Agent Lifecycle Toolkit (ALTK): Reusable Middleware Components for Robust AI Agents

Zidane Wright
IBM Research
Yorktown Heights, NY, USA
Zidane.D.Wright@ibm.com

Jason Tsay*
IBM Research
Yorktown Heights, NY, USA
jason.tsay@ibm.com

Anupama Murthi
IBM Research
Yorktown Heights, NY, USA
anupama.murthi@ibm.com

Osher Elhadad
IBM Research
Haifa, Israel
Osher.Elhadad@ibm.com

Diego Del Rio
IBM
Capital Federal, Argentina
diego.del.rio@gmail.com

Saurabh Goyal
IBM Research
Yorktown Heights, NY, USA
saurabh.goyal1@ibm.com

Kiran Kate
IBM Research
Yorktown Heights, NY, USA
kakate@us.ibm.com

Jim A. Laredo
IBM Research
Yorktown Heights, NY, USA
laredoj@us.ibm.com

Koren Lazar
IBM Research
Haifa, Israel
koren.lazar@ibm.com

Vinod Muthusamy
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
vmuthus@us.ibm.com

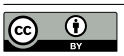
Yara Rizk
IBM Research
Yorktown Heights, NY, USA
yara.rizk@ibm.com

Abstract

As AI agents move from demos into enterprise deployments, their failure modes become consequential: a misinterpreted tool argument can corrupt production data, a silent reasoning error can go undetected until damage is done, and outputs that violate organizational policy can create legal or compliance risk. Yet, most agent frameworks leave builders to handle these failure modes ad hoc, resulting in brittle, one-off safeguards that are hard to reuse or maintain. We present the Agent Lifecycle Toolkit (ALTK), an open-source collection of modular middleware components that systematically address these gaps across the full agent lifecycle.

Across the agent lifecycle, we identify opportunities to intervene and improve, namely, post-user-request, pre-LLM prompt conditioning, post-LLM output processing, pre-tool validation, post-tool result checking, and pre-response assembly. ALTK provides modular middleware that detects, repairs, and mitigates common failure modes. It offers consistent interfaces that fit naturally into existing pipelines. It is compatible with low-code and no-code tools such as the ContextForge MCP Gateway and Langflow. Finally, it significantly reduces the effort of building reliable, production-grade agents.

*Corresponding author.



CAIS '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2415-2/26/05
https://doi.org/10.1145/3786335.3813206

CCS Concepts

• **Computing methodologies** → **Intelligent agents**; • **Software and its engineering** → **Software libraries and repositories**.

Keywords

AI Systems, AI Agents, Agentic Middleware

ACM Reference Format:
Zidane Wright, Jason Tsay, Anupama Murthi, Osher Elhadad, Diego Del Rio, Saurabh Goyal, Kiran Kate, Jim A. Laredo, Koren Lazar, Vinod Muthusamy, and Yara Rizk. 2026. Agent Lifecycle Toolkit (ALTK): Reusable Middleware Components for Robust AI Agents. In *ACM Conference on AI and Agentic Systems (CAIS '26), May 26–29, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786335.3813206

1 Introduction

The agentic paradigm has accelerated rapidly as developers build increasingly capable LLM-powered agents that can reason, call tools, and produce structured outputs. Yet, these systems remain fundamentally brittle: as complexity grows, so do issues like hallucinated tool calls, silent failures, inconsistent outputs, and reasoning errors that break workflows. To address these challenges, we introduce ALTK, an open-source, framework-agnostic package that improves agent reliability, predictability, and production readiness. Agent Lifecycle Toolkit (ALTK) can integrate into any agent pipeline and add deterministic safeguards and recovery mechanisms that elevate agents from "cool demos" to dependable, enterprise-grade systems.

Early agents often rely on a simple loop of repeated LLM tool calls, useful for prototypes but insufficient for enterprise reliability. Production agents need additional logic to ensure robustness, especially in domains like sales where a single misinterpreted field can trigger incorrect APIs and distort downstream forecasts. Agent

orchestration frameworks such as LangChain [4], LangGraph [10], CrewAI [16] offer building blocks such as tools, memory, and popular agent architectures. However, they expect the developers to write custom code handling tool call errors or checking for policy conformance.

