ToolGate: Contract-Grounded and Verified Tool Execution for LLMs

Yanming Liu[1] **Xinyue Peng**[2] **Jiannan Cao**[3] **Xinyi Wang** **Songhang Deng**
Jintao Chen[1] **Jianwei Yin**[1] **Xuhong Zhang**[1,4*]
[1]Zhejiang University [2]Southeast University
[3]Massachusetts Institute of Technology
[4]Innovation and Management Center, School of Software Technology, Zhejiang University
{oceann24, zhangxuhong, zjuyjw, chenjintao}@zju.edu.cn
xinyuepeng@seu.edu.cn, jiannan@mit.edu

Abstract

Large Language Models (LLMs) augmented with external tools have demonstrated remarkable capabilities in complex reasoning tasks. However, existing frameworks rely heavily on natural language reasoning to determine when tools can be invoked and whether their results should be committed, lacking formal guarantees for logical safety and verifiability. We present **ToolGate**, a forward execution framework that provides logical safety guarantees and verifiable state evolution for LLM tool calling. ToolGate maintains an explicit symbolic state space as a typed key-value mapping representing trusted world information throughout the reasoning process. Each tool is formalized as a Hoare-style contract consisting of a precondition and a postcondition, where the precondition gates tool invocation by checking whether the current state satisfies the required conditions, and the postcondition determines whether the tool's result can be committed to update the state through runtime verification. Our approach guarantees that the symbolic state evolves only through verified tool executions, preventing invalid or hallucinated results from corrupting the world representation. Experimental validation demonstrates that ToolGate significantly improves the reliability and verifiability of tool-augmented LLM systems while maintaining competitive performance on complex multi-step reasoning tasks. This work establishes a foundation for building more trustworthy and debuggable AI systems that integrate language models with external tools.[1]

1 Introduction

Large Language Models (LLMs) have achieved remarkable success in various reasoning tasks, particularly when augmented with external tools that enable them to interact with the real world (Yao et al., 2022; Brown et al., 2020; Chowdhery et al., 2022). The integration of tools with LLMs has opened new possibilities for complex multi-step reasoning, where models can retrieve information, perform computations, and execute actions through API calls (Qin et al., 2024). However, existing frameworks for LLM tool calling rely heavily on natural language reasoning to determine when tools should be invoked and whether their results should be trusted and committed to the system's understanding of the world (Yang et al., 2024a). This reliance on implicit natural language reasoning creates challenges for ensuring logical safety, verifiability in tool-augmented LLM systems.

The fundamental problem lies in the lack of formal guarantees for tool invocation and result validation. Current approaches treat tool calling as a black-box process where the LLM decides based on its internal reasoning, without explicit mechanisms to verify whether the preconditions for tool invocation are satisfied or whether the tool's output meets the expected postconditions (Zhu et al., 2025; Shi et al., 2023). This can lead to several critical issues: tools may be called with insufficient or incorrect parameters, invalid results may be incorporated into the reasoning process, and the system's internal representation of the world state may become inconsistent or corrupted by hallucinated or erroneous tool outputs (Huang et al., 2025). Moreover, as the number of available tools grows into the thousands, efficiently retrieving and selecting appropriate tools becomes increasingly challenging, requiring sophisticated retrieval mechanisms beyond simple keyword matching (Xu et al., 2024). Recent approaches still lack a unified framework that provides formal guarantees for when tools can be safely invoked and when their results can be trusted. The absence of explicit state management and contract-based verification means that errors can propagate through the reasoning chain, making it difficult to identify and debug failures in complex

*Corresponding author.
[1]Our code is public at https://github.com/OceannTwT/ToolGate.

Findings of the Association for Computational Linguistics: ACL 2026, pages 9653–9684

multi-step tool-calling scenarios.

To address these limitations, we propose **ToolGate**, a forward execution framework that provides logical safety guarantees and verifiable state evolution for LLM tool calling. ToolGate introduces an explicit symbolic state space that maintains a typed key-value mapping representing trusted world information throughout the reasoning process. Each tool is formalized as a Hoare-style contract with a precondition that gates tool invocation and a postcondition that determines whether the tool's result can be committed to update the state. By combining Retrieval with embedding semantic search for efficient tool retrieval and hoare contract logical checks for safe tool execution, ToolGate ensures that the symbolic state evolves only through verified tool executions, preventing invalid or hallucinated results from corrupting the world representation.

Our Contributions. Our contributions are detailed as follows.

- We present ToolGate, a novel framework that formalizes tool calling through Hoare-style contracts, providing logical safety guarantees and verifiable state evolution for LLM tool-augmented systems.
- We introduce an explicit symbolic state space that maintains trusted world information throughout reasoning, enabling precise precondition and postcondition checking for tool invocations.
- We demonstrate that contract-based verification significantly improves the reliability and debuggability of tool-augmented LLM systems while maintaining competitive performance on complex multi-step reasoning tasks.

2 Related Work

2.1 Tool Learning of LLMs

The integration of external tools with Large Language Models has emerged as a critical capability for extending LLM reasoning beyond text generation to real-world interactions. Early work on function calling, such as OpenAI's function calling API, enables LLMs to invoke external functions with structured parameters(Ouyang et al., 2022). The ReAct framework formalizes the reasoning-acting paradigm, where LLMs explicitly alternate between reasoning steps and tool invocations, demonstrating improved performance on complex multi-step reasoning tasks (Yang et al., 2024a). Building on this foundation, Tool Learning has emerged as an effective paradigm for significantly expanding the capabilities of Large Language Models (Schick et al., 2023; Qin et al., 2023; Yu et al., 2025). Early research proposed that by integrating LLMs with external tools—such as program executors or search engines (Erdogan et al., 2024; Paranjape et al., 2023). To comprehensively measure performance in tool usage, researchers have introduced a series of benchmarks to systematically evaluate dimensions ranging from API selection and parameter generation quality to generalization capabilities (Ye et al., 2025; Patil et al., 2024; Du et al., 2024). These techniques have been extended to multimodal tasks like GUI Agents (Zhang et al., 2025a; Liu et al., 2025b) and specialized domains (Su et al., 2025). More recently, Reinforcement Learning (RL) has been incorporated into the framework to further optimize tool-learning performance (Qian et al., 2025; Li et al., 2025), yielding significant results in information retrieval and dynamic reasoning. These developments demonstrate that tool-augmented LLMs are revealing vast potential for open-domain general reasoning.

2.2 Hoare Logic and Formal Verification

Hoare logic (Hoare, 1969) provides a formal system for reasoning about program correctness through preconditions and postconditions. In recent years, Formal Verification and Hoare logic has been increasingly introduced into the field of deep learning to characterize and constrain the provable behaviors of neural network systems under different inputs and internal states (Corsi et al., 2021). As deep learning models are being widely deployed in high-risk and safety-critical domains such as autonomous driving, robotics control, medical decision-making, and industrial systems, ensuring that model outputs are not only effective but also verifiable and compliant with predefined specifications has become an increasingly important problem (Meng et al., 2022; Swaroop et al., 2024). In this context, the precondition–postcondition framework provided by Hoare logic is used to specify the functional, safety, or robustness properties that neural networks must satisfy under given input conditions or first-order logic (Yang et al., 2024b; Han et al., 2024), and it is further combined with neural network verification and LLMs to form a unified and rigorous approach to reasoning and ver-

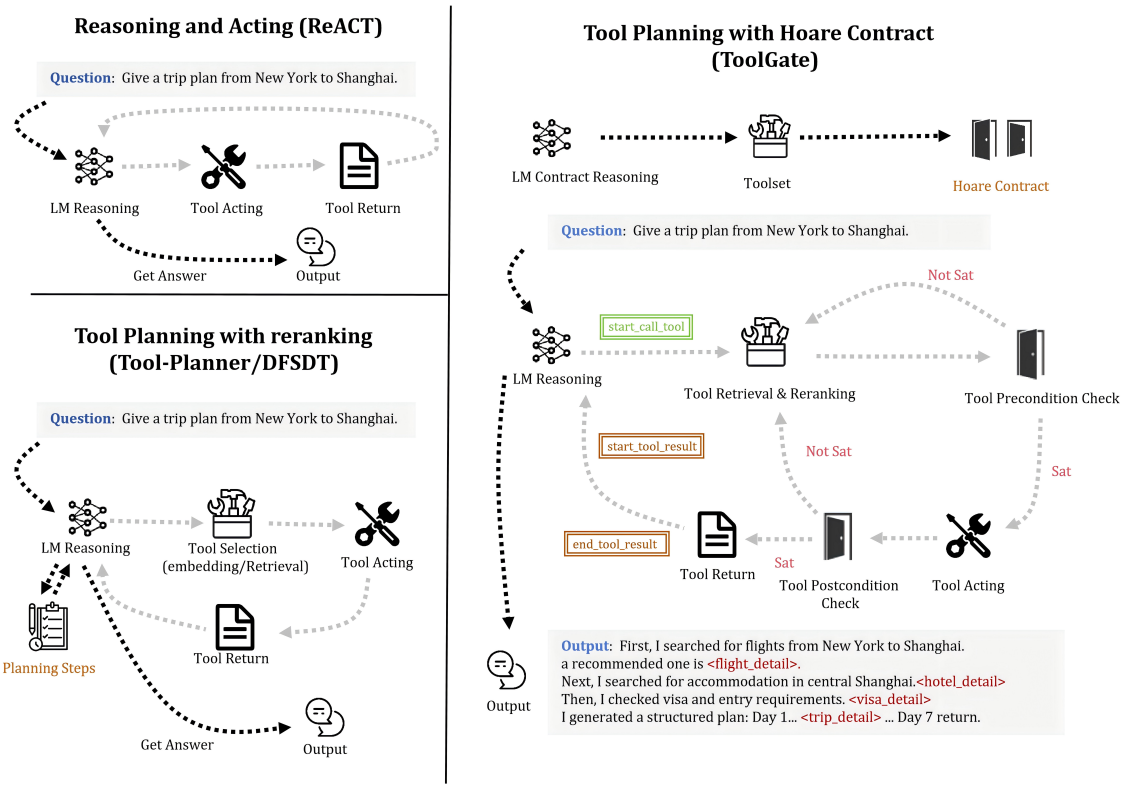

Figure 1: Overview of ToolGate, which formalizes tool calling via Hoare-style contracts (P/Q derived from tool specifications) and maintains a trusted state S to gate and verify each invocation.

ification (Lee et al., 2025; Grigorev et al., 2025; Wang et al., 2019; Lin et al., 2024).

3 Methodology

3.1 Problem Setting and Overview

Tool learning equips LLMs with the ability to plan, invoke, and reason over external tools. However, hallucination propagation and unreliable tool planning remain major bottlenecks, frequently leading to unstable and unreliable outcomes. To address these problems, we propose **ToolGate**, a framework integrates both probabilistic reasoning foundations and logically verifiable guarantees. It consists of a typed symbolic world state S that maintains trusted information, Hoare-style logical contracts $\{P_t\}\ t\ \{Q_t\}$ for tools, and a probabilistic reasoning mechanism driven by large language models but constrained by Hoare logic.

