

AutoTool: Efficient Tool Selection for Large Language Model Agents

Jingyi Jia, Qinbin Li*

School of Computer Science and Technology
Huazhong University of Science and Technology
{jingyijia, qinbin}@hust.edu.cn

Abstract

Large Language Model (LLM) agents have emerged as powerful tools for automating complex tasks by leveraging the reasoning and decision-making abilities of LLMs. However, a major bottleneck in current agent frameworks lies in the high inference cost of tool selection, especially in approaches like ReAct that repeatedly invoke the LLM to determine which tool to use at each step. In this work, we propose AutoTool, a novel graph-based framework that bypasses repeated LLM inference by exploiting a key empirical observation: tool usage inertia—the tendency of tool invocations to follow predictable sequential patterns. AutoTool constructs a directed graph from historical agent trajectories, where nodes represent tools and edges capture transition probabilities, effectively modeling the inertia in tool selection. It further integrates parameter-level information to refine tool input generation. By traversing this structured representation, AutoTool efficiently selects tools and their parameters with minimal reliance on LLM inference. Extensive experiments across diverse agent tasks demonstrate that AutoTool reduces inference costs by up to 30% while maintaining competitive task completion rates, offering a practical and scalable enhancement for inference-heavy frameworks. Our work highlights the promise of integrating statistical structure into LLM agent design for greater efficiency without sacrificing performance.

Code — <https://github.com/jiajingyyyyyy/AutoTool>

1 Introduction

Large Language Models (LLMs) have experienced explosive growth, demonstrating impressive capabilities in various tasks (Achiam et al. 2023; Dubey et al. 2024b; Yang et al. 2025) from natural language understanding (Devlin et al. 2019) and reasoning (Guo et al. 2025) to automation of complex workflows (Wang et al. 2025). LLM-powered agents, which leverage these capabilities for interactive (Yi et al. 2024) and decision-making (Wei et al. 2025) tasks, have become increasingly prevalent in numerous domains, including software development (Jin et al. 2024), intelligent personal assistants (Li et al. 2024) and scientific research automation (Team et al. 2025).

However, despite their versatility, a significant drawback of current LLM-based agents is the substantial computational overhead (Kim et al. 2025), particularly evident in frameworks like ReAct (Yao et al. 2023b) that involve many LLM inferences. Among these inference processes, one major goal is to repeatedly infer appropriate tools to use based on dynamic contexts, leading to high inference costs and latency. Given these challenges, a natural question arises: *Can we utilize statistical methods instead of relying heavily on LLM inference to select tools efficiently?*

Addressing this issue, we empirically observe a critical phenomenon termed *tool usage inertia*, where tool selections demonstrate predictable sequential patterns. For instance, when an agent searches in an academic database, the invocation of *AuthorNodeCheck* is often followed by *LoadAuthorNet* to retrieve detailed information. This sequential dependence is observable across diverse agent tasks (Qin et al. 2023; Ma et al. 2024), confirming that prior tool selections significantly influence subsequent choices. Such inertia is widely present in many domain tasks, where LLM agents are usually applied.

Inspired by these insights, we introduce **AutoTool**, a novel graph-based method for automatic tool selection in LLM agents. AutoTool constructs a graph representation capturing the observed inertia in tool selection from historical workflows, where nodes correspond to tools and edges encode observed sequential dependencies. Additionally, AutoTool intelligently integrates parameter-level information into this graph, thereby enabling automated parameter filling. By efficiently traversing this structured representation, AutoTool selects the appropriate tools and parameters significantly faster than traditional inference-based approaches. Experimental evaluations demonstrate that AutoTool substantially reduces token consumption and LLM call counts while maintaining comparable agent task progress rates to LLM-based methods.

Our contributions are summarized as follows:

- We empirically identify and analyze the phenomenon of tool usage inertia in LLM-based agents, both in tool selection and parameter filling.
- We design a method to construct an inertia-aware tool graph that captures sequential patterns and data flow in agent behavior.
- We develop a graph-based selection algorithm that effi-

*Corresponding Author

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

ciently determines the next tool and its parameters with minimal LLM intervention.

- We conduct extensive experiments showing that AutoTool achieves significant reductions in LLM inference cost while preserving task performance.

2 Background and Related Work

2.1 LLM Agent Frameworks

LLM agents have significant potential in solving complex problems, primarily through effective task planning, reasoning, and interaction with external tools (Qin et al. 2024; Qu et al. 2025). The seminal work ReAct (Yao et al. 2023b) introduces the core paradigm of driving agent decisions through interleaved “Thought-Act-Observe” cycles, which has become a cornerstone for numerous open-source frameworks, including Langchain (LangChain 2023) and MetaGPT (Hong et al. 2023). Subsequent research has expanded agents’ capabilities to interact with a vast array of external tools, such as RESTful APIs in RestGPT (Song et al. 2023) or various AI models orchestrated by HuggingGPT (Shen et al. 2023). However, a shared limitation across these powerful frameworks is that the fundamental decision of which tool to use at each step still predominantly relies on a costly LLM inference (Belcak et al. 2025). This reliance creates a significant computational bottleneck, which is the primary issue our work aims to address.

2.2 Automated Tool Selection

Research in tool selection can be broadly categorized into two types: fine-tuning-dependent and tuning-free methods. Fine-tuning methods like Toolformer (Schick et al. 2023), Gorilla (Patil et al. 2024) and ToolRL (Qian et al. 2025) aim to improve intrinsic tool-use capabilities. While these approaches demonstrably enhance a model’s intrinsic ability to call tools correctly, their reliance on high-quality data or carefully crafted reward signals presents a significant barrier to scalability and adaptability. Tuning-free methods employ different strategies at runtime. Approaches like AnyTool (Du, Wei, and Zhang 2024) and ToolNet (Liu et al. 2024b) focus on improving retrieval completeness and handling large-scale APIs. Many search strategies are introduced to find the optimized action sequences, such as the DFSDT in ToolLLM (Qin et al. 2023), BFS in ITS (Koh et al. 2024), and the toolkit-based planning in ToolPlanner (Liu et al. 2024c).

2.3 Automated Workflow Generation

To accomplish many complex tasks, agents need to execute a sequence of interdependent actions. Some approaches focus on search and planning. For instance, Tree of Thoughts (Yao et al. 2023a) explores diverse action paths, while ToolChain (Zhuang et al. 2023) employs the A* search algorithm over a decision tree. Other works, such as LLM-Compiler (Kim et al. 2024), target execution efficiency by enabling parallel function calling. In parallel, another key approach is learning from past interactions: ART (Paranjape et al. 2023) automates multi-step reasoning and tool use by retrieving and composing examples from a task library, whereas Agent Workflow Memory (Wang et al. 2024)

extracts reusable workflows to guide web agents in long-horizon tasks. Furthermore, several works have begun to apply graph-based methods to enhance planning and workflow automation (Zhuge et al. 2024; Wu et al. 2024). Although most of these methods primarily focus on improving success rates and workflow quality, their planning or search processes can be computationally intensive. A comparison between our method and related studies is summarized in Table 1.

3 Motivation

Observation 1: LLM Agents have high inference costs for tool selection The predominant focus in LLM agent research has been on maximizing task success rates, often leaving significant room for improvement in operational efficiency—a crucial factor for practical deployment. A multi-step task can trigger numerous LLM calls, posing significant challenges for real-time or resource-constrained applications.

We argue that this universal reliance on the LLM is not only costly but also *sub-optimal*. The core issue is that not all decision steps in a task are of equal complexity or importance. Many tool invocations, both for selection and parameter filling, occur in highly patterned or repetitive contexts that do not require the LLM’s full, nuanced reasoning power. This universal application of a resource-intensive model for both complex and simple decisions constitutes an over-utilization of its powerful abilities, creating unnecessary computational overhead.

Observation 2: Tool Invocation Exhibits Predictable, Low-Entropy Inertia Motivated by this efficiency challenge, we conducted an empirical analysis to test the hypothesis that tool usage is not a series of independent events, but rather a process governed by sequential patterns. We utilize a ReAct agent within the ScienceWorld environment to generate 322 trajectories, yielding 6014 tool invocations. Analyzing these sequences, we discover strong “Sequential Inertia”: tool selection is not independent but follows predictable, low-entropy patterns.

We quantify this predictability by modeling the sequence as a k -th order Markov chain and measuring the reduction in conditional entropy (Shannon 1948). The analysis reveals a substantial drop in uncertainty: a 0-order model (assuming independence) yields a baseline entropy of 3.50 bits, this value decreases to 2.52 bits for a 1st-order model and drops further to 1.93 bits for a 2nd-order model.

To validate the statistical significance of these sequential dependencies, we performed likelihood ratio tests (G^2 -tests, $N = 6014$). The results confirm that each increase in model order yields a highly significant improvement in fit. Specifically, the transitions from a 0-order to a 1st-order model, and from a 1st-order to a 2nd-order model, are both statistically significant, with $G^2(361) = 9390.70$ and $G^2(3914) = 5437.03$ respectively ($p < .001$ for both).

We provide a rigorous theoretical foundation for AutoTool in **Appendix F**.

