ToolSmith: A Multi-Agent Framework for Enterprise Tool Creation

Purna Chandra Sekhar Vakudavathu[1,2*], Kushal Mukherjee[2], Jayachandu Bandlamudi[2], Renuka Sindhgatta[2], Sameep Mehta[2]

[1]Department of Mathematics, Indian Institute of Technology, Delhi
[2]IBM Research

Abstract

Although LLMs can generate tools for generic domains and tasks, they struggle with enterprise-related domains that involve proprietary APIs and data schemas. We present ToolSmith, a framework for autonomously generating and validating agent-compatible tools. Given an API specification and a Tool Specification Requirement (TSR), ToolSmith produces a tool function and verifies it through a closed-loop process: it creates natural language (NL) tests and executes the tool in a secure agent sandbox for validation. For state-changing tools, ToolSmith confirms outcomes by querying the API with parameters derived from the NL tests. If the tool fails to produce the desired output, ToolSmith generates diagnostic feedback to iteratively regenerate it. By ensuring both functional correctness and agent compatibility, ToolSmith enables reliable automation of enterprise workflows.

Introduction

Advances in large language models (LLMs) have enabled AI agents to perform complex real-world tasks using natural language instructions (Yao et al. 2023; Liu et al. 2023; Parisi, Zhao, and Fiedel 2022; Schick et al. 2023). These agents are increasingly adopted in domains such as enterprise automation, customer service, and software engineering (Masterman et al. 2024).

Agentic systems are enabled with tools that expand an agent's ability to interact with its environment. In enterprises, these tools often correspond to wrapper services around APIs[1] that encapsulate business functionality and operations. To make agents effective in such settings, it is necessary to construct bespoke tools that combine APIs with auxiliary code and can be reliably invoked by the agent.

The primary challenge in tool creation lies in ensuring that the tools are robust, reliable, and secure. Each tool should be self-contained, functionally accurate, and accompanied by comprehensive metadata—encompassing descriptions, inputs, and outputs—to facilitate seamless integration with the agentic system.

We introduce ToolSmith, a framework that leverages the Langgraph[2]-based agentic system to generate and validate enterprise tools. ToolSmith creates tools that incorporate enterprise APIs and supporting logic, while systematically testing them for correctness and agent compatibility.

Prior Work

Prior work in tool creation involves two main paradigms, both of which exhibit critical limitations for enterprise use cases. The first work is on-the-fly code generation(Qian et al. 2023; Zheng et al. 2024; Wang et al. 2024). In this approach, the agents generate and execute code when attempting to respond to a user's prompt. On-the-fly code generation is fundamentally insecure, lacking governance over LLM's actions, and unreliable, as the code is ephemeral and not validated. The second approach is a more structured paradigm and is found in frameworks like LATM(Cai et al. 2024). This process involves the identification of what tools need to be created and then using separate LLM calls to generate the tool code and unit tests for the tool. The generated tool is executed against these unit tests directly as a native function and validated. This approach exhibits several key limitations that inhibit its usage for enterprise use cases. **Domain Knowledge Gap:** The unit test generation lacks knowledge of proprietary enterprise domains, leading to hallucinated test cases and unreliable verification (Eghbali and Pradel 2024; Gu et al. 2025; Yu et al. 2025). **Native testing:** Direct unit tests only verify the code's logic in the tool, ignoring vital tool properties such as well-formed docstrings. (Trilcke et al. 2025). High-quality docstrings are essential metadata for agents to work with tools.

Our Contributions

Our key contributions are as follows:

Context Grounded Test Generation: Many API-wrapped tools require input values that are only available at runtime (such as valid user IDs or account numbers), which cannot be fully captured in the API specifications. As a result, purely LLM-generated test cases may be syntactically valid but contain irrelevant data values, leading to empty responses or 'no results found' errors. Our framework generates Natural Language tests by exploring the available data using the APIs within the agentic flow. This grounding of

[*]Work done during an internship at IBM Research.