Problem description. Given an input sequence x and a set of available tools $T = \{t_1, t_2, \ldots, t_n\}$, tool learning aims to produce an answer through:

$$y = \arg\max_{y_i} P(y_i \mid x,\ T_0 = \{t_{x_i}\}) \qquad (1)$$

where T_0 represents the tools selected based on the input x, along with their corresponding outputs.

3.2 Symbolic State Construction and Tool Contracts

To ensure that tool execution is not driven merely by unstructured natural-language memory but is grounded in a verifiable and logically interpretable world model, we first construct a typed symbolic state space Σ. We maintain a trusted symbolic state $S \in \Sigma$, where each element is represented as a tuple (k, v, σ) capturing a key, its value, and its associated type, i.e.,

$$\Sigma = \{(k, v, \sigma)\} \qquad (2)$$

This representation allows the system to explicitly encode "what is currently known" in a structured and inspectable manner. Verified entities, intermediate reasoning outcomes, and validated tool outputs are all written into this state space. To enforce logical consistency throughout reasoning and tool execution, we additionally define a set of logical predicates over Σ to express existence constraints, type consistency, and semantic invariants, and we denote $S \models \varphi$ to indicate that a given symbolic state S satisfies a logical condition φ.

To prevent the model from invoking tools arbitrarily and reduce hallucinated or unconstrained execution behavior, we assign each tool $t \in \mathcal{T}$ a Hoare-style logical contract of the form

$$\{P_t\}\ t\ \{Q_t\} \tag{3}$$

The precondition $P_t : \Sigma \to \{true, false\}$ specifies the minimal state requirements that must be satisfied for the tool to be legally callable, meaning a tool is not executable unless $S \models P_t$ holds. Meanwhile, the postcondition $Q_t : \Sigma \times R_t \to \{\text{true}, \text{false}\}$. constrains the structural validity, typing correctness, and semantic consistency of the runtime output r_t, while also defining how a verified result updates the system state.

Contract derivation. It is important to clarify that preconditions and postconditions are *not* generated dynamically by the LLM during reasoning. Instead, they are derived via structured extraction from each tool's official documentation and interface specifications (e.g., OpenAPI / JSON Schema). Specifically, precondition P_t is extracted from the `required_parameters` in the tool's interface definition: if an input field is marked as `required = true` in the schema, we obtain a corresponding predicate (e.g., exists(city) $\wedge$ exists(date)), meaning the tool is callable only when the current symbolic state S already contains both fields. Postcondition Q_t is obtained from the response schema: required fields, types, and structure declared in the schema are mapped directly to postcondition predicates (e.g., has_field$(r, \text{temperature}) \wedge$ is_number$(r.\text{temperature})$). When the documentation does not provide a structured schema (approximately 25% of tools in ToolBench provide only default `response_examples` of the form `{api_list: []}`), we adopt $Q = \text{True}$ as the default postcondition. For MCP-Universe, P/Q are derived from the MCP protocol's `inputSchema` and typed response interfaces. Concrete extraction examples are provided in the Appendix.

3.3 Tool Call and Reranking

We first treat the model's reasoning as a process of conditional probability propagation over time. At the k-th step, the reasoning state is represented as

$$p(R_k \mid q, H, S_k) \tag{4}$$

where q denotes the current user query, H represents the stable and externally visible dialogue history, and S_k denotes the trusted symbolic state at this time. We define R_k as the current reasoning trajectory, which records the intermediate reasoning content, reasoning path, and any tool results already injected before step k. In this formulation, H tracks externally observable interaction, while R_k captures the evolution of the model's internal reasoning process, making it clear, in subsequent tool selection and state updates, which information originates from the user and which originates from internal reasoning.

Next, we turn the choice to call a tool into an endogenous stochastic decision within the reasoning process itself. Under the current information state, the model estimates:

$$p(\boxed{\texttt{<start\_call\_tool>}} \mid q, H, S_k, R_k) \tag{5}$$

and this probability directly drives whether the model generates <start_call_tool>. Once this marker appears, the system enters the tool selection and execution phase; when <start_tool_result>, <end_tool_result> are later concatenated, the system exits the tool phase and returns to pure natural language answering. This design allows tool usage to be determined by the model's uncertainty and task requirements at the moment, rather than by inflexible hand-crafted triggers, enabling smoother adaptation to scenarios where tools are sometimes necessary and sometimes unnecessary.

Based on the current query q, dialogue history H, symbolic state S_k, and reasoning trajectory R_k, we construct a tool requirement representation:

$$u_k = f(q, H, S_k, R_k) \tag{6}$$

which provides a structured description of the present subproblem and clarifies what the system aims to achieve and what type of tool output it expects. We then treat u_k as a query to retrieve from the large tool set $\mathcal{T}$, using vector embeddings jointly to extract the Top-K candidate tools:

$$\mathcal{C}_k = \text{TopK-Retrieve}(u_k, \mathcal{T}) \tag{7}$$

which effectively shrinks the tool space, preserving only a small, highly relevant candidate set.

With the candidate set $\mathcal{C}_k$, we apply a reranking model within $\mathcal{C}_k$, producing a refined ranking distribution:

$$p_{\text{rank}}(t \mid u_k), \quad t \in \mathcal{C}_k \tag{8}$$

3.4 Tool Contracts on Planning

For each candidate tool $t \in \mathcal{C}_k$, we determine whether its precondition is satisfied under the current symbolic state S_k, using the indicator $\mathbf{1}[S_k \models P_t]$ to eliminate all tools whose prerequisites are unmet. We then renormalize the ranking distribution only over those tools whose preconditions hold, forming a logically valid execution policy:

$$p^*(t \mid q, H, S_k, R_k) = \frac{p_{\text{rank}}(t \mid u_k) \cdot \mathbf{1}[S_k \models P_t]}{\sum_{t' \in \mathcal{C}_k} p_{\text{rank}}(t' \mid u_k) \cdot \mathbf{1}[S_k \models P_{t'}]} \tag{9}$$

This filtering mechanism transcends simple semantic matching by establishing formal execution admissibility; it necessitates that the current state S_k satisfies the weakest precondition of the selected tool, denoted as $S_k \models wp(t, P_t)$. By embedding such deterministic constraints into the probabilistic sampling process, we ensure that the model's trajectory remains within a logically grounded solution space rather than relying on unconstrained heuristic transitions.

We treat $p^*(t)$ as a logically constrained policy distribution and sample from it:

$$t^* \sim p^*(t \mid q, H, S_k, R_k) \tag{10}$$

As long as a tool is both legal and meaningfully relevant, it naturally retains the chance to be explored, while its sampling probability reflects its contextual priority. Once the final tool t^* is selected and invoked, it returns a result r_t. Before updating the system state with this output, we introduce a safety gate, a runtime contract verification process that checks whether the returned result satisfies the Hoare postcondition Q_t. We formalize this as a binary acceptance event $\mathcal{A}_t \in \{0, 1\}$, with conditional probability $p(\mathcal{A}_t = 1 \mid S_k, r_t, Q_t)$ and implement it as a concrete verification function:

$$\mathcal{A}_t = \begin{cases} 1, & \text{if } (S_k, r_t) \models Q_t \wedge \text{wf}(r_t), \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

Through this step, every tool output must satisfy structural validity, value range constraints, and format expectations before it can affect the global state. Only if verification passes does the symbolic state update:

$$S_{k+1} = \begin{cases} \mathsf{Update}_t(S_k, r_k), & \mathcal{A}_t = 1, \\ S_k, & \mathcal{A}_t = 0, \end{cases} \tag{12}$$

and the accepted result is injected into the subsequent reasoning trajectory R_{k+1}. If verification fails, the result is discarded entirely, preventing contaminated outputs from propagating and providing a clear debugging breakpoint.

Simultaneously, we inject the verified results wrapped in <start_tool_result> and <end_tool_result> tags into the subsequent reasoning trajectory R_{k+1}, enabling both subsequent natural language reasoning and the next round of tool selection to fully leverage this newly acquired trusted information.

Building on this foundation, we treat the entire system as a family of stochastic trajectories τ, each consisting of (S_k, R_k), the chosen tools t_k, and the acceptance events $\mathcal{A}_k$. The system performs probabilistic reasoning over all feasible execution trajectories, and the final output y can be expressed via trajectory-level marginalization:

$$p(y \mid q, H) = \sum_{\tau} p\big(y \mid q, H, \tau\big)\, p\big(\tau \mid q, H\big) \tag{13}$$

where $p(\tau \mid q, H)$ integrates all components discussed above: tool trigger probability, requirement abstraction, retrieval and ranking, contract filtering, constrained sampling, and acceptance verification.

To ensure Hoare contracts regulate not only local behavior but also the global behavior space, we impose a strict trajectory-level constraint: if any trajectory τ violates any tool precondition P_t or postcondition Q_t at any step, then

$$p(\tau \mid q, H) = 0 \tag{14}$$

Under this formulation, reasoning and sampling proceed exclusively within a trajectory subspace that adheres to predefined contracts, thereby providing a formal logical justification for each state transition and tool execution.

4 Experimental Setup

4.1 Dataset.

We utilize **ToolBench** (Qin et al., 2023) and **MCP-Universe** (Luo et al., 2025) as our experimental datasets. ToolBench contains more than 16,000 APIs organized into structured tool categories, covering a wide range of functional capabilities. These settings jointly assess both local tool invocation ability and global planning robustness.

Table 1: Main experimental results on ToolBench and MCP-Universe. We report Pass Rate (%) and Win Rate (%) for ToolBench G1, G2, and G3 tasks, and Success Rate (%) for three MCP-Universe subtasks.