Empirically, this manifests as highly skewed successor distributions, as illustrated in Figure 1. The *go_to* action is followed by *look_around* in 88.7% of cases. Similarly, after the sequence *focus_on* \rightarrow *wait*, the next action is *look_around*

Feature	AutoTool	DFSdT	AnyTool	ToolChain	ToolNet	ToolPlanner	LLMCompiler
Efficiency	✓	✗	✓	✗	✓	✓	✓
LLM Offloading	✓	✗	✗	✗	✗	✗	✗
Inertia Aware	✓	✗	✗	✗	✗	✗	✗
Parameter Flow	✓	✗	✗	✗	✗	✗	✗
Tool Graph	✓	✗	✓	✓	✓	✗	✓

Table 1: A comparison of AutoTool with other tuning-free tool selection methods

Action	Src Tool	Src Param	%
use(target)	move	source	44.8
	pick_up	OBJ	22.1
	move	target	11.5
	pour	OBJ	11.0
	Other	–	10.6
pick_up(OBJ)	focus_on	OBJ	40.1
	move	source	24.3
	pour	from	16.4
	look_at	OBJ	5.9
	Other	–	13.3

Table 2: Parameter source analysis for the ‘use’ and ‘pick_up’ actions.

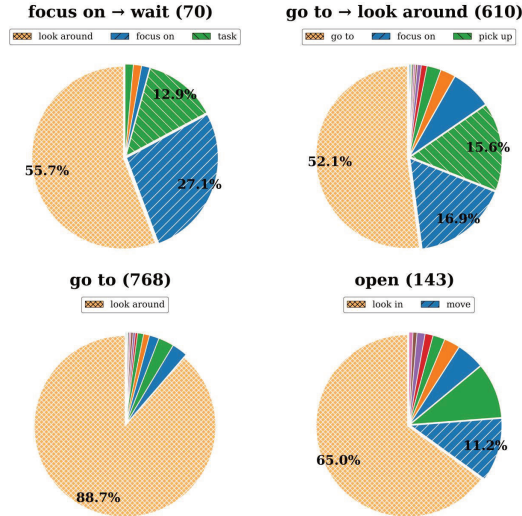


Figure 1: The distribution of successor tools for different tools or tool sequences.

with a high probability of 55.7%. These skewed distributions, where the next action often concentrates on one or two highly likely candidates, confirm the low-entropy nature of the task and demonstrate that non-LLM prediction is not only possible but highly feasible. Furthermore, the sources for directly transferred parameters are highly concentrated. As detailed

in Table 2, a narrow set of preceding actions often provides a substantial portion of the necessary inputs. For instance, the top source alone accounts for 44.8% of parameters for *use(target)* and 40.1% for *pick_up(OBJ)*.

This quantifiable inertia—as revealed through theoretical and empirical analysis in ScienceWorld—forms the foundation for our proposed method.

4 Methodology

4.1 Problem Statement

In LLM-driven agent systems, an agent selects an action $a_t = (\text{tool}_{t+1}, \text{params}_{t+1})$ at each timestep t based on an observation o_t , a task goal G , and a set of available tools \mathcal{T} . Current methods, such as ReAct, typically infer $a_t \sim p_{\text{LLM}}(a|o_t, G, \mathcal{T})$ via a complete LLM inference. This incurs significant computational costs, limiting their applicability in resource-constrained or real-time scenarios.

To address this limitation, we define our problem as follows: Given a dataset of trajectories $D_{\text{hist}} = \{\tau_1, \tau_2, \dots, \tau_N\}$ collected from historical task executions, where each trajectory $\tau_i = (o_0^{(i)}, a_0^{(i)}, o_1^{(i)}, a_1^{(i)}, \dots)$ records a sequence of observations and actions. Our objective is to construct M_{AutoTool} , a training-free decision-making algorithm based on the *Tool Inertia Graph*, which can selectively bypass the LLM for predictable actions to improve efficiency.

4.2 Overview

Figure 2 illustrates the AutoTool framework, which attempts an ‘inertial invocation’ before each standard LLM call to bypass costly inference. This process operates in two stages: first, the Inertia Sensing module (Fig. 2b) predicts the next likely tool by combining historical frequency and contextual relevance. If a tool is identified, the Parameter Filling module (Fig. 2c) then populates its arguments by backtracking the parameter flow on the graph. A tool is executed directly only if both stages succeed, thus bypassing a costly LLM inference. We introduce each module below and **defer more details and pseudo code of the algorithms to Appendix A due to page limit**.

4.3 Graph Construction

The **Tool Inertia Graph** (TIG) is a dynamic graph, denoted as $G_t = (V_t, E_t, W_t)$. It is incrementally constructed from execution trajectories and can also be bootstrapped with prior knowledge. The TIG’s node structure (V_t) is hierarchical:

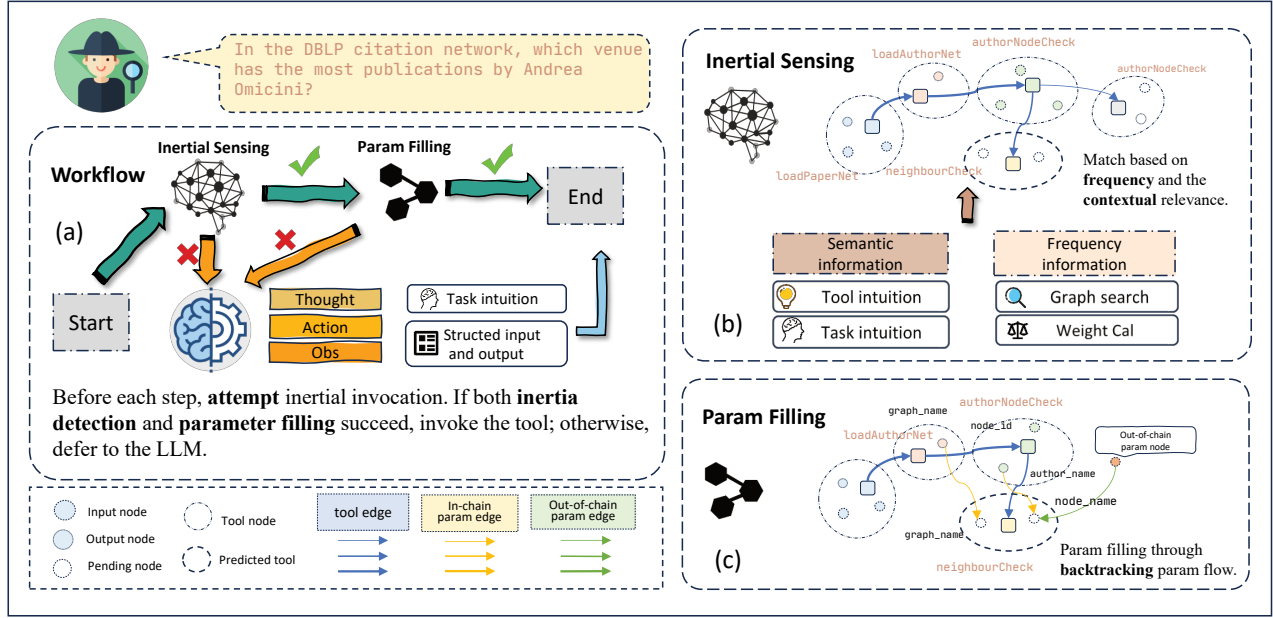


Figure 2: An overview of the AutoTool framework. This figure illustrates our proposed workflow, which initiates with the Inertia Sensing module predicting the next likely tool by exploiting historical usage patterns. If a high-confidence tool is identified, the Parameter Filling module then populates its required parameters. Only when both stages succeed is the tool executed directly via the inertial path, bypassing a costly LLM call. Otherwise, the system reverts to a standard LLM invocation.

- **Tool Nodes** (v_i): Primary nodes, each representing an available tool, $tool_k \in \mathcal{T}$. They store the tool’s functional description and track execution-level attributes, such as success or failure status. Notably, these nodes can be initially constructed from basic tool documentation alone, ensuring the graph’s scalability. Each Tool Node encapsulates a dynamic subgraph containing:
 - **Parameter Nodes** (p_i): Secondary nodes within the subgraph, each representing a specific input or output parameter of the parent Tool Node.

This hierarchical structure allows the TIG to meticulously track not only the sequence of tools but also the parameter-level data flows between them.

These nodes are connected by two types of directed edges (E_t):

- **Tool Sequence Edges** ($e_{ij}^{\text{tool}} = (v_i, v_j)$): Connect Tool Nodes to represent sequential dependencies.
- **Parameter Dependency Edges** ($e_{xy}^{\text{param}} = (p_x, p_y)$): Connect Parameter Nodes to model the data flow between tools.

The initial creation and continuous update of these edges and their corresponding weights (W_t) are detailed in Section 4.4. This online construction mechanism ensures the TIG promptly reflects the agent’s latest experiences and learned patterns, making inertial predictions both adaptive and effective.

4.4 Edge Filling

The TIG learns and adapts from execution trajectories by dynamically creating and updating the edges that model both sequential tool dependencies and parameter-level data flows.

Tool Sequence Edges. When a tool $tool_j$ is executed immediately after $tool_i$, a Tool Sequence Edge (e_{ij}^{tool}) is created between them if it does not exist, typically with an initial weight of 1. The edge’s weight is reinforced exclusively by high-confidence sequences generated from the LLM. This design choice prevents error propagation from potentially sub-optimal inertial calls.

More critically than just recording frequencies, AutoTool enables the TIG to differentiate between effective and ineffective pathways. Each sequence edge is associated with a posterior efficacy score, which is continuously updated based on execution feedback. If an inertia-driven tool call is successful (judged by environmental feedback or task progression), the weight of the corresponding edge is incremented. Conversely, a failure penalizes the edge by decrementing its weight. This online learning mechanism allows the TIG not only to record historical co-occurrence frequencies but also to learn the *actual effectiveness* of tool sequences, enabling robust adaptation to changing task dynamics.