ALTK is a modular toolkit that comes with pre-built, hardened drop-in components to strengthen reasoning, tool execution, and output validation in agents. Rather than enforcing a particular agent framework (such as LangChain, LangGraph, or AutoGPT), its framework-agnostic design allows teams to introduce targeted reliability improvements without re-architecting their agents.

ALTK currently includes 10 components, each addressing a distinct failure mode in the agent lifecycle, as summarized in Figure 1. For example, the lifecycle stage of "Pre-Tool" indicates a step in the agent's execution when the LLM has generated a tool call but the tool call is yet to be executed. A "Pre-Tool" ALTK component such as SPARC takes the generated tool call, tool specifications, and agent context as input and performs different checks on the tool call to check for correctness. SPARC flags an invalid tool call with reasoning and suggestions for correcting it so the agent skips executing the wrong tool call and asks the LLM to take this feedback and generate a correct call.

What distinguishes ALTK is its flexibility through precise, surgical improvements rather than all-in-one orchestration. This allows for multiple paths of integration into existing agentic systems, including existing low-code and no-code systems such as LangFlow and the ContextForge MCP Gateway. All ALTK components have been rigorously evaluated on public benchmarks and show clear gains over baseline settings.

2 Toolkit Approach

ALTK is organized around key stages in the agent lifecycle, as shown in Figure 1: build-time, pre-LLM prompt preparation, post-LLM, pre-tool call, post-tool call, and pre-final response. The main design goals are *separation of concerns* and *modularity*: each component targets one dominant error mode and can be enabled independently or in combination with others.

ALTK is currently implemented as an open source Python library called *altk-boost* that provides a simple interface into these lifecycle components. We intend for this library and its components to be plug-and-play and as framework-agnostic as possible. Integrating an ALTK component into an agent during runtime has three phases which can be done in three lines of code: 1) defining the input to the component, 2) instantiating and configuring the component, and 3) processing the given input and reviewing the result. Concretely, for the code example in Figure 2, we use the Silent Error Review component in the ALTK as a post-tool hook to check for silent errors in a problematic tool. This component expects the current log of messages and tool response as input and then after instantiating the Silent Error Review component, these messages are processed and a result is given whether or not a silent error was detected. This result can be given back to the agent to prevent unintended behavior. Each component in the ALTK library follows the same basic interface and phases. The library is available on GitHub[1] and PyPi[2]. For more code examples, please see the *examples* folder in repository. A video walkthrough[3] is available with additional demonstration videos on the ALTK YouTube channel[4].

At the time of writing, ALTK has 10 components (see Figure 3), spanning lifecycle stages from build-time to right before the response is given to the user. For this demonstration, we focus on three components: SPARC as a *pre-tool* call validator, JSON Processor as a *post-tool* processor of long responses, and a *post-tool* Silent Error Review.

2.1 Pre-tool: SPARC - pre-execution validation

In enterprise settings, semantically incorrect tool calls may waste API quota, corrupt external state, or trigger irreversible actions. Production agents need an inline runtime mechanism that decides whether a specific call should even be allowed to execute. SPARC is a component that works at the pre-tool lifecycle stage. Based on the message history, the provided tool specifications and any candidate tool calls, SPARC performs *syntactic* validation, *semantic* validation, and *transformation* validation. Syntactic validation is rule-based and catches non-existent tools, unknown arguments, missing required parameters, type mismatches, and JSON-schema violations. Semantic validation uses one or more LLM judges to assess function-selection appropriateness, parameter grounding, hallucinated values, value-format alignment, and unmet prerequisites. Transformation validation handles format or unit mismatches (for example, date, currency formats) and performs automatic conversions as demanded by the tool specifications. The output of the component identifies whether tool call is invalid. If the tool call is not valid, SPARC identifies issues and suggests remediation.