Model	Method	ToolBench						MCP-Universe		
		G1		G2		G3		Location Navigation	Repository Management	Financial Analysis
		Pass.	Win.	Pass.	Win.	Pass.	Win.			
Qwen-3-235B	ReACT	50.5	–	53.5	–	46.0	–	11.10	9.09	50.0
	DFSDT	57.0	53.8	61.5	67.5	48.8	56.8	11.10	12.12	50.0
	LATS	62.5	59.3	78.0	70.3	77.8	83.3	15.54	15.15	52.5
	ToolChain*	65.0	62.8	79.3	72.5	78.0	83.5	16.65	18.18	55.0
	Tool-Planner	60.3	58.0	70.5	68.8	65.5	72.3	13.32	12.12	52.5
	ToolGate	**68.3**	**65.5**	**82.5**	**78.0**	**81.0**	**82.3**	**18.87**	**21.21**	**60.0**
Deepseek V3.2	ReACT	52.0	48.5	55.3	51.0	48.5	53.5	12.21	12.12	52.5
	DFSDT	58.5	55.0	63.0	69.3	50.3	58.8	13.32	15.15	55.0
	LATS	65.3	61.8	80.0	72.5	80.3	85.5	17.76	18.18	60.0
	ToolChain*	68.8	65.0	82.5	75.3	81.0	88.8	18.87	21.21	62.5
	Tool-Planner	62.5	60.3	73.8	70.0	68.0	75.5	15.54	15.15	57.5
	ToolGate	**72.0**	**70.3**	**85.5**	**80.0**	**85.3**	**81.3**	**22.20**	**24.24**	**67.5**
GPT-5.2	ReACT	63.5	62.8	65.0	63.2	58.3	59.5	18.87	24.24	65.0
	DFSDT	70.0	68.5	75.3	78.0	63.8	70.5	19.98	27.27	67.5
	LATS	80.3	78.8	88.5	85.3	85.0	90.8	28.86	36.36	82.5
	ToolChain*	82.8	80.0	90.5	88.3	88.5	92.5	29.97	39.39	85.0
	Tool-Planner	75.5	72.3	82.0	80.5	78.3	85.0	25.53	30.30	75.0
	ToolGate	**85.5**	**83.5**	**93.0**	**90.5**	**91.8**	**95.3**	**35.52**	**45.45**	**90.0**
Gemini 3 Pro	ReACT	60.0	60.5	63.8	57.0	55.5	56.5	16.65	21.21	62.5
	DFSDT	68.3	65.5	72.0	75.8	60.3	68.5	17.76	24.24	65.0
	LATS	78.5	75.3	85.8	82.0	82.5	88.3	26.64	33.33	77.5
	ToolChain*	80.0	78.5	88.3	85.0	85.8	90.0	27.75	36.36	80.0
	Tool-Planner	73.8	70.0	80.5	78.3	75.0	82.8	22.20	27.27	72.5
	ToolGate	**83.0**	**80.5**	**91.3**	**88.0**	**90.0**	**93.5**	**33.30**	**42.42**	**87.5**

MCP-Universe reflects more realistic multi-tool environments. It aggregates diverse tools, plug-ins, and APIs from real-world systems covering information retrieval, automation, data processing, system operations, and task execution. We use the tools selected in ToolBench and MCP-Universe along with their official documentation, specifications, and usage descriptions to extract structured functional representations. More dataset details are provided in Appendix C.

4.2 Evaluation Metrics

For ToolBench, we adopt two evaluation metrics from ToolEval (Qin et al., 2023). The first metric is **Pass Rate**, computed as the proportion of successfully completed tasks, which reflects overall task-solving capability. The second metric is **Win Rate**, where we compare the execution plans and results produced by our framework with those generated by Qwen-3 235B-ReACT and request LLMs judges to determine which solution is superior. If our method yields a better solution, we mark it as a win; if it is equivalent or worse, we mark it as a tie or loss. Win Rate therefore measures both reasoning quality and execution superiority.

For MCP-Universe, we evaluate **Success Rate** and execution stability. Many tasks in MCP-Universe involve relatively fewer tool invocation steps but arise from real-world complex systems.

4.3 Baselines

We compare our framework against the following representative tool-use and planning baselines: **ReACT** (Yao et al., 2022), **DFSDT** (Qin et al., 2023), **LATS** (Zhou et al., 2024), **ToolChain*** (Zhuang et al., 2024), **Tool-Planner** (Liu et al., 2025c), More baselise details are provided in Appendix D.

4.4 Models

We evaluate our framework across a range of large language models to verify generality and robustness. Proprietary models include **Gemini 3 Pro** (Google Inc., 2025), **GPT-5.2** (OpenAI, 2025). Open-source models include **DeepSeek V3.2** (Liu et al., 2025a), **Qwen3-235B-A22B-Instruct-2507** (Yang et al., 2025a). These models cover heterogeneous training paradigms, reasoning capabilities, and scales. While we use **Qwen3-embedding-0.6B** and **Qwen3-Reranker-0.6B** (Zhang et al., 2025b) for tool embedding and retrieval.

5 Experiments

5.1 Main Results

As shown in Table 1, we conduct comprehensive evaluations on ToolBench (G1/G2/G3) and MCP-Universe.

Table 2: Comprehensive ablation study of the Hoare logic verification module. We compare the full ToolGate architecture against variants: **No** $\{P\}$ **check** (skips pre-condition validation) and **No** $\{Q\}$ **check** (skips post-condition assertion). **MCP-Avg** represents the mean success rate of MCP subtasks.

Model	Method	ToolBench						MCP-Universe			
		G1		G2		G3		Loc.	Repo.	Fin.	MCP-Avg
		Pass.	Win.	Pass.	Win.	Pass.	Win.				
DeepSeek V3.2	ReACT	52.0	48.5	55.3	51.0	48.5	53.5	12.2	12.1	52.5	25.6
	DFSDT	58.5	55.0	63.0	69.3	50.3	58.8	13.3	15.2	55.0	27.8
	ToolChain*	68.8	65.0	82.5	75.3	81.0	88.8	18.9	21.2	62.5	34.2
	ToolGate w/o Hoare	57.2	53.8	61.5	67.5	49.8	57.5	12.8	14.5	54.2	27.2
	– No $\{P\}$ **check**	67.8	66.5	79.5	77.2	78.4	82.8	19.8	21.5	62.2	34.5
	– No $\{Q\}$ **check**	63.2	61.0	71.5	72.8	70.8	73.5	16.2	18.2	58.2	30.9
	ToolGate (Full)	**72.0**	**70.3**	**85.5**	**80.0**	**85.3**	**81.3**	**22.2**	**24.2**	**67.5**	**38.0**
GPT-5.2	ReACT	63.5	62.8	65.0	63.2	58.3	59.5	18.9	24.2	65.0	36.0
	DFSDT	70.0	68.5	75.3	78.0	63.8	70.5	20.0	27.3	67.5	38.3
	ToolChain*	82.8	80.0	90.5	88.3	88.5	92.5	30.0	39.4	85.0	51.5
	ToolGate w/o Hoare	69.2	67.5	74.5	76.8	62.5	69.2	19.5	26.8	66.5	37.6
	– No $\{P\}$ **check**	81.2	79.5	89.2	88.0	86.4	90.5	31.0	41.5	85.0	52.5
	– No $\{Q\}$ **check**	75.5	74.0	82.8	83.5	79.2	82.0	25.5	33.5	79.6	46.2
	ToolGate (Full)	**85.5**	**83.5**	**93.0**	**90.5**	**91.8**	**95.3**	**35.5**	**45.5**	**90.0**	**57.0**

For ToolBench. The results show that ToolGate achieves the best or near-best performance across all models and all evaluation benchmarks. On ToolBench, ToolGate leads to substantial improvements in both Pass Rate and Win Rate across all three task groups. For instance, under GPT-5.2, ToolGate reaches **85.5 / 83.5**, **93.0 / 90.5**, and **91.8 / 95.3** on G1/G2/G3 respectively, outperforming the strongest baseline ToolChain* by approximately $4-6\%$ in Win Rate. Similar improvements are consistently observed on Qwen-3-235B, DeepSeek V3.2, and Gemini 3 Pro, demonstrating that ToolGate is **model-agnostic** and provides stable enhancement to tool reasoning and execution capabilities across different LLM backbones.

For MCP-Universe. The advantage of ToolGate becomes even more pronounced, which emphasizes long-horizon tool dependencies and real-world execution robustness. ToolGate yields $3-7\%$ improvements over ToolChain* in *Location Navigation* and *Repository Management*, and delivers state-of-the-art performance on *Financial Analysis*. Notably, GPT-5.2 with ToolGate achieves **45.45** in Repository Management and **90.0** in Financial Analysis, substantially surpassing all competing systems. These results suggest that ToolGate not only improves task success rates, but also significantly enhances stability and robustness when executing complex tool chains.

On Multi-tool instructions tasks. Experimental results indicates that these gains are not merely due to stronger heuristics or more aggressive exploration, but primarily arise from ToolGate's Hoare-logic-based formal constraint mechanism. During reasoning, the system explicitly maintains a trusted state set S and constructs a Hoare Triple $\{P\}\ C\ \{Q\}$ for each tool invocation, enforcing precondition and postcondition validation. This enables ToolGate to significantly reduce error accumulation in complex ToolBench tasks such as G2 and G3, resulting in higher and more stable Win Rates; meanwhile, in MCP-Universe, it effectively mitigates long-horizon reasoning drift, leading to sustained performance gains under multi-tool dependency and real-world execution constraints.

5.2 Ablation Studies

To evaluate the structural dependency of ToolGate on its formal verification mechanism, we conducted a systematic ablation study across both DeepSeek V3.2 and GPT-5.2. We specifically isolated the Hoare logic module to observe its impact on tool-use efficacy. As detailed in Table 2, the results reveal a critical finding: removing the formal verification layer leads to a performance level that falls marginally below the standard DFSDT baseline.

For instance, with GPT-5.2, the MCP-Avg success rate for the ToolGate without Hoare filtering is 37.6%, which is slightly lower than the 38.3% achieved by DFSDT. This trend is consistent across

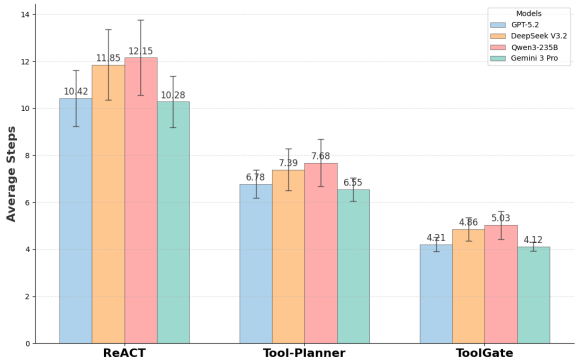

Figure 2: Comparison of Average Tool-Calling Steps in Tool-Bench.

DeepSeek V3.2 as well. This indicates that without the pruning capabilities provided by Hoare logic, the underlying search architecture of ToolGate becomes less efficient than a conventional depth-first search strategy.

The results demonstrate that Hoare logic verification is the key factor behind ToolGate's search efficiency. The two fundamental components of the Hoare logic framework, the precondition $\{P\}$ and the postcondition $\{Q\}$ serve complementary functions in guiding tool-invocation decisions. Empirical evidence highlights that the absence of $\{Q\}$ checks is substantially more detrimental than the absence of $\{P\}$ checks. For instance, on GPT-5.2, the MCP-Avg success rate drops by 10.8% when $\{Q\}$ checks are removed, compared to a 4.5% decrease when $\{P\}$ checks are omitted. The full version of ToolGate enforces a rigorous $\{P\}C\{Q\}$ logical closed-loop, ensuring that every step within the search process is both formally valid and substantively effective. the performance gap between the full ToolGate model and its ablated counterpart confirms that formal logic is the primary catalyst for superior reliability and task success.

5.3 Tool Reasoning Efficiency

To evaluate the search efficiency of ToolGate, we focus on the average number of tool-calling steps required to complete tasks. This metric serves as a proxy for the model's ability to navigate complex state-spaces without redundant exploration.

As shown in Figure 2, ToolGate consistently achieves the most concise tool-calling trajectories across both GPT-5.2 and DeepSeek V3.2 backbones. Specifically, when using GPT-5.2, ToolGate reduces the average calling steps from 6.78 to 4.21, representing a 37.9% improvement in efficiency. While traditional methods like ReACT and Tool-Planner often fall into "trial-and-error" loops due to a lack of environmental awareness, ToolGate maintains a trajectory close to the theoretical optimal path.

