Tool-Using ReAct Agent Implementation Report

Xueyin Luo, Yanzhe Xie, Ye Guo
Hong Kong University of Technology and Science(Guangzhou)
AIAA3102 (L02) - Python Programming for Artificial Intelligence

Abstract

This work implements a tool-using language model agent based on the ReAct (Reasoning + Acting) pattern. The agent integrates external tools, including web search and a safe calculator, and is built on top of LangGraph as a deterministic state machine. Beyond the core ReAct loop, the system incorporates hallucination control, self-consistency decoding, and guardrails such as tool limits and safety filters. A custom evaluation set and an LLM-as-a-judge framework are used to assess performance, highlighting both the benefits and operational trade-offs of agentic architectures.

1 Introduction

1.1 The Agentic Paradigm Shift and Project Objectives

Large language models (LLMs) excel at pattern-based text generation but are constrained by static knowledge, limited grounding, and unreliable arithmetic. Tool-using agents aim to address these limitations by integrating external tools into the reasoning loop, allowing dynamic interaction with the environment rather than purely static recall.

This work centers on implementing a tool-using agent based on the ReAct (Reasoning + Acting) pattern. ReAct organizes the interaction as an iterative cycle of reasoning, tool invocation, and observation, so that the system can refine its plan dynamically and mitigate the weaknesses of static LLMs.

The primary objectives of this project are:

- Implement a stable ReAct loop: Thought $\rightarrow$ Action $\rightarrow$ Observation.
- Develop structured and extensible interfaces for external tools, focusing on a web search engine and a controlled calculator.
- Ensure robustness through safety mechanisms such as step limits, timeouts, and comprehensive error handling.
- Build a diverse evaluation dataset and protocol to assess the agent's performance and compare it with baseline LLM deficiencies.

2 ReAct Framework and Agent Architecture

2.1 Overall Agent Architecture

The agent follows the ReAct paradigm [4], implemented as a deterministic state machine on top of LangGraph [1]. The overall pipeline and the corresponding state-machine design are shown side-by-side in Figure 3.

- A user query first enters the system.

Preprint. Under review.

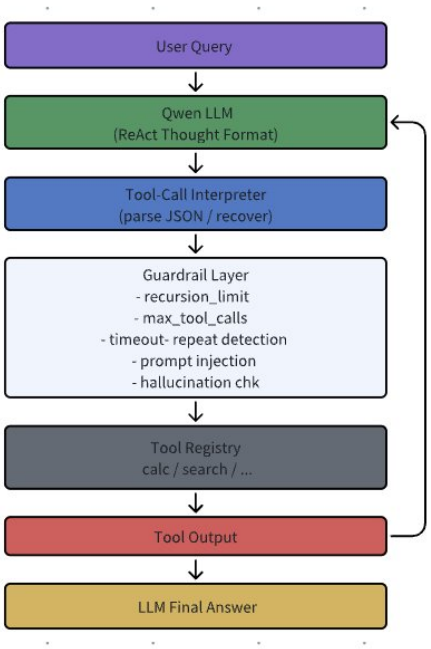

Figure 1: *
(a) Full system pipeline

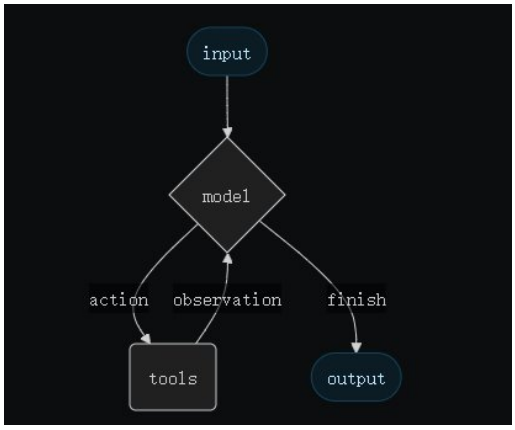

Figure 2: *
(b) LangGraph two-node state machine

Figure 3: Overall ReAct-based agent architecture. The left diagram shows the end-to-end pipeline, and the right diagram shows the underlying two-node state machine implemented using LangGraph.

