. In addition, SLATE evaluates tool-invocation decisions under a fixed reference plan, so it does not directly assess an agent's ability to synthesize plans from scratch. Finally, since both the benchmark and simulator were generated through a closed-source LLM pipeline (Claude 3.7), the dataset may contain model-specific biases. This can make cross-backbone comparisons less strictly fair; therefore, our main conclusions emphasize comparisons across tool-use paradigms under the same backbone LLM.

Limitations of EGB. Although EGB-Sampling is compatible with black-box models, its reliance on multiple samplings ($m = 10$) to estimate entropy introduces higher inference latency and API costs than single-pass greedy methods, potentially limiting its use in real-time or resource-constrained scenarios. In addition, EGB can only branch over sampled or logit-ranked alternatives, and entropy can be imperfect when a model is confidently wrong or uncertain among functionally equivalent tools. Deterministic replay in our experiments is a property of the SLATE simulator and a practical implementation of backtracking, rather than a fundamental constraint of EGB. This assumption is also shared by tree-search baselines such as LATS, and practical systems can often support limited backtracking through checkpointing, idempotent API design, and rollback mechanisms; extending EGB to fully non-deterministic stateful environments remains future work. Moreover, this study focuses on e-commerce tasks; further work is needed to validate entropy-guided search in other tool-rich environments, such as software engineering or scientific research.

7 Ethical Considerations

This work focuses on improving the evaluation and planning efficiency of tool-augmented LLM agents through a synthetic benchmark and an uncertainty-aware search strategy. The proposed dataset does not contain real user data, personal information, or proprietary APIs, thereby minimizing privacy and data misuse risks. However, more efficient planning and execution may lower the barrier to deploying highly autonomous agents, which could be misused if applied without appropriate safeguards. In particular, aggressive exploration strategies may be undesirable in safety-critical or high-stakes environments. We emphasize that our methods are intended for controlled research settings and should be combined with external safety mechanisms, access controls, and human oversight in real-world deployment. Additionally, as the benchmark is synthetically generated, it may inherit biases from the underlying language model, and results should be interpreted with appropriate caution. UltraTool is publicly available under the Apache License 2.0, and our research use is consistent with its intended use. The proprietary and open-source models used in our experiments are listed in Section 4. Portions of the synthetic data, code, and manuscript were prepared with assistance from AI tools, including ChatGPT and Claude Code; however, the human authors proposed the research ideas, validated the methodology and results, and retain full responsibility for the work.

References

Michael Ahn, Debidatta Dwibedi, Chelsea Finn, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Karol Hausman, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, and 1 others. 2024. Autort: Embodied foundation models for large scale orchestration of robotic agents. *arXiv preprint arXiv:2401.12963*.

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

Richard E Bellman and Stuart E Dreyfus. 2015. *Applied dynamic programming*. Princeton university press.

Daniil A Boiko, Robert MacKnight, and Gabe Gomes. 2023. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332*.

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. 2025. Acebench: Who wins the match point in tool usage? *Preprint*, arXiv:2501.12851.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. 2025. Traject-bench:a trajectory-aware benchmark for evaluating agentic tool use. *Preprint*, arXiv:2510.04550.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, and 1 others. 2024. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. *arXiv preprint arXiv:2401.17167*.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and 1 others. 2023. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, Junteng Liu, Zhaochen Su, Yiyang Guo, Fan Zhou, Lueyang Zhang, Juan Michelini, Xingyao Wang, Xiang Yue, Shuyan Zhou, and 2 others. 2026. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. *Preprint*, arXiv:2510.25726.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023a. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023b. Api-bank: A comprehensive benchmark for tool-augmented llms. *Preprint*, arXiv:2304.08244.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. 2025. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. *Preprint*, arXiv:2408.04682.

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

Rithesh Murthy, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Le Xue, Weiran Yao, Yihao Feng, Zeyuan Chen, Akash Gokul, Devansh Arpit, and 1 others. 2023. Rex: Rapid exploration and exploitation for ai agents. *arXiv preprint arXiv:2307.08962*.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, and 108 others. 2025. gpt-oss-120b and gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

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

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

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

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning, 2023. *URL https://arxiv.org/abs/2303.11366*, 1.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Jun Wang, Jiamu Zhou, Muning Wen, Xiaoyun Mo, Haoyu Zhang, Qiqiang Lin, Cheng Jin, Xihuai Wang, Weinan Zhang, Qiuying Peng, and Jun Wang. 2025. Hammerbench: Fine-grained function-calling evaluation in real mobile device scenarios. *Preprint*, arXiv:2412.16516.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, and 1 others. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, and 1 others. 2025. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2526–2547.

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

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Hongyang Yang, Boyu Zhang, Neng Wang, Cheng Guo, Xiaoli Zhang, Likun Lin, Junlin Wang, Tianyu Zhou, Mao Guan, Runjia Zhang, and 1 others. 2024. Finrobot: An open-source ai agent platform for financial applications using large language models. *arXiv preprint arXiv:2405.14767*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

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

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

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

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

Appendices

A Supplementary Experiments

A.1 Leveraging Past Executions with Memory

Table 6: Evaluation of search methods with vs. without memory on the SLATE synthetic dataset with Claude-Sonnet-4.

