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 (Synthetic Large-scale API Toolkit for E-commerce), a large-scale context-aware benchmark designed for the automated assessment of tool-integrated agents. Unlike static metrics, SLATE accommodates diverse yet functionally valid execution trajectories, revealing 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 optimizes the exploration-exploitation trade-off, significantly enhancing both task success rates and computational efficiency. Extensive experiments on SLATE demonstrate that our dual contribution provides a robust foundation for developing reliable and scalable LLM agents in tool-rich environments.

1 Introduction

The rapid progress of Large Language Models (LLMs) has substantially enhanced the reasoning and decision-making capabilities of AI agents (Wang et al., 2024; Guo et al., 2024; Xi et al., 2025), enabling them to autonomously interact with external tools and APIs to solve real-world tasks (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).

Despite this potential, the transition toward context-grounded, long-horizon tool-use planning reveals a significant performance gap. In these sophisticated scenarios, agents are required to navigate massive tool libraries and execute multi-step plans that involve intricate dependencies (Qin et al., 2023). We argue that the development of robust tool-use agents is primarily bottlenecked by two intertwined challenges. The first is the *absence of rigorous, plan-level evaluation frameworks* capable of assessing agents in high-complexity, non-deterministic settings (Qu et al., 2025). The second challenge stems from the *limitations of existing agentic methods in navigating the vast decision spaces* inherent to these tasks, which often results in computational inefficiency and reduced autonomy (Huang et al., 2023; Xu et al., 2025).

The first challenge lies in evaluation. Current benchmarks are inadequate, primarily in two respects. Many employ limited toolsets that do not reflect the scale or diversity of real-world applications (Yao et al., 2022, 2024; Shridhar et al., 2020). Conversely, benchmarks that do incorporate large tool libraries often assess only single-step tool invocation, neglecting the sequential dependencies crucial for multi-step task execution (Tang et al., 2023; Patil et al., 2024; Li et al., 2023). Furthermore, attempts at plan-level evaluation frequently depend on subjective LLM-as-a-judge assessments (Qin et al., 2023; Huang et al., 2024), which cannot reliably measure an agent's true end-to-end task completion capabilities. This is especially problematic as such methods may penalize valid, alternative solution paths (for instance, those with redundant but harmless tool calls or different but effective execution orders), thereby failing to capture a holistic view of agent proficiency.

Complementary to the evaluation gap, the sec-

ond challenge pertains to the intrinsic limitations of current agentic algorithms in navigating expansive decision spaces. While foundational methods like ReAct (Yao et al., 2023b) and self-reflection (Shinn et al., 2023; Madaan et al., 2023) integrate reasoning with environmental feedback, they often struggle to systematically explore the solution space of long-horizon tasks. To address this, an alternative line of research treats the reasoning process as explicit planning by employing search algorithms to navigate decision trees (Koh et al., 2024; Ye et al., 2025). Prominent among these are methods based on Monte Carlo Tree Search (MCTS), which aim to balance exploration and exploitation in decision-making (Zhou et al., 2023; Murthy et al., 2023; Herr et al., 2025). However, the efficacy of these search-based approaches is frequently hindered by prohibitive computational costs, especially in scenarios involving large toolsets and long planning horizons. Consequently, despite ongoing efforts to optimize this trade-off, current methods lack the efficiency and reliability required for practical deployment in complex environments.