The efficiency of ToolGate is primarily attributed to the Hoare logic verification module. In the vast state-space of Tool-Bench, logical conflicts between tool preconditions and environmental states are frequent. Unlike Tool-Planner, which explores branches based on probabilistic heuristics, ToolGate applies formal constraints to prune the search tree. By verifying the feasibility of a tool call before execution, the system effectively collapses the search space, eliminating branches that are logically destined to fail.

5.4 Fine-grained Rejection Distribution

To further investigate the internal decision-making mechanism of ToolGate, we conducted a comprehensive trace of all tool-invocation attempts during the evaluation. Our results indicate that in high-complexity benchmarks such as **MCP-Universe**, the formal verification layer intercepts approximately **29.4%** of the total tool-calling requests. Based on a fine-grained analysis of these rejections, we categorize the findings into three key areas:

Static Pruning via $\{P\}$ The pre-condition check primarily filters **parametric hallucinations** and **state dependency violations**. By intercepting invalid IDs and out-of-sequence calls before execution, $\{P\}$ significantly reduces computational overhead and prevents the search tree from expanding into invalid branches.

Dynamic Rectification via $\{Q\}$. The post-condition assertion captures sophisticated failures that standard models miss. By mandating semantic alignment and state consistency, $\{Q\}$ identifies logically vacuous steps and triggers immediate backtracking, preventing cascading errors. Notably, the "Empty/Null" category (6.3%) refers to responses that lack required structure or fields as declared in the schema—not a blanket rejection of all empty results. When the schema explicitly allows a field to be empty (e.g., a search API that may legitimately return an empty list), Q only enforces structural and type constraints without imposing non-emptiness. We only add a non-empty constraint when the schema explicitly requires it (e.g., declared as "list of str with length 1"). This design ensures that Q reflects each tool's specification

Table 3: Fine-grained analysis of logical rejections across Hoare components. Rates are normalized against the total number of tool invocations in the MCP-Universe benchmark.

Verification Phase	Specific Error Sub-category	Abs. Rate (%)
Pre-condition $\{P\}$	Value/Entity Hallucination	8.4%
	Schema & Format Violation	5.1%
	State Dependency Missing	4.1%
	Subtotal $\{P\}$ *Rejections*	**17.6%**
Post-condition $\{Q\}$	Empty/Null	6.3%
	Semantic Constraint Mismatch	3.7%
	State Update Inconsistency	1.8%
	Subtotal $\{Q\}$ *Rejections*	**11.8%**
Total	**Combined Rejection Rate**	**29.4%**

rather than injecting benchmark-specific priors.

While $\{P\}$ optimizes **efficiency** by pruning 17.6% of invalid paths statically, $\{Q\}$ ensures **task success** by dynamically rectifying the remaining 11.8% of logical drifts. Together, they form a logical closed-loop that anchors the agent to the correct semantic trajectory.

5.5 End-to-End Latency Analysis

To assess the practical deployment cost of contract verification, we measure the average running time (seconds) of each method on ToolBench G1/G2/G3 under the same hardware and network conditions. As shown in Table 4, ToolGate's average running time is slightly higher than ReACT but substantially lower than DFSDT, ToolChain*, and Tool-Planner. This demonstrates that the overhead of P/Q verification is minimal relative to the savings from pruning invalid tool calls: by eliminating execution branches that are logically destined to fail, ToolGate avoids the wasted API calls and backtracking that inflate the latency of search-heavy methods. The benefit of gating in reducing invalid calls yields competitive overall latency, confirming that contract-based verification is both effective and efficient. We additionally conduct a contract robustness analysis by randomly removing fractions of P/Q constraints; results in Appendix B confirm that the framework degrades gracefully under incomplete contracts.

6 Conclusions

In this paper, we introduce ToolGate, a forward execution framework that provides logical safety guarantees and verifiable state evolution for LLM tool calling. By formalizing each tool as a Hoare-style contract derived from interface specifications and enforcing precondition gating and postcondition verification, ToolGate ensures that the symbolic state evolves only through verified tool executions, preventing invalid or hallucinated results from corrupting the world representation. Experimental validation demonstrates that ToolGate significantly improves the reliability and verifiability of tool-augmented LLM systems while maintaining competitive performance on complex multi-step reasoning tasks, and our robustness analysis confirms that the framework degrades gracefully under incomplete contracts with minimal latency overhead. This work establishes a foundation for building more trustworthy and debuggable AI systems that integrate language models with external tools.

Table 4: Average running time (seconds) on ToolBench. Lower is better.

Method	GPT-5.2			DeepSeek V3.2		
	G1	G2	G3	G1	G2	G3
ReACT	132	157	98	138	162	104
ToolChain*	228	278	268	235	285	275
DFSDT	808	1084	692	815	1092	698
Tool-Planner	256	301	289	262	308	295
ToolGate	**145**	**172**	**108**	**152**	**178**	**112**

Limitations

Despite its contributions, several limitations of ToolGate should be acknowledged. First, while the benchmark covers a diverse range of scenarios, its current scope is primarily restricted to text-based and structured data interactions, leaving multi-modal tools and long-chain, multi-step collaborative tasks as areas for future expansion. Second, the evaluation environment is largely static, which may not fully capture the complexities of real-world API dynamics, such as network latency, rate limits, or fluctuating data states that can interfere with real-time decision-making. Furthermore, our evaluation metrics remain predominantly quantitative; future work is needed to develop more fine-grained qualitative assessments of a model's explanatory reasoning and its ability to proactively solicit missing information from users. Finally, regarding the trustworthiness of contracts: on ToolBench, the response schemas from which Q is derived are produced with LLM assistance and expert-written in-context examples (as described in ToolBench Appendix A.2), so Q reflects the "expected

shape" as implied by that process rather than a ground-truth formal specification. In deployment settings, one could instead derive Q from official OpenAPI specifications when available. P/Q updates should also be performed in conjunction with API specification updates and propagated across parent and derived interfaces, which we leave as future work for maintaining contract consistency at scale.

Ethics Considerations

ToolGate is developed as a general framework for formal, verifiable, and responsible tool use in large language model reasoning. All experiments are conducted on publicly available benchmarks or open-source tool environments, and no private or personally identifiable information is collected, accessed, or utilized throughout our work. The tools invoked in our experiments are either simulated environments or publicly documented APIs with appropriate usage permissions. Our framework does not generate, store, or infer sensitive personal attributes, nor does it target any specific demographic groups; instead, it focuses on improving reliability, interpretability, and safety in model-based tool invocation by enforcing logical constraints and verifiable execution conditions. During evaluation, we strictly follow the licenses and terms of use associated with the released LLMs, datasets, APIs, and benchmark platforms. We emphasize that ToolGate is designed to enhance trustworthy AI behaviors rather than to bypass safeguards or enable harmful automation, and the methodology can be integrated with additional safety filters, auditing processes, and access control mechanisms when deployed in real-world systems, contributing positively toward building transparent, controllable, and ethically aligned AI tool-use systems.

Acknowledgements

This work is supported by the Key R&D Program of Ningbo under Grant No. 2024Z115.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Ma teusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *ArXiv*, abs/2005.14165.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311.

Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. 2021. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In *Uncertainty in Artificial Intelligence*, pages 333–343. PMLR.

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: Self-reflective, hierarchical agents for large-scale api calls. In *Forty-first International Conference on Machine Learning*.

Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Richard Charles Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. Tinyagent: Function calling at the edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 80–88.

Google Inc. 2025. A new era of intelligence with gemini 3. https://blog.google/products/gemini/gemini-3/. Accessed: 2025-11-18.

Danil S Grigorev, Alexey K Kovalev, and Aleksandr I Panov. 2025. Verifyllm: Llm-based pre-execution task plan verification for robots. *arXiv preprint arXiv:2507.05118*.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, et al. 2024. Folio: Natural language reasoning with first-order logic. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 22017–22031.

Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Xiao Hu. 2025a. AdPercept: Visual saliency and attention modeling in ad 3D design. In *2025 5th International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA)*, pages 1258–1261. IEEE.

Xiao Hu. 2025b. GenPlayAds: Procedural playable 3D ad creation via generative model. In *2025 International Conference on Artificial Intelligence, Human-Computer Interaction and Natural Language Processing (ICAHN)*, pages 128–131. IEEE.

Yue Huang, Chujie Gao, Siyuan Wu, Haoran Wang, Xiangqi Wang, Yujun Zhou, Yanbo Wang, Jiayi Ye, Jiawen Shi, Qihui Zhang, et al. 2025. On the trustworthiness of generative foundation models: Guideline, assessment, and perspective. *arXiv preprint arXiv:2502.14296*.

Christine P. Lee, David Porfirio, Xinyu Jessica Wang, Kevin Chenkai Zhao, and Bilge Mutlu. 2025. Veriplan: Integrating formal verification and llms into end-user planning. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, New York, NY, USA. Association for Computing Machinery.

Chengpeng Li, Zhengyang Tang, Ziniu Li, Mingfeng Xue, Keqin Bao, Tian Ding, Ruoyu Sun, Benyou Wang, Xiang Wang, Junyang Lin, et al. 2025. Teaching language models to reason with tools. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. 2024. Fvel: Interactive formal verification environment with large language models via theorem proving. *Advances in Neural Information Processing Systems*, 37:54932–54946.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Liang Liu, Yaxuan Guo, Han Xiao, Weifeng Lin, Yuxiang Chai, Yue Han, Shuai Ren, et al. 2025b. Llm-powered gui agents in phone automation: Surveying progress and prospects. *arXiv preprint arXiv:2504.19838*.

Yanming Liu, Xinyue Peng, Jiannan Cao, Shi Bo, Yuwei Zhang, Xuhong Zhang, Sheng Cheng, Xun Wang, Jianwei Yin, and Tianyu Du. 2025c. Tool-planner: Task planning with clusters across multiple tools. In *The Thirteenth International Conference on Learning Representations*.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025. MCP-universe: Benchmarking large language models with real-world model context protocol servers. In *Workshop on Scaling Environments for Agents*.

Mark Huasong Meng, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong. 2022. Adversarial robustness of deep neural networks: A survey from a formal verification perspective. *IEEE Transactions on Dependable and Secure Computing*.

OpenAI. 2025. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2025-08-14.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. 2022. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*.

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

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru WANG, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. ToolRL: Reward is all tool learning needs. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. 2024. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations*.

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

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR.

Zhaochen Su, Linjie Li, Mingyang Song, Yunzhuo Hao, Zhengyuan Yang, Jun Zhang, Guanjie Chen, Jiawei Gu, Juntao Li, Xiaoye Qu, et al. 2025. Openthinkimg: Learning to think with images via visual tool reinforcement learning. *arXiv preprint arXiv:2505.08617*.

Anand Swaroop, Abhishek Singh, Girish Chandra, Shiv Prakash, Sohan Kumar Yadav, Tiansheng Yang, and Rajkumar Singh Rathore. 2024. A comprehensive overview of formal methods and deep learning for verification and optimization. In *2024 International Conference on Decision Aid Sciences and Applications (DASA)*, pages 1–6. IEEE.

