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. We have also shown an improved performance of our approach compared to the standard LATM (LLM as tool maker) baseline on a generated benchmark dataset.

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

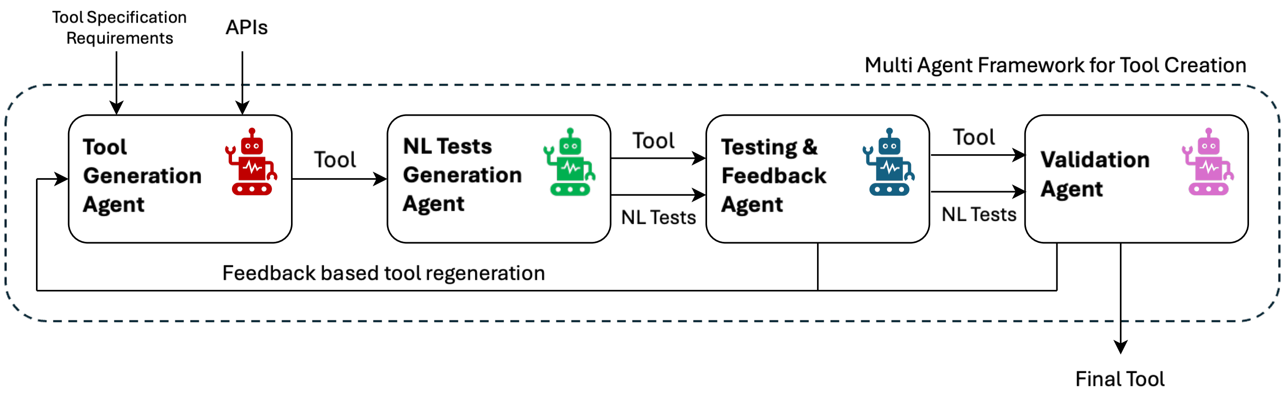
Our key contributions are as follows:

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

*Work done during an internship at IBM Research.

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

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



Tool Specification Requirements
APIs
Multi Agent Framework for Tool Creation
Tool Generation Agent
Tool
NL Tests Generation Agent
Tool
NL Tests
Testing & Feedback Agent
Tool
NL Tests
Validation Agent
Feedback based tool regeneration
Final Tool

Figure 1: ToolSmith Framework

using the APIs within the agentic flow. This grounding of 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. Using these two inputs, the agent's task is to generate a complete Python function. This function implements the API call and the necessary logic to fulfil the TSR, such as filtering, aggregating and transforming the data returned by the API call. The generated Python function includes a comprehensive, schema-compliant docstring [4]. The docstring acts as metadata, ensuring the tool is agent-compatible by describing its purpose, inputs, and outputs for agents. This initial, generated tool is then passed on to the next agents in the framework for testing and validation.

NL Test Generation Agent: The API specification along with helper API specifications, TSR and the tool docstring are provided to this 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 and TSR, 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 to retrieve the required data from the backend service.

Testing & Feedback Agent: The API specification, generated Tool and the NL Test are given to this agent. The Testing & Feedback Agent classifies whether the generated tool is state-altering by analyzing its code, the TSR, and the NL test. This agent orchestrates the agent-centric integration testing by passing the generated tool and the 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. This method validates both the tool's logic and its agent compatibility. Based on the tool execution in the sandbox, the Testing & Feedback Agent determines the next step by following one of three pathways: (1) *Self-Correction*: If the tool fails to execute, contains malformed docstrings, returns an output that violates the API's response schema or provides an output inconsistent with the NL tests parameters, the agent generates diagnostic feedback (eg, a comment like `Tool failed with 404 error, the endpoint may be incorrect`) to guide the Tool Generation Agent in regeneration of the tool code, repeating the verification loop. (2) *Advance to State-Change Validation*: If the tool executes successfully and is state-altering (e.g., a POST request), it is passed to the Validation Agent. (3) *Success*: If the tool executes successfully and is fetch-only (e.g., a GET request), the process terminates, and the verified tool is returned.

State-Change Validation Agent: The State-Change Validation Agent acts as the final verifier for tools that modify system data. After a tool successfully executes in the test sandbox, this agent is provided with the API specification and the NL Test. It first combines the API's semantics with the parameters from the NL tests. It then generates and exe-

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

cutes 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.

Benchmark Dataset

We constructed our dataset using an Enterprise API collection consisting of 55 enterprise APIs, each exposing a single operation. The APIs are related to sales engagement (Salesloft[5]) and vary in complexity with respect to their input and output parameters.

Benchmark Dataset Generation

Tool Specification Requirement generation: We utilized the API specifications, along with carefully designed prompts, to guide a Creator LLM (`llama4`[6]) in generating comprehensive tool specification requirements. For each API specification, we generated five candidate requests. These requests were then evaluated and filtered by a Judge LLM (`llama4`) based on the following criteria: (i) alignment with the original API specification, (ii) clarity and completeness of the request, (iii) correctness of the target endpoint and its operational status, and (iv) overall feasibility for tool generation.