We address these intertwined deficiencies by introducing a rigorous evaluation framework and a principled search strategy for tool-augmented agents. We first present SLATE (Synthetic Large-scale API Toolkit for E-commerce), a large-scale, context-aware benchmark for objective plan-level evaluation. Each SLATE instance consists of a query, an executable plan, and an associated toolset, supported by context-grounded simulation outputs for valid invocations and explicit default outputs for invalid ones, enabling automated end-to-end evaluation at scale while accommodating trajectory diversity. Leveraging SLATE, we conduct a systematic study of representative agent architectures, revealing three key findings: (1) agents struggle to self-correct under binary execution feedback in large action spaces; (2) execution history improves tool and argument selection but has limited influence on the decision of whether to invoke a tool; (3) existing search-based methods, particularly MCTS-based approaches, exhibit an unfavorable utility-to-computation trade-off in long-horizon settings. Motivated by these insights, we propose Entropy-Guided Branching (EGB), an uncertainty-aware search algorithm that selectively allocates computation by branching only when predictive entropy over tool choices is high, while following a greedy path under confidence. This adaptive strategy improves the exploration–exploitation balance, reducing unnecessary search overhead and increasing end-to-end task success. Together, SLATE, our empirical diagnosis, and EGB establish a more rigorous foundation for reliable LLM agents operating in large action spaces and long-horizon tasks.

2 Related Work

2.1 Benchmarks and Evaluation Frameworks for Tool Use in LLMs

Existing benchmarks for tool-augmented language models primarily fall into two categories. The first includes small-scale environments such as τ-BENCH (Yao et al., 2024) and ALFWORLD (Shridhar et al., 2020), where the tool or action space is limited to a few dozen predefined options. While these settings support controlled evaluations of decision-making and action selection, they lack the scale and complexity required to assess general-purpose agents operating over realistic tool libraries. The second category comprises benchmarks designed for large-scale tool selection (Huang et al., 2023; Tang et al., 2023; Patil et al., 2024; Li et al., 2023; Huang et al., 2024; Qin et al., 2023). These benchmarks typically rely either on step-level metrics (e.g., tool awareness, retrieval accuracy) or on proxy signals for plan-level evaluation, most commonly subjective judgments from LLM judges. However, they do not adequately account for end-to-end execution correctness on long-horizon plans, which is essential.

2.2 Reasoning and Search in Tool-Augmented Agents

Research on LLM agents has transitioned from eliciting internal reasoning to navigating complex interactions with external tools. Early techniques such as Chain-of-Thought (CoT) (Wei et al., 2022) and Self-Consistency (SC) (Wang et al., 2022) established foundational reasoning capabilities. These were later extended via structured search methods such as Tree-of-Thought (ToT) (Yao et al., 2023a) and Reasoning via Planning (RAP) (Hao et al., 2023) to explore diverse cognitive paths.

A distinct line of research grounds agent decisions in external environments. The ReAct framework (Yao et al., 2023b) pioneered the interleaving of reasoning with tool execution, a paradigm later refined by mechanisms for self-refinement (Madaan et al., 2023) and reflective feedback (Shinn et al., 2023) that enable agents to learn from historical errors and environmental observations. As tool libraries scale, tool selection is increas-

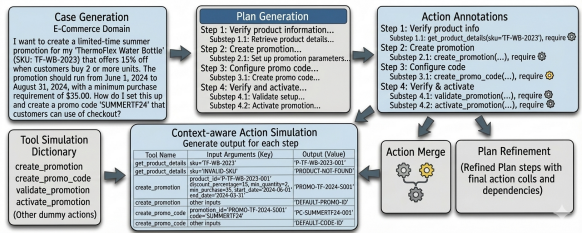

Figure 1: **Illustration of the SLATE dataset structure.**

ingly framed as a formal search problem. Methods such as LATS (Zhou et al., 2023) and REX (Murthy et al., 2023) adapt classical algorithms like Monte Carlo Tree Search (MCTS) to guide agents through multi-step decision spaces. However, the prohibitive computational cost of near-exhaustive exploration in long-horizon tasks remains a critical bottleneck. This tension between search completeness and efficiency motivates our development of Entropy-Guided Branching (EGB), an adaptive and uncertainty-aware search strategy.