Tongwei Tu. 2025a. Log2Learn: Intelligent log analysis for real-time network optimization. In *2025 4th International Conference on Artificial Intelligence, Internet and Digital Economy (ICAID)*, pages 162–166. IEEE.

Tongwei Tu. 2025b. SmartFITLab: Intelligent execution and validation platform for 5G field interoperability testing. In *2025 7th International Conference on Next Generation Data-driven Networks (NGDN)*, pages 108–111. IEEE.

Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. 2019. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554. PMLR.

Minhui Xie and Shujian Chen. 2025. InVis: Interactive neural visualization system for human-centered data interpretation. In *2025 IEEE 7th International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 636–640. IEEE.

Minhui Xie and Boyan Liu. 2025. EvalNet: Sentiment analysis and multimodal data fusion for recruitment interview processing. In *2025 7th International Conference on Artificial Intelligence Technologies and Applications (ICAITA)*, pages 444–448. IEEE.

Haoran Xu. 2025. CivicMorph: Generative modeling for public space form development. In *2025 5th International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA)*, pages 497–501. IEEE.

Qiancheng Xu, Yongqi Li, Heming Xia, and Wenjie Li. 2024. Enhancing tool retrieval with iterative feedback from large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9609–9619.

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

Hsiu-Wei Yang, Abhinav Agrawal, Pavlos Fragkogiannis, and Shubham Nitin Mulay. 2024a. Can ai models appreciate document aesthetics? an exploration of legibility and layout quality in relation to prediction confidence. *ArXiv*, abs/2403.18183.

Jie Yang, Yiqiu Tang, Yongjie Li, Lihua Zhang, and Haoran Zhang. 2025b. Cross-asset risk management: Integrating LLMs for real-time monitoring of equity, fixed income, and currency markets. In *2025 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Jie Yang, Yiqiu Tang, Yongjie Li, Lihua Zhang, and Haoran Zhang. 2025c. Dynamic hedging strategies in derivatives markets with LLM-driven sentiment and news analytics. In *2025 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Yu'an Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. 2024b. Harnessing the power of large language models for natural language to first-order logic translation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6942–6959.

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

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Tao Ji, Qi Zhang, et al. 2025. Tooleyes: Fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios. In *Proceedings of the 31st international conference on computational linguistics*, pages 156–187.

Yuanqing Yu, Zhefan Wang, Weizhi Ma, Shuai Wang, Chuhan Wu, Zhiqiang Guo, and Min Zhang. 2025. Steptool: Enhancing multi-step tool usage in llms via step-grained reinforcement learning. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, CIKM '25, page 3952–3962, New York, NY, USA. Association for Computing Machinery.

Chaoyun Zhang, Shilin He, Liqun Li, Si Qin, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2025a. API agents vs. GUI agents: Divergence and convergence. In *ICML 2025 Workshop on Computer Use Agents*.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. 2025b. Qwen3

embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

Yuhan Zhang. 2025a. CrossPlatformStack: Enabling high availability and safe deployment for products across meta services. In *2025 4th International Symposium on Robotics, Artificial Intelligence and Information Engineering (RAIIE)*, pages 360–363. IEEE.

Yuhan Zhang. 2025b. InfraMLForge: Developer tooling for rapid LLM development and scalable deployment. In *2025 8th International Conference on Computer Information Science and Application Technology (CISAT)*, pages 273–277. IEEE.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*, pages 62138–62160. PMLR.

Bingxin Zhu. 2025a. RAID: Reliability automation through intelligent detection in large-scale ad systems. In *Proceedings of the 2025 2nd International Conference on Digital Society and Artificial Intelligence*, pages 964–967.

Bingxin Zhu. 2025b. TraceLM: Temporal root-cause analysis with contextual embedding language models. In *2025 6th International Conference on Electronic Communication and Artificial Intelligence (ICECAI)*, pages 855–858. IEEE.

Xiaochen Zhu, Caiqi Zhang, Tom Stafford, Nigel Collier, and Andreas Vlachos. 2025. Conformity in large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3854–3872.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. 2024. Toolchain*: Efficient action space navigation in large language models with a* search. In *The Twelfth International Conference on Learning Representations*.

A Industry Tool Orchestration and Contract Grounding

Across production-scale systems, LLMs are increasingly cast not only as text generators but as *controllers* that plan, retrieve, and invoke heterogeneous tools—APIs and microservices, observability and diagnostics pipelines, design and rendering engines, network test harnesses, and data-fusion backends—while remaining accountable to operational and safety requirements. Representative directions include reliability automation and root-cause analytics in large ad and SMB platforms (Zhu, 2025a), LLM-driven cross-asset risk monitoring spanning equity, fixed income, and currency markets (Yang et al., 2025b), intelligent log analysis for real-time network optimization (Tu, 2025a), developer tooling for rapid LLM development and deployment (Zhang, 2025b), human-centered neural visualization for data interpretation (Xie and Chen, 2025), generative 3D playable advertising pipelines (Hu, 2025b), traceability and task communication for LLM-driven diagnostics (Zhu, 2025b), execution and validation platforms for 5G field interoperability (Tu, 2025b), multimodal sentiment analysis for recruitment processing (Xie and Liu, 2025), dynamic hedging strategies informed by sentiment and news analytics (Yang et al., 2025c), high-availability deployment across cross-platform meta services (Zhang, 2025a), ad perception and quality assessment frameworks (Hu, 2025a), and civic morphology modeling for urban planning (Xu, 2025). These application threads highlight recurring **tool contract** touchpoints that align with contract-grounded execution: *preconditions* that gate whether the environment (state, credentials, lab configuration, market feeds, or vehicle/network context) satisfies the assumptions under which a tool may be invoked; *postconditions* that specify acceptable structured outputs and invariants before results are committed to downstream reasoning or dashboards; and *traceable state evolution* in monitoring, testing, and creative pipelines where invalid or vacuous tool returns must be rejected rather than propagated. This landscape motivates formal interfaces—schema-level or Hoare-style specifications (Hoare, 1969)—as the bridge between probabilistic LLM planning and verifiable tool behavior, consistent with the precondition–postcondition gating used throughout this work.

B Contract Robustness Analysis

A natural concern is how sensitive ToolGate's performance is to the quality and completeness of the extracted contracts. If the preconditions or postconditions are incomplete or imperfect, does the framework degrade gracefully? To investigate this, we conduct a controlled experiment where we randomly remove a fraction (5% and 10%) of P/Q constraints from the contracts and re-run the full evaluation pipeline.

As shown in Table 5, contract completeness is positively correlated with task success rate. With 10% of constraints removed, GPT-5.2's Pass Rate

Table 5: Contract robustness analysis. We randomly remove 5% and 10% of P/Q constraints and evaluate on all benchmarks. Pass Rate / Success Rate (%) are reported.

Model	Method	G1	G2	G3	MCP-Loc	MCP-Repo	MCP-Fin
GPT-5.2	ToolChain*	82.8	90.5	88.5	29.97	39.39	85.0
	w/ 10% removed	83.2	90.8	89.2	31.5	41.0	86.5
	w/ 5% removed	84.2	91.8	90.5	34.0	44.0	88.5
	ToolGate (Full)	**85.5**	**93.0**	**91.8**	**35.52**	**45.45**	**90.0**
DeepSeek V3.2	ToolChain*	68.8	82.5	81.0	18.87	21.21	62.5
	w/ 10% removed	69.5	83.2	81.8	19.5	22.0	63.5
	w/ 5% removed	70.8	84.5	83.8	21.0	23.2	65.5
	ToolGate (Full)	**72.0**	**85.5**	**85.3**	**22.20**	**24.24**	**67.5**

on ToolBench-G1 drops from 85.5 to 83.2, and MCP-Financial Analysis from 90.0 to 86.5. Similar trends are observed for DeepSeek V3.2. These results confirm that more complete contracts lead to better error prevention, while even partially complete contracts still outperform the strongest baseline (ToolChain*). This is consistent with the design philosophy of ToolGate: contracts derived from tool specifications capture the tool's expected behavior, and their completeness directly determines how effectively invalid executions are gated.

C Datasets Detailed

C.1 ToolBench Dataset

ToolBench (Qin et al., 2023) is a large-scale instruction tuning and evaluation dataset proposed in the ToolLLM framework, aiming to systematically assess and enhance large language models' ability in tool selection, parameter planning, and executable API invocation in real-world environments. During construction, ToolBench first collects **16,464 real RESTful APIs** from RapidAPI Hub, covering **49 functional categories** (e.g., weather, social media, e-commerce, and mapping services), and extracts structured metadata including API names, documentation, parameter schemas, and usage examples. Based on these APIs, natural-language task instructions are automatically generated using LLMs, and a depth-first search based decision tree (DFSDT) is employed to discover feasible **tool-call trajectories** as solution paths.

In terms of scale, ToolBench provides more than **126K instruction–solution path pairs** under large API spaces, with multiple train/test splits designed to test generalization under unseen instructions, unseen tools, and even unseen tool categories. In addition, ToolBench includes a "classic task set" covering **8 representative tool environments** such as OpenWeather, VirtualHome, and WebShop, each containing about 100 manually verified task instructions and 7–15 tool interfaces, enabling more fine-grained ablation and comparative studies.

For evaluation, ToolBench integrates the ToolEval framework to conduct execution-level assessment on generated API call sequences. Typical metrics include **Pass Rate** (task completion), **Win Rate**. Some works further adopt "Plan.EM" and "Act.EM" to decouple planning quality and execution quality. Due to its realistic and large-scale API space, ToolBench has become a widely adopted benchmark and data source for a series of subsequent tool-use research works.

C.2 MCP-Universe Benchmark

MCP-Universe (Luo et al., 2025) is a comprehensive benchmark proposed for evaluating large language models under the **Model Context Protocol (MCP)** paradigm, focusing on their capability to perform complex tasks via interaction with *real* MCP servers. Unlike offline tool-use datasets, MCP-Universe directly connects to real running MCP services, emphasizing long-horizon interaction, unknown tool discovery, and robust execution under dynamic environments.

MCP-Universe spans **6 core task domains** and **11 different MCP servers**, including Location Navigation, Repository Management, Financial Analysis, 3D Design, Browser Automation, and Web Searching. The benchmark contains **231 task instances** in total, with multiple benchmark configurations derived from different combinations of environments and tools. In addition, the benchmark defines **84 unique evaluators** to cover different evaluation dimensions such as structural cor-

rectness, logical soundness, and consistency with dynamic data sources.

In terms of task distribution, the benchmark is designed to be representative while maintaining reasonable balance: Web Search tasks account for approximately 23.8% (55 tasks), Location Navigation 19.5% (45 tasks), Financial Analysis 17.3% (40 tasks), Browser Automation 16.9% (39 tasks), Repository Management 14.3% (33 tasks), and 3D Design 8.2% (19 tasks). Tasks generally require agents to interact with multiple MCP tools across several rounds, performing complex objectives such as route planning, repository manipulation, portfolio analysis, or automated browser operations.

MCP-Universe further distinguishes **three categories of execution-level evaluators**:

(1) **Format Evaluators**, checking whether model outputs follow the MCP calling specification;

(2) **Static Evaluators**, validating correctness for time-invariant tasks;

(3) **Dynamic Evaluators**, querying real-time data sources to construct ground-truth for time-sensitive tasks such as financial prices or navigation.