Parameter Dependency Edges. Tracking parameter flow is crucial for automating parameter filling. When a tool is invoked, our framework parses its structured input and output. It then backtracks through the execution history to identify the source of each input parameter. When a parameter of the

current tool inherits its value from any parameter of a preceding tool, a Parameter Dependency Edge (e_{xy}^{param}) is established or reinforced between the corresponding Parameter Nodes. These edges encode recurring data transfer patterns, laying the groundwork for efficient, non-LLM parameter filling.

4.5 Graph Searching

At each decision step, instead of immediately invoking the LLM, AutoTool first performs a graph search to attempt an inertial call. This process involves two sequential stages: tool selection and parameter filling.

Tool Selection via CIPS First, AutoTool searches the TIG to identify candidate tools that have historically followed the sequence of the most recent k tools. For each candidate, we calculate a **Comprehensive Inertia Potential Score (CIPS)**, which balances historical patterns with the current task context:

$$\text{CIPS} = (1 - \alpha) \cdot \text{Score}_{\text{freq}} + \alpha \cdot \text{Score}_{\text{ctx}} \quad (1)$$

The **Frequency Score** ($\text{Score}_{\text{freq}}$) quantifies historical usage patterns extracted from TIG edge weights, while the **Contextual Score** ($\text{Score}_{\text{ctx}}$) evaluates semantic alignment between the agent’s current intuition and the candidate tool’s description. We select the tool v^* achieving the highest CIPS. If this score surpasses the predefined threshold ($\text{CIPS}(v^*) > \theta_{\text{inertial}}$), the process proceeds to parameter filling. Otherwise, the inertial attempt aborts and control reverts to the standard LLM inference module.

Hierarchical Parameter Filling If a tool candidate v^* passes the threshold, AutoTool attempts to populate its required parameters using a hierarchical, non-LLM strategy that follows a strict priority order, ensuring that the most reliable information sources are always prioritized.

The primary method is dependency backtracking, which traverses the parameter dependency edges in the TIG to find an input’s source in a preceding tool’s output. If this fails, the framework attempts environmental state matching, using key states maintained by the agent (e.g., current location). As a final non-LLM attempt, it resorts to heuristic filling based on the agent’s current state or task goal. The inertial call is executed only if all required parameters are successfully populated through this hierarchy. If any parameter remains undetermined, the inertial call is aborted, and the decision-making process falls back to the LLM, ensuring that only high-confidence, fully specified actions are executed via inertia.

5 Evaluation

We present the primary experimental results below. Due to the page limit, **additional results including inertia analysis on ToolBench, case studies, dynamic analysis, and parameter filling accuracy are provided in the appendix.**

5.1 Experimental Setup

Datasets We evaluate AutoTool on three diverse and challenging multi-hop benchmarks to assess its generalizability

across different domains. We begin with **Alfworld** (Shridhar et al. 2020), a classic text-based simulator for embodied household tasks. For more complex procedural tasks, we employ **ScienceWorld** (Wang et al. 2022), where agents must follow logical sequences in scientific experiments. Finally, to evaluate multi-step API usage, we use **ToolQuery-Academic** (Ma et al. 2024), a benchmark from AgentBoard involving queries to an academic database. Together, these datasets provide a comprehensive test environment, covering challenges from physical reasoning and procedural following to structured API interactions.

Evaluation Metrics Our evaluation focuses on two core aspects: task execution accuracy and efficiency.

To measure accuracy, we adopt the **progress rate** metric from the AgentBoard testing framework (Ma et al. 2024). AgentBoard manually defines key sub-goals for each task. It then calculates the progress rate by comparing the sequence of sub-goals an agent actually achieves against the predefined sequence. This metric serves as our primary indicator of an agent’s ability to successfully complete tasks.

To evaluate task execution efficiency, we focus on two key metrics. We use the average number of **LLM calls** as a robust, hardware-agnostic proxy for temporal efficiency, since API latency is the primary runtime bottleneck. Concurrently, we measure the average **token consumption** to quantify the computational cost.

Baselines To our knowledge, there is no other published open-source work designed to improve the efficiency of tool selection in LLM agents without calling LLMs. Given this, and because our framework is designed for easy integration, we evaluate AutoTool by applying it to two foundational agent paradigms: ReAct (Yao et al. 2023b) and Reflexion (Shinn et al. 2023). This approach allows us to demonstrate its effectiveness as an enhancement module.

ReAct, a foundational LLM agent paradigm, uses a “Thought-Action-Observation” loop for multi-step reasoning and interaction. Reflexion enhances ReAct by introducing a self-reflection mechanism powered by an LLM-based evaluator. This evaluator assesses the agent’s trajectories and generates feedback to prompt the agent to reflect on its errors. Our experiments mirror the original paper’s heuristic self-reflection: reflection is triggered by failed tool calls or three consecutive identical observations, with the LLM-generated reflection added to the agent’s memory.

Settings All experiments are conducted on a server equipped with four Intel(R) Xeon(R) Gold 5117 CPUs and four NVIDIA Tesla V100-SXM2-32GB GPUs. We use Llama4-Scout-17b as the default model. The sampling temperature is consistently set to 0.

5.2 Speedup

Our core objective is to verify that AutoTool can significantly reduce computational overhead (measured by the number of LLM calls and token consumption) while maintaining comparable task performance (measured by progress rate).

Notably, the graph construction time is negligible, as analyzed in Section 5.3. We primarily assess efficiency improve-

Method	AlfWorld				ScienceWorld				Tool-Query-Academia			
	PR	tok-in	tok-out	LLMC	PR	tok-in	tok-out	LLMC	PR	tok-in	tok-out	LLMC
ReAct	0.394	6560	2310	24.1	0.716	9574	1072	23.3	0.901	1230	658	7.58
+ AutoTool	0.531	4110	804	20.4	0.708	7377	758	17.8	0.895	1070	717	6.32
<i>SpeedUp</i>	—	1.60x	2.87x	1.18x	—	1.30x	1.41x	1.31x	—	1.15x	0.92x	1.20x
Reflexion	0.481	6813	2379	30.7	0.730	7282	1211	24.9	0.917	1680	680	8.85
+ AutoTool	0.453	5130	1976	23.7	0.712	7842	1012	19.5	0.923	1260	569	7.05
<i>SpeedUp</i>	—	1.33x	1.20x	1.29x	—	0.93x	1.20x	1.28x	—	1.33x	1.19x	1.26x

Table 3: Performance and resource consumption comparison of baseline methods with and without AutoTool. progress rate (PR) measures task accuracy, while SpeedUp row shows the cost reduction ratio.

ment by comparing the average LLM calls and token consumption of AutoTool-enhanced agents (ReAct+AutoTool and Reflexion+AutoTool) with their original baselines. We use SimCSE (Gao, Yao, and Chen 2021) to calculate the contextual relevance score.

In our experimental setup, core AutoTool hyperparameters were tuned to achieve a 10-30% reduction in LLM calls, with $\theta_{inertial} = 0.1$ controlling the inertia trigger threshold and $\alpha = 0.5$ weighting contextual relevance. To maximize task progress, we implement a uniform constraint that limits inertia calls to no more than 30% of the total operations and explicitly prohibits consecutive inertia calls. To ensure fair comparison, we keep all agent components and prompts consistent with their respective baselines.

All experiments are conducted in a pure cold-start setting. The core graph is built online from scratch **without any prior trajectories**, ensuring fairness across comparisons.

As shown in Table 3, introducing AutoTool’s optimization mechanism yields substantial efficiency gains across all datasets. On average, it reduces the LLM call count by 15% to 25% and the total token consumption by 10% to 40%.

For ReAct+AutoTool We integrate into the TIG a specialized fault tolerance mechanism—designed as a preconfigured recovery path that activates upon detecting consecutive tool failures. It triggers a check operation to retrieve the current list of available tools, enabling the agent to re-orient itself and break out of ineffective exploration loops.

On the AlfWorld dataset, the effectiveness of this approach is particularly striking, as the TIG not only delivers substantial efficiency gains, with a 1.18x SpeedUp in LLM calls and over 1.60x in total token consumption, but also boosts progress rate, which in turn leads to fewer total execution steps. On the ScienceWorld and ToolQuery-Academic datasets, ReAct+AutoTool also demonstrates stable efficiency gains in both LLM call counts and token consumption, with total SpeedUp values of 1.3x/1.3x/1.4x (for tok-in, tok-out, and LLMC respectively) and 1.2x/1.2x/1.0x. In essence, the efficiency gains are attributable to the inertia graph’s predictive capability, whereas the progress rate improvement (especially on AlfWorld) and overall stability are ensured by the fault-tolerance mechanism and the 30% cap on inertial calls respectively.

For Reflexion+AutoTool To avoid conflicting with its inherent reflection mechanism, we do not introduce a preconfigured recovery path. Despite this, AutoTool is still able to effectively utilize tool usage inertia and maintain competitive performance. A comparison between Reflexion+AutoTool and ReAct+AutoTool reveals that the former yields a superior progress rate, primarily because the Reflexion mechanism improves the quality of collected inertia trajectories by reducing meaningless trial operations.

Furthermore, to assess the model-agnostic nature of our framework, we replicated experiments on ScienceWorld using diverse LLMs, including **Llama-3.3-70B** (Dubey et al. 2024a), **Qwen2.5-72B** (Hui et al. 2024), and **DeepSeekV3** (Liu et al. 2024a). As detailed in **Appendix C**, AutoTool consistently delivered significant efficiency gains across all models, demonstrating that our method is a robust architectural improvement, not dependent on a specific model’s characteristics.

