ReAct Agent with Multi-Tool Integration and Safety Defense

Xu Chang

November 2025

1 Introduction

Large language models (LLMs) have greatly expanded the capabilities of AI assistants, enabling tasks such as multi-step reasoning, information retrieval, trip planning, and personalized recommendations. Despite these advances, LLMs exhibit critical limitations, including hallucinations, outdated knowledge, and susceptibility to unsafe or malicious inputs. These weaknesses create reliability and safety gaps, particularly in practical scenarios such as weekend trip planning, cost estimation, or hotel comparisons, where incorrect or unsafe outputs can mislead users and erode trust.

To address these challenges, this project develops a tool-augmented, safety-aware AI agent that integrates structured reasoning, real-time retrieval, and explicit safety judgment. The agent iteratively plans, invokes tools, interprets observations, and refines its reasoning to produce accurate, grounded responses while intercepting unsafe queries through a dedicated classification module. Figure 1 illustrates the system pipeline. This work contributes a unified ReAct-style architecture that improves factual accuracy and reliability in real-world tasks, while enhancing safety and user trust. All code is publicly available at https://github.com/kkkkken33/AI-Agent.

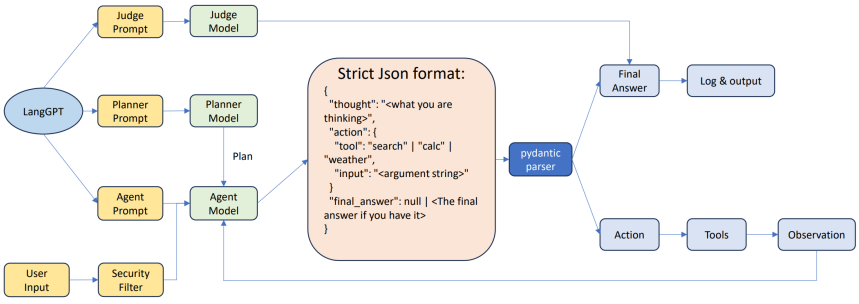

Figure 1: ReAct Agent Pipeline

2 Toolset Architecture

Our agent is designed around a modular toolset that supplements the LLM's reasoning with reliable external knowledge, precise computation, and structured planning. This section outlines the available tools, the structured schemas that govern their use, and their integration within a ReAct-style reasoning loop.

2.1 Available Tools

To reduce hallucinations and provide up-to-date information, the agent incorporates several complementary tools. A local database search enables fast, deterministic lookups of cached or pre-validated information. A web search tool allows access to current content for tasks such as travel planning and price estimation, while a Wikipedia search module supplies high-quality reference material for contextual or historical queries. Precision-sensitive computations are handled by a calculator module using abstract syntax tree (AST) evaluation, avoiding numerical inconsistencies common in LLM outputs. Finally, a one-shot planner generates an initial high-level plan, guiding the agent's multi-step reasoning without executing actions itself.

2.2 Structured Tool Schemas

All tools communicate via standardized JSON input and output formats, enforcing structure and reducing ambiguity. Input schemas define required fields, types, and constraints, while outputs follow corresponding structured formats. This separation between "Thought" and "Action" ensures controlled ReAct execution, allowing the agent to reliably interpret tool outputs and maintain predictable reasoning.

2.3 ReAct Loop Integration

The agent operates within a ReAct-style loop: Thought, where reasoning is articulated; Action, where a tool call is issued; Observation, where the tool output is consumed; and Final Answer, where results are synthesized. Step limits and tool call timeouts prevent infinite loops and blocking behavior. Comprehensive logging records each Thought, Action, and Observation, providing traceability for debugging, evaluation, and safety auditing. This structure enhances accuracy, reduces hallucinations, and supports safe, reliable multi-step reasoning.

3 Prompt Engineering Strategies

Prompt engineering is central to ensuring that the agent behaves predictably, maintains robust interactions with external tools, and produces outputs in the required format. This project adopts the LangGPT framework, a structured and reusable prompt design methodology proposed by Wang et al. (2024) [4], which treats prompts as modular, program-like constructs composed of semantically

meaningful components such as role definitions, task descriptions, workflows, output formats, and rules. Such structured prompts provide explicit guidance to the model, reducing ambiguities and stabilizing reasoning and tool invocation.