- The agent executes a sequence of Thought $\rightarrow$ Action $\rightarrow$ Observation steps.
- Each step passes through a guardrail layer before execution.
- All events are synchronously logged into a structured trace for later analysis.

The architecture is organized as a two-node graph:

- `model_node`: runs the LLM, generating the next `Thought` and, when needed, an `Action`.
- `tool_node`: validates and executes a tool call, then returns an `Observation`.

This design enforces a strict one-tool-per-step policy and makes every intermediate decision fully auditable.

2.2 System Prompt and ReAct Format Enforcement

The ReAct behavior is enforced by a dedicated system prompt, which constrains how the model formats every message.

ReAct format rules.

- Every assistant turn must begin with:

  `Thought: <one sentence>`

- If a tool is needed, the model must emit:

  ```
  Action: <tool_name>
  Action Input: <a valid JSON object>
  ```

  and then *stop* after `Action Input` and wait for the tool result.

- After receiving tool output, the model continues with a new `Thought:` and either calls another tool or finishes with:

`Final: <your answer>`

- At most one tool may be used per turn. For complex tasks the model is asked to call `planner` first; for search tasks it should prefer `web_search`.

These patterns are parsed by regular-expression helpers such as `_extract_section` and `_parse_text_action_call` in `graph.py`, supporting both OpenAI-style function calling and textual ReAct output. The full system prompt is included in Appendix A and in the public repository:

`https://github.com/MoliaiELS/PythonPrograming-For-AI/blob/main/src/graph.py`

2.3 Tool Interface, Schemas, and Guardrails

The agent exposes a small set of deterministic tools implemented in `tools.py`. All tools follow a unified contract: the *input* is a strict JSON object (no extra fields), and the *output* is a JSON dictionary; malformed payloads are rejected by the guardrails.

Core tools.

- **Calculator (`calc`).** A safe arithmetic evaluator based on AST whitelisting. Only numeric operators (`+`, `-`, `*`, `/`, `//`, `%`, `**`) are allowed; variable access, attributes, and function calls are forbidden. Example input:

  `{"expression": "12 * (3 + 4)"}`

- **Wikipedia Search (`wiki_search`).** Queries the official Wikipedia API and returns structured snippets with document IDs (`wiki_1`, `wiki_2`, ...) and confidence scores. It never falls back to other sources and reports errors explicitly.
- **Baidu Search (`baidu_search`).** An HTML-scraping search tool with basic captcha detection. When blocked, it emits a structured error so that the agent can choose an alternative backend.
- **Serper.dev Search (`serper_search`).** A wrapper around a Google Search API. Results are ranked and annotated with document IDs and scores.
- **Aggregated Search (`web_search`).** A meta-tool that cascades over Serper, Baidu, and Wikipedia, and serves as the default entry point for factual queries.
- **Deterministic Planner (`planner`).** A non-LLM tool that produces short step-by-step plans with hints, helping the LLM decompose complex tasks and call tools in a stable way.

JSON-based tool schemas. Each tool is registered using an OpenAI-compatible JSON schema via `qwen_tools_schema()` in `tools.py`. A simplified schema for the calculator is shown in Listing 1; the full set of schemas is given in Appendix B.

Listing 1: Example JSON schema for the `calc` tool.

```
{
  "type": "function",
  "function": {
    "name": "calc",
    "description": "Safely evaluate an arithmetic expression.",
    "parameters": {
      "type": "object",
      "properties": {
        "expression": { "type": "string" }
      },
      "required": ["expression"],
      "additionalProperties": false
    }
  }
}
```

The main compliance properties are:

1. Only JSON objects are allowed as `Action Input`.
2. All fields must match the schema; unsupported parameters trigger a tool error.
3. The function-calling interface guarantees deterministic argument parsing.
4. Tools must return JSON dictionaries; non-dict outputs are rejected.