Method	Plan-level	Step-wise Evaluation	
	Execution Success Rate	Tool Match Rate	Action Identification Accuracy
ReAct	29.3 ± 1.2	66.4 ± 0.4	85.7 ± 0.1
Reflexion	44.7 ± 2.3	61.5 ± 1.6	83.4 ± 0.5
EGB-Sampling (Ours)	54.0 ± 2.0	68.5 ± 1.5	87.9 ± 0.2
ReAct-with-Memory	86.3 ± 0.6	92.7 ± 0.3	98.7 ± 0.1
Reflexion-with-Memory	91.0 ± 1.5	92.7 ± 1.1	98.4 ± 0.3
EGB-with-memory (Ours)	$\mathbf{92.7 \pm 1.0}$	92.6 ± 0.5	98.7 ± 0.2

Experiment Setup. While EGB demonstrates strong performance through entropy-guided exploration, we investigate whether incorporating memory from previous task executions can further enhance its capabilities. We create a pseudo in-distribution setting by randomly partitioning the dataset $\mathcal{D}$ into a validation set $\mathcal{D}_{\text{val}}$ (50 samples) and a test set $\mathcal{D}_{\text{test}}$ (50 samples), where both sets are drawn from the same task distribution but contain no shared tasks. We accumulate execution traces from the validation set as memory and evaluate on the test set.

Step-wise Evaluation Metrics. Beyond plan-level execution success, we also track step-wise behavior. Action Identification Accuracy measures the proportion of steps where the agent correctly determines whether a tool call is needed, and Tool Match Rate evaluates, among steps requiring tools, the alignment rate between the agent's tool selection and the reference tool in the plan.

Analysis. As shown in Table 6, memory substantially improves performance across all methods: ReAct improves from 29.3% to 86.3% (+57.0%), Reflexion from 44.7% to 91.0% (+46.3%), and EGB from 54.0% to 92.7% (+38.7%). Notably, even with memory enabled, EGB maintains its advantage over baselines, outperforming ReAct-with-Memory by 6.4% and Reflexion-with-Memory by 1.7%. These results demonstrate that while memory provides a powerful mechanism for leveraging past experiences in similar tasks, EGB continues to offer systematic improvements regardless of whether memory is available, confirming its effectiveness as a complementary approach to memory-based methods.

A.2 Error Entropy Correlation

Table 7: Entropy-error relationship on SLATE using Qwen-logits.

Entropy Range	# Steps	# Errors	Error Rate
$[0.000, 0.640)$	365	184	50.7%
$[0.640, 1.281)$	116	79	68.1%
$[1.281, 1.921)$	67	52	77.6%
$[1.921, 2.561)$	21	20	95.2%
$[2.561, 3.202)$	15	14	93.3%

Experiment Setup. We use the Qwen-logits experiment results because they provide calibrated per-step logit distributions for entropy estimation. Entropy is computed at each tool-selection step, excluding fallback steps where logit computation fails. We compare steps whose first-pass selected tool matches the reference tool (*Correct*, $n = 233$) against those that do not (*Error*, $n = 349$), for a total of 582 analyzed

steps. Error rate in Table 7 is defined as the fraction of steps in each entropy bin whose first-pass tool does not match the reference.

Analysis. Correct steps have substantially lower entropy (mean $= 0.052$, median $= 0.000$) than error steps (mean $= 0.119$, median $= 0.081$), indicating stronger confidence when the tool choice is correct. Across bins, error rate rises from 50.7% in the lowest-entropy range to above 93% in the highest ranges, showing a near-monotonic trend. The lowest-entropy bin still has non-trivial error due to benchmark difficulty: with 1000 tools and functionally similar candidates, the model can be confidently wrong. Overall, entropy is best interpreted as a relative prioritization signal for branching rather than a binary error detector.

How Branching Reduces Error. We further analyze 68 successful branch corrections (68 out of 114 first-pass failed plans corrected by entropy-guided exploration). We identify three mechanisms: (1) **Direct fix** (17.6%), where the branched step switches to the reference tool; (2) **Cascade fix** (30.9%), where the branched step changes execution context and downstream tools become correct (on average, each branch changes 0.9 subsequent steps); and (3) **Functional equivalence** (51.5%), where the branched tool name differs from the reference but yields equivalent outputs. These mechanisms explain why entropy-guided branching remains effective even when exact tool-name matching is not the only path to success.

A.3 Robustness to Inaccurate Plan Specifications

Table 8: Execution success rate under accurate vs. inaccurate plans on adapted UltraTool with GPT-OSS-120b.

Methods	Accurate Plan	Inaccurate Plan
ReAct	69.0 ± 2.0	65.0 ± 5.0
Self-Consistency	69.3 ± 1.3	65.3 ± 0.7
EGB-Sampling	79.7 ± 0.7	72.0 ± 2.0

Experiment Setup. To test robustness to imprecise planning, we modify original UltraTool plans by rewriting some tool-call sub-steps into broader descriptions while preserving intent (e.g., "Use file writing tool to create and write content" $\rightarrow$ "Perform file operation", "Search for files matching pattern" $\rightarrow$ "Find relevant files"). In total, 10.5% of tool-call sub-steps are modified.

Analysis. All methods degrade when plan specificity drops, but EGB-Sampling remains strongest (72.0%). Notably, EGB-Sampling under inaccurate plans still exceeds ReAct and Self-Consistency under accurate plans, indicating that entropy-guided branching provides robustness to upstream plan imprecision.