5.3 Overhead Analysis

To quantify the computational overhead introduced by our framework, we systematically analyzed the time consumption of its core components for both ReAct+AutoTool and Reflexion+AutoTool across all three datasets. We dissect this overhead into two primary categories: the non-semantic modules (e.g., graph search and construction), detailed in Table 5, and the semantic relevance calculation modules, which constitute the primary overhead, presented in Table 4.

As detailed in Table 5, our analysis reveals that the overhead from AutoTool’s non-semantic components is minimal, typically remaining on the order of seconds even when LLM inference time for a task exceeds a thousand seconds. The primary overhead stems from contextual relevance calculation, which involves embedding both the agent’s intuition and tool descriptions. However, as Table 4 shows, this cost is negligible: computing contextual relevance accounts for only $2.7\% \pm 1.5\%$ of total task execution time, a mere fraction of standard LLM inference.

5.4 Sensitivity Analysis

To investigate the impact of AutoTool’s key hyperparameters on its performance, we conduct a sensitivity analysis on the inertia trigger threshold $\theta_{inertial}$ and the semantic

Method	Dataset	Init	Semantic Modules (s)			Total	Overhead % of
		(SimSCE)	Intuition Emb.	Tool Emb.	Similarity Comp.	Overhead (s)	Total Task Time
ReAct+ AutoTool	AlfWorld	4.04	23.5793	3.0367	1.1830	31.84	1.21
	ScienceWorld	4.03	21.1180	1.0173	0.4775	26.64	1.38
	Academic	3.76	6.1420	0.1704	0.0210	10.09	4.16
Reflexion+ AutoTool	AlfWorld	4.02	35.3731	7.2560	2.1583	48.81	1.72
	ScienceWorld	4.07	26.6970	2.2535	0.7911	33.81	1.53
	Academic	3.77	5.1539	0.1105	0.0153	9.05	3.75

Note: ‘Init (SimSCE)’ refers to the one-time cost of loading the SimSCE model into memory.

Table 4: Time cost analysis of semantic modules in AutoTool. This table details the overhead from computing contextual relevance and its percentage of the total task execution time. All time values are in seconds (s).

Method	Dataset	AutoTool Core Modules (s)					LLM Time	Action Counts	
		Graph Const.	Graph Search	Parsing	Param Filling	Gen. Action	(s)	Inertial	Total
ReAct+ AutoTool	AlfWorld	0.3159	1.5026	0.2370	0.4653	0.0267	2604.1	1076	3605
	ScienceWorld	0.2414	0.9907	0.4578	0.1839	0.0192	1909.4	690	2316
	Academic	0.0286	0.0201	0.0218	0.0195	0.0031	232.1	49	203
Reflexion+ AutoTool	AlfWorld	0.4460	0.9933	0.2531	0.6226	0.0064	2799.8	1080	3763
	ScienceWorld	0.3500	0.6857	0.5055	0.2914	0.0075	2173.8	689	2406
	Academic	0.0320	0.0139	0.0213	0.0119	0.0014	232.5	34	184

Note: ‘LLM Time’ is the LLM inference time within the AutoTool framework, while ‘Parsing’ is the time spent parsing LLM outputs and tool outputs.

Table 5: Time cost and action count analysis of AutoTool’s core modules. This table details the performance of non-semantic components within the AutoTool framework.

θ_{in}	α	PR	Tok-In	Tok-Out	LLMC	Avg Act
0.1	0.3	0.719	67724	710	17.13	24.45
	0.5	0.694	69 010	702	16.85	24.18
	0.7	0.686	73 897	785	18.38	26.47
0.15	0.3	0.706	71 077	765	17.72	25.36
	0.5	0.715	70 640	744	17.66	25.22
	0.7	0.687	69 287	726	17.33	24.75
0.2	0.3	0.696	78 213	812	19.14	26.93
	0.5	0.695	73 515	854	17.94	24.88
	0.7	0.696	73 846	861	18.12	24.67

Table 6: Sensitivity analysis of AutoTool on the ScienceWorld dataset. We vary the inertia trigger threshold θ_{in} and the contextual relevance weight α .

similarity weight α . The experiments are performed on the ScienceWorld dataset with ReAct+AutoTool, and the results are shown in Table 6. The experimental results largely meet our expectations: a lower $\theta_{inertial}$ (e.g., 0.1) tends to trigger more inertia calls, thereby most effectively reducing the average number of LLM calls (e.g., reaching the lowest value of 16.85 among all tested combinations when $\alpha = 0.5$) and the corresponding token consumption.

Interestingly, the progress rate remains stable despite the lower threshold. We attribute this to the two key constraints: a 30% cap on total inertia calls, and a prohibition on consecutive ones. Indeed, even with a relatively high $\theta_{inertial}$ of 0.2, the number of triggered inertia calls already approaches or reaches this 30% ceiling. This reveals an insight into our design: under these constraints, a more lenient $\theta_{inertial}$ is not only safe but beneficial. It allows AutoTool to capture a more diverse range of effective inertia patterns without being overwhelmed by low-quality calls, thus avoiding the rigidity and insufficiency of an overly strict threshold.

6 Conclusion

We propose AutoTool, a graph-based and lightweight tool selection framework. Rather than simply following observed tool usage inertia, AutoTool treats it as a behavior to be actively managed. By innovatively integrating statistical structure into the design of LLM agents, AutoTool leverages the observed tool usage inertia to effectively address the high latency and resource consumption associated with multi-step tool selection in existing frameworks. While we detail the framework’s current limitations in **Appendix G**, our method has demonstrated strong generalization and adaptability, showing great potential for practical applications.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 62502174).

References

- Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Belcak, P.; Heinrich, G.; Diao, S.; Fu, Y.; Dong, X.; Muralidharan, S.; Lin, Y. C.; and Molchanov, P. 2025. Small Language Models are the Future of Agentic AI. *arXiv preprint arXiv:2506.02153*.
- Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 4171–4186.
- Du, Y.; Wei, F.; and Zhang, H. 2024. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253*.
- Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; and et al. 2024a. The Llama 3 Herd of Models. *arXiv:2407.21783*.
- Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. 2024b. The llama 3 herd of models. *arXiv e-prints, arXiv:2407*.
- Duhigg, C. 2012. *The power of habit: Why we do what we do in life and business*, volume 34. Random House.
- Gao, T.; Yao, X.; and Chen, D. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.
- Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Hong, S.; Zheng, X.; Chen, J.; Cheng, Y.; Wang, J.; Zhang, C.; Wang, Z.; Yau, S. K. S.; Lin, Z.; Zhou, L.; et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4): 6.
- Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Jin, H.; Huang, L.; Cai, H.; Yan, J.; Li, B.; and Chen, H. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.
- Kim, J.; Shin, B.; Chung, J.; and Rhu, M. 2025. The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective. *arXiv preprint arXiv:2506.04301*.
- Kim, S.; Moon, S.; Tabrizi, R.; Lee, N.; Mahoney, M. W.; Keutzer, K.; and Gholami, A. 2024. An llm compiler for parallel function calling. In *Forty-first International Conference on Machine Learning*.
- Koh, J. Y.; McAleer, S.; Fried, D.; and Salakhutdinov, R. 2024. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*.
- LangChain. 2023. Langchain: Build context-aware reasoning applications. URL: <https://github.com/langchain-ai/langchain>.
- Li, Y.; Wen, H.; Wang, W.; Li, X.; Yuan, Y.; Liu, G.; Liu, J.; Xu, W.; Wang, X.; Sun, Y.; et al. 2024. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459*.
- Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.
- Liu, X.; Peng, Z.; Yi, X.; Xie, X.; Xiang, L.; Liu, Y.; and Xu, D. 2024b. Toolnet: Connecting large language models with massive tools via tool graph. *arXiv preprint arXiv:2403.00839*.
- Liu, Y.; Peng, X.; Cao, J.; Bo, S.; Zhang, Y.; Zhang, X.; Cheng, S.; Wang, X.; Yin, J.; and Du, T. 2024c. Tool-Planner: Task Planning with Clusters across Multiple Tools. *arXiv preprint arXiv:2406.03807*.
- Ma, C.; Zhang, J.; Zhu, Z.; Yang, C.; Yang, Y.; Jin, Y.; Lan, Z.; Kong, L.; and He, J. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. *arXiv preprint arXiv:2401.13178*.
- Paranjape, B.; Lundberg, S.; Singh, S.; Hajishirzi, H.; Zettlemoyer, L.; and Ribeiro, M. T. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*.
- Patil, S. G.; Zhang, T.; Wang, X.; and Gonzalez, J. E. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37: 126544–126565.
- Qian, C.; Acikgoz, E. C.; He, Q.; Wang, H.; Chen, X.; Hakkani-Tür, D.; Tur, G.; and Ji, H. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.
- Qin, Y.; Hu, S.; Lin, Y.; Chen, W.; Ding, N.; Cui, G.; Zeng, Z.; Zhou, X.; Huang, Y.; Xiao, C.; et al. 2024. Tool learning with foundation models. *ACM Computing Surveys*, 57(4): 1–40.
- Qin, Y.; Liang, S.; Ye, Y.; Zhu, K.; Yan, L.; Lu, Y.; Lin, Y.; Cong, X.; Tang, X.; Qian, B.; et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.
- Qu, C.; Dai, S.; Wei, X.; Cai, H.; Wang, S.; Yin, D.; Xu, J.; and Wen, J.-R. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8): 198343.
- Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36: 68539–68551.