2.2 Post-tool: JSON Processor

In a tool-augmented agentic pipeline, the JSON processor acts as a critical middleware layer between raw API output and downstream reasoning. Passing voluminous, deeply nested JSON directly into the agent's context competes for attention with the task prompt and has been shown to degrade accuracy as responses grow longer [8]. Instead, this component delegates the parsing to a code generation step: the LLM is prompted to write a short Python function that navigates the JSON structure, applies any necessary filtering or aggregation logic, and returns only the extracted answer. This approach treats the LLM as a programmer rather than a reader, playing to its strength in producing structured code. When augmented with the API's JSON response schema, the generated parser becomes even more reliable; the model can reason about field names, data types, and nesting relationships without needing to infer them from the raw data. The resulting agent architecture is both more token efficient, since the code output is far smaller than the full response it processes, and more composable, as the deterministic output of an executed script slots cleanly into the next stage of an agent loop without any formatting noise and verbosity.

[1] https://github.com/AgentToolkit/altk-boost

[2] https://pypi.org/project/altk-boost/

[3] https://www.youtube.com/watch?v=FsTuf9fmgM4

[4] https://www.youtube.com/@AgentToolkit

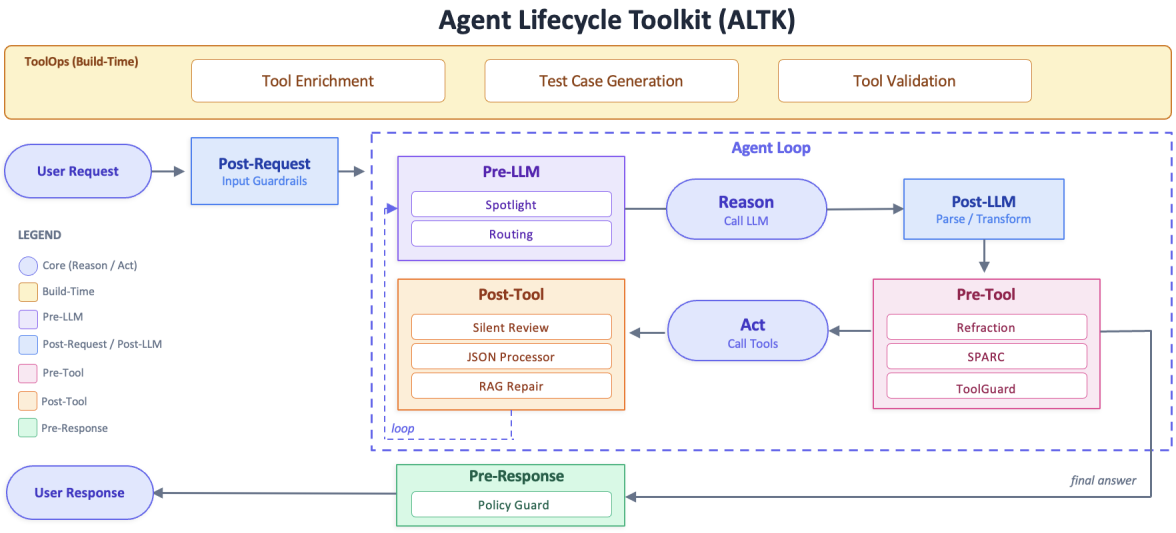


Figure 1: Agent lifecycle and corresponding ALTK components

```python
def post_tool_hook(state: AgentState) -> dict:
    # Post-tool node to check the output of a problematic tool
    tool_response = state["messages"][-1].tool_outputs
    review_input = SilentReviewRunInput(messages=state["messages"],
    ↪  tool_response=tool_response)
    reviewer = SilentReviewForJSONDataComponent()
    review_result = reviewer.process(data=review_input,
    ↪  phase=AgentPhase.RUNTIME)
    if review_result.outcome == Outcome.NOT_ACCOMPLISHED:
        return "Silent error detected, retry the tool!"
    else:
        # (allow tool call to go through)

agent_graph.add_edge("flaky_tool", "post_tool_hook")
```

Figure 2: Code example in Python integrating the Silent Error Review component from ALTK to a LangGraph agent as a post-tool hook.