B Algorithms

B.1 Pseudocode

Algorithm 1 Entropy-Guided Branching (EGB)

Require: Query q, Plan $p = \{ss_{1,1}, \ldots, ss_{n,m_n}\}$, Tool library $\mathcal{T}$, Simulator $\mathcal{S}$, Global branch limit $B = 50$, Per-step branch limit $B_s = 5$

// Phase 1: Initial Execution with Entropy Recording
$H \leftarrow \langle\rangle$ {Execution history}
$\mathcal{E} \leftarrow \langle\rangle$ {Entropy tree}
for each substep $ss_{i,j} \in p$ **do**
  $\mathcal{C}_{i,j} \leftarrow \text{GETCANDIDATES}(q, ss_{i,j}, H, \mathcal{T})$ {Retrieve candidate tools via embedding search}
  $(a^*_{i,j}, E_{i,j}, \mathcal{D}_{i,j}) \leftarrow \text{COMPUTEENTROPY}(q, ss_{i,j}, H, \mathcal{C}_{i,j})$ {Alg. 2 or 3}
  $o_{i,j} \leftarrow \mathcal{S}(a^*_{i,j})$ {Execute selected action}
  $H \leftarrow H \cup \langle(ss_{i,j}, a^*_{i,j}, o_{i,j})\rangle$
  $\mathcal{E} \leftarrow \mathcal{E} \cup \langle(i, j, E_{i,j}, \mathcal{D}_{i,j})\rangle$ {Store entropy and distribution}
end for
if $\text{TASKSUCCESS}(H)$ **then**
  return H
end if
// Phase 2: Entropy-Guided Branching
$\mathcal{E}_{\text{sorted}} \leftarrow \text{SORTBYENTROPY}(\mathcal{E}, \text{descending})$
$b \leftarrow 0$ {Global branch counter}
for each $(i, j, E_{i,j}, \mathcal{D}_{i,j}) \in \mathcal{E}_{\text{sorted}}$ **do**
  $t^*_{i,j} \leftarrow \text{EXTRACTTOOLNAME}(a^*_{i,j})$ {Tool selected in first pass}
  $\mathcal{T}_{\text{alt}} \leftarrow \{t : (t, p_t) \in \mathcal{D}_{i,j}, t \neq t^*_{i,j}\}$ {Alternative tools from distribution}
  Sort $\mathcal{T}_{\text{alt}}$ by probability in descending order
  $b_s \leftarrow 0$ {Per-step branch counter}
  for each $t' \in \mathcal{T}_{\text{alt}}$ **do**
    if $b \geq B$ **or** $b_s \geq B_s$ **then**
      break {Budget exhausted}
    end if
    $a'_{i,j} \leftarrow \text{GETORGENERATECALL}(t', q, ss_{i,j}, H_{<(i,j)}, \mathcal{D}_{i,j})$ {Use cached call or generate params[†]}
    $H' \leftarrow H_{<(i,j)} \cup \langle(ss_{i,j}, a'_{i,j}, \mathcal{S}(a'_{i,j}))\rangle$ {Branch from step (i, j)}
    for each subsequent substep $ss_{r,s}$ where $(r, s) \succ (i, j)$ **do**
      $a_{r,s} \leftarrow \text{SELECTTOOL}(q, ss_{r,s}, H', \mathcal{T})$ {Single-pass selection}
      $H' \leftarrow H' \cup \langle(ss_{r,s}, a_{r,s}, \mathcal{S}(a_{r,s}))\rangle$
    end for
    if $\text{TASKSUCCESS}(H')$ **then**
      return H'
    end if
    $b \leftarrow b + 1; b_s \leftarrow b_s + 1$
  end for
end for
return H {Return first-pass result}

[†] For Alg. 2, uses pre-sampled calls stored in $\mathcal{D}_{i,j}$. For Alg. 3, generates parameters on-demand via LLM.

Algorithm 2 Entropy Computation via Sampling (EGB-Sampling)

Require: Query q, Substep $ss_{i,j}$, History $H_{i,j}$, Candidates $\mathcal{C}_{i,j}$, Sample count m
Ensure: Selected action $a^*_{i,j}$, Entropy $E_{i,j}$, Action distribution $\mathcal{D}_{i,j}$

```
votes ← {} {Vote counts per tool}
calls ← {} {Sampled tool calls per tool}
for k = 1 to m do
    a_{i,j}^{(k)} ~ π(a_{i,j} | H_{i,j}, C_{i,j}) {Sample complete action (tool + params) from LLM}
    t^{(k)} ← ExtractToolName(a_{i,j}^{(k)})
    votes[t^{(k)}] ← votes[t^{(k)}] + 1
    calls[t^{(k)}] ← calls[t^{(k)}] ∪ {a_{i,j}^{(k)}} {Store sampled call for potential branching}
end for
// Compute entropy from vote distribution
for each tool t ∈ votes do
    p_t ← votes[t]/m
end for
E_{i,j} ← −Σ_t p_t log p_t
// Select majority tool (use pre-sampled call)
t* ← arg max_t votes[t]
a*_{i,j} ← SelectOne(calls[t*]) {Use one of the sampled calls}
D_{i,j} ← {(t, p_t, calls[t]) : t ∈ votes} {Store distribution with pre-sampled calls}
return (a*_{i,j}, E_{i,j}, D_{i,j})
```

Algorithm 3 Entropy Computation via Logits (EGB-Logits)