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

[1]https://swagger.io/specification/

[2]https://www.langchain.com/langgraph

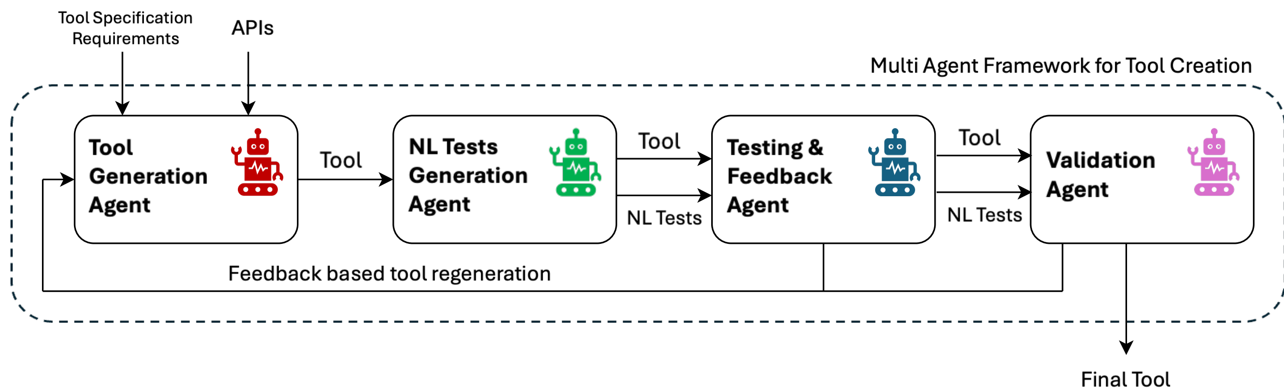

Figure 1: ToolSmith Framework

NL tests ensures that the entities and values used correspond to what is actually present in the service.

Agent-Centric Verification & State Validation: We introduce a testing process that validates the tool's logic and agent-compatible properties (e.g., docstrings) in a sandbox containing the agent and the target LLM. For state-altering tools, it confirms outcomes by querying the API with parameters grounded in the NL tests.

Autonomous Self-Correction Loop: Upon NL test failures and validation failures, the framework generates diagnostic feedback on the error to guide the regeneration of the tool code and repeats the verification cycle.

Framework

The ToolSmith Framework, presented in Figure 1, employs four key agents in an iterative workflow. The process is initiated by the Tool Generation Agent, which generates the tool. This tool then enters a verification loop driven by the NL Test Generation Agent, the Testing & Feedback Agent, and the Validation Agent.

Tool Generation Agent: This agent takes API specification[3] and accompanying Tool Specification Requirement (TSR) as input. It then generates a Python function that implements the required logic and includes a comprehensive, schema-compliant docstring[4]. Docstrings act as metadata (purpose, inputs, outputs) for agent compatibility.

NL Test Generation Agent: To create grounded, API-compatible NL tests, this system employs a two-step method separating the NL test structure from its data. First, the NL Test Generation Agent generates the linguistic structure of the NL test based on the tool's docstring, using placeholders for actual data values. (eg, `<USERID>`). Second, it analyzes the API specification to identify a suitable data-fetching endpoint to populate the NL tests. Then, it generates a dynamic code snippet that calls the endpoint to retrieve the required data from the backend service.

Testing & Feedback Agent: This agent orchestrates agent-centric integration testing by passing the generated tool and NL tests to a Tool Testing Sandbox. In the sandbox, we create a ReACT(Yao et al. 2023) style agent with the target tool and specified LLM using the Langgraph framework. The ReACT agent then invokes the appropriate tool based on the NL test as a part of its reasoning process, simulating real-world execution. Tool output is validated by cross-referencing the response against both its predefined API schema for structural correctness and the NL Tests parameters for contextual consistency. Based on these results, the agent determines the next step: (1) *Self-Correction*: If the tool fails execution, violates the schema, or contradicts the NL Tests parameters, the agent generates diagnostic feedback (eg, `Tool failed with 404 error , the endpoint may be incorrect`) to guide the Tool Generation Agent in regenerating the code. (2) *Advance to State-Change Validation*: If execution is successful and state-altering (e.g., a POST request), it passes to the Validation Agent. (3) *Success*: If successful and fetch-only (e.g., a GET request), the process terminates, and the verified tool is returned.