Problem	Solution	Component
Agent ignores prompt instructions	Highlight key prompt spans	Spotlight
Routing picks wrong target	Topic hints direct selection	Routing
Business policies not followed	Deterministic policy enforcement	ToolGuard
Broken or invalid tool syntax	Validate and fix syntax	Refraction
Hallucinated or wrong tool calls	Ensure correct tool calls	SPARC
Overloaded by large JSON	Extract relevant JSON data	JSON Processor
Missed subtle tool errors	Detect subtle tool errors	Silent Error Review
Failure after tool execution	Recover failed tool calls	RAG Repair
Responses violate defined policies	Enforce output policies	Policy Guard
Misuse from poor tool specs	Enrich, test, validate tools	ToolOps

Figure 3: List of components in ALTK

2.3 Post-tool: Silent Error Review

A common scenario for agentic tool usage is "soft failures" where an API may return responses that seem correct by returning a HTTP status code of "200 OK" but the body of the response may contain text like "Service under maintenance" or "No results found." Traditional agents often interpret this tool response as a correct final answer and this may cause unintended behavior. The Silent Error Review component works at the post-tool stage to identify these failures using a prompt-based approach. The component takes as input the user query, the tool response, and optionally the tool specification, to review the response as "ACCOMPLISHED", "PARTIALLY ACCOMPLISHED", or "NOT ACCOMPLISHED".

3 Evaluations

As ALTK is comprised of many components, each component is individually evaluated for effectiveness in their particular task. We present evaluations for the subset of ALTK components we focus on for this demonstration.

3.1 Pre-tool: SPARC Evaluation

We evaluate SPARC on the airline API subset of the τ-bench dataset [21] in a ReAct loop. If the candidate call is approved, it executes. If rejected, the reflection artifact (issue type, evidence, and correction suggestion) is fed back to the agent, which retries. Figure 4 reports the experimental results. The main pattern is that gains grow with k. For GPT-4o self-reflection, pass^1 moves from 0.470 to 0.485, while pass^4 improves from 0.260 to 0.300. SPARC is most helpful for near-miss trajectories: it turns many incorrect first proposals into recoverable tool decisions on subsequent retries.

3.2 Post-tool: JSON Processor

We evaluate the JSON processor component on 15 models from various families and sizes on a dataset of approximately 1,300 JSON responses queries of varying complexity. Using the JSON processor leads to improvements over directly prompting a model to retrieve the answers without using the JSON processor component. Figure 5 shows 16% improvement on average across models when using the JSON processor [8].

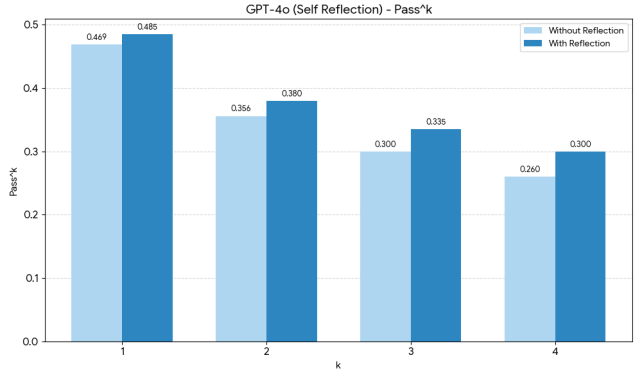


Figure 4: τ-bench airline pass^k with and without SPARC. "With reflection" inserts SPARC before each tool call and returns critique and/or corrections to the agent when a call is rejected.

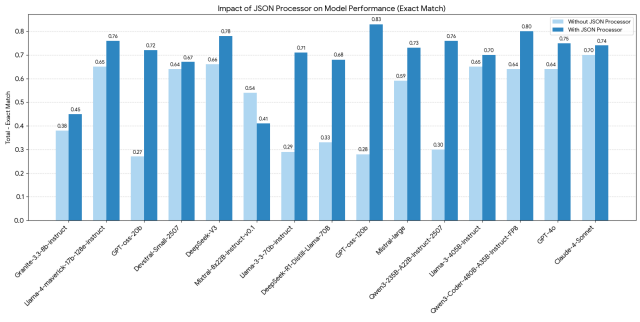


Figure 5: Model performance with and w/o JSON Processor[8]

3.3 Post-tool: Silent Error Review Evaluation

We evaluate the Silent Error Review component on the LiveAPIBench dataset for SQL queries in a ReAct loop [5]. We evaluate using two metrics: Micro Win Rate and Macro Win Rate. The Micro Win Rate is the average performance across individual data subsets and the Macro Win Rate is the overall performance across all samples in all subsets. As seen in figure 6, adding Silent Error Review to the ReAct loop nearly doubles the micro win rate, indicating more queries are fully or partially accomplished. Additionally, the average loop counts decrease showing that fewer iterations are needed to reach success.