Table 1: SLATE Dataset Statistics

Total Tools	1000	**Relevant Tools in Plans**	253
Avg. Arguments/Tool	3.05	**Avg. Calls/Tool**	6.30

3 SLATE Dataset Construction

To address the limitations of existing benchmarks, we introduce SLATE by grounding its construction in comprehensive planning-trajectory factual contexts. This section details the development of the SLATE dataset, which is specifically engineered to reflect the complexities of real-world e-commerce scenarios. These environments naturally require long-horizon reasoning across intricate dependencies, such as comparative shopping or inventory-aware checkout processes (Yao et al., 2022; Shridhar et al., 2020). Furthermore, tasks in this domain typically operate over expansive API libraries that span diverse functionalities ranging from product retrieval to logistics management, which provides a realistic testbed for agent scalability in large decision spaces (Qin et al., 2023; Qu et al., 2025).

The resulting dataset comprises user queries paired with multi-step solution plans, precise tool annotations, and simulated execution results for each intermediate step. By providing these grounded execution traces, SLATE facilitates a rigorous assessment of tool-augmented LLM agents, supporting both step-wise verification and end-to-end plan-level evaluation at scale. Our construction pipeline consists of the following stages: query collection, plan generation, tool annotation and normalization, plan refinement, tool simulation, and manual post-processing. This systematic workflow ensures that each trajectory is both logically sound and executionally valid. Detailed prompt designs for each stage are provided in App. D.

3.1 Definition

Let $\mathcal{D} = (\mathcal{Q}, \mathcal{P}, \mathcal{T}, \mathcal{S})$ denote the SLATE dataset, where $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ is a set of n complex user queries, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ is the corresponding set of structured, hierarchical plans (Huang et al., 2024), $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ is a library of m tools, and $\mathcal{S}$ is a context-aware tool simulator. Each query $q_i \in \mathcal{Q}$ is a natural language instruction representing a complex task. Its associated plan $p_i \in \mathcal{P}$ is represented as a sequence of triplets $p_i = [(hs_1, ss_{1,1}, \text{tool}_{1,1}), \ldots, (hs_k, ss_{k,l}, \text{tool}_{k,l})]$, where hs_i denotes a high-level step (e.g., "i"), written in natural language and describing an intermediate subgoal, while $ss_{i,j}$ denotes a finer-grained substep (e.g., "i,j") that may invoke a tool calling $\text{tool}_{i,j} \in \mathcal{T}$ or indicate "No Tool Required." Each tool $t \in \mathcal{T}$ is defined in function-call format as $t(\text{arg}_1, \text{arg}_2, \ldots)$, with arguments specified at runtime. Since directly implementing all tools within a large-scale interactive environment is often infeasible, SLATE incorporates a simulator $\mathcal{S}$ to approximate the behavior of tool executions. The simulator is designed to generate coherent outputs for each tool call conditioned on the input query and the plan context, thereby enabling consistent and interpretable execution traces. This simulation mechanism plays a crucial role in supporting plan-level evaluation: unlike prior benchmarks that rely on LLM-as-a-judge assessments or isolated step-wise accuracy, our simulator allows objective, automated measurement of end-to-end plan success. Fig. 1 illustrates the dataset structure, and Table 1 gives summary statistics.

3.2 Query Collection

To ensure that the SLATE dataset captures realistic tool-use demands, we focus on domains that inherently require large-scale tool invocation. We select the e-commerce domain as our target setting due to its diverse and interdependent task structure involving frequent interactions between buyers and sellers. This domain naturally necessitates the coordination of numerous APIs and tool calls, making it an ideal testbed for evaluating tool-augmented language agents.

To balance diversity and topical coherence, we

design the query set to cover a broad spectrum of task types while maintaining internal consistency across scenarios. Specifically, we collaborate 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 select a category and employ Claude 3.7 to generate and progressively complicate the query, ensuring both linguistic richness and operational complexity. The prompt template used for generation is provided in Appendix D, with illustrative examples shown in Appendix A. Fig. 2 shows the distribution of tools over the categories.