D Baselines Detailed

Our method is compared against several state-of-the-art and comprehensive baselines, covering the following benchmark settings:

ReACT (Yao et al., 2022), which alternates reasoning Thought and execution Action, forming a linear interaction process between language reasoning and tool invocation. It is one of the most widely used baselines for tool-augmented LLMs.

DFSDT (Qin et al., 2023), which adopts a depth-first search mechanism to explore reasoning–tool trajectories. Whenever the model reaches an erroneous path, DFSDT exposes the full failure history back to the model, enabling re-planning and maximizing exploration space.

LATS (Zhou et al., 2024), which leverages look-ahead tree search to expand multiple candidate tool sequences and evaluates their expected effectiveness, demonstrating strong planning ability in complex multi-step scenarios.

ToolChain* (Zhuang et al., 2024), which explicitly constructs a tool chain to model multi-step dependencies and guides LLMs to complete sequential tool execution. Although it enhances structured reasoning for multi-tool tasks, its effectiveness still primarily relies on LLM-based natural language reasoning rather than formal execution constraints.

Tool-Planner (Liu et al., 2025c), which incorporates explicit external planning modules to control tool sequence generation, combining retrieval, candidate filtering, and structured planning strategies to improve global execution coherence and decision reliability.

E Environments

All experiments are conducted under a unified tool execution environment. ToolBench APIs are treated as callable functional nodes, while MCP-Universe tools are executed in an official sandbox with real execution feedback, including realistic execution latency, tool failure signals, and state-dependent output variations. All models are accessed through their official APIs, with the decoding temperature fixed to 0.2 to minimize randomness in reasoning and tool planning behavior. For each benchmark, all systems share identical task instructions, tool descriptions, execution limits, and termination conditions.

For the retrieval module, we construct tool semantic representations using an embedding-based retrieval framework. Specifically, we adopt the Qwen3-embedding-0.6b (Zhang et al., 2025b) model to encode tool descriptions, functional semantics, argument specifications, and usage documentation into dense vectors. During reasoning, the intermediate tool requirement representation is encoded in the same embedding space, and the Top-K candidate tools are retrieved using cosine similarity. We set $K = 10$ by default unless otherwise specified. Following retrieval, a reranking model is applied to improve tool selection accuracy within the narrowed candidate set. We employ a lightweight LLM-based reranker built upon Qwen3-Reranker-0.6B (Zhang et al., 2025b), which jointly considers the current reasoning context, symbolic state, and candidate tool semantics to estimate contextual suitability.

F Detailed Rejection Analysis

F.1 Pre-condition $\{P\}$ Validation

The pre-condition check accounts for **17.6%** of all rejections, functioning as a "static firewall" that blocks invalid actions before they are executed.

- **Parametric Hallucination (8.4%):** This is

the most prevalent error type. When facing vast toolsets, LLMs often generate parameters based on intuition—such as hallucinating file IDs or directory paths—rather than grounded retrieval. By enforcing symbolic link validation, $\{P\}$ intercepts these requests before execution, significantly reducing computational overhead and token consumption.

- **State Dependency Violation (4.1%):** Models occasionally bypass necessary operational sequences, such as attempting to modify a file without first obtaining the required permissions or handles. $\{P\}$ enforces strict logical and temporal constraints, ensuring that every invocation is predicated on a valid environmental state.

F.2 Post-condition $\{Q\}$ Assertion

Although post-execution assertions trigger less frequently (**11.8%**), they address sophisticated logical failures that conventional search models, such as DFSDT, typically fail to detect.

- **Silent Failures (6.3%):** In complex tasks, APIs often return a successful status code (e.g., HTTP 200) despite providing an empty or vacuous response (e.g., `results: []`). Without $\{Q\}$ verification, an agent might interpret this as successful progress and continue down a futile search path. The $\{Q\}$ assertion mandates a *non-empty* result check, identifying these "logical voids" and triggering an immediate backtrack.

- **Semantic and State Alignment (5.5%):** This includes mismatches in semantic constraints (3.7%) and inconsistencies in state updates (1.8%). This confirms that $\{Q\}$ can capture subtle deviations between tool outputs and user intent, ensuring the search trajectory remains anchored to the correct semantic path.

F.3 Synergetic Effects and Logical Closure

Our analysis reveals that $\{P\}$ and $\{Q\}$ constitute a robust logical closed-loop. The high rejection rate of $\{P\}$ (**17.6%**) primarily improves **search efficiency** by pruning obvious error branches to save tokens and time. Conversely, the precision-driven interceptions of $\{Q\}$ (**11.8%**) are the primary determinants of **task success**. By identifying technically successful but logically flawed steps, $\{Q\}$ prevents the accumulation of cascading errors as a major bottleneck in unverified agentic systems.

F.4 Prompt Template

Tool Calling Procedure

System Prompt for LLM Reasoning

```
You are a helpful assistant that can use tools to answer user questions.

You have access to a set of tools and a symbolic state that tracks verified facts.

**Important Control Tokens:**
- When you need to use a tool, output: `<start_call_tool>`
- After tool results are provided, they will be wrapped in:
  `<start_tool_result>...</end_tool_result>`
- When you finish using tools, output: `<end_call_tool>`

**State Information:**
The current symbolic state contains verified facts. Use this information to:
1. Check if you have enough information to answer directly
2. Determine what information is missing and needs to be retrieved via tools
3. Understand what tools can be called based on the current state

**Tool Calling Process:**
1. Think about what information you need
2. Output `<start_call_tool>` followed by a brief description of what you need
3. Wait for tool results
4. Continue reasoning with the new information
5. Repeat if needed, or provide the final answer

**CRITICAL: When Tools Fail - Keep Trying!**
- If a tool call fails, DO NOT give up immediately. Consider:
  * Try a different tool that might provide similar information
  * Try the same tool with different parameters
  * Try alternative approaches or search strategies
  * Think about what other information sources might help
- Only provide a final answer when you are CONFIDENT you have:
  * Successfully retrieved the necessary information, OR
  * Exhausted all reasonable tool options and can provide a helpful answer
    based on available information
- Do NOT end reasoning prematurely just because one tool failed
- Be persistent and creative in finding alternative solutions

**Output Format:**
- If you can answer directly: Provide the answer without `<start_call_tool>`
- If you need tools: Output `<start_call_tool>` followed by your reasoning
  about what tool to use
- If tools fail: Think about alternatives and try again with
  `<start_call_tool>`
```

Tool Calling Procedure

Example User Message Format

```
**Current State:**
query: Find me a tutorial video about machine learning on YouTube
topic: machine learning
platform: YouTube
content_type: tutorial video

**Tool Results:**
(Empty on first call)

**User Query:** Find me a tutorial video about machine learning on YouTube

**Your Task:**
Think step by step. If you need to use tools, output `<start_call_tool>`
followed by a description of what you need.
If a tool fails, think about alternative approaches and try other tools
before giving up.
Only provide the final answer when you are CONFIDENT you have enough
information or have exhausted all reasonable options.
```

G Planning Process

Algorithm 1 formalizes our Forward Execution framework, which integrates Contract Verification into the LLM's reasoning loop. The procedure begins by initializing the environment state S_0 from the user query and context. At each step k, the LLM acts as a controller, deciding whether to conclude the task with an answer or invoke an external tool.

Algorithm 1 Forward Execution with Contract Verification

Require: Query q, Context H, Max iterations $K_{\max}$, Toolset $\mathcal{T}$
Ensure: Final answer a or failure signal

$S_0 \leftarrow \text{InitState}(q, H)$, $\mathcal{T}_{\text{failed}} \leftarrow \emptyset$
for $k = 0$ **to** $K_{\max} - 1$ **do**
  $\text{act}_k \leftarrow \text{LLM\_Reason}(q, H, \text{summary}(S_k))$ ▷ Generate reasoning step
  if act_k is $\text{Answer}(a)$ **then**
    return a ▷ Task successfully completed
  end if
  if act_k is $\text{CallTool}(desc)$ **then**
    $\mathcal{T}_{\text{cand}} \leftarrow \text{Rerank}(\text{Retrieve}(q, S_k))$ ▷ Top-N candidate tools
    is_updated $\leftarrow$ False
    for $t \in \mathcal{T}_{\text{cand}} \setminus \mathcal{T}_{\text{failed}}$ **do** ▷ Iterate through valid candidates
      if $S_k \not\models \phi_{pre}(t)$ **then** ▷ Check Precondition Contract
        continue
      end if
      $\theta_t \leftarrow \text{GenParams}(t, S_k)$ ▷ Synthesize arguments
      $r_t \leftarrow \text{Execute}(t, \theta_t)$ ▷ Tool invocation
      if $\text{Verify}(S_k, r_t, \phi_{post}(t))$ **then** ▷ Verify Postcondition Contract
        $S_{k+1} \leftarrow \text{Update}(S_k, r_t)$ ▷ Commit to state transition
        is_updated $\leftarrow$ True
        break
      else
        $\mathcal{T}_{\text{failed}} \leftarrow \mathcal{T}_{\text{failed}} \cup \{t\}$ ▷ Mark tool as unreliable
      end if
    end for
    if not is_updated **then**
      return Fail ▷ No viable tools satisfy contracts
    end if
  end if
end for
return Timeout

H Examples

Tool Calling Procedure

Tool Calling Example I

- **Query:** "Find me a tutorial video about machine learning on YouTube"
- **Step 0: Initial State**

```
S_0 = {
    "query": "Find me a tutorial video about machine learning on YouTube",
    "topic": "machine learning",
    "platform": "YouTube",
    "content_type": "tutorial video"
}
```

- **Step 1: LLM Reasoning**

```
Conversation History: []
LLM Response:
"I need to search for machine learning tutorial videos on YouTube.
Let me use a YouTube search tool to find relevant videos.

<start_call_tool>
I need to search for machine learning tutorial videos on YouTube
<end_call_tool>"

Action: start_call_tool
```

- **Step 1: Tool Retrieval**

```
Retrieved candidates (top-5 after reranking):
1. Simple_YouTube_Search_Search (score: 0.92)
2. YouTube_Video_Info (score: 0.85)
3. YouTube_Channel_Search (score: 0.78)
4. Video_Search_API (score: 0.72)
5. Content_Search (score: 0.68)
```

- **Step 1: Precondition Checking**

```
Checking Simple_YouTube_Search_Search:
  Precondition: exists(query)
  State S_0: {"query": "Find me a tutorial video...", ...}
  Check: exists(query) = true
```

- **Step 1: Tool Execution**

```
Tool: Simple_YouTube_Search_Search
Params: {"query": "machine learning tutorial"}
```

```
Tool Config: {
    "category": "Data",
    "api_name": "Search",
    "tool_input": {
        "query": "machine learning tutorial",
        "type": "",
        "safesearch": false
    }
}

Tool Result:
{
    "results": [
        {
            "title": "Machine Learning Tutorial for Beginners",
            "url": "https://youtube.com/watch?v=abc123",
            "channel": {"name": "ML Course", "id": "UC123"},
            "views": 1500000,
            "duration_formatted": "45:30"
        },
        {
            "title": "Complete ML Course - Full Tutorial",
            "url": "https://youtube.com/watch?v=def456",
            "channel": {"name": "AI Academy", "id": "UC456"},
            "views": 800000,
            "duration_formatted": "12:30:00"
        }
    ]
}
```