- Shannon, C. E. 1948. A mathematical theory of communication. *The Bell system technical journal*, 27(3): 379–423.
- Shen, Y.; Song, K.; Tan, X.; Li, D.; Lu, W.; and Zhuang, Y. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36: 38154–38180.
- Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36: 8634–8652.
- Shridhar, M.; Yuan, X.; Côté, M.-A.; Bisk, Y.; Trischler, A.; and Hausknecht, M. 2020. Alfworlde: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.
- Song, Y.; Xiong, W.; Zhu, D.; Wu, W.; Qian, H.; Song, M.; Huang, H.; Li, C.; Wang, K.; Yao, R.; et al. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.
- Team, T. D.; Li, B.; Zhang, B.; Zhang, D.; Huang, F.; Li, G.; Chen, G.; Yin, H.; Wu, J.; Zhou, J.; et al. 2025. Tongyi DeepResearch Technical Report. *arXiv preprint arXiv:2510.24701*.
- Wang, Q.; Wang, T.; Tang, Z.; Li, Q.; Chen, N.; Liang, J.; and He, B. 2025. MegaAgent: A large-scale autonomous LLM-based multi-agent system without predefined SOPs. In *Findings of the Association for Computational Linguistics: ACL 2025*, 4998–5036.
- Wang, R.; Jansen, P.; Côté, M.-A.; and Ammanabrolu, P. 2022. Scienceworld: Is your agent smarter than a 5th grader? *arXiv preprint arXiv:2203.07540*.
- Wang, Z. Z.; Mao, J.; Fried, D.; and Neubig, G. 2024. Agent workflow memory. *arXiv preprint arXiv:2409.07429*.
- Wei, H.; Zhang, Z.; He, S.; Xia, T.; Pan, S.; and Liu, F. 2025. Plangenllms: A modern survey of llm planning capabilities. *arXiv preprint arXiv:2502.11221*.
- Wu, X.; Shen, Y.; Shan, C.; Song, K.; Wang, S.; Zhang, B.; Feng, J.; Cheng, H.; Chen, W.; Xiong, Y.; et al. 2024. Can graph learning improve planning in LLM-based agents? *Advances in Neural Information Processing Systems*, 37: 5338–5383.
- Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; and Narasimhan, K. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36: 11809–11822.
- Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023b. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- Yi, Z.; Ouyang, J.; Liu, Y.; Liao, T.; Xu, Z.; and Shen, Y. 2024. A survey on recent advances in llm-based multi-turn dialogue systems. *arXiv preprint arXiv:2402.18013*.
- Zhuang, Y.; Chen, X.; Yu, T.; Mitra, S.; Burszty, V.; Rossi, R. A.; Sarkhel, S.; and Zhang, C. 2023. Toolchain*: Efficient action space navigation in large language models with a* search. *arXiv preprint arXiv:2310.13227*.
- Zhuge, M.; Wang, W.; Kirsch, L.; Faccio, F.; Khizbullin, D.; and Schmidhuber, J. 2024. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*.

A Details of AutoTool

Data Structure

The core data structure of AutoTool is the Tool Inertia Graph (TIG), designed to capture and leverage sequential patterns in tool usage and parameter dependencies. The TIG is primarily composed of several interconnected components that work in concert.

At the foundational level are **ToolNodes**, each precisely representing an available tool within the system. A ToolNode encapsulates the tool’s name, functional description, and a formalized list of its input arguments (args) and output arguments (returns). To manage data flow within individual tools more granularly, each ToolNode embeds a **ParamGraph**. This dedicated subgraph is responsible for maintaining **ParamNodes** associated with the tool. These ParamNodes not only distinguish between input and output attributes but also cache example values observed during runtime, thereby providing a basis for subsequent parameter filling.

To track the transfer of data across different tool invocations, the TIG incorporates **ParamEdges**. This constitutes a core mapping structure, conceptually organized as `{target_tool: {target_param: {(source_tool, source_param): ParamEdge_instance}}}`. Each individual `ParamEdge_instance` meticulously records a specific historical data flow: it signifies that an output value or input value from a particular “source_param” of a “source_tool” has previously served as an input for a specific “target_param” of a subsequent “target_tool.” A counter within the `ParamEdge` quantifies the strength and frequency of this observed dependency.

Complementing the node and data-flow structures, the TIG maintains **ToolPaths** to learn and utilize sequential patterns of tool invocations. Each ToolPath object stores an observed complete sequence of tool calls (represented as a list of tool names) along with the frequency of that specific sequence’s occurrence. For erroneous tool invocations, AutoTool simply add a tool call chain with a negative frequency, but ensure that its length is greater than the inertia observation window.

For efficient retrieval and management of these paths, the TIG internally employs several key mappings. `self.path_index` provides a rapid index from an individual tool name to a set of all ToolPath IDs that contain that tool. The `self.paths` list allows direct access to the ToolPath object itself (containing the tool sequence, frequency, etc.) via its unique ID. Furthermore, `self.paths_lookup` enables quick lookup of a ToolPath ID from a tuple representation of a tool sequence, which is crucial for efficiently checking path existence and updating path frequencies.

Collectively, these components—ToolNodes with their embedded ParamGraphs, the graph of ParamEdges, and the systematically managed ToolPaths—form the foundation of the TIG. This architecture enables the TIG to dynamically learn from execution history and effectively support inertia-driven tool selection.

Tool Prediction

AutoTool predicts the next tool by analyzing a segment of recently executed tools called the **current_sequence**. This `current_sequence` is a truncated portion of the full tool execution history, and its length is defined by a variable called the *inertia window*. The inertia window is typically set to 2, this means AutoTool rely on two consecutive action sequences for inertia prediction. When the length is 1, the lack of contextual information results in lower accuracy for inertia prediction. Increasing the length to 3 or more theoretically allows for capturing longer dependencies, but on the current datasets, the number of historical paths that meet the criteria significantly decreases, leading to a decline in the model’s generalization ability and an increased risk of overfitting to specific long sequences. Exploring a dynamically changing inertia window is a direction for further optimization.

The graph searching process begins by identifying candidate historical ToolPaths that contain the `current_sequence` as a sub-sequence. Tool Prediction module first use `TIG.path_index` to get the set of path IDs associated with the first tool in `current_sequence`. Then, for each subsequent tool in `current_sequence`, this set of path IDs is intersected with the set of path IDs associated with that tool. This efficiently narrows down to `potential_path_indices` that are supersets of the current execution window.

Once these `potential_path_indices` are obtained, Tool Prediction module iterate through each corresponding ToolPath. It first verifies whether the `current_sequence` is indeed a sub-sequence of the historical path. Then it checks if there is at least one tool following the `current_sequence` in that historical path. If both conditions are satisfied, the tool that immediately follows the `current_sequence` in the historical path is considered a candidate for the next action. The weight for each unique candidate next tool in the `candidate_tool_dic` is then updated based on the `frequency` attribute of the historical ToolPath, denoted as $w(c_j)$.

The Inertia Confidence Score (ICS) calculation often yields multiple candidate tools for the next action. AutoTool first computes the total aggregated weight $W_{\text{total}} = \sum w(c_i)$ across all candidate tools c_i , reflecting the overall historical evidence strength for any known subsequent tool. To prevent premature high-confidence inertial calls when historical data is sparse (i.e., W_{total} is small), an Inertia Confidence Factor (ICF) is introduced, derived by applying a slowly increasing, bounded-by-1 exponential function to W_{total} : $\text{ICF} = 1 - k^{-W_{\text{total}}}$, where k is a constant greater than 1 (e.g., $k = 1.1$). When W_{total} is small, ICF approaches 0, suppressing overall confidence; as W_{total} grows, ICF approaches 1, indicating higher confidence. Finally, the Inertia Confidence Score (ICS) for each candidate tool c_j is calculated as: $\text{ICS}(c_j) = \left(\frac{w(c_j)}{W_{\text{total}}} \right) \times \text{ICF}$. This score is then combined with contextual relevance to form the final comprehensive inertia potential score. This mechanism ensures AutoTool effectively leverages inertia when sufficient data is available while maintaining conservative decision-making in data-scarce scenarios.

Algorithm 1: ToolPrediction

```
Input : current_history, current_memory, tool_graph
Output : predicted_tool (String or None),
         confidence_score (Float)

1 // Goal: Predict the next tool
  based on historical patterns and
  current context.
2 current_sequence :=
  GetRecentToolSequence(current_history);
3 intuition := ExtractRelevantThought(current_memory);

4 // Step 1: Candidate Path
  Identification
5 candidate_paths :=
  FindMatchingPaths(current_sequence, tool_graph);
6 if candidate_paths is empty then
7 |   return None, 0.0;
8 end

9 // Step 2: Scoring Candidate Next
  Tools
10 scored_next_tools := EmptyMap;
11 foreach path in candidate_paths do
12 |   next_tool := GetNextToolInPath(path,
13 |   current_sequence);
14 |   if next_tool exists then
15 |   |   freq_score :=
16 |   |   CalculateFrequencyScore(next_tool, path,
17 |   |   tool_graph);
18 |   |   sem_score :=
19 |   |   CalculateSemanticSimilarity(next_tool's
20 |   |   description, intuition);
21 |   |   combined_score := WeightedSum(freq_score,
22 |   |   sem_score);
23 |   |   UpdateOrAdd(scored_next_tools, next_tool,
24 |   |   combined_score);
25 |   end
26 end

27 // Step 3: Prediction Decision
28 best_tool, highest_score :=
29 GetToolWithMaxScore(scored_next_tools);
30 return best_tool, highest_score;
```

Param Filling

A significant challenge in designing a general-purpose parameter filling mechanism arises from the considerable heterogeneity across different datasets, particularly concerning the input/output structures of tools and varying environment states. To ensure broad applicability and maintain a modular architecture, AutoTool adopts an **Adaptor pattern**. This approach effectively decouples the core parameter filling framework from dataset-specific intricacies.