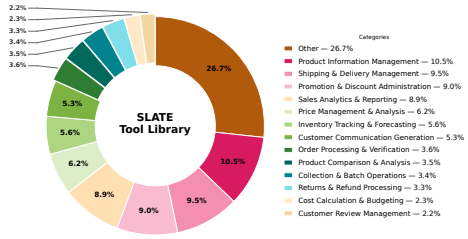

Figure 2: **SLATE Tool Library category distribution.**

3.3 Hierarchical Execution Plans with Tool Invocations

Given a complex natural language query $q \in \mathcal{Q}$, we construct a hierarchical execution plan $p \in \mathcal{P}$ in conjunction with a coherent toolkit $\mathcal{T}$, where each substep of the plan is grounded through suitable tool interactions. Our design follows a top-down reasoning paradigm inspired by human problem-solving: the query is recursively decomposed into high-level goals, each of which is further refined into executable substeps. This decomposition enables interpretable multi-stage resolution and systematic tool grounding aligned with task semantics.

The construction pipeline consists of two key components: (1) Hierarchical Plan Generation, where each query is formulated as an ordered sequence of high-level goals and corresponding substeps; and (2) Tool Definition and Grounding, where tools are identified, created, or reused to support the execution of substeps in a coherent and reusable manner.

Hierarchical Plan Generation. Given a query q, we use Claude 3.7 to generate a multi-step plan $[(hs_1, ss_{1,1}), (hs_1, ss_{1,2}), \ldots, (hs_k, ss_{k,l})]$, where hs_i denotes a high-level objective, and $ss_{i,j}$ represents a substep that may or may not invoke a tool. Some substeps are purely summarization or decision steps without explicit tool usage. Notably, plan generation is independent of any toolset to ensure that the resulting structure is free from any tool-specific bias, solely reflects the natural decomposition of the query.

Tool Definition and Grounding. For each substep requiring tool support, we define a tool in a normalized schema specifying its name, input arguments, functionality, and output format. Several principles guide the tool creation process. First, tool definitions must be specific and not trivially match substep descriptions; overly generic tools or name overlaps with substep text diminish the challenge of tool selection. Second, execution plans often exhibit strong sequential dependencies: the inputs to tools in later steps may rely on the outputs of earlier executions. We explicitly encode such dependencies by ensuring tool arguments are derivable from prior outputs or the original query context. Third, to promote tool reusability and avoid redundancy across queries, we introduce a tool deduplication strategy. Specifically, we embed all generated tool names and descriptions using the ModernBERT (Warner et al., 2025), and for each new query, we embed the query itself and retrieve the top-50 most semantically related tools from the existing library. The LLM is then encouraged to reuse existing tools where appropriate; otherwise, it generates new tools when novel functionality is required. After initial generation, we apply a tool normalization step to ensure consistency across similar tools. Tools are first grouped into functional categories, and within each category, the LLM merges tools with semantically equivalent functionality, reconciling input/output arguments to ensure compatibility. The execution plans are then updated to reflect any merged tool definitions, resulting in a coherent and unified plan-tool structure across the dataset.

3.4 Tool Simulator

Implementing every tool in a massive library is often infeasible. However, tool execution feedback is essential for end-to-end evaluation and providing a closed-loop signal for agentic planning. Since end-users are primarily concerned with final task resolution rather than the specific trajectory of tool calls, we prioritize an evaluation metric centered on execution success. To facilitate this at scale, we develop a Tool Execution Simulator.

The simulator's core function is to provide contextually appropriate outputs while differentiating between correct and incorrect invocations. If a tool call aligns with the ground-truth resolution, the simulator returns a context-coherent result; otherwise, it provides an uninformative response to signal a failed step. Our simulator, along with all other SLATE components, is generated using Claude 3.7 and consists of two primary modules:

(1) Context-Aware Simulation Database. This module serves as a repository of ground-truth execution traces. During dataset generation, each execution plan is annotated with the correct sequence of tool calls and their argument dependencies (e.g., `ToolA(arg1=OUTPUT_FROM_STEP_1)`). Claude 3.7 generates a coherent trace of simulated outputs for each plan, ensuring consistency by conditioning on the query, the full execution plan, and historical input-output pairs. These outcomes are stored in a lookup table structured as `(tool_name, arguments)` $\rightarrow$ `outcome`. For each tool, we also define a default, context-agnostic response for mismatched invocations.