Require: Query q, Substep $ss_{i,j}$, History $H_{i,j}$, Candidates $\mathcal{C}_{i,j} = \{c_0, \ldots, c_{K-1}\}$ with $K \leq 100$, Threshold $\tau = 0.01$

```
prompt ← BuildPrompt(q, ss_{i,j}, H_{i,j}, C_{i,j}) {Ask LLM to output tool index 0 to K−1}
x ← Tokenize(prompt)
// Get first token distribution over digits 0-9
l^{(1)} ← ForwardPass(x)[−1] {Logits at last position}
p^{(1)} ← Softmax(l^{(1)}_{['0':'9']}) {Probability over digit tokens}
// Compute conditional second token distribution
P ← 0^K {Probability for each candidate}
for d_1 = 0 to 9 do
    x' ← Concat(x, Token(d_1))
    l^{(2)} ← ForwardPass(x')[−1]
    p^{(2)} ← Softmax(l^{(2)}_{['0':'9']})
    p_end ← 1 − Σ_{d_2=0}^{9} p^{(2)}_{d_2} {Probability of non-digit token (sequence end)}
    if d_1 < K then
        P[d_1] ← P[d_1] + p^{(1)}_{d_1} · p_end {Single-digit index}
    end if
    for d_2 = 0 to 9 do
        idx ← d_1 × 10 + d_2
        if 10 ≤ idx < K then
            P[idx] ← P[idx] + p^{(1)}_{d_1} · p^{(2)}_{d_2} {Two-digit index}
        end if
    end for
end for
P ← P / Σ_k P[k] {Normalize}
// Compute entropy over all candidates
E_{i,j} ← −Σ_{k=0}^{K−1} P[k] log P[k]
// Filter candidates by probability threshold for branching
C^{filtered}_{i,j} ← {c_k : P[k] ≥ τ}
// Select highest probability tool and generate parameters
k* ← arg max_k P[k]
a*_{i,j} ← GenerateParams(c_{k*}, q, ss_{i,j}, H_{i,j}) {Generate params only for selected tool}
D_{i,j} ← {(c_k, P[k]) : c_k ∈ C^{filtered}_{i,j}} {Params generated lazily during branching}
return (a*_{i,j}, E_{i,j}, D_{i,j})
```

B.2 Complexity Analysis

Let n denote the number of plan substeps, m the number of entropy samples per step in EGB-Sampling, and b the number of executed branch attempts in Phase 2 (bounded by B). We summarize the dominant computational costs in terms of LLM-scale operations.

Phase 1 (initial execution + entropy recording). For EGB-Sampling, each step performs entropy estimation with m sampled calls, yielding $O(nm)$ dominant LLM complexity. For EGB-Logits, entropy is derived from direct token-level probabilities (effectively m=1), reducing Phase 1 to $O(n)$ dominant LLM complexity.

Phase 2 (entropy-guided branching). Each branch reuses the prefix before the branch point and re-executes only the suffix with single-pass tool selection. In the worst case, each attempt may replay $O(n)$ remaining steps, so Phase 2 is $O(bn)$; with branch points roughly centered, the average replay length is about $n/2$, i.e., $O(bn)$ up to constants.

Total complexity. Combining both phases gives:

- EGB-Sampling (black-box entropy): $O(n(m + b))$,
- EGB-Logits (white-box entropy): $O(n + bn) = O(bn)$ when branching dominates.

These expressions match the complexity trend in the accompanying algorithm diagram: entropy sampling dominates Phase 1 for black-box models, while branch replay dominates when branch budgets are large.

C SLATE Construction

C.1 Prompt Templates

Prompt for Generating Diverse E-Commerce Cases

Generate ONE focused e-commerce query that an {user_type} might have.
User type: {user_type} Focus area: {focus} Complexity level: {complexity}
Requirements based on COMPLEXITY LEVEL:
{complexity} COMPLEXITY GUIDELINES: {self._get_complexity_guidelines(complexity)}
For ALL queries, ensure:

1. Include SPECIFIC DETAILS that can be used as direct arguments in tools:
   - Product names and identifiers (e.g., "Sunset Yoga Mat (SKU: YM-2023-BL)")
   - Order numbers (e.g., "Order #AB-12345678")
   - FULL DATES WITH YEARS (e.g., "January 15, 2023" not just "January 15")
   - Prices with currency (e.g., "$49.99")
   - Specific quantities (e.g., "3 units")
2. Include MULTIPLE DATA POINTS for tools to extract as parameters
3. For COMPLEX and ADVANCED queries, include:
   - Multiple constraints or conditions
   - Timing requirements or deadlines
   - Preferences with priorities
   - Historical context or previous actions
   - Special exceptions or unusual circumstances

Examples of queries at different complexity levels:
SIMPLE: "I need to track my order #RT-78256391 for the Samsung Galaxy Buds that I ordered on May 3, 2023. When will it be delivered?"
MODERATE: "I need to return two items from my Order #112-9384756 placed on March 12, 2023: the Samsung Galaxy S22 with a cracked screen and the protective case. I want to keep the screen protector and charging cable. Can I get a return label for just those two items?"
COMPLEX: "I need to modify my bulk order #BLK-2023-4872 placed on February 28, 2023 for my company. We ordered 50 Lenovo ThinkPad T14 laptops with i7 processors at $1,299 each, but now I need to change 15 of them to the i9 model which costs $1,599 each. We've already paid the deposit of $25,000 and delivery is scheduled for June 15, 2023. I need to know if this change will affect our delivery date and what additional payment is required."