4 Usage and Integrations

To maximize accessibility and facilitate adoption across diverse enterprise environments, ALTK is architected to support multiple deployment approaches. Rather than forcing a single integration pattern, the toolkit targets three distinct developer profiles: pro-code, low-code, and no-code.

In the pro-code case, for software engineers and AI researchers requiring fine-grained control over the execution flow, ALTK is distributed as a modular set of open-source Python libraries hosted on GitHub. Developers install the core packages and directly modify their existing agent source code, embedding ALTK's lifecycle hooks (Pre-Tool, Post-Tool, Pre-Response) natively within their

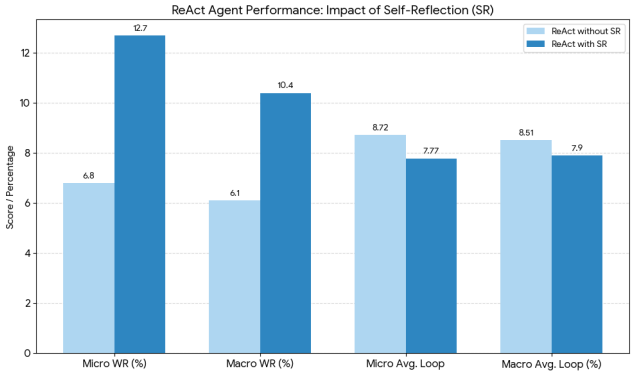


Figure 6: ReAct agent performance comparison with and without Silent Error Review on LiveAPIBench [5]; WR: Win Rate

logic. There are several Jupyter notebooks with examples of how to use the various components[5].

For rapid prototyping and visual workflow builders, ALTK provides deep integration with the Langflow[6] low-code tool. ALTK capabilities are exposed via a custom agent within Langflow's visual authoring environment. The ToolGuard , JSON Processor , and SPARC components are available in Langflow with demos on our YouTube channel[7].

For system administrators, platform engineers, and operations teams who need to secure and enhance legacy or third-party agents without altering their underlying code (no-code), ALTK integrates seamlessly as a middleware layer via the ContextForge MCP Gateway[8]. Administrators configure the ALTK integrations in the gateway to intercept outbound tool calls and inbound responses, applying ALTK's SPARC or JSON Processor dynamically. Tool enrichment and testing are available at the gateway for more effective agent tool use. See our YouTube channel[7] for demos of these integrations.

5 Related Work

A wide ecosystem of agent orchestration frameworks, such as LangChain [4], LangGraph [10], LlamaStack [2], LlamaIndex [11], CrewAI [16], AutoGen [20], Claude's Agent SDK [3], Bee [7], Smolagents [6], and Haystack [15], provides tooling to build LLM-based agents. These frameworks largely focus on workflow composition, infrastructure, or developer-side robustness. They offer hooks and middleware [9] for retries, fallbacks, and tool routing, but they generally do not detect or prevent semantic errors in agent reasoning or tool use.

ALTK fills this gap by acting as a framework-agnostic reliability layer that integrates at lifecycle hook points within any agent system. For example, ALTK can plug into LangChain middleware or be exposed as skills or MCP components within Claude's SDK.

Compared to data and model-centric approaches (e.g., APIGen [13], ToolACE [12], Granite function-calling models [1]), which

[5]https://agenttoolkit.github.io/altk-boost/examples/
[6]https://www.langflow.org/blog/langflow-1-7
[7]https://www.youtube.com/@AgentToolkit
[8]https://ibm.github.io/mcp-context-forge/

improve average tool-calling quality through better training, ALTK provides an inference-time gate to determine whether a specific call should run.

Compared to reflection- and repair-based methods (e.g., Reflexion [18], REBACT [22], ToolReflection [17], Failure Makes the Agent Stronger [19], and Tool-MVR [14]), ALTK's SPARC module is similar in spirit but differs by operating before execution, producing structured outputs rather than free-form critiques, and combining semantic reflection with deterministic schema and execution-verification checks.