(2) Semantic Equivalence Matching. To handle diverse runtime inputs, the simulator employs a semantic equivalence module to determine if an agent's tool call matches a ground-truth entry. This approach moves beyond simple string comparison, accounting for variations in format (e.g., "YYYY-MM-DD" vs. "MM/DD/YYYY") and semantic phrasing. If a match is successful, the simulator retrieves the corresponding context-aware outcome; otherwise, it returns the default failure signal. This architecture enables dynamic, realistic simulation for robust end-to-end evaluation.

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.

4 Preliminaries: Plan-Guided Tool Use

This section formally defines the task of *Plan-Guided Tool Utilization* and reviews the core reasoning and agentic paradigms.

4.1 Problem Formulation

We formulate the plan-guided tool utilization task 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 $\mathcal{A}$ must sequentially process each substep $ss_{i,j}$ defined in p. At each substep $ss_{i,j}$, the algorithm's policy π selects an action $a_{i,j}$, which consists of choosing a tool $t_{i,j} \in \mathcal{T} \cup \{\text{NO\_OP}\}$ and generating its corresponding arguments. The policy, $\pi(a_{i,j}|H_{i,j})$, conditions on the execution history $H_{i,j}$, defined as the sequence of outcomes from all preceding substeps: $H_{i,j} = \langle (hs_r, ss_{r,s}, a_{r,s}, o_{r,s}) \rangle_{(r,s) \prec (i,j)}$, where $(r,s) \prec (i,j)$ denotes all index pairs preceding substep (i,j) in the plan's execution order. Each observation $o_{r,s}$ is the result returned by a tool simulator $\mathcal{S}$ upon executing action $a_{r,s}$. Note that pure reasoning-based approaches commit to a sequence of actions without leveraging intermediate observations from the simulator.

4.2 Reasoning and Acting Strategies

Under the MDP framework defined above, agent strategies $\mathcal{A}$ can be broadly categorized into two paradigms. *Reasoning-based* approaches synthesize a complete action sequence $\{a_{i,j}\}$ from a single, self-contained prompt, committing to a fixed execution plan without incorporating intermediate observations from the simulator. In contrast, *acting-based* approaches tightly interleave reasoning with environmental interaction, enabling the agent to continuously incorporate execution feedback and adapt subsequent decisions on the fly.

Reasoning-based approaches decouple searching from execution, generating action sequences $\{a_{i,j}\}$ based on a self-contained, deliberative process without leveraging environmental feedback $\{o_{i,j}\}$.

In contrast, acting-based approaches treat task resolution as an interactive process, where the agent's policy $\pi(a_{i,j}|H_{i,j})$ is continuously informed by environmental feedback $\{o_{i,j}\}$. The foundational method, *ReAct* (Yao et al., 2023b), establishes an iterative cycle of *Thought-Action-Observation*, using feedback from the simulator $\mathcal{S}$ to inform the next step. *Reflexion* (Shinn et al., 2023) enhances this with a self-correction mechanism, where case-level failures trigger a reflective process to generate guiding principles for subsequent attempts. More sophisticated strategies employ lookahead search over the state-action space. For instance, MCTS-based methods like *LATS* (Zhou et al., 2023) build a search tree by using the simulator $\mathcal{S}$ for environment-aware rollouts, allowing the agent to evaluate the long-term consequences of its actions before making a decision.

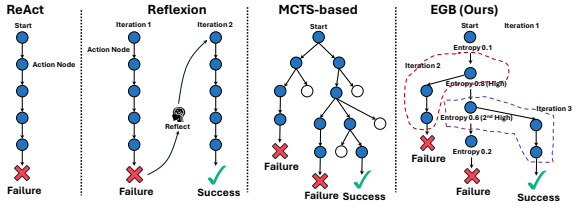

Figure 3: Comparison of different paradigms.