Figure 7: Prompt to generate diverse e-commerce cases from different user perspectives. **Example Variable Values:** `user_type` alternates between "buyer" and "seller" based on query ID; `focus` is randomly selected from buyer focus areas ("product search with multiple conflicting requirements", "order tracking for multiple items with shipping complications", "complex product return with partial refund request", etc.) or seller focus areas ("updating complex product variations and attributes", "inventory management across multiple warehouses", "implementing tiered pricing strategy with conditions", etc.); `complexity` is one of "SIMPLE", "MODERATE", "COMPLEX", or "ADVANCED" with weighted distribution of 0%, 10%, 80%, 10% respectively.

Prompt for Creating Specialized Tool Plans to Resolve E-Commerce Queries (Part 1)

ADVANCED: "I'm managing our company's quarterly office supply order (PO #BZ-45721) placed on April 2, 2023 with scheduled delivery on April 20, 2023 across three locations. For the Chicago office (Location ID: CHI-005), we need to cancel the 12 ergonomic chairs ($259 each) due to their recent merger, but expedite the 15 monitor stands ($89 each) to arrive by April 15. For the Boston office (Location ID: BOS-002), we need to add 8 wireless keyboards ($65 each) and 8 wireless mice ($45 each) for new hires. The New York office (Location ID: NYC-001) shipment is fine as is, but we need to change the delivery window to after 2:00 PM. We're eligible for the 12% corporate discount and already applied the SPRING2023 promo code for 5% off. How will these changes affect our total, and can all these modifications be accommodated before processing begins on April 10?"
Return only the query text with no additional explanation.
Create a plan with specialized tools to resolve this e-commerce query:
User Query ({user_type}, {complexity} complexity): "{query}"
{existing_tools_text}
Please provide:
1. A HIERARCHICAL PLAN with: {plan_requirements}
2. For each SUBSTEP, indicate:

- If it's a high-level step: "tool": "null" - NO EXCEPTIONS!
- If no tool is needed: "tool": "No tool required"
- If a tool is needed: "tool": "tool_name(param1='value1', param2='value2')"

3. CRITICAL: FIRST TOOL CALL arguments must come DIRECTLY from the query

- Example: If query mentions "Order #AB-12345", first tool should use order_id='AB-12345'
- Extract actual values from the query text, don't invent new values

4. CRITICAL: SUBSTEP DESCRIPTIONS MUST BE DISTINCT FROM TOOL NAMES

- Make each substep description MEANINGFUL and CONTEXT-RICH
- Do not create tool names that are Identical or Very Similar to substep descriptions.
- BAD: "1.1 Get order details" when using "get_order_details" tool
- GOOD: "1.1 Retrieve customer's purchase history for Order #AB-123" when using "get_order_details" tool
- Describe the PURPOSE and CONTEXT of the step, not just the action
- Include relevant business context and specific goals for each step

Figure 8: Prompt for creating specialized tool plans (Part 1). **Example Variable Values:** `user_type` is either "buyer" or "seller"; `complexity` is one of "SIMPLE", "MODERATE", "COMPLEX", or "ADVANCED"; `query` is the actual query text generated in stage 1. **existing_tools_text:** Dynamically generated list of top 30 most relevant tools from global tool library. **plan_requirements:** SIMPLE (2-3 high-level steps, 1-2 substeps each, 3-5 total); MODERATE (3-4 high-level steps, 1-3 substeps each, 6-8 total); COMPLEX (4-5 high-level steps, 2-3 substeps each, 8-12 total); ADVANCED (5-7 high-level steps, 2-4 substeps each, 12-16 total).

Prompt for Creating Specialized Tool Plans to Resolve E-Commerce Queries (Part 2)

5. TOOL SELECTION AND DESIGN:
   - TRY REUSE EXISTING TOOLS whenever appropriate (from the list above)
   - If none of the existing tools are suitable, design NEW SPECIALIZED TOOLS that:
     - Have SPECIFIC, FOCUSED functionality (not general-purpose)
     - Use SIMPLE arguments (2-3 parameters maximum)
     - Return SIMPLE, FOCUSED results (1-3 fields maximum)
     - Follow snake_case naming convention
     - Avoid creating monolithic "do everything" tools
6. CRITICAL - AVOID TOOL REPETITION:
   - DO NOT use the same tool in consecutive substeps (e.g., avoid "2.1 xxx tool_A; 2.2 xxx tool_A")
   - If you need to call the same tool multiple times, space them out with other operations
   - Each substep should ideally use a DIFFERENT tool to create diversity
   - Create specialized tools for different aspects rather than reusing generic ones
7. IMPORTANT - SEQUENTIAL DEPENDENCIES:
   - Later steps should use results from previous steps for sequential dependencies
   - Use the format OUTPUT_FROM_STEP_X.Y.field to reference previous outputs
   - Example: product_id=OUTPUT_FROM_STEP_1.2.product_id
   - Create a CHAIN of dependencies where each step builds on previous results
   - Ensure that tool outputs from early steps provide necessary inputs for later steps
8. FINAL STEP should produce a DIRECT RESULT that resolves the query
   - The last tool should return a clear outcome or answer
   - For example: confirmation message, success status, or direct result
   - FINAL STEPS should utilize outputs from earlier steps
9. FOR {complexity} COMPLEXITY:
   - {self._get_plan_complexity_guidelines(complexity)}

Figure 9: Prompt for creating specialized tool plans (Part 2). Focus on tool selection, reuse strategy, avoiding repetition, and establishing sequential dependencies between steps.