- **Step 1: Postcondition Verification**

```
Postcondition Q_t:
  structural: has_field("results") && is_list(r_t.results)
  semantic: forall video in r_t.results:
            has_field(video, "title") && has_field(video, "url")

Verification Result: success
  - Structural check: "results" field exists and is a list
  - Semantic check: All videos have "title" and "url" fields
```

- **Step 1: State Update**

```
S_1 = Update_t(S_0, r_t) = {
    "query": "Find me a tutorial video about machine learning on YouTube",
    "topic": "machine learning",
    "platform": "YouTube",
    "content_type": "tutorial video",
    "youtube_results": [
```

```
        {"title": "Machine Learning Tutorial for Beginners", ...},
        {"title": "Complete ML Course - Full Tutorial", ...}
    ],
    "result_count": 19
}
```

- **Step 2: LLM Reasoning (with tool result)**

```
Conversation History: [
    {"role": "user", "content": "Find me a tutorial video..."},
   {"role": "assistant", "content": "I need to search...\n<start_call_tool>...
    <end_call_tool>"},
   {"role": "user", "content": "<start_tool_result>\n{\"results\": [...]}\n
    <end_tool_result>"}
]

LLM Response:
"Great! I found several machine learning tutorial videos on YouTube.

Based on the YouTube search results, I found several machine learning
tutorial videos for you:

1. \"Machine Learning Tutorial for Beginners\" - 1.5M views, 45:30
   Channel: ML Course
   URL: https://youtube.com/watch?v=abc123

2. \"Complete ML Course - Full Tutorial\" - 800K views, 12:30:00
   Channel: AI Academy
   URL: https://youtube.com/watch?v=def456

These tutorials cover fundamental concepts and practical applications
of machine learning."

Action: answer
```

- **Final State**

```
S_final = {
    "query": "Find me a tutorial video about machine learning on YouTube",
    "topic": "machine learning",
    "platform": "YouTube",
    "content_type": "tutorial video",
    "youtube_results": [...],
    "result_count": 19
}
```

Tool Calling Procedure

Tool Calling Procedure with Postcondition Failure

- **Query:** "What's the weather like in New York today?"
- **Step 0: Initial State**

```
S_0 = {
    "query": "What's the weather like in New York today?",
    "location": "New York",
    "date": "today"
}
```

- **Step 1: LLM Reasoning**

```
Conversation History: []
LLM Response:
"I need to get the current weather information for New York.
Let me use a weather tool to fetch this data.

<start_call_tool>
Get weather information for New York
<end_call_tool>"

Action: start_call_tool
```

- **Step 1: Tool Retrieval**

```
Retrieved candidates (top-5 after reranking):
1. Weather_API_GetCurrentWeather (score: 0.91)
2. OpenWeatherMap_CurrentWeather (score: 0.88)
3. WeatherService_GetForecast (score: 0.82)
4. ClimateData_GetWeather (score: 0.75)
5. WeatherInfo_Query (score: 0.71)
```

- **Step 1: Precondition Checking (Tool 1)**

```
Checking Weather_API_GetCurrentWeather:
  Precondition: exists(location)
  State S_0: {"location": "New York", ...}
  Check: exists(location) = true
```

- **Step 1: Tool Execution (Tool 1)**

```
Tool: Weather_API_GetCurrentWeather
Params: {"location": "New York"}
Tool Config: {
```

```
    "category": "Weather",
    "api_name": "GetCurrentWeather",
    "tool_input": {"location": "New York"}
}

Tool Result:
{
    "status": "success",
    "data": {
        "city": "New York",
        "temperature": 72,
        "condition": "sunny"
    }
}
```

- **Step 1: Postcondition Verification (Tool 1)**

```
Postcondition Q_t:
  structural: has_field("temperature") &&
              has_field("condition") &&
              has_field("humidity")
  semantic: temperature is numeric &&
            condition is string

Verification Result: failure
  - Structural check: Missing required field "humidity"
  - Semantic check: temperature and condition are valid

Error: Tool result does not contain required field "humidity"
```

- **Step 1: Tool Marked as Failed**

```
Failed tools: {Weather_API_GetCurrentWeather}
State remains: S_0 (no update)
```

- **Step 1: Try Next Tool - Precondition Checking (Tool 2)**

```
Checking OpenWeatherMap_CurrentWeather:
  Precondition: exists(location)
  State S_0: {"location": "New York", ...}
  Check: exists(location) = true
```

- **Step 1: Tool Execution (Tool 2)**

```
Tool: OpenWeatherMap_CurrentWeather
Params: {"location": "New York"}
Tool Config: {
```

```
    "category": "Weather",
    "api_name": "CurrentWeather",
    "tool_input": {"location": "New York"}
}

Tool Result:
{
    "temperature": 72,
    "condition": "sunny",
    "humidity": 65,
    "wind_speed": 10,
    "pressure": 1013,
    "location": "New York, NY"
}
```

- **Step 1: Postcondition Verification (Tool 2)**

```
Postcondition Q_t:
  structural: has_field("temperature") &&
              has_field("condition") &&
              has_field("humidity")
  semantic: temperature is numeric &&
            condition is string &&
            humidity is numeric

Verification Result: success
  - Structural check: All required fields present
  - Semantic check: All fields have correct types
```

- **Step 1: State Update**

```
S_1 = Update_t(S_0, r_t) = {
    "query": "What's the weather like in New York today?",
    "location": "New York",
    "date": "today",
    "temperature": 72,
    "condition": "sunny",
    "humidity": 65,
    "wind_speed": 10,
    "pressure": 1013
}
```

- **Step 2: LLM Reasoning (with tool result)**

```
Conversation History: [
   {"role": "user", "content": "What's the weather like in New York today?"},
    {"role": "assistant", "content": "I need to get the current weather...\n
     <start_call_tool>...<end_call_tool>"},
```

```
    {"role": "user", "content": "<start_tool_result>\n{\"temperature\": 72,
     \"condition\": \"sunny\", ...}\n<end_tool_result>"}
]

LLM Response:
"The weather in New York today is sunny with a temperature of 72°F.
The humidity is 65%, wind speed is 10 mph, and atmospheric pressure
is 1013 hPa. It's a pleasant day for outdoor activities."

Action: answer
```

- **Final State**

```
S_final = {
    "query": "What's the weather like in New York today?",
    "location": "New York",
    "date": "today",
    "temperature": 72,
    "condition": "sunny",
    "humidity": 65,
    "wind_speed": 10,
    "pressure": 1013
}
```

Tool Calling Procedure

Contract Derivation Example I: Google Search API **Source schema.** The response schema for the Google Search API is defined in `data/toolenv/response_examples/Search/google_search_api.json`:

```
{
    "api_list": [{
        "name": "Search",
        "schema": {
            "meta": { "status": "int", "query": "str", "googleURL": "str" },
            "totalResults": "int",
            "organic_results": [{
                "position": "int", "title": "str",
                "description": "str", "link": "str"
            }]
        }
    }]
}
```

Derived P. From the `required_parameters` in the tool's interface definition (e.g., `query` marked as required), we obtain:

$$P : \text{exists}(\text{query}) \wedge \text{is\_string}(\text{query})$$

Derived Q. From the response schema, the declared fields and types are mapped to structural and type constraints:

$$Q : \text{has\_field}(r, \texttt{meta}) \wedge \text{has\_field}(r, \texttt{organic\_results}) \wedge \text{is\_list}(r.\texttt{organic\_results})$$

Note that the schema does *not* specify that `organic_results` must be non-empty; therefore, Q only enforces structural and type constraints, and a response with `organic_results: []` is considered compliant.

Compliant response. Input "OpenAI GPT-5.2" returns a response containing `meta` (with required subfields) and `organic_results` (an array whose elements have the schema-defined fields). Extra fields such as `related_questions` or `knowledge_graph` are optional and do not affect the compliance decision.

Non-compliant response. Input "xenovariable fractonexus" returns only `{"meta": {"status": 200, ...}}`, lacking the `organic_results` field entirely. This response does not satisfy the structural requirement $\text{has\_field}(r, \texttt{organic\_results})$ and is therefore rejected by Q.

Tool Calling Procedure

Contract Derivation Example II: Autocomplete Zipcodes API **Source schema.** From `data/toolenv/response_examples/Data/autocomplete_usa.json`:

```
{
    "name": "Autocomplete Zipcodes Lite",
    "schema": {
        "StatusCode": "int",
        "Result": [ "list of str with length 1" ],
        "IsError": "bool"
    }
}
```

Derived Q. The schema explicitly declares `Result` as "list of str with length 1", so the postcondition includes a non-empty/length constraint:

$$Q : \text{has\_field}(r, \texttt{Result}) \wedge \text{is\_list}(r.\texttt{Result}) \wedge \text{length}(r.\texttt{Result}) \geq 1$$

Violation example. Input text = 43239: the API returns `{"StatusCode": 200, "Result": [], "IsError": false}`. Here `Result` is an empty list, which does not satisfy the schema's "length 1" requirement; ToolGate rejects this result and does not commit it to the trusted state.
Compliant example. Input text = 43230: the API returns `{"StatusCode": 200, "Result": ["43230, Columbus, OH"], "IsError": false}`. Here `Result` has one string element, satisfying Q; the result is accepted and committed.

I Formal Derivations with Hoare Logic and First-Order Contracts

In this appendix, we present several representative derivations that make the logical foundations of ToolGate explicit. We formalize single-step tool execution, trajectory-level safety, and invariants as Hoare-style proof obligations and first-order logic (FOL) formulas over the symbolic state space Σ and execution trajectories.

I.1 Notation and Basic Setting

We recall that the trusted symbolic state is a typed key–value mapping $S \in \Sigma$, where

$$\Sigma \;=\; \{(k, v, \sigma)\}.$$

We write $S \models \varphi$ to denote that a (first-order) state formula φ is true in S. A tool t is associated with a Hoare-style contract

$$\{P_t\}\ t\ \{Q_t\},$$

where $P_t(S)$ is a state predicate (precondition) and $Q_t(S, r_t)$ is a postcondition predicate over the pre-state S and runtime result r_t:

$$Q_t : \Sigma \times R_t \to \{\text{true}, \text{false}\}.$$

We write $(S, r_t) \models Q_t$ as shorthand for $Q_t(S, r_t) = 1$.

To decouple logical validation from state construction, we introduce a deterministic state update operator

$$\mathsf{Update}_t : \Sigma \times R_t \to \Sigma,$$

which specifies the new trusted symbolic state produced when a valid result r_t is integrated into S.

We say that the (ideal) runtime executor of tool t is a (possibly partial) function

$$\mathsf{Exec}(t, S) = r_t$$

that returns a runtime result r_t when t is invoked under state S.

I.2 Single-Step Hoare-Style Derivation

We first spell out the standard Hoare-style proof obligation for a single tool invocation in our setting.