For each new dataset or environment, a corresponding concrete Adaptor subclass must be implemented to bridge these specific details with the generic filling logic. The core framework then orchestrates the Adaptor and other components to perform the comprehensive parameter filling process. The base class definition for the Adaptor interface is presented here.

```
import abc
from typing import Dict, List, Any, Tuple,
Optional, Set

class EnvironmentAdapter(abc.ABC):
    """
    Environment adapter base class, defines
    interfaces related to specific
    environments/datasets.
    Each new environment should implement a
    concrete adapter subclass.
    """

    def __init__(self, debug: bool = False):
        """
        Initialize the environment adapter
        Args:
            debug: Whether to enable debug
        mode
        """
        self.debug = debug

    @abc.abstractmethod
    def reset(self, init_observation: str =
    None):
        pass

    @abc.abstractmethod
    def parse_action(self, action_text: str,
    tool_descriptions: Dict) -> Optional[
    Dict[str, Any]]:
        """
        Parse action text and return
        structured action data
        Args:
            action_text: Action text
            tool_descriptions: Tool
            description dictionary
        Returns:
            Parsed action dictionary, format
            : {"tool_name": str, "inputs": Dict[str,
            Any]}
        """
        pass

    @abc.abstractmethod
```

```

def infer_output(self, tool_name: str,
inputs: Dict[str, Any], result: Any) ->
Dict[str, Any]:
    """
    Infer structured output based on
    tool execution result
    Args:
        tool_name: Tool name
        inputs: Input parameters
        result: Execution result
    Returns:
        Inferred structured output
    dictionary
    """
    pass

@abc.abstractmethod
def update_state(self, action_parsed:
Dict[str, Any], structured_outputs: Dict
[str, Any]) -> None:
    """
    Update environment state based on
    action and output
    Args:
        action_parsed: Parsed action
    dictionary
        structured_outputs: Structured
    output dictionary
    """
    pass

@abc.abstractmethod
def get_contextual_params(self, action_
type: str, missing_params: Set[str],
required_params_info: Dict) -> Dict[str,
Any]:
    """
    Infer parameter values based on
    current environment state
    Args:
        action_type: Target action type
        missing_params: Set of missing
    parameters
        required_params_info: Dictionary
    of required parameter descriptions
    Returns:
        Inferred parameter dictionary
    """
    pass

@abc.abstractmethod
def generate_action_from_params(self,
action_type: str, params: Dict[str, Any
]) -> str:
    pass

```

Listing 1: Abstract Base Class for Environment Adapters

Future work will explore applying some prompt engineering to achieve a more structured presentation in status and tool output observation, which will make parameter filling adaptable to different datasets and environments much more easily, while minimizing the introduction of additional token overhead.

Algorithm 2: ParameterFilling

```

Input : target_tool, exec_history, param_dep_graph,
        env_adapter, tool_graph
Output: filled_parameters (Map), was_successful
        (Boolean)

1 // Goal: Automatically populate
  input parameters for the
  target_tool.
2 required_params := GetInputSchema(target_tool,
  tool_graph_schema);
3 filled_params := EmptyMap;
4 // Priority 1: Parameter Dependency
  Graph (Learned direct
  output-to-input links)
5 foreach param_name in required_params do
6   if param_name not yet filled then
7     sources := QuerySources(param_dep_graph,
      target_tool, param_name);
8     if value found in sources then
9       value :=
        GetValueFromHistory(exec_history,
        source_tool, source_param);
10      if value is valid and type-compatible then
11        | filled_params[param_name] := value;
12      end
13    end
14  end
15 end
16 // Priority 2: Environment Context
17 foreach param_name in required_params do
18   if param_name not yet filled then
19     value :=
      env_adapter.GetContextualValue(target_tool,
      param_name);
20     if value is available and valid then
21       | filled_params[param_name] := value;
22     end
23   end
24 end
25 all_critical_filled :=
  CheckIfAllCriticalParamsAreFilled(filled_params,
  required_params);
26 return filled_params, all_critical_filled;

```

When a source parameter value is retrieved from historical data via backtracking and matching, AutoTool handles it based on its type. If the source value is singular (e.g., a string or number), it is directly assigned to the target parameter. However, a challenge arises when the source value is a list of multiple elements while the target parameter typically expects a single value, necessitating a selection strategy. While an ideal strategy would deeply integrate tool semantics and current task context, this often results in dataset-specific solutions that compromise generality. Although sophisticated, scenario-specific heuristic rules could be developed, our current AutoTool implementation employs a more general method to balance efficacy with simplicity: the framework maintains a record of all values already used for parameter filling within the ongoing task. When sourcing from a list, AutoTool randomly selects an element from that source list that has not yet appeared in this record. This approach provides a foundational mechanism for diversifying parameter inputs from list-type sources within the same task, acknowledging that more contextually-aware selection strategies represent an avenue for future enhancement.

B Macro-Level Inertia Analysis on ToolBench

To validate the generalizability of our "Sequential Inertia" hypothesis beyond task-oriented domains, we conducted a supplementary macro-level analysis on the ToolBench G3 (Figure 3). ToolBench is a large-scale benchmark featuring over 16,000 real-world APIs, with its G3 subset being a comprehensive collection of over 15,000 diverse tools. This dataset provides an ideal testbed for evaluating the structural properties of tool-calling behavior in a more open-ended, less constrained environment.

We processed 15,980 valid invocation trajectories from the G3 dataset. To ensure the analysis reflects meaningful sequential patterns, we applied a preprocessing step to remove repetitive tool calls (e.g., $A \rightarrow A$) and overly short trajectories (length less than 2). Based on the cleaned trajectories, we constructed a transition probability matrix representing the likelihood of transitioning from any given tool to another.

Our primary metric for this macro-level analysis is the **system-wide conditional entropy**, which quantifies the average uncertainty of predicting the next tool, given the current one. The results are as follows:

- **Calculated Conditional Entropy $H(Y|X)$:** Our analysis revealed a system-level conditional entropy of only **3.62 bits**.
- **Maximum Theoretical Entropy (Uniform Baseline):** For a system with 1,595 unique tools, the maximum possible entropy, assuming completely random transitions, is $\log_2(1595) \approx 10.64$ bits.
- **Entropy Reduction:** The observed entropy of 3.62 bits represents a **65.96% reduction** from the theoretical maximum.

The nearly 66% reduction in entropy provides a powerful, formal guarantee of the system’s non-random and highly predictable structure. It confirms that even in a vast and diverse ecosystem of real-world APIs like ToolBench, tool invocation

is not a random walk. Instead, it is governed by strong, underlying patterns of usage. This macro-level evidence strongly supports our central claim that tool-calling inertia is a fundamental and generalizable property of LLM agents, making our AutoTool approach broadly applicable.

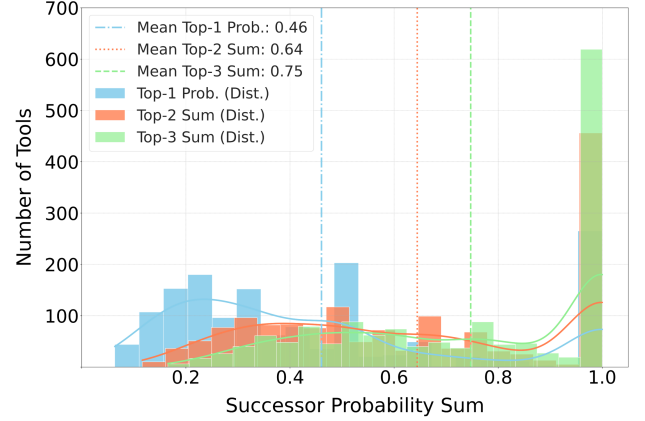


Figure 3: ToolBench Inertia Analysis. We calculated the frequency distribution of successor tools for each tool and derived the proportion of its top- k successor tools among all its successor tools.

C Model Robustness

To verify the generalization ability of our method across different models, we conducted experiments on the ScienceWorld dataset using a diverse set of models with varying parameter sizes and architectures. Specifically, we tested the following models: Llama-4-Scout-17B-16E-Instruct, DeepSeekV3, Llama3.3-70B-Instruct, and Qwen2.5-72B-Instruct-Turbo. The results are presented in Table 7.

Our method demonstrates strong generalization across different models. Despite significant performance differences among these models on the same dataset, our approach consistently reduces token consumption and the number of LLM calls while maintaining a comparable progress rate. This provides empirical evidence for further generalization in multiple scenarios.

For DeepSeekV3, the poor instruction-following ability of the base model leads to a significant waste of solution steps as it fails to output action in the prescribed format. However, our preconfigured recovery path (two consecutive erroneous calls force a call to *check valid action*, to verify the set of available tools) significantly improves the progress rate while reducing token consumption. Similar experimental phenomena were also observed in the test results of ReAct + AutoTool on alfworld, as detailed in Table 3.

D Case Study

To provide a comprehensive and balanced evaluation of AutoTool, this section presents both a detailed success case and an analysis of characteristic failure scenarios. This approach allows us to not only demonstrate the practical benefits of

Table 7: Performance Comparison of AutoTool with ReAct across Different LLMs on ScienceWorld. Metrics include Progress Rate (PR), Token Consumption (token-in, token-out), and Number of LLM Calls.