Prompt for Creating Specialized Tool Plans to Resolve E-Commerce Queries (Part 3)

10. TOOL DIVERSITY REQUIREMENTS:

- Create tools that span different functional domains (retrieval, validation, processing, notification, etc.)
- Avoid generic tools like "process_request" or "handle_query"
- Instead create specific tools like "validate_return_window", "calculate_refund_amount", "send_confirmation_email"
- Each tool should have a clear, single responsibility

TOOL DESIGN PRINCIPLES:

1. FOCUSED: Each tool should do ONE thing well
2. COMPOSABLE: Tools should work together through their inputs/outputs
3. REUSABLE: Create tools that could be useful in other scenarios
4. SIMPLE: Prefer multiple simple tools over one complex tool
5. DIVERSE: Create tools spanning different functional domains
6. SEQUENTIAL: Design tools to create natural dependencies and data flow

Remember:

- REUSE existing tools whenever appropriate
- FIRST TOOL must use arguments DIRECTLY from the query
- Keep tools SPECIALIZED with SIMPLE inputs and outputs
- Ensure steps are SEQUENTIAL with clear dependencies
- AVOID repeating the same tool in consecutive steps
- FINAL STEP should provide a DIRECT RESOLUTION to the query
- Return valid JSON only

Figure 10: Prompt for creating specialized tool plans (Part 3). Format response as JSON object with "plan" array (high-level steps with "tool": "null", and substeps with tool calls or "No tool required") and "tools" array (defining all tools with "name", "description", "arguments", and "results" fields). **Complexity Guidelines:** SIMPLE (1-2 tool calls, linear, minimal dependencies); MODERATE (2-4 tool calls, at least one dependency and conditional step); COMPLEX (4-7 tool calls, multiple dependencies, validation, branching paths); ADVANCED (7+ tool calls, multi-stage workflow, complex dependencies, edge case handling, maximum tool diversity).

Prompt for Generating Diverse E-Commerce Tools

Generate {batch_size} DIVERSE e-commerce tools with COMPLETELY UNIQUE NAMES.

TOOL NAMING REQUIREMENTS:

1. Each tool name MUST be UNIQUE and use snake_case format
2. NEVER use any of these existing names: {names_str}
3. Create DISTINCTIVE names that avoid generic patterns
4. Consider these name patterns for inspiration: {name_pattern_text}
5. Each tool name should reflect its SPECIFIC FUNCTION and DOMAIN
6. VERIFY that each name is different from all others in your response

TOOL ASSIGNMENTS - Create exactly one tool for EACH of these specific domains: {domains_text}

SAMPLE TOOL STRUCTURE: {json.dumps(sample_tool, indent=2)}

TOOL REQUIREMENTS:

1. Match each tool precisely to its assigned domain above
2. Include 1-3 SIMPLE arguments with clear purposes
3. Return 1-3 FOCUSED result fields
4. Include "query_id": "dummy" in each tool
5. Follow the exact JSON structure of the sample

Return a JSON array containing {batch_size} tools with UNIQUE names:

```
[
  {tool1},
  {tool2},
  ...
]
```

ONLY return the JSON array with no additional text.

Figure 11: Prompt for generating diverse e-commerce tools with unique names. **Example Variable Values:** `batch_size` is configurable (typically 10 or 20, default 10). `names_str` is a comma-separated string of 20 random existing tool names to avoid duplication (e.g., "get_order_details, validate_return_window, calculate_refund, search_products, update_inventory, generate_shipping_label, process_payment, send_notification, track_shipment, analyze_sales, manage_promotions, verify_address, calculate_tax, check_stock, create_invoice, update_customer, generate_report, schedule_delivery, process_return, validate_coupon"). `name_pattern_text` consists of 4 randomly selected naming patterns from: "feature_specific_domain, domain_specific_action, specialized_task_handler, domain_analyzer, action_target_generator, domain_insight_provider, specialized_workflow_automation, target_specific_optimizer". `domains_text` is a numbered list of specific domains (one per tool) from 100 predefined diverse domains covering: Product management (catalog, variants, bundling, photography assessment, competitive analysis, recommendations, sourcing, limited editions, warranties, etc.), Shipping & logistics (delivery, warehousing, returns, tracking, etc.), etc. `sample_tool` is a randomly selected existing tool or default structure containing: `name`, `description`, `query_id`, `arguments` (with type, properties, and argument details), and `results` (with type, properties, and result field details).

Prompt for Categorizing E-Commerce Tools into Functional Domains

Analyze these e-commerce tools and categorize them into functional domains.

TOOLS TO CATEGORIZE: {json.dumps(batch_tools, indent=2)}

PREDEFINED CATEGORIES: {json.dumps(self.categories, indent=2)}

INSTRUCTIONS:

1. For each tool, assign it to the MOST APPROPRIATE category from the predefined list
2. If a tool could fit multiple categories, choose the PRIMARY function
3. Consider the tool's main purpose and typical use case
4. Return a JSON object mapping tool names to their categories

Return format:

```
{
  "tool_name_1": "Category Name",
  "tool_name_2": "Category Name",
  ...
}
```

ONLY return the JSON object with no additional text.