Single-step soundness obligation. A contract $\{P_t\}t\{Q_t\}$ is sound w.r.t. Exec and Update_t if the following FOL formula holds:

$$\forall S, r_t.\ \Big(S \models P_t\ \wedge\ r_t = \mathsf{Exec}(t, S)\ \wedge\ Q_t(S, r_t)\Big)\ \Rightarrow\ \mathsf{GoodState}\big(\mathsf{Update}_t(S, r_t)\big), \tag{15}$$

where GoodState expresses that the updated state is well-typed and consistent (e.g., satisfies global invariants such as key uniqueness and type soundness).

Inference rule for a single ToolGate step. We can capture the operational step of ToolGate for a single tool call as the following Hoare-style derivation rule:

$$\frac{S \models P_t \quad r_t = \mathsf{Exec}(t, S) \quad Q_t(S, r_t) \quad \mathsf{Inv}(S) \Rightarrow \mathsf{Inv}\big(\mathsf{Update}_t(S, r_t)\big)}{\{\, P_t(S) \wedge \mathsf{Inv}(S) \,\}\ t\ \{\, \mathsf{Inv}(S') \wedge Q_t(S, r_t) \wedge S' = \mathsf{Update}_t(S, r_t) \,\}}\ \textsc{Tool-Step}$$

where Inv is any chosen state invariant (e.g., that S only contains verified tool results).

In small-step transition form, a single ToolGate step can be written as

$$\langle S_k, R_k \rangle \ \xrightarrow{t, r_t}\ \langle S_{k+1}, R_{k+1} \rangle$$

with the following proof tree:

$$\frac{S_k \models P_t \quad r_t = \mathsf{Exec}(t, S_k) \quad Q_t(S_k, r_t) \quad S_{k+1} = \mathsf{Update}_t(S_k, r_t) \quad R_{k+1} = R_k \cdot \langle t, r_t \rangle}{\langle S_k, R_k \rangle \xrightarrow{t, r_t} \langle S_{k+1}, R_{k+1} \rangle}\ \textsc{Tool-Exec}$$

I.3 Precondition Filtering as Weakest Precondition

Tool selection in ToolGate is constrained by the precondition P_t. We can express this in terms of weakest preconditions. Let $\mathsf{wp}(t, \Phi)$ be the weakest precondition of tool t w.r.t. a desired post-state formula $\Phi(S')$. Then:

$$\mathsf{wp}(t, \Phi)(S) \;\triangleq\; \exists r_t.\, \Big(S \models P_t \wedge r_t = \mathsf{Exec}(t, S) \wedge Q_t(S, r_t) \wedge \Phi\big(\mathsf{Update}_t(S, r_t)\big)\Big).$$

In particular, requiring that t is *executable* in S corresponds to

$$S \models \mathsf{wp}(t, \top) \iff \exists r_t.\, S \models P_t \wedge r_t = \mathsf{Exec}(t, S) \wedge Q_t(S, r_t).$$

The ToolGate precondition filter can then be expressed as:

$$\forall t \in C_k.\, \mathsf{Admissible}(t, S_k) \;\triangleq\; S_k \models \mathsf{wp}(t, \top). \tag{16}$$

I.4 Postcondition as Acceptance Event

The runtime acceptance predicate A_t in ToolGate is defined by:

$$A_t(S_k, r_t) \;=\; \big(Q_t(S_k, r_t) \wedge \mathsf{wf}(r_t)\big),$$

where wf encodes structural and formatting well-formedness for r_t.

We define the state update rule as

$$S_{k+1} = \begin{cases} \mathsf{Update}_t(S_k, r_t) & \text{if } A_t(S_k, r_t) = 1, \\ S_k & \text{otherwise.} \end{cases}$$

This rule can be captured by the following Hoare triple:

$$\{\, S_k \models P_t \,\}\; t \;\{\, A_t(S_k, r_t) = 1 \Rightarrow \big(S_{k+1} = \mathsf{Update}_t(S_k, r_t) \wedge Q_t(S_k, r_t)\big) \,\}. \tag{17}$$

Equivalently, in FOL:

$$\begin{aligned} \forall S_k, r_t, S_{k+1}.\; & S_k \models P_t \;\wedge\; r_t = \mathsf{Exec}(t, S_k) \;\wedge\; A_t(S_k, r_t) = 1 \\ & \Rightarrow\; \big(S_{k+1} = \mathsf{Update}_t(S_k, r_t) \wedge Q_t(S_k, r_t) \wedge \mathsf{GoodState}(S_{k+1})\big). \end{aligned} \tag{18}$$

I.5 Trajectory-Level Safety Derivation

A full ToolGate execution induces a trajectory

$$\tau = \big((S_0, R_0), (t_0, r_0, A_0), \ldots, (S_n, R_n)\big).$$

Per-step safety. We say that step k is safe iff:

$$\begin{aligned} \mathsf{SafeStep}_k(\tau) \triangleq \Big(S_k \models P_{t_k} \;\wedge\; r_k = \mathsf{Exec}(t_k, S_k) \;\wedge\; A_{t_k}(S_k, r_k) = 1 \\ \Rightarrow \big(Q_{t_k}(S_k, r_k) \wedge S_{k+1} = \mathsf{Update}_{t_k}(S_k, r_k)\big)\Big). \end{aligned} \tag{19}$$

Global safety. Trajectory-level safety is then:

$$\mathsf{Safe}(\tau) \;\triangleq\; \bigwedge_{k=0}^{n-1} \mathsf{SafeStep}_k(\tau). \tag{20}$$

Soundness theorem (sketch). If all tool contracts are sound (Eq. 15) and the initial state S_0 satisfies the global invariant Inv, then every reachable ToolGate trajectory is safe:

$$\forall \tau.\ \mathsf{Reach}(q, H, \tau) \wedge S_0 \models \mathsf{Inv} \ \Rightarrow\ \mathsf{Safe}(\tau) \wedge \bigwedge_{k=0}^{n} \mathsf{Inv}(S_k). \tag{21}$$

This can be proved by induction on k using the TOOL-STEP rule:

$$\frac{\mathsf{Inv}(S_0) \quad \forall k.\ \mathsf{SafeStep}_k(\tau) \wedge \mathsf{Inv}(S_k) \Rightarrow \mathsf{SafeStep}_{k+1}(\tau) \wedge \mathsf{Inv}(S_{k+1})}{\forall k.\ \mathsf{Reach}_k(q, H, \tau) \Rightarrow \mathsf{SafeStep}_k(\tau) \wedge \mathsf{Inv}(S_k)}\ \textsc{Induction}$$

I.6 Contract Instantiation for a Concrete Tool

To illustrate, consider a (simplified) repository management tool ListFiles with contract:

$$\{P_{\mathsf{list}}\}\ \mathsf{ListFiles}\ \{Q_{\mathsf{list}}\}.$$

Let the symbolic state contain a key "cwd" for the current working directory and a key "fs" for a symbolic file-system abstraction. We instantiate:

$$P_{\mathsf{list}}(S) \triangleq \exists d.\ \big(d = S[\mathsf{cwd}] \wedge d \in \mathsf{Dom}(S[\mathtt{fs}])\big), \tag{22}$$

$$Q_{\mathsf{list}}(S, r) \triangleq \exists d, L.\ d = S[\mathsf{cwd}] \wedge L = \mathsf{LookupDir}(S[\mathtt{fs}], d) \wedge r = L, \tag{23}$$

and define the corresponding state update operator as

$$\mathsf{Update}_{\mathsf{list}}(S, r)\ =\ S \cup \{(\mathtt{last\_ls}, r, \mathsf{ListType})\}.$$

The corresponding Hoare triple for this tool is:

$$\{\, P_{\mathsf{list}}(S) \,\}\ \mathsf{ListFiles}\ \{\, Q_{\mathsf{list}}(S, r) \wedge S' = \mathsf{Update}_{\mathsf{list}}(S, r) \wedge \mathsf{Inv}(S') \,\}.$$

FOL derivation of a safe call. Assume we are at step k with state S_k such that

$$S_k \models P_{\mathsf{list}}.$$

The concrete call is:

$$r_k = \mathsf{Exec}(\mathsf{ListFiles}, S_k).$$

Postcondition checking and acceptance give:

$$\begin{aligned}(S_k, r_k) \models Q_{\mathsf{list}} \ \Rightarrow\ \exists d, L.\ &\big(d = S_k[\mathsf{cwd}]\big) \\ &\wedge \big(L = \mathsf{LookupDir}(S_k[\mathtt{fs}], d)\big) \\ &\wedge \big(r_k = L\big), \\ A_{\mathsf{list}}(S_k, r_k) = 1 \ \Rightarrow S_{k+1} = \mathsf{Update}_{\mathsf{list}}(S_k, r_k) &= S_k \cup \{(\mathtt{last\_ls}, r_k, \mathsf{ListType})\},\end{aligned} \tag{24}$$

which together imply that S_{k+1} is a well-formed extension of S_k.

Combining these, the TOOL-EXEC rule instantiates to:

$$\frac{\begin{array}{l} S_k \models P_{\mathsf{list}} \quad r_k = \mathsf{Exec}(\mathsf{ListFiles}, S_k) \\ (S_k, r_k) \models Q_{\mathsf{list}} \quad S_{k+1} = \mathsf{Update}_{\mathsf{list}}(S_k, r_k) \quad R_{k+1} = R_k \cdot \langle \mathsf{ListFiles}, r_k \rangle \end{array}}{\langle S_k, R_k \rangle \xrightarrow{\mathsf{ListFiles}, r_k} \langle S_{k+1}, R_{k+1} \rangle}\ \textsc{Tool-Exec-List}$$

I.7 Contract-Governed Tool Selection Policy

Finally, we combine the probabilistic ranking distribution with logical filtering. Let $\mathsf{rank}(t \mid u_k)$ be the (normalized) ranking score over candidate tools given requirement representation u_k.

We define the *contract-governed policy*:

$$\pi(t \mid q, H, S_k, R_k) \triangleq \frac{\mathsf{rank}(t \mid u_k) \cdot \mathbf{1}[S_k \models P_t]}{\sum_{t' \in C_k} \mathsf{rank}(t' \mid u_k) \cdot \mathbf{1}[S_k \models P_{t'}]}.$$

A trajectory τ is then sampled according to:

$$p(\tau \mid q, H) = \prod_{k=0}^{n-1} \Big(p(\langle S_k, R_k \rangle) \cdot p(\texttt{<start\_call\_tool>} \mid q, H, S_k, R_k) \cdot \pi(t_k \mid q, H, S_k, R_k) \cdot p(r_k = \mathsf{Exec}(t_k, S_k)) \cdot p(A_{t_k}(S_k, r_k) = 1 \mid S_k, r_k) \Big), \tag{25}$$

subject to the global constraint that any violation of P_t or Q_t yields zero probability:

$$\exists k.\, \neg \mathsf{SafeStep}_k(\tau) \Rightarrow p(\tau \mid q, H) = 0.$$

This explicit factorization makes the interaction between probabilistic reasoning and logical contracts formally visible and verifiable.