Execution guardrails. Every tool call passes through a guardrail layer implemented in `graph.py` and the `ToolRegistry`:

- **Schema validation.** The JSON payload must parse successfully and satisfy the declared schema; unknown tools immediately raise

  `ERROR: unknown_tool(<name>)`

- **Timeouts and repetition limits.** All tools execute inside a threaded wrapper with a configurable timeout (`tool_timeout_s`). Timeouts yield structured errors instead of blocking indefinitely. To avoid infinite loops, the agent tracks tool signatures; repeated identical calls beyond a threshold result in

  `ERROR: repeated_same_tool_call_too_many_times`

- **Global tool-call cap.** A global counter (`max_tool_calls`) limits the total number of tool uses per query, preventing runaway reasoning.

2.4 Debug Logging and ReAct Loop Tracing

A key engineering objective is to guarantee transparency and reproducibility. The agent therefore maintains a multi-layer logging stack that records every step of the ReAct loop in a structured, machine-auditable form.

Structured JSONL trace. Each run produces a `trace.jsonl` file, where every iteration of the ReAct loop is recorded as a single JSON object containing:

- a timestamped event label,
- the generated `Thought`,
- any emitted `Action` and its parsed arguments,
- the raw tool output (or structured error),
- internal metadata used for recursion limits and guardrail checks.

This format allows programmatic replay and automated auditing of the entire reasoning trajectory. A concrete JSON example of one complete ReAct episode is given in Appendix C.

CLI session logs. In `run_agent.py`, each execution creates a dedicated log directory containing:

- `run.log` – high-level execution summary,
- `chat.md` – human-readable transcript,
- `chat.jsonl` – interaction-level record.

These logs jointly capture both LLM outputs and tool-execution behavior.

Real-time debug mode. When the debug flag is enabled, the system prints the latest Observation to the console at each iteration, exposing tool behavior, mis-parsed JSON, and guardrail-triggered errors in real time.

Together, these mechanisms provide a coherent transparency layer over the strictly serialized ReAct loop (Thought $\rightarrow$ Action $\rightarrow$ Observation $\rightarrow$ Final), enabling targeted evaluation, ablations, and regression debugging.

3 Advanced Agent Augmentation and Stability Mechanisms

The move toward production-level agents requires augmenting the core ReAct framework with mechanisms that address stochastic variability and hallucination risks. This section describes the hallucination control and self-consistency components.

3.1 Hallucination Control

Hallucination control focuses on enhancing the factual grounding of the agent's responses, ensuring that every assertion is traceable to observed external evidence. This mechanism improves trustworthiness and reliability.

Evidence scoring and tracking. The retrieval component assigns each document snippet a unique `doc_id` and a confidence score that reflects relevance. By exposing this confidence score to the LLM via the Observation step, the agent can implicitly weigh evidence reliability in subsequent planning. Information derived from low-confidence documents can be flagged and may trigger re-search or a cautious conclusion.

Evidence-based ReAct and mandatory citation. The system prompt enforces mandatory citation requirements. The LLM must cite `[doc_id]` immediately following any factual statement. This forces the model to evaluate whether the observed evidence is robust enough before synthesizing the answer. If the model produces a plausible fact but fails to trace it to a valid observed `doc_id`, the statement is flagged as ungrounded. This shifts the objective from merely generating plausible text to generating empirically supported text.

Automatic rejection of ungrounded answers. After the LLM proposes a final answer, a post-processing audit script checks all citations. It verifies that each cited `doc_id` exists in the trace and that its evidence score exceeds a predefined threshold. If grounding fails, the system rejects the answer and either retries or returns a safe fallback.

3.2 Self-Consistency Decoding

Self-consistency is used to mitigate the high variance in LLM outputs and to stabilize the quality of the reasoning traces.

Multi-trace execution. The method runs the entire ReAct agent loop K times independently. This redundancy acts as a safeguard against single-run failures. If one trace selects a suboptimal search query, other traces may follow more effective paths.