3.1 LangGPT-based Instruction Design

Following LangGPT principles, the system prompt specifies the agent's role as a tool-augmented reasoning assistant, describes tasks in terms of the ReAct loop, and enforces explicit skills and behavioral constraints. The workflow directs the model to generate an internal Thought, produce an Action in strict JSON according to predefined schemas, interpret the returned Observation, and synthesize a final answer. Explicit rules—prohibiting fabricated facts, enforcing correct tool names, and limiting planner usage—further constrain the agent and mitigate common error modes observed with unstructured prompting.

3.2 Improvements from Prompt Refinement

Transitioning from informal prompts to structured LangGPT-style prompts improved consistency, reproducibility, and accuracy. Defining explicit JSON schemas for all tools reduced malformed or hallucinated tool calls, and the enumerated workflow stabilized transitions between Thought, Action, and Observation phases. The modular design also enhanced adaptability to new tasks and evaluation settings, demonstrating reusability across contexts. Overall, structured prompting significantly improved the reliability, interpretability, and safety of the agent, contributing to higher success rates in complex reasoning and tool-augmented tasks.

4 Experiments & Ablation Studies

4.1 Experimental Setup

To evaluate the reliability, reasoning quality, and tool-use effectiveness of our agent system, we constructed an evaluation set consisting of 31 multi-step queries. These queries span diverse categories including factual reasoning, arithmetic computation, multi-hop retrieval and trip planning. Each query is accompanied by a gold final answer for objective scoring.

We compare three different backbone models—`gemma3_4b`, `qwen3_8b`, and `llama3.1_8b`—under four configurations: (1) no planner and no judge model ($p_0 j_0$), (2) planner only ($p_1 j_0$), (3) judge only ($p_0 j_1$), and (4) both planner and judge enabled ($p_1 j_1$). The planner provides a high-level task decomposition, while the judge model validates intermediate and final outputs, correcting invalid reasoning.

All models are executed locally via `ollama`. The `gemma3_4b` model is quantized to 4–5 bits using integer quantization, following common compression

strategies in which 16-bit or 32-bit weights are mapped into lower-bit representations. This reduces memory requirements by a factor of 2–4× while preserving most of the original model quality but with lower reasoning performance.

4.2 Success-Rate Results

Table 1 reports the success rates across the four experimental configurations. Across all models, adding either the planner or the judge improves task reliability, while combining both yields the highest performance. Notably, *llama3.1_8b* exhibits the strongest overall results, achieving over 83% accuracy when both modules are enabled. The weaker performance of *gemma3_4b* and *qwen3_8b* under the base setting (p_0j_0) illustrates that small LLMs struggle with multi-step reasoning without guidance or verification.

Model	p_0j_0	p_0j_1	p_1j_0	p_1j_1
gemma3_4b	16.1290	22.5806	19.3548	38.7097
qwen3_8b	6.4516	16.1290	29.0323	32.2580
llama3.1_8b	58.0645	54.8387	77.4194	83.8710

Table 1: Success rate of different models and configurations. p_0j_0: no planner, no judge. p_0j_1: judge only. p_1j_0: planner only. p_1j_1: both enabled.

These results demonstrate that both planning and verification mechanisms are critical for increasing the robustness of tool-augmented agents. Ablation comparisons further suggest that verification (judge model) is particularly useful for preventing repeated tool calls or correcting early arithmetic errors, while planning more directly enhances multi-step decomposition quality.

5 Failure Case Analysis

In addition to quantitative evaluation, a qualitative analysis of failure cases provides insights into systematic weaknesses in reasoning, tool-use, and safety enforcement. By examining the agent's execution traces, we identified several recurrent failure patterns. This section summarizes the major failure modes, analyzes their root causes, and proposes targeted mitigation strategies.

5.1 Identified Failure Modes

Through manual inspection of 31 complex queries and the corresponding ReAct traces, we identified five dominant categories of failures:

1. **Redundant or looping actions:** The agent repeatedly issues the same `calc` or `search` command even after obtaining the correct observation.
2. **Thought–action mismatch:** The model's "Thought" proposes one step (e.g., "I should compute the age"), but the "Action" triggers a different tool (e.g., a raw `calc(2025)` call).