Base LLM	Method	Progress Rate	token-in	token-out	llm.call
Llama-4-Scout-17B-16E-Instruct	ReAct	0.7159	95 743.62	1072.03	23.31
	ReAct+AutoTool	0.7082	73 779.72	758.87	17.81
Llama-3.3-70B-Instruct-Turbo	ReAct	0.8126	90 001.91	1687.08	19.15
	ReAct+AutoTool	0.7891	63 886.16	1130.70	14.60
Qwen2.5-72B-Instruct-Turbo	ReAct	0.7600	81 466.92	1168.15	19.71
	ReAct+AutoTool	0.7484	66 563.08	876.54	15.93
DeepSeekV3	ReAct	0.4479	112 062.43	2204.80	25.00
	ReAct+AutoTool	0.5889	72 608.53	871.43	17.26

our method but also to transparently define its operational boundaries.

Success Case Analysis

Here we present case studies comparing the ReAct agent with the AutoTool+ReAct agent. We illustrate examples from the 30th Scienceworld task in Figure 4.

As shown on the right, *Look around* is executed twice by inertial tool calling, which significantly reduces token consumption. Meanwhile, the baseline has to take one step at a time. This obvious inertial calling is harmless for the agent to solve the task. Even if the inertial calling leads to inefficient or erroneous actions, it barely causes any waste of tokens and time.

Case Study for Scienceworld	
Your task is to find a counter in the kitchen.. First, focus on the thing. Then, move it to the purple box in the kitchen.	
ReAct	ReAct+AutoTool
go to hallway 2750 You move to the hallway.	go to hallway 2776 You move to the hallway.
look around 2863 This room is called the hallway. In it, you see: the agent, a painting	look around 0 This room is called the hallway. In it, you see: the agent, a painting
go to kitchen 3028 You move to the kitchen.	go to kitchen 3029 You move to the kitchen.
look around 3118 This room is called the kitchen. In it, you see: the agent, a chair	look around 0 This room is called the kitchen. In it, you see: the agent, a chair
focus on counter 3426 You focus on the counter.	focus on counter 3418 You focus on the counter.
...	...

Figure 4: Case study for scienceworld

Failure Case Analysis

In the typical usage scenarios of AutoTool, there are three classic failure scenarios. An analysis of these scenarios not only reveals the methodological boundaries of AutoTool but also inversely demonstrates the necessity of our designed feedback and fallback mechanisms. All cases presented below are derived from authentic experimental logs.

Scenario 1: Parameter Filling with Incorrect Context

When an agent needs to fill parameters for an action, AutoTool might retrieve a “frequently used” parameter from a completely unrelated historical memory, entirely ignoring the agent’s current physical location and task constraints. This leads to action failure due to unsatisfied preconditions.

Goal	put two cd in safe.
Context	The agent is located near desk 2 and is currently carrying cd 1, as confirmed by an inventory check in Step 22. The final destination for the object is the safe.
Inertial Call	put cd 1 in/on drawer 1
Analysis	AutoTool’s inertial call put cd 1 in/on drawer 1 failed due to two critical errors. (1) Logical Mismatch : The parameter drawer 1 contradicted the task goal (safe). (2) Physical Precondition Violation : The agent was at desk 2, making an action at drawer 1 physically impossible. This demonstrates a severe contextual disconnect in parameter filling.

Figure 5: Failure Scenario 1

Scenario 2: Inertia-Induced Redundant Actions A more subtle risk of inertia is that it can cause the agent to “spin its wheels.” If an action sequence is historically chained, AutoTool might mechanically execute the next step without first evaluating whether the current state already satisfies the desired condition. While this may not cause a task-level “failure,” it generates useless steps, wasting both time and execution turns.

Scenario 3: Overgeneralization of Tool Usage Patterns

AutoTool builds its inertia graph by learning high-frequency action sequences. However, it might learn a general pattern (e.g., go to → open) without understanding the semantic

Goal	put two soapbar in garbagecan.
Context	The agent arrived at countertop 1. In Step 12, it picked up soapbar 1 from countertop 1. At this moment, the agent's physical location was still at countertop 1.
Inertial Call	go to countertop 1
Analysis	This action was entirely redundant because the agent was already at that location. The environment correctly returned Nothing happens, and the step was wasted.

Figure 6: Failure Scenario 2

constraints of the objects to which it applies. A failure of "overgeneralization" occurs when it applies this general pattern to an object that does not support the action.

Goal	put a clean spatula in drawer.
Context	In last step, the agent executed go to sinkbasin 1 and arrived at "Sink Basin 1."
Inertial Call	open sinkbasin 1
Analysis	In many tasks, after reaching a container (e.g., a drawer), the next step is to open it. Based on the high-frequency pattern go to -> open, AutoTool directly made an inertial call: open sinkbasin 1 . However, a "sinkbasin" is an open container and lacks a door or lid that can be "opened" .

Figure 7: Failure Scenario 3

E Dynamic Analysis

To demonstrate AutoTool’s dynamic learning capability and its progressive optimization effect on agent performance, we track the execution process of Reflexion+AutoTool on the Alf-world dataset, which had the most severe progress rate loss. Figure 8 shows the evolution trends of Reflexion+AutoTool and the original Reflexion in terms of Progress Rate and average LLM call counts (or Token consumption) as the number of executed tasks increases.

From the Figure 8, after the experimental data converge (after 40 tasks), our method slightly underperforms the original Reflexion in terms of progress rate. However, as more execution trajectories are collected, around the 80th task execution, we observe that the performance of Reflexion+AutoTool begins to steadily improve. Considering the influence of the previous data, our method eventually maintains a comparable performance with Reflexion in the later stages of the experiment. More crucially, throughout the learning process, ReflexionAutoTool consistently demonstrates and gradually expands its advantage in execution efficiency. As shown in the Figure 8, its average LLM call counts and Token consumption were significantly lower than those of the original

Reflexion from the start, and this gap became increasingly evident as the inertia graph was refined. This highlights AutoTool’s online learning capability. Even starting from scratch, it can build an effective inertia model through continuous interaction and feedback, significantly enhancing the execution efficiency of complex agents like Reflexion. This suggests that with more historical trajectory data, AutoTool’s performance potential will be further unleashed.

F Theoretical Analysis

The efficiency of AutoTool is rooted in a profound insight into the agent’s decision-making process, centered on the "tool usage inertia" phenomenon. We establish the theoretical foundation for this phenomenon on principles from information theory, cognitive science, and graph search algorithms.

Low-Entropy Markov Process of Tool Selection: We model the agent’s tool selection sequence as a **Markov Decision Process (MDP)**. In this model, the set of available tools forms the state space $S = \{tool_1, \dots, tool_N\}$, and a sequence of invocations is a trajectory of states (s_1, s_2, \dots, s_t) where $s_t \in S$. The "tool usage inertia" observed in our work corresponds, in information theory, to a transition matrix with very low **Conditional Entropy** (Shannon 1948). This is formally expressed as the uncertainty of the next state s_{t+1} given the current state s_t :

$$H(S_{t+1}|S_t) = - \sum_{i,j \in S} p(s_i, s_j) \log_2 p(s_j|s_i) \quad (2)$$

where $p(s_j|s_i)$ is the transition probability from s_i to s_j . Our empirical findings show that for most tools s_i , the distribution $p(s_j|s_i)$ is highly skewed. Consequently, the system’s conditional entropy is significantly lower than the maximum possible entropy for a random selection, $H_{\max} = \log_2 |S|$. This condition, $H(S_{t+1}|S_t) \ll \log_2 |S|$, provides the theoretical basis for why fast, non-LLM statistical prediction is feasible.

Cognitive Heuristics: Priming and Habit Loops: This inertia phenomenon is analogous to concepts in cognitive science. The execution of a preceding tool acts as a **Priming Effect**. Furthermore, AutoTool’s dynamic graph update mechanism mimics the **Habit Loop** (Cue \rightarrow Routine \rightarrow Reward) (Duhigg 2012). The feedback from the environment acts as a reward, captured by a dynamic weight update rule for the edge from tool s_i to s_j in our Tool Inertia Graph (TIG):

$$w_{t+1}(s_i, s_j) = w_t(s_i, s_j) + \begin{cases} \Delta w_{\text{success}} & \text{if inertial call is successful} \\ -\Delta w_{\text{failure}} & \text{if inertial call fails} \end{cases} \quad (3)$$

where $\Delta w > 0$. This mechanism allows the agent to continuously learn and reinforce effective tool sequences from experience.