Helper APIs collation: Since each API specification in this collection exposes only one operation, additional helper APIs were required to validate the generated tools. To identify suitable helpers, `llama4` was prompted to generate concise (2–3 line) summaries for each of the 55 APIs. These summaries were then aggregated and combined with the selected tool specification requirement. Using this combined context, `llama4` retrieved the three most relevant helper APIs whose operations were most likely to support tool validation.

Note that helper API collation was necessary only for the Salesloft data set, as each API exposes a single endpoint. In practical enterprise environments, APIs typically provide multiple endpoints, reducing the need for such a collation.

Through this process, a total of 186 pairs of API-tool specification requirements were generated. Figure 2 shows an example benchmark data sample.

We classify the TSR data samples into four classes based on the tool specification requirement generated. The classification is as follows:

1. **Read-without data fetch:** Tool involving *read-only* requests with inputs that can be generated independently of the data in the backend system. Example: Create a tool to retrieve the last five active accounts.
2. **Read-with data fetch:** Tool involving *read-only* requests where the inputs are dependent on the data in the backend system. Example: Create a tool to fetch meeting details for a user id. The user id must already exist in the backend for the tool to work.

[5]https://developers.salesloft.com/docs/api/

[6]https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct-FP8

Request Type	Number of Data Samples
Read-without data fetch	66
Read-with data fetch	80
Write-without data fetch	8
Write-with data fetch	32
Total	**186**

Table 1: Dataset Composition by Request Type.

```
{
  "api_spec_filename":
  "Salesloft_Get_all_cadence_memberships.json",
  "Generated TSR": "create a tool to retrieve
  cadence membership details, filter by
  cadence ID, and count active memberships",
  "auth_type": "Bearer",
  "Ground Truth API Sequence":
  GET /v2/cadence_memberships
}
```

Figure 2: An example benchmark data sample containing the API spec filename and TSR.

3. **Write-without data fetch:** Tool involving *write* requests with inputs that can be generated independently of the data in the backend system. Example: Create a tool to create a new account in the database.
4. **Write-with data fetch:** Tool involving *write* requests where the inputs are dependent on the data in the backend system. Example: Create a tool to modify the meeting details of a user id.

Table 1 represents the composition of the dataset based on the type of tool specification requirement.

Experimental Setup

We benchmark our proposed framework against the existing tool creation framework, LATM. Our LATM implementation follows the setup described in (Cai et al. 2024), employing two LLMs. The first LLM generates executable tool code from the provided API specification and Task-Specific Requirement (TSR). The second LLM uses the generated code and API specification to produce a corresponding Natural Language (NL) test. Both outputs—the tool code and NL test—are evaluated in a Tool Testing Sandbox, where a ReAct agent executes the tool to determine functional correctness. We use `llama4` for both tool code and NL test generation, and `mistral-small`[7] as the sandbox ReAct agent.

For ToolSmith, we input the same API specification and TSR. If the framework fails to invoke the tool, a full process re-execution is triggered up to three times. In contrast, when a generated tool contains internal errors, only the internal feedback loop is retried, also up to three times. Instances unresolved after the allotted attempts are marked as failures. For ToolSmith's agent models, we use `llama4` for tool gen-

[7]https://huggingface.co/mistralai/Mistral-Small-3.1-24B-Instruct-2503

	LATM (Baseline)		ToolSmith (Ours)	
Request Type	**Success**	**Failure**	**Success**	**Failure**
Read-without data fetch	39	27	**61**	**5**
Read-with data fetch	12	68	**68**	**12**
Write-without data fetch	**6**	**2**	3	5
Write-with data fetch	0	32	**21**	**11**
Total	**57**	**129**	**153**	**33**

Table 2: Comparison of ToolSmith and LATM frameworks: Success and Failure Counts by Request Type

eration and Testing & Feedback agents, and `gpt-oss`[8] for NL Test Generation and Validation agents. As in our LATM implementation, we use `mistral-small` as the sandbox ReAct agent for consistent results.

Results

Table 2 presents a comparative analysis of ToolSmith and the LATM framework, demonstrating that ToolSmith achieves a significantly higher success rate across the dataset.

LATM's performance degrades significantly in the *Read-with data fetch* and *Write-with data fetch* categories. Lacking the ability to fetch the required backend data, LATM hallucinates parameters (e.g., user IDs), resulting in a high volume of execution failures. In many cases, the generated API calls are syntactically correct but semantically invalid, as the missing contextual information leads to incorrect parameter substitution.

ToolSmith, in contrast, handles data-fetching tasks more effectively and performs reliably across most categories. Its main weakness is observed in the *Write-without data fetch* category, where it exhibits a lower success rate. We attribute these failures to (1) **tool-calling errors**, where the model generates incomplete or incorrectly formatted API requests, and (2) **LLM-specific failures**, such as empty or non-executable responses. These issues appear to be implementation-related rather than fundamental, and could likely be mitigated through improved prompt engineering, more robust tool invocation handling, or by adopting better frontier LLM.

Overall, these results indicate that ToolSmith provides a more reliable framework for autonomous API-based tool-generation, particularly for tasks requiring access to external data and services for testing and for validation.

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.

[8]https://huggingface.co/openai/gpt-oss-120b

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.