Figure 12: Prompt for categorizing e-commerce tools into functional domains. **Example Variable Values:** `batch_tools` is a batch of 10-20 tools to categorize, each containing `name`, `description`, `arguments`, and `results` fields (e.g., "get_order_details", "calculate_shipping_cost", etc.). `self.categories` is a predefined list of 15 functional domain categories: "Order Management", "Product Management", "Inventory Management", "Shipping & Logistics", "Payment Processing", "Customer Management", "Returns & Refunds", "Pricing & Promotions", "Analytics & Reporting", "Search & Discovery", "Reviews & Ratings", "Notifications & Communication", "Authentication & Security", "Marketplace Integration", and "Content Management".

Prompt for Generating Tool Simulation Outputs

Generate a realistic JSON output for this e-commerce tool call.
QUERY: "{context['query']}"
CURRENT STEP: "{context['step']}"
TOOL:

- Name: {tool_name}
- Description: {tool_def.get('description', 'No description available')}

ARGUMENTS USED: {json.dumps(args, indent=2)}
PREVIOUS TOOL OUTPUTS: {json.dumps(context['previous_outputs'], indent=2) if context['previous_outputs'] else "No previous outputs"}
REQUIRED OUTPUT PROPERTIES: {json.dumps(result_props, indent=2)}
EXAMPLE FORMAT: {json.dumps(example_output, indent=2)}
COMPLEXITY LEVEL: {complexity} {complexity_guidance}
INSTRUCTIONS:

1. Generate SPECIFIC, REALISTIC values consistent with the original query
2. Maintain CONSISTENCY with previous tool outputs
3. Include ALL required properties in the tool's result schema
4. Values should match their expected data types
5. {"This is the FINAL TOOL in the plan. Make sure the output provides a DIRECT RESULT that clearly resolves the user's request (e.g., confirmation, status, or answer)" if is_final_step else "Keep output focused and relevant to the step"}

RETURN ONLY THE JSON OUTPUT OBJECT.

Figure 13: Prompt for generating tool simulation outputs. **Example Variable Values:** `context['query']` is the original e-commerce query text; `context['step']` is the current step description from the plan; `tool_name` is the name of the tool being called; `tool_def` is the tool definition object containing description and other metadata; `args` is a JSON object containing the arguments passed to the tool; `context['previous_outputs']` is a JSON object or array containing outputs from previously executed tools (null or empty if no previous outputs); `result_props` is a JSON schema object defining required output properties with types and descriptions; `example_output` is a sample JSON object showing the expected output format; `complexity` is one of "SIMPLE", "MODERATE", "COMPLEX", or "ADVANCED"; `complexity_guidance` is text providing specific guidance for generating outputs at the given complexity level; `is_final_step` is a boolean indicating whether this tool is the final step in the plan (affects instruction 5).

C.2 An Example of SLATE Data

This section presents a representative example from the SLATE dataset, illustrating the transformation from a user query to a hierarchical execution plan, associated tool calls, and deterministic simulation outputs. The example demonstrates the plan structure and execution semantics used for step-wise and plan-level evaluation.

Case

I want to create a limited-time summer promotion for my **"ThermoFlex Water Bottle"** (**SKU:** `TF-WB-2023`) that offers **15% off** when customers **buy 2 or more units**. The promotion should run from **June 1, 2024** to **August 31, 2024**, with a minimum purchase requirement of **$35.00**. How do I set this up and create a promo code **"SUMMERTF24"** that customers can use at checkout?

Plan

- **step**: 1. Verify product information
  - **step**: 1.1 Retrieve product details
  - **action**: `get_product_details` (**sku**=`"TF-WB-2023"`)
- **step**: 2. Create the promotion
  - **step**: 2.1 Set up promotion parameters
  - **action**: `create_promotion` (**product_id**=`OUTPUT_FROM_STEP_1.1.product_id`, **discount_percentage**=15, **min_quantity**=2, **min_purchase**=35.00, **start_date**=`"2024-06-01"`, **end_date**=`"2024-08-31"`)
- **step**: 3. Configure promo code
  - **step**: 3.1 Create promo code
  - **action**: `create_promo_code` (**promotion_id**=`OUTPUT_FROM_STEP_2.1.promotion_id`, **code**=`"SUMMERTF24"`)
- **step**: "4. Verify and activate the promotion"
  - **step**: "4.1 Validate promotion setup"
  - **action**: `validate_promotion` (**promotion_id**=`OUTPUT_FROM_STEP_2.1.promotion_id`)
  - **step**: "4.2 Activate the promotion"
  - **action**: `activate_promotion` (**promotion_id**=`OUTPUT_FROM_STEP_2.1.promotion_id`, **promo_code_id**=`OUTPUT_FROM_STEP_3.1.promo_code_id`)

Figure 14: Example of plan generation and tool execution.