Graph Search as a Decisional Shortcut: An unconstrained LLM, at each decision step, faces a **combinatorially vast action space**. If an agent can choose from N tools at each step for a sequence of length T , the number of potential trajectories is on the order of N^T . The LLM must implicitly navigate this enormous space. In contrast, by constructing historical trajectories into a Dynamic Tool Inertia Graph (TIG),

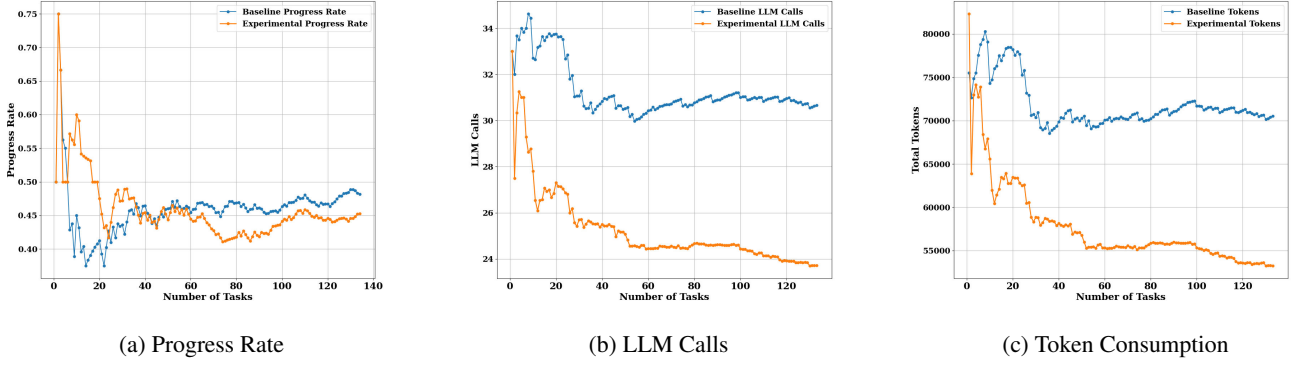


Figure 8: Dynamic learning process of Reflexion+AutoTool on the Alfworld dataset. Compared to the baseline Reflexion, our method consistently reduces (b) LLM calls and (c) token consumption, while maintaining a competitive (a) progress rate that improves over time as more trajectories are collected.

Backbone	Mode	AlfWorld		ScienceWorld		Academic	
		tried	success	tried	success	tried	success
ReAct	PDG	68.87%	27.88%	70.22%	51.24%	74.29%	76.92%
	context_filled	31.13%	29.94%	29.78%	45.83%	25.71%	66.67%
Reflexion	PDG	71.42%	22.33%	77.03%	55.90%	78.12%	76.00%
	context_filled	28.58%	25.35%	22.97%	36.46%	21.88%	71.43%

Table 8: Parameter Filling Statistics

AutoTool transforms this open-ended decision problem into a constrained **Graph Search** problem.

The agent’s choice is guided by the Comprehensive Inertia Potential Score (CIPS), whose decision-making philosophy is akin to the evaluation function in an A* search algorithm:

$$\text{CIPS}(s_{t+1}|h_t) = (1-\alpha) \cdot \text{Score}_{\text{freq}}(s_{t+1}|h_t) + \alpha \cdot \text{Score}_{\text{ctx}}(s_{t+1}|h_t) \quad (4)$$

While A* aims to minimize cost ($f(n) = g(n) + h(n)$) and CIPS aims to maximize a score, their core structure combines historical information with forward-looking heuristics. Here, $\text{Score}_{\text{freq}}$ acts like the known cost $g(n)$, exploiting historical data to evaluate a path’s proven reliability. Meanwhile, $\text{Score}_{\text{ctx}}$ functions as the heuristic $h(n)$, predicting future success by evaluating a tool’s relevance to the current task. This A*-like mechanism allows AutoTool to balance **exploitation** (relying on ‘habits’) and **exploration** (responding to the current ‘stimulus’).

The agent then selects the tool that maximizes this score, bypassing a full LLM inference call:

$$s^* = \arg \max_{s_{t+1} \in \text{Candidates}} (\text{CIPS}(s_{t+1}|h_t)) \quad (5)$$

The work by Zhuang et al. (Zhuang et al. 2023) also demonstrates the effectiveness of such guided search for navigating the LLM action space.

G Limitations

As a data-driven acceleration method, AutoTool’s performance relies on the quality and quantity of historical data,

which may present cold-start challenges, particularly when dealing with extensive toolsets. Its inertia-based predictions may also be less effective in highly dynamic environments or tasks requiring intricate reasoning. A further consideration involves AutoTool’s dependency on parsing and integrating tool outputs. Designing targeted parsing functions for tools that give highly unstructured outputs may require additional engineering effort. Finally, the current manual approach to setting hyperparameters also presents an opportunity for future optimization. To advance AutoTool’s robustness and usability, future work will concentrate on exploring adaptive hyperparameter tuning and investigating deeper integration of semantic information to bolster decision-making in complex environments.

H Parameter Filling Accuracy

To further validate the robustness of AutoTool, we quantitatively analyzed the accuracy of our parameter filling module. A parameter filling event is deemed successful if it leads to a tool call that is both syntactically valid and get successful environment feedback.

Table 8 presents the proportion of attempts (tried) and success rates (success) for each parameter filling mode across different environments. We observe that the Parameter Dependency Graph (PDG) mode is predominantly utilized across all environments and backbones. Parameter filling success rates vary by environment: Academic tasks exhibit the highest success rates, likely due to their structured inputs, while

AlfWorld consistently presents the lowest success rates for both PDG and context_filled modes, suggesting inherent challenges in its open-ended and linguistically diverse environment, with ScienceWorld showing intermediate performance.

I Other Baseline

Given the sequential nature of tool calls, we implemented an N-gram model (n=3) as a new baseline, which also starts collecting data from scratch. Furthermore, we conducted separate experiments for this baseline under two conditions: with and without a recovery mechanism. For parameter filling, this baseline reuses the parameter filling module from AutoTool. The final results are presented in the Table 9.

Method	PR	tok-in	tok-out	LLMC
AutoTool	0.6698	80 220	1324	18.6
Ngram_w_recovery	0.6476	81 838	1403	18.9
Ngram_wo_recovery	0.6140	78 316	1135	18.6

Table 9: Comparison of AutoTool with N-gram Baselines on the ScienceWorld Dataset with Qwen2.5-32B.

J Prompt

We have provided the instruction prompts for the agent on three datasets. The specific details are shown in Figures 9, 11, and 10. For the placeholders in the prompts, we can selectively fill them in. The *reflection instructions* are the reflective messages generated by the LLM in Reflexion, and no reflection is provided for ReAct. Except for the Scienceworld dataset, all other datasets use the zero-shot method without providing examples. Our AutoTool method does not modify the system prompt, that is, it uses the same instruction prompt as the baseline.

For actions generated by inertia calls, AutoTool specifically note them when encapsulating them into temporary memory. Specifically, the prompt is: **Think: Using graph inertia to predict next action {predicted_tool} with parameters {filled_params}. \nAction: {final_action}**

Instruction Prompt for Alfworld

You are a helpful assistant. Your task is to interact with a virtual household simulator to accomplish a specific task.
Your MAIN GOAL is: **[[goal]]**
You MUST achieve ALL aspects of this goal. Do not simplify or ignore parts of the goal.

Follow the ReAct (Reasoning + Acting) paradigm:
1. **Think:**
What you think about the current situation and how to achieve the goal.
2. **Action:** Output EXACTLY ONE command from the list below based on your thought process.

Available Actions:
- **take** **[[obj]]** from **[[recep]]**: Pick up an object **[[obj]]** from a location **[[recep]]**.
- **put** **[[obj]]** in/on **[[recep]]**: Place an object **[[obj]]** into/onto a location **[[recep]]**.
- **open** **[[recep]]**: Open a container **[[recep]]** to access its contents.
- **close** **[[recep]]**: Close an open container **[[recep]]**.
- **toggle** **[[target]]**: Toggle the state of an object or device **[[target]]** (e.g., turn on/off a desk lamp or microwave).
- **clean** **[[obj]]** with **[[recep]]**: Clean an object **[[obj]]** with a tool/receptacle **[[recep]]**.
- **cool** **[[obj]]** with **[[recep]]**: Cool an object **[[obj]]** with a cooling device/receptacle **[[recep]]**.
- **heat** **[[obj]]** with **[[recep]]**: Heat an object **[[obj]]** with a heating device/receptacle **[[recep]]**.
- **examine** **[[target]]**: Look closely at an object or receptacle **[[target]]** to get details. Use this to fulfill 'look at [object]' parts of a goal.
- **go to** **[[recep]]**: Move to a specific location or receptacle **[[recep]]**.
- **look**: Observe your current surroundings to get a general view of the area and visible items.
- **use** **[[target]]**: Use or interact with an object or device **[[target]]** in its default way (often similar to 'toggle' for devices).
- **check_inventory**: Check what objects you are currently carrying.
- **check_actions**: List all currently valid actions based on the environment's state. (Use if unsure or if standard actions fail).

Guidelines for Action:
- **CRITICAL FORMATTING:** Your response MUST be structured EXACTLY as follows, with "Think:" and "Action:" on SEPARATE lines and NO extra text before or after:
Think: [Your reasoning and plan here, addressing points a-d under Think:]
Action: [Your single, valid Alfworld command here from 'Available Actions']

Interaction Examples:
...
[examples_str]
Now, begin the task. Remember the MAIN GOAL: **[[goal]]**
[reflection_instructions]

Figure 9: Instruction prompt for alfworld

Instruction Prompt for Academic

Your goal is to: **[[goal]]**
[tools_description]
If you are finished, call the "finish" action with your final answer.

You should follow the ReAct (Reasoning + Acting) paradigm:
1. Think: Analyze the current situation and plan your next move
2. Action: Execute a specific action with appropriate arguments

Your response must be one of the following two formats:
(1) Think: [your thinking]
(2) Action: [action_name] with Action Input: [parameters in JSON format]

For example:
[examples_str]

Remember to:
1. Use the tools in a logical sequence to solve the problem
2. If unsure about available actions, use 'check_valid_actions'
3. Use precise parameter names and values as specified in the tool descriptions
4. Format JSON parameters correctly
5. Submit your final answer using the 'finish' action when you complete the task

[reflection_instructions]

Figure 10: Instruction prompt for academic



Figure 11: Instruction prompt for scienceworld