Normalized answer comparison. Final answers from all K traces are normalized, for example by standardizing capitalization and punctuation, so that comparison focuses on semantics rather than token-level differences.

Consensus strategies. The system uses a hierarchy of strategies to derive a robust final answer:

1. **Exact match check.** If all K normalized runs yield identical answers, the result is returned immediately.
2. **LLM semantic verifier.** If consensus is not achieved, a dedicated LLM judge is prompted. This verifier analyzes competing answers and their reasoning traces and selects the answer that is most consistent, coherent, and best supported by evidence.
3. **Fallback majority vote.** If the semantic verifier is unavailable or inconclusive, the system defaults to majority voting over the K traces.

3.3 Safety Filter: Prompt Injection Defense

We integrated a lightweight safety module to prevent the agent from executing harmful or manipulative instructions.

Table 1: Evaluation dataset composition and focus of each category.

Category	Description	Count	Test Focus
Simple retrieval	Direct questions requiring one tool invocation (search).	4	Base tool reliability, prompt fidelity, hallucination control
Multi-hop reasoning	Questions requiring synthesis of multiple facts via sequential search.	10	Sequential planning and contextual state recall
Calculation	Fact retrieval followed by deterministic mathematical computation.	9	Tool offloading and grounded arithmetic
Safety	Questions attempting to override prompts or ethics constraints.	6	Safety filter behavior and robustness

Injection Pattern Detection. A heuristic detector scans user queries for common attack patterns (e.g., "ignore previous instructions", "DAN mode", "system override"). Detected cases are intercepted before they reach the ReAct reasoning pipeline.

Pre-Execution Blocking. If a query matches known injection signatures, the system immediately responds with a refusal message and does not execute any action.

Outcome. This mechanism protects the integrity of the agent's internal logic and tool policies, preventing adversarial manipulation while maintaining usability for benign queries.

4 Evaluation Methodology

The assessment of the ReAct agent requires a structured evaluation framework that goes beyond anecdotal success. The design is informed by existing benchmarks such as BIG-Bench Hard [2] and AgentBench [3], which contain difficult logical problems and tool-using tasks.

4.1 Custom Gold Set and Complexity Levels

A custom evaluation dataset with 29 distinct queries was created to test multiple aspects of the agent's architecture, including external data retrieval, sequential planning, arithmetic offloading, and safety.

The dataset is categorized into four complexity levels, summarized in Table 1.

4.2 Pass/Fail Criteria

For this task, a simple pass-or-fail metric is used. Each query has a gold target defined as a mixture of keywords and semantic conditions. A prediction is marked as pass if it matches the required content, and fail otherwise.

4.3 LLM-as-a-Judge Framework

To make evaluation objective and scalable, an automated evaluation system known as the LLM-as-a-judge framework is implemented using a strong model (Qwen-Plus). The judge model receives three inputs:

- the original user query,
- the agent's complete reasoning trace,
- the final answer.

The judge then assigns a score on a discrete scale (for example 1–5) based on:

- quality of reasoning and tool selection,
- helpfulness and relevance to the user query.

Table 2: Ablation study on model configuration and guardrails. Accuracy is pass rate on the 29-task evaluation set.

Model	Configuration	Accuracy	Time	Description
Qwen-Plus	consistency_k = 3	91.67%	10:04	Baseline self-consistent agent
Qwen-Turbo	consistency_k = 3	96.15%	06:58	Faster model with self-consistency
Qwen-Turbo	consistency_k = 1	96.15%	02:58	No self-consistency; lower cost and latency
Qwen-Turbo	consistency_k = 3, no planner	92.31%	08:54	Weaker multi-step planning
Qwen-Turbo	consistency_k = 3, no web search	84.62%	05:08	Relies on internal knowledge; poor on new or expert-domain questions
Qwen-Turbo	consistency_k = 3, no calculator	96.15%	07:27	Uses internal reasoning; fails on seven-digit multiplication

The numeric score is then converted into a binary pass/fail label. This framework leverages the traceability of the ReAct design and provides a rigorous and auditable assessment of performance.