5 EGB for Long-Horizon Planning

We propose *Entropy-Guided Branching (EGB)* (illustrated in Fig. 3), an uncertainty-aware acting strategy designed to navigate expansive tool spaces through iterative trajectory refinement in black-box settings. 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 $\mathcal{S}$.

5.1 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. For each substep $ss_{i,j} \in p$, we assume a black-box setting where internal model logits are inaccessible. To quantify decision uncertainty, we generate m candidate actions $\{a_{i,j}^{(1)}, a_{i,j}^{(2)}, \ldots, a_{i,j}^{(m)}\}$ through independent samplings from the policy:

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

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. While EGB can utilize internal signals if available, as demonstrated in our experiments, this sampling-based approach ensures that our strategy remains fully compatible with black-box proprietary models. The initial execution continues until the agent either reaches a terminal state or the plan fails to achieve the goal.

5.2 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 $\{a_{i,j}^{(1\ldots m)}\}$ 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.

6 Experiments

We setup experiments on the SLATE synthetic dataset to evaluate EGB against representative baselines, and conduct studies on the impact of hyper-parameter 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)	$\mathbf{54.0 \pm 2.0}$	68.5 ± 1.5	87.9 ± 0.2

6.1 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

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)	$\mathbf{51.3 \pm 6.0}$	33.6 ± 2.4	83.0 ± 0.6
EGB-Logits (Ours)	$\mathbf{67.8 \pm 4.5}$	36.4 ± 0.8	85.1 ± 0.4

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 entironment-aware rollouts.

Model: We conduct primary experiments on Claude-4-Sonnet, 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, an open-source model with accessible logits. We selected Qwen2.5-7B based on preliminary experiments showing it achieves the best performance on both instruction-following capability and entropy-uncertainty correlation among models of comparable size. We use AWS bedrock accessing the Claude model and 5 NVIDIA L4 GPUs for Qwen model 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.

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.

6.2 Experimental Results

Results on a Proprietary Model: We first conduct comprehensive experiments on Claude-4-Sonnet 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 (2.1%-5.1% on Tool Match Rate), reveals a critical limitation of baseline methods: without access to intermediate step-level feedback from the simulator, they struggle to identify error-prone decisions under constrained computation. Unlike EGB, which leverages internal policy uncertainty (entropy) to pinpoint 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 LATS, which exhibit *decreased* Tool Match Rates compared to Baseline-LLM and ReAct, suggesting they modify previously correct tool selections into incorrect ones during their search process. Their plan-level improvements thus stem primarily from exploring more execution paths rather than intelligently targeting problematic decisions, resulting in occasional successes by chance rather than systematic error correction.

Results on an Open-source Model: To further validate our approach, we conduct experiments on Qwen-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

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 includes 44.6 generation calls and 149.6 lightweight forward passes. * Parallelization across all methods are disabled for Qwen experiments.

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 reinforces our core assumption that entropy serves as a reliable indicator of error-prone decisions. Besides, accurate entropy computation from logits proves more effective than sampling-based estimation.

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 (**9.4% faster than Reflexion**). EGB-Logits demonstrates the decent efficiency, requiring only 254 seconds (**2.8× 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 confirm that entropy-guided branching achieves strong performance without prohibitive computational overhead.

Impact of Hyperparameters: We systematically investigate EGB's two critical hyperparameters, as illustrated in Figure 4. 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 $b = 10$.

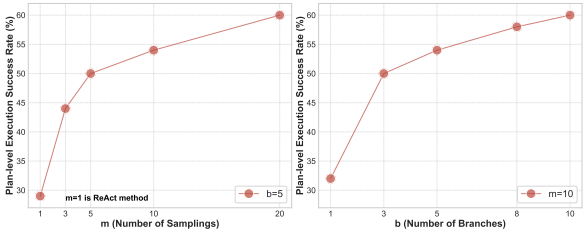

Figure 4: Evaluation of EGB search with Claude-3.5-Sonnet on the SLATE synthetic dataset with varied configurations: (1) number of samplings, m, to calculate entropy, and (b) budget, b, as the maximum numbers of iterations to try a different branch.