Overall, ALTK complements rather than replaces existing agent frameworks, providing systematic runtime safeguards that enhance correctness and reliability.

6 Conclusion

ALTK is motivated by a simple observation: robust agents need more than a capable base LLM; they need to address the dominant failure modes that occur at various points in the agent lifecycle. ALTK provides a flexible toolkit of components to address common problems at these lifecycle stages. This flexibility is open-ended, as we invite agent builders to extend the toolkit with their own solutions to problems as well as integrate components into agentic systems. We believe lifecycle-based components are key to building agents that are intelligent, reliable, and adaptable.

Beyond runtime reliability, ALTK can also support analytics and evaluation workflows by applying its lifecycle checks to agent trajectories, enabling fine-grained analysis of failure modes and model behavior. These same components can provide structured signals for reward model training or tuning, turning ALTK's reflectors into supervision signals that improve tool-use fidelity and policy adherence.

References

[1] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Shrivatsa Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. 2024. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 1131–1139.

[2] Meta AI. 2024. Llama Stack. https://github.com/meta-llama/llama-stack.

[3] Anthropic. 2024. Claude Agent SDK. https://docs.anthropic.com.

[4] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain.

[5] Benjamin Elder, Anupama Murthi, Jungkoo Kang, Ankita Rajaram Naik, Kiran Kate, Kinjal Basu, and Danish Contractor. 2026. Live API-Bench: 2500+ Live APIs for Testing Multi-Step Tool Calling. arXiv:2506.11266 [cs.SE] https://arxiv.org/abs/2506.11266

[6] HuggingFace. 2024. Smolagents. https://github.com/huggingface/smolagents.

[7] IBM Research. 2024. Bee Agent Framework. https://github.com/i-am-bee/bee-agent-framework.

[8] Kiran Kate, Yara Rizk, Poulami Ghosh, Ashu Gulati, Tathagata Chakraborti, Zidane Wright, and Mayank Agarwal. 2025. How Good Are LLMs at Processing Tool Outputs? *arXiv preprint arXiv:2510.15955* (2025).

[9] LangChain. 2024. LangChain Built-in Middleware. https://docs.langchain.com/oss/python/langchain/middleware/built-in.

[10] LangChain. 2024. LangGraph. https://github.com/langchain-ai/langgraph.

[11] Jerry Liu. 2022. LlamaIndex. https://github.com/run-llama/llama_index.

[12] Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. 2024. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920* (2024).

[13] Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems* 37 (2024), 54463–54482.

[14] Zhiyuan Ma, Jiayu Liu, Xianzhen Luo, Zhenya Huang, Qingfu Zhu, and Wanxiang Che. 2025. Advancing tool-augmented large language models via meta-verification and reflection learning. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 2078–2089.

[15] Malte Möller, Marius Mosbach, Terry Ruas, Silvestro Severini, and Iryna Gurevych. 2020. Haystack: An end-to-end NLP framework. In *Proc. EMNLP (Demos)*.

[16] João Moura. 2023. CrewAI. https://github.com/crewAIInc/crewAI.

[17] Gregory Polyakov, Ilseyar Alimova, Dmitry Abulkhanov, Ivan Sedykh, Andrey Bout, Sergey Nikolenko, and Irina Piontkovskaya. 2025. ToolReflection: Improving Large Language Models for Real-World API Calls with Self-Generated Data. In *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)*. 184–199.

[18] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems* 36 (2023), 8634–8652.

[19] Junhao Su, Yuanliang Wan, Junwei Yang, Hengyu Shi, Tianyang Han, Junfeng Luo, and Yurui Qiu. 2025. Failure makes the agent stronger: Enhancing accuracy through structured reflection for reliable tool interactions. *arXiv preprint arXiv:2509.18847* (2025).

[20] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155* (2023).

[21] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. tau-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045* (2024).

[22] Qiuhai Zeng, Sarvesh Rajkumar, Di Wang, Narendra Gyanchandani, and Wenbo Yan. 2025. Reflect before Act: Proactive Error Correction in Language Models. *arXiv preprint arXiv:2509.18607* (2025).