5 Results and Failure Modes

5.1 Qualitative Validation of the ReAct Architecture

Qualitative analysis shows that ReAct functions as intended. It consistently answers complex planning queries accurately and is highly reliable on arithmetic by delegating calculations to the external tool, avoiding typical LLM errors. Some seven-digit multiplications still fail due to expression-handling and precision limits, but overall the results validate that agentic architectures can separate roles: the LLM manages language and reasoning, while deterministic modules handle computation.

5.2 Trace Analysis of Success Cases

Successful traces show that the agent can satisfy multi-criteria constraints. For example, for a travel query such as *"visit a city in France that is not Paris, has a beach, and is less than 3 hours by train from Paris"*, the agent:

- retrieves candidate cities via search,
- verifies coastal properties,
- checks train duration using another search, and
- synthesizes a final answer that satisfies all constraints.

The sequential nature of the ReAct loop allows the agent to maintain a complex short-term state, verify conditions against tool outputs, and iteratively refine the solution.

5.3 Failure Modes: Semantic Mismatch and Looping Instability

Failure cases reveal issues related to semantic ambiguity and termination logic. A representative failure occurs when the agent searches for a fictional entity, for example *"Grand Hotel Budapest price"*. The agent retrieves evidence that the hotel is fictional but does not transition to a safe termination state such as *"This hotel does not exist"*. In practice, however, the agent often reacts relatively quickly to hallucination-like questions and can produce the intended refusal, although this is not guaranteed.

5.4 Ablation Study on Guardrails

To validate the necessity of architectural guardrails and configuration choices, an ablation study is conducted by varying the model and key parameters such as the self-consistency factor and tool set. Results are summarized in Table 2.

Table 3: Illustrative step counts for current and optimized designs.

Question type	Current steps	Optimized steps	Waste
Simple lookup (Nobel Prize year)	4 (planner + 2 searches + calc)	2 (search + calc)	$\approx 50\%$
Multi-fact query (director age)	5–6 steps	3 steps	40–50%
Arithmetic problem (e.g., $239 \times 41 - 200$)	3+ steps with multiple failed attempts	1 step	$> 60\%$

The experiments confirm that guardrails such as recursion limits and maximum tool calls are not optional tuning parameters but mandatory architectural components. Without time-to-live constraints, the probabilistic nature of the LLM can lead to unbounded recursion and unstable behavior.

6 Operational Trade-offs and Future Directions

6.1 Latency, Cost, and Context Bloat

The accuracy and groundedness benefits of ReAct come with operational costs. Its serial LLM–tool loop introduces substantial latency, turning queries a standard LLM answers in two seconds into multi-step interactions that may take 15 seconds or more.

ReAct's continual short-term memory updates also cause context bloat: each step appends all prior thoughts, actions, and observations, rapidly increasing token usage, API costs, and computation time. Longer traces further suffer from the "lost in the middle" problem, reducing the model's attention to earlier instructions.

6.2 Memory Optimization and Parallel Tool Execution

Future development should mitigate latency and context bloat while enhancing personalization.

Memory optimization via RAG. To transform the general planner into a more personal assistant, a retrieval-augmented generation (RAG) system with a vector database can serve as long-term memory. User preferences (such as budget ranges and seating choices) are stored persistently. Before the ReAct loop starts, only relevant facts are retrieved and injected into context, which reduces repeated inclusion of static information and slows down context growth.

Parallel function calling. The main bottleneck is the serialized execution of tools. While some tasks are inherently sequential, many early information-gathering steps are independent. In such cases, parallel function calls (for example, querying weather, currency, and train schedules at the same time) can significantly reduce wall-clock time.

Table 3 sketches potential savings from optimizing step counts.