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 $b = 1$, although no error-correction branches are triggered, self-consistency estimation still provides measurable benefits. The key takeaway is that performance improves with both more accurate entropy estimation and increased branching opportunities for error correction.

7 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, 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.

8 Limitations

Despite the robust performance of EGB and the comprehensive nature of SLATE, several limitations warrant further investigation. First, while SLATE incorporates intricate sequential dependencies and context-aware responses, it remains a synthetic benchmark; real-world e-commerce APIs often involve non-deterministic environment states, transient network failures, and complex side effects that our simulator may only partially approximate. Second, although EGB-Sampling is fully compatible with black-box models, its reliance on multiple independent samplings ($m = 10$ or 20) to estimate entropy introduces higher inference latency and API costs compared to single-pass greedy methods, potentially limiting its application in real-time or resource-constrained scenarios. Third, this study focuses exclusively on the e-commerce domain; further research is needed to validate the generalizability of entropy-guided search in other tool-rich environments, such as software engineering or scientific research. Finally, since both the benchmark and simulator were generated via a closed-source LLM pipeline, the evaluation framework may inherit latent biases or systematic reasoning patterns inherent to the underlying model, which could affect the diversity of the generated edge cases.

9 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.

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*.

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*.

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*.

Nathan Herr, Tim Rocktäschel, and Roberta Raileanu. 2025. Llm-first search: Self-guided exploration of the solution space. *arXiv preprint arXiv:2506.05213*.

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*.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. 2024. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*.

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

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*.

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.

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*.

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.

Weikai Xu, Chengrui Huang, Shen Gao, and Shuo Shang. 2025. Llm-based agents for tool learning: A survey: W. xu et al. *Data Science and Engineering*, pages 1–31.

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. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv 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)*.

Guanghao Ye, Khiem Duc Pham, Xinzhi Zhang, Sivakanth Gopi, Baolin Peng, Beibin Li, Janardhan Kulkarni, and Huseyin A Inan. 2025. On the emergence of thinking in llms i: Searching for the right intuition. *arXiv preprint arXiv:2502.06773*.

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 Examples

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 5: 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 6: Deterministic simulation dictionary for tool calls.

B Supplementary Experiments

Table 5: Evaluation of search methods with vs. without memory on the SLATE synthetic dataset with Claude-4-Sonnet.

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

Impact of Memory. While EGB demonstrates strong performance through entropy-guided exploration, we investigate whether incorporating memory from previous task executions can further enhance its capabilities. To evaluate this, 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. As shown in Table 5, 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.

C EGB Algorithms

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**
  $\mathcal{T}_{\text{alt}} \leftarrow \{t : (t, p_t) \in \mathcal{D}_{i,j}, t \neq t^*_{i,j}\}$ {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 $\mathcal{D}_{i,j}$

$\texttt{votes} \leftarrow \{\}$ {Vote counts per tool}
$\texttt{calls} \leftarrow \{\}$ {Sampled tool calls per tool}
for $k = 1$ **to** m **do**
  $a^{(k)}_{i,j} \sim \pi(a_{i,j} \mid H_{i,j}, \mathcal{C}_{i,j})$ {Sample complete action (tool + params) from LLM}
  $t^{(k)} \leftarrow \textsc{ExtractToolName}(a^{(k)}_{i,j})$
  $\texttt{votes}[t^{(k)}] \leftarrow \texttt{votes}[t^{(k)}] + 1$
  $\texttt{calls}[t^{(k)}] \leftarrow \texttt{calls}[t^{(k)}] \cup \{a^{(k)}_{i,j}\}$ {Store sampled call for potential branching}
end for
// Compute entropy from vote distribution
for each tool $t \in \texttt{votes}$ **do**
  $p_t \leftarrow \texttt{votes}[t]/m$
end for
$E_{i,j} \leftarrow -\sum_t p_t \log p_t$
// Select majority tool (use pre-sampled call)
$t^* \leftarrow \arg\max_t \texttt{votes}[t]$
$a^*_{i,j} \leftarrow \textsc{SelectOne}(\texttt{calls}[t^*])$ {Use one of the sampled calls}
$\mathcal{D}_{i,j} \leftarrow \{(t, p_t, \texttt{calls}[t]) : t \in \texttt{votes}\}$ {Store distribution with pre-sampled calls}
return $(a^*_{i,j}, E_{i,j}, \mathcal{D}_{i,j})$