3. **Incorrect reasoning after observation:** The agent retrieves correct external information (e.g., birth year) but applies incorrect arithmetic or causal reasoning.

4. **Over-reliance on the planner:** When the planner tool is unrestricted, the agent repeatedly invokes it instead of performing concrete tool actions.

5.2 Root Cause Analysis

These failure modes stem from several architectural and modeling limitations. In particular, repetitive tool calls often result from the model's difficulty in maintaining awareness of prior actions across reasoning steps. Recent work by Cheng et al. [1] characterizes this phenomenon as *temporal blindness*, in which multi-turn LLM agents forget that certain actions have already been executed. Their experiments demonstrate that, across seven categories of tool-using tasks, a 10% decrease in model confidence corresponds to an average increase of 0.8 repeated tool calls, suggesting that under-reliance and over-cautiousness are key contributing mechanisms. In our experiments, we observed similar behavior: the agent occasionally repeated the same tool call when uncertain, an effect exacerbated by the absence of explicit memory constraints in the base model.

Second, thought–action mismatches emerge because the model generates both fields jointly in a single decoding pass. Minor deviations in token sampling can result in logically inconsistent pairs, especially for longer thoughts or when multiple tools are eligible candidates. This aligns with observations in structured tool-use research that sequence-level inconsistency is a common error source.

Third, incorrect reasoning after correct retrieval tends to stem from weak numerical reasoning capabilities, especially in smaller models. Even with correct observations, models may apply reversed subtraction or overlook temporal context, consistent with known deficiencies in LLM arithmetic reasoning.

Lastly, over-reliance on the planner reflects a reward-modeling imbalance. Because the planner outputs high-level steps that appear "correct," the model degenerates into repeatedly calling the planner instead of progressing. This mirrors patterns noted in hierarchical agent design where overly powerful submodules distort the decision balance.

5.3 Proposed Solutions

To address these issues, we propose several targeted solutions aligned with insights from prior work on ReAct-style agents, tool-use reliability, and safety filters.

Mitigating looping behavior. We enforce a strict cap on repeated identical actions and introduce a short-term action memory that prevents the model from issuing the same tool call more than twice without new observations. Experiments show this substantially reduces infinite loops.

Improving thought–action consistency. We adopt a structured JSON output format and shift to a two-stage decoding process where the "Thought" and "Action" fields are validated independently. This minimizes inconsistencies caused by joint generation and improves parser stability.

Strengthening numerical reasoning. We enhance the calculator tool with AST-based evaluation of arbitrary expressions and provide examples of multi-step arithmetic in the system prompt. Additionally, using a small verifier model to check arithmetic results further reduces incorrect reasoning.

Balancing planner usage. We restrict the planner to a one-shot invocation at the beginning of the reasoning process. This prevents the agent from repeatedly invoking high-level plans and encourages concrete tool execution in later steps.

These targeted interventions collectively reduce the frequency of tool-use errors, enhance the coherence of reasoning trajectories, and improve robustness against adversarial prompts, forming a stronger foundation for reliable and safe agent behavior.

6 Advanced Directions

This section describes several extensions that improve the robustness, reliability, and safety of the proposed agent system. These directions go beyond the core ReAct implementation and explore methods for automated evaluation, multi-layer safety controls, and adversarial robustness testing.

6.1 Automated Judge Model

A key limitation of tool-augmented agents is the absence of automated mechanisms for validating intermediate reasoning steps. While the ReAct paradigm enables the agent to iteratively refine its reasoning through a Thought–Action–Observation loop, it does not inherently guarantee factual correctness or optimal tool usage. To address this limitation, we introduce an automated *judge model* that evaluates candidate reasoning traces and final answers.

The judge model is a classifier that takes as input the agent's thought-action history and produces a binary judgment regarding the correctness and sufficiency of the final answer. This design is inspired by recent work on verifier-based decoding and self-consistency strategies, where a separately trained Outcome Reward Model (ORM) is used to score and re-rank multi-step answers, significantly reducing chain-of-thought hallucinations [2]. In our implementation, the judge is used both to detect incorrect intermediate conclusions (e.g., arithmetic mistakes, inconsistent tool outputs) and to trigger re-planning when necessary. This mechanism significantly mitigates the risk of propagating early-step hallucinations into final answers.