State-Change Validation Agent: This agent validates the created tool by inferring the expected state by combining the API's semantics with the parameters from the NL tests. It then generates and executes a dynamic code snippet to query the system's current state, verifying that it matches the expected outcome. In case of a validation failure, the agent generates diagnostic feedback and guides the framework to regenerate the tool.

Discussion and Conclusion

This paper proposes ToolSmith, a multi-agent framework for the autonomous generation and validation of agent-compatible tools. The framework is designed to accelerate the adoption of autonomous agents in enterprise workflows by securely and scalably transforming API specifications into usable tools. Future work includes benchmarking the framework to determine the best LLMs to use under cost constraints. In addition, we plan to analyze frequent patterns in ToolSmith agentic flow traces and optimize the graph to reduce cost and improve performance.

[3]https://swagger.io/specification/

[4]https://peps.python.org/pep-0257/#multi-line-docstrings

References

Cai, T.; Wang, X.; Ma, T.; Chen, X.; and Zhou, D. 2024. Large Language Models as Tool Makers. In *The Twelfth International Conference on Learning Representations*.

Eghbali, A.; and Pradel, M. 2024. De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding. arXiv:2401.01701.

Gu, X.; Chen, M.; Lin, Y.; Hu, Y.; Zhang, H.; Wan, C.; Wei, Z.; Xu, Y.; and Wang, J. 2025. On the Effectiveness of Large Language Models in Domain-Specific Code Generation. *ACM Trans. Softw. Eng. Methodol.*, 34(3).

Liu, R.; Wei, J.; Gu, S. S.; Wu, T.-Y.; Vosoughi, S.; Cui, C.; Zhou, D.; and Dai, A. M. 2023. Mind's Eye: Grounded Language Model Reasoning through Simulation. In *The Eleventh International Conference on Learning Representations*.

Masterman, T.; Besen, S.; Sawtell, M.; and Chao, A. 2024. The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey. *ArXiv*, abs/2404.11584.

Parisi, A.; Zhao, Y.; and Fiedel, N. 2022. TALM: Tool Augmented Language Models. arXiv:2205.12255.

Qian, C.; Han, C.; Fung, Y.; Qin, Y.; Liu, Z.; and Ji, H. 2023. CREATOR: Tool Creation for Disentangling Abstract and Concrete Reasoning of Large Language Models. In Bouamor, H.; Pino, J.; and Bali, K., eds., *Findings of the Association for Computational Linguistics: EMNLP 2023*, 6922–6939. Singapore: Association for Computational Linguistics.

Schick, T.; Dwivedi-Yu, J.; Dessi, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Trilcke, P.; Börner, I.; Sluyter-Gäthje, H.; Skorinkin, D.; Fischer, F.; and Milling, C. 2025. Agentic DraCor and the Art of Docstring Engineering: Evaluating MCP-empowered LLM Usage of the DraCor API. arXiv:2508.13774.

Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2024. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

Yu, H.; Chen, T.; Huang, J.; Li, Z.; Ran, D.; Wang, X.; Li, Y.; Marron, A.; Harel, D.; Xie, Y.; and Xie, T. 2025. DeCon: Detecting Incorrect Assertions via Postconditions Generated by a Large Language Model. *CoRR*, abs/2501.02901.

Zheng, T.; Zhang, G.; Shen, T.; Liu, X.; Lin, B. Y.; Fu, J.; Chen, W.; and Yue, X. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics: ACL 2024*, 12834–12859. Bangkok, Thailand: Association for Computational Linguistics.