D Prompt Templates

Algorithm 3 Entropy Computation via Logits (EGB-Logits): Qwen2.5 Adaptation

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 $\tau = 0.01$

$\texttt{prompt} \leftarrow \textsc{BuildPrompt}(q, ss_{i,j}, H_{i,j}, \mathcal{C}_{i,j})$ {Ask LLM to output tool index 0 to $K-1$}
$\mathbf{x} \leftarrow \textsc{Tokenize}(\texttt{prompt})$
// Get first token distribution over digits 0-9
$\mathbf{l}^{(1)} \leftarrow \textsc{ForwardPass}(\mathbf{x})[-1]$ {Logits at last position}
$\mathbf{p}^{(1)} \leftarrow \textsc{Softmax}(\mathbf{l}^{(1)}_{['0':'9']})$ {Probability over digit tokens}
// Compute conditional second token distribution
$\mathbf{P} \leftarrow \mathbf{0}^K$ {Probability for each candidate}
for $d_1 = 0$ **to** 9 **do**
  $\mathbf{x}' \leftarrow \textsc{Concat}(\mathbf{x}, \textsc{Token}(d_1))$
  $\mathbf{l}^{(2)} \leftarrow \textsc{ForwardPass}(\mathbf{x}')[-1]$
  $\mathbf{p}^{(2)} \leftarrow \textsc{Softmax}(\mathbf{l}^{(2)}_{['0':'9']})$
  $p_{\text{end}} \leftarrow 1 - \sum_{d_2=0}^{9} \mathbf{p}^{(2)}_{d_2}$ {Probability of non-digit token (sequence end)}
  if $d_1 < K$ **then**
    $\mathbf{P}[d_1] \leftarrow \mathbf{P}[d_1] + \mathbf{p}^{(1)}_{d_1} \cdot p_{\text{end}}$ {Single-digit index}
  end if
  for $d_2 = 0$ **to** 9 **do**
    $idx \leftarrow d_1 \times 10 + d_2$
    if $10 \leq idx < K$ **then**
      $\mathbf{P}[idx] \leftarrow \mathbf{P}[idx] + \mathbf{p}^{(1)}_{d_1} \cdot \mathbf{p}^{(2)}_{d_2}$ {Two-digit index}
    end if
  end for
end for
$\mathbf{P} \leftarrow \mathbf{P} / \sum_k \mathbf{P}[k]$ {Normalize}
// Compute entropy over all candidates
$E_{i,j} \leftarrow -\sum_{k=0}^{K-1} \mathbf{P}[k] \log \mathbf{P}[k]$
// Filter candidates by probability threshold for branching
$\mathcal{C}^{\text{filtered}}_{i,j} \leftarrow \{c_k : \mathbf{P}[k] \geq \tau\}$
// Select highest probability tool and generate parameters
$k^* \leftarrow \arg\max_k \mathbf{P}[k]$
$a^*_{i,j} \leftarrow \textsc{GenerateParams}(c_{k^*}, q, ss_{i,j}, H_{i,j})$ {Generate params only for selected tool}
$\mathcal{D}_{i,j} \leftarrow \{(c_k, \mathbf{P}[k]) : c_k \in \mathcal{C}^{\text{filtered}}_{i,j}\}$ {Params generated lazily during branching}
return $(a^*_{i,j}, E_{i,j}, \mathcal{D}_{i,j})$

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."
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.

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)

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).