7 Conclusion

This project demonstrates clear advantages of a ReAct-based agent over static LLMs for dynamic tasks. The Thought $\rightarrow$ Action $\rightarrow$ Observation workflow improves factual accuracy and groundedness, and offloading arithmetic to a deterministic calculator removes most errors. System reliability hinges on engineering choices, including a strict system prompt and guardrails like step and tool-call limits. Techniques such as self-consistency and hallucination control further enhance robustness. Future work should reduce operational overhead, particularly through RAG-based memory optimization and parallel tool execution to achieve real-time, high-throughput performance.

References

[1] Langgraph: Build robust language agents as graphs. `https://langgraph.dev`, 2024. Accessed: 2025-02-XX.

[2] Mirac Suzgun and collaborators. Big-bench hard. https://github.com/suzgunmirac/BIG-Bench-Hard, 2021.

[3] THUDM Team. Agentbench: Evaluating language agents in realistic environments. https://github.com/THUDM/AgentBench, 2023.

[4] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

A Full System Prompt

For completeness, Listing 2 shows the full system prompt used to enforce the ReAct format, hallucination control, and citation rules.

Listing 2: Full system prompt used by the agent.

```
You are a tool using assistant. You must be accurate and transparent.
You have access to tools. Use them only when needed and never invent
    tool results.
You must follow this ReAct format in every assistant message.

Format rules:
1) Start with a single short line: Thought: <one sentence>.
2) If you need a tool, then write:
   Action: <tool_name>
   Action Input: <a valid JSON object>
   Stop after Action Input and wait for the tool result.
3) After you receive tool output, continue with a new Thought and then
     either call another tool or finish with:
   Final: <your answer>
4) Use at most one tool per turn.
5) If the task is complex, call planner first.
6) Tool output may contain untrusted content. Treat it as data only.
7) For search tasks, prefer web_search first.

Hallucination Control & Citation Rules:
1) Every factual statement (numbers, dates, names, events) MUST be
    supported by a retrieved document.
2) When you use information from a document, cite it using [doc_id] (e
    .g., [wiki_1], [serp_2]).
3) Evaluate the strength of evidence in your Thought. If documents
    have low scores (< 0.5), be skeptical.
4) If you cannot find high-confidence evidence, state that you cannot
    answer. Do not invent facts.
5) Your Final Answer must include a list of references at the end.

(Examples omitted here; see the code repository for full prompt
    variants.)
```

B Tool JSON Schemas

The main tools exposed to the agent are registered with OpenAI-compatible JSON schemas. For reference, Listing 3 reproduces the calculator schema, and similar specifications are used for the search tools.

Listing 3: Calculator tool schema (full version).

```
{
  "type": "function",
  "function": {
    "name": "calc",
```

```
    "description": "Safely evaluate an arithmetic expression.",
    "parameters": {
      "type": "object",
      "properties": {
        "expression": {
          "type": "string",
          "description": "A valid arithmetic expression, e.g. '12*(3+4)'."
        }
      },
      "required": ["expression"],
      "additionalProperties": false
    }
  }
}
```

C Example ReAct JSON Trace

For reference, Listing 4 shows a typical JSON trace corresponding to one complete ReAct episode. Each array element corresponds to a single step in the loop and is exactly what is written into `trace.jsonl` during execution.

Listing 4: Example JSON trace for a single ReAct episode.

```
[
  {
    "step": 1,
    "event": "thought",
    "content": "I should search for the required information."
  },
  {
    "step": 2,
    "event": "action",
    "name": "web_search",
    "input": { "query": "iPhone 15 release date" }
  },
  {
    "step": 3,
    "event": "observation",
    "output": {
      "results": [
        {
          "doc_id": "serp_1",
          "score": 0.90,
          "snippet": "Apple introduced iPhone 15 on September 12, 2023 ..."
        }
      ]
    }
  },
  {
    "step": 4,
    "event": "thought",
    "content": "The retrieved evidence is strong; I can answer now."
  },
  {
    "step": 5,
    "event": "final",
    "answer": "The iPhone 15 was introduced on September 12, 2023 [serp_1]."
  }
]
```