Simulate Outputs with context

- get_product_details (**sku**="TF-WB-2023")
  ⇒ Output: **product_id**="P-TF-WB-2023-001"
- create_promotion (**product_id**="P-TF-WB-2023-001", **discount_percentage**=15, **min_quantity**=2, **min_purchase**=35.00, **start_date**="2024-06-01", **end_date**="2024-08-31")
  ⇒ Output: **promotion_id**="PROMO-TF-2024-S001"
- create_promo_code (**promotion_id**="PROMO-TF-2024-S001", **code**="SUMMERTF24")
  ⇒ Output: **promo_code_id**="PC-SUMMERTF24-001"
- validate_promotion (**promotion_id**="PROMO-TF-2024-S001")
- activate_promotion (**promotion_id**="PROMO-TF-2024-S001", **promo_code_id**="PC-SUMMERTF24-001")
  ⇒ Output: **success**="true"

Deterministic Simulation Dictionary

1. get_product_details (Args: sku)
TF-WB-2023 ⇒ P-TF-WB-2023-001
Other value ⇒ different default value

2. create_promotion (Args: product_id, discount_percentage, min_quantity, min_purchase, start_date, end_date)
P-TF-WB-2023-001, 15, 2, 35, 2024-06-01, 2024-08-31 ⇒ PROMO-TF-2024-S001
Other value ⇒ different default value

3. create_promo_code (Args: promotion_id, code)
PROMO-TF-2024-S001, SUMMERTF24 ⇒ PC-SUMMERTF24-001
Other value ⇒ different default value

Figure 15: Deterministic simulation dictionary for tool calls.

D Related Work

D.1 Plan-Guided Tool Utilization

This appendix first formalizes plan-guided tool utilization, then reviews benchmarks and agent strategies most closely related to SLATE and EGB.

We formulate plan-guided tool utilization as a Markov Decision Process (MDP) (Bellman and Dreyfus, 2015). Given a case query $q \in \mathcal{Q}$, its corresponding plan p, and a tool library $\mathcal{T}$, an algorithm $\mathcal{A}$ sequentially processes each substep $ss_{i,j}$ specified by p. At each substep, the policy π selects an action $a_{i,j}$ consisting of a tool choice $t_{i,j} \in \mathcal{T} \cup \{\text{NO_OP}\}$ and its corresponding arguments. The policy conditions on the execution history $H_{i,j}$, written as $\pi(a_{i,j} \mid H_{i,j})$, where $H_{i,j} = \langle (hs_r, ss_{r,s}, a_{r,s}, o_{r,s}) \rangle_{(r,s) \prec (i,j)}$ records all previous high-level steps, substeps, actions, and observations. Here, $(r, s) \prec (i, j)$ denotes all index pairs preceding substep (i, j) in the plan execution order, and each observation $o_{r,s}$ is returned by the tool simulator $\mathcal{S}$ after executing action $a_{r,s}$. This formulation highlights the central distinction between approaches that commit to a complete action sequence in advance and approaches that adapt decisions using intermediate execution feedback.

D.2 Benchmarks and Evaluation Frameworks for Tool Use

Existing tool-use benchmarks can be grouped by environment scale and evaluation granularity. One line studies compact interactive environments, such as τ-BENCH (Yao et al., 2024a) and ALFWORLD (Shridhar et al., 2020), where the available tool or action space is limited to a few dozen predefined options. These environments enable controlled evaluation of decision-making and action selection, but their limited scale does not fully stress-test general-purpose agents operating over realistic, large tool libraries.

A second line of work targets large-scale tool selection and API use (Huang et al., 2023; Tang et al., 2023; Patil et al., 2024; Li et al., 2023a; Huang et al., 2024; Qin et al., 2023). These benchmarks expand candidate tool spaces and improve realistic coverage, but they often rely on step-level metrics, such as tool awareness and retrieval accuracy, or proxy plan-level signals, most commonly subjective LLM-judge ratings. As a result, they do not fully capture end-to-end execution correctness for long-horizon plans. SLATE addresses this gap by emphasizing plan-level execution with a large tool library and a simulator that supports controlled evaluation of whether the complete sequence of tool invocations satisfies the task.

D.3 Reasoning, Acting, and Search Strategies

Under this MDP formulation, tool-use agents fall into three increasingly interactive families. *Reasoning-based* methods generate a complete action sequence $\{a_{i,j}\}$ from a self-contained prompt, decoupling deliberation from execution and committing to a fixed plan without using intermediate observations $\{o_{i,j}\}$. Early methods such as Chain-of-Thought (CoT) (Wei et al., 2022) and Self-Consistency (SC) (Wang et al., 2022) established core reasoning capabilities, while Tree-of-Thought (ToT) (Yao et al., 2023a) and Reasoning via Planning (RAP) (Hao et al., 2023) extended this paradigm by exploring multiple cognitive paths before selecting an answer or action sequence.

Acting-based methods instead treat task completion as an interactive process, where the policy $\pi(a_{i,j} \mid H_{i,j})$ continuously incorporates environmental feedback. ReAct (Yao et al., 2023b) introduced the iterative *Thought-Action-Observation* loop, using observations from the environment or simulator $\mathcal{S}$ to guide subsequent decisions. Later work improves this loop through self-refinement (Madaan et al., 2023) and reflective feedback, as in Reflexion (Shinn et al., 2023), where case-level failures trigger a reflection step that produces guidance for future attempts.

Search-based methods further extend acting-based agents by explicitly exploring the state-action space before committing to decisions. LATS (Zhou et al., 2023) and REX (Murthy et al., 2023) adapt classical search algorithms, including Monte Carlo Tree Search (MCTS), to guide agents through multi-step decision spaces and evaluate the long-term consequences of candidate actions through environment-aware rollouts. However, near-exhaustive exploration becomes computationally prohibitive in long-horizon tasks and large tool libraries. This trade-off between search completeness and inference efficiency motivates EGB, which uses model uncertainty to branch selectively at high-entropy decision points rather than searching uniformly across all steps.