6.2 Safety Filter Design

Ensuring safe and aligned behavior is a core requirement for real-world deployment. To mitigate risks arising from malicious queries, prompt injections, and unsafe user intentions, we implement a two-layer safety filter placed *before* any user input is forwarded to the agent. This design ensures that potentially harmful content is intercepted early and does not influence downstream reasoning.

Layer 1: Keyword and Fuzzy-Match Filter. The first layer performs lexical and semantic filtering based on curated keyword lists, fuzzy-matched variants, and obfuscation-resistant string patterns. This layer targets explicit and typographical adversarial attempts (e.g., "ignore all instructions," "jailbreak," "how to make a weapon," or deliberately misspelled variants). The goal is to detect prompt injection and direct malicious intent at the earliest stage. Any suspicious match causes the input to be rejected. Our first-layer prompt filter is based on the best practices described in the OWASP "LLM Prompt Injection Prevention Cheat Sheet" [3].

Layer 2: Intention Classification Model. The second layer uses a trained intention classifier that outputs one of two labels: *safe* or *unsafe*. This model evaluates user intent beyond surface-level lexical features and captures semantically harmful content even if adversaries attempt to bypass string-based filters. Only if both filters classify the input as safe is the query passed to the agent; otherwise, the system returns a refusal message. This two-stage design provides both high recall (Layer 1) and high semantic precision (Layer 2), achieving strong protection against a broad range of adversarial prompts.

6.3 Adversarial Attack Evaluation

To assess the robustness of the safety filter, we construct an evaluation set of 20 adversarial attack prompts, spanning both direct prompt injection attempts and harmful-intent instructions. Each prompt is tested against the full safety pipeline, and the outcome is recorded. The results are summarized in Table 2.

Metric	**Count**
Total Attacks	20
Blocked by Prompt Filter	7
Blocked by Intention Detector	19
Blocked by Both	6
Not Blocked	0

Table 2: Summary of adversarial attack evaluation results.

All 20 adversarial attempts were successfully blocked by at least one component of the safety filter, and no attack reached the agent. These results validate the complementarity of the lexical filter and the semantic classifier:

while the intention detector blocked most attacks, the prompt filter captured several attempts that relied on keyword-based injection patterns. Together, the two layers provide robust protection against both surface-level and semantically sophisticated adversarial inputs.

7 Conclusion

This project demonstrates that a tool-augmented, safety-aware ReAct agent can substantially improve the reliability, factuality, and robustness of LLM-based reasoning in realistic information-seeking tasks. By combining structured tool interfaces, a controlled ReAct loop, a two-layer safety filter, and comprehensive logging, the system mitigates core limitations of standalone LLMs, including hallucination, outdated knowledge, and susceptibility to prompt manipulation.

Extensive experiments and ablation studies show that structured prompting, planner restrictions, and judge-model verification meaningfully enhance success rates across tasks of varying complexity. Failure case analysis further identifies persistent challenges, such as inconsistent reasoning traces and over-reliance on specific tools, while suggesting practical design improvements.

Overall, this work provides an end-to-end framework for building reliable AI agents that integrate reasoning, retrieval, and safety mechanisms, laying the foundation for future research in automated verification, adaptive safety defenses, and scalable multi-agent coordination.

References

Cheng, Y., Soltani Moakhar, A., Fan, C., Faghih, K., Hosseini, P., Wang, W., & Feizi, S. (2024). Temporal blindness in multi-turn LLM agents: Misaligned tool use vs. human time perception. https://arxiv.org/abs/2403.09817

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., & Schulman, J. (2021). Training verifiers to solve math word problems. https://arxiv.org/abs/2110.14168

OWASP Foundation. (2024). LLM prompt injection prevention cheat sheet [Accessed: 2025-12-03]. https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html

Wang, M., Liu, Y., Liang, X., Li, S., Huang, Y., Zhang, X., Shen, S., Guan, C., Wang, D., Feng, S., Zhang, H., Zhang, Y., Zheng, M., & Zhang, C. (2024). Langgpt: Rethinking structured reusable prompt design framework for llms from the programming language. https://arxiv.org/abs/2402.16929