Automated Creation and Enrichment Framework for Improved Invocation of Enterprise APIs as Tools

Prerna Agarwal, Himanshu Gupta, Soujanya Soni, Rohith D Vallam, Renuka Sindhgatta, Sameep Mehta

IBM Research, India

preragar@in.ibm.com, higupta8@in.ibm.com, soujanya.soni@ibm.com, rovallam@in.ibm.com, renuka.sr@ibm.com, sameepmehta@in.ibm.com

Abstract

Recent advancements in Large Language Models (LLMs) has lead to the development of agents capable of complex reasoning and interaction with external tools. In enterprise contexts, the effective use of such tools that are often enabled by application programming interfaces (APIs), is hindered by poor documentation, complex input or output schema, and large number of operations. These challenges make tool selection difficult and reduce the accuracy of payload formation by up to 25%. We propose ACE, an automated tool creation and enrichment framework that transforms enterprise APIs into LLM-compatible tools. ACE, (i) generates enriched tool specifications with parameter descriptions and examples to improve selection and invocation accuracy, and (ii) incorporates a dynamic shortlisting mechanism that filters relevant tools at runtime, reducing prompt complexity while maintaining scalability. We validate our framework on both proprietary and open-source APIs and demonstrate its integration with agentic frameworks. To the best of our knowledge, ACE is the first end-to-end framework that automates the creation, enrichment, and dynamic selection of enterprise API tools for LLM agents.

Introduction

Recent advancements in Large Language Models (LLMs) has significantly improved their capabilities for reasoning over complex tasks and interacting with external tools, permitting the development of agents (Schneider 2025). A central paradigm in enabling actions is *tool learning*, that aims to combine the strengths of specialized tools and LLMs, thus extending their capabilities beyond text generation (Qin et al. 2024). Existing research on tool learning focuses on improving the agent's ability to decompose tasks and select tools, typically through supervised fine-tuning on tool-use datasets (Schick et al. 2023; Li et al. 2023). However, an equally critical aspect is the tool interface as an agent's performance is constrained not only by its reasoning ability but also by the semantic information presented by the tools.

In many enterprise settings, tools take the form of application programming interfaces (APIs) to perform tasks. These APIs are typically designed for use by developers, with technical specifications and parameters that are not immediately suitable for invocation by LLM-based agents. Integrating enterprise APIs as tools for LLM-based agents presents several challenges. First, converting API specifications into agent-compatible tools is often a manual, time-consuming, and error-prone process, further complicated by the need to support diverse agentic frameworks. Second, enterprise APIs often lack structured, detailed, and usable documentation. This includes tool-level descriptions and parameter semantics. Lack of details limits the language agent's ability to (i) select the appropriate tool for a given natural language query and (ii) construct valid, schema-compliant input payloads. The problem is compounded by complex, nested input/output schema, which are difficult for agent's to interpret without explicit guidance or examples. Third, APIs with large operation sets; (for example; Jira, a service management platform, has over 900 API operations[1]) result in scalability issues, as passing all tool details may exceed the LLM's context length during tool invocation.

To address these challenges, we propose an automated enrichment framework that enriches API specifications, and a tool creation framework that transforms APIs as Python tools and augments tool definitions with structured, model-friendly docstrings. The framework parses schema definitions to generate clear tool-level descriptions, parameter-level documentation with type information, and illustrative example values, including concise representations of complex structures. These example values act as few-shot demonstrations, enabling the LLM to form valid input payloads. By providing enriched docstrings with sufficient context and examples, the framework supports more reliable tool selection, accurate input construction, and scalable API integration for LLM-based agents. Additionally, a shortlisting component dynamically selects the top-k relevant tools for a given user query, reducing the candidate set and improving selection accuracy. Together, these capabilities improve tool selection, input construction, and scalability when integrating large and complex APIs into LLM-based agents.

Contributions: To summarize, we make the following contributions in our work:

(1) We propose **Automated Creation and Enrichment (ACE) framework**, end-to-end system for automated cre-

[1]https://developer.atlassian.com/cloud/jira/service-desk-ops/rest/v2/intro/

ation, enrichment, and shortlisting of enterprise API tools for LLM-based agents.

(2) We introduce an automated creation process that is framework-agnostic and an enrichment process that augments tool metadata with detailed descriptions, parameter information, and illustrative examples to support accurate tool selection and calling.

(3) We present a shortlisting mechanism that identifies the top-k most relevant tools for a given user query, enabling efficient and accurate tool usage in large tool repositories.

(4) We evaluate ACE framework on both proprietary and open-source enterprise APIs, demonstrating its effectiveness and seamless integration with existing agentic frameworks.

To the best of our knowledge, ACE framework is the first one to automate the API tool creation and enrichment in an end-to-end manner with tool shortlisting capability.

Related Work

We present existing work in the context of Tool learning: Tool Creation, Usage, and Shortlisting (Qin et al. 2024).

API Tool Creation Recent work such as LangChain's `OpenAPI Toolkit`[2] enables LLMs to interact with APIs by parsing the OpenAPI schema and allowing the agent to explore the API documentation at runtime. Rather than exposing individual endpoints as structured tools, this approach relies on intermediate reasoning over the Open API Specification (OAS). While effective for smaller APIs, enterprise APIs often include hundreds of operations and large, complex schemas, which can exceed the context capacity of smaller models and reduce scalability.

Hence, transforming an OAS into framework or protocol specific Python tools, is essential for enabling seamless integration with LLM-based agents. The proposed framework implements a transformation pipeline that parses the OAS, infers operations, input and output schemas, and generates Python tool definitions, allowing them to be imported into an agentic framework for real-time interaction.

Tool Usage Recent work has focused on improving LLMs ability to interact with external APIs and tools. Tool usage research has progressed through model-side improvements: (i) generating tool calling data and fine-tuning LLMs (Patil et al. 2023; Qin et al. 2023), and (ii) benchmarking design to evaluate tool selection, sequencing, and invocation under realistic constraints (Yao et al. 2024). However, enhancing the effectiveness of LLMs in using tools also requires improving how tools are constructed and presented to the model.

To address limitations in tool specifications, OASBuilder (Lazar et al. 2025) generates OpenAPI schemas from webpages and adds missing endpoint descriptions, parameter documentation, and constraints (e.g., formats, enum values). This improves the completeness of the OAS, which is critical for generating effective tool representations. EasyTool (Yuan et al. 2024) extracts tool descriptions and guidelines from diverse and often inconsistent documentation sources, helping LLMs understand tool purpose and usage more reliably. ToolCoder (Ding et al. 2025) takes a generative approach, producing both Python tool code and natural language descriptions from a user query.

While these methods either generate an OAS or a Python tool, they do not specifically address the challenges of transforming large, complex OAS into agent-usable tools at enterprise scale. Our work complements these efforts by proposing an automated OAS-to-tool framework that integrates creation, enrichment, and shortlisting to support accurate and scalable tool use in LLM-agent systems.

Tool Shortlisting Selecting the appropriate tool by LLM agents has become increasingly important as the number of available tools continues to grow. Early approaches rely on the LLM agent choose from static tool lists or on rule-based mappings, which are difficult to scale and prone to failure when handling ambiguous, under-specified, or multi-intent queries. To address these limitations, recent work has explored *retrieval-augmented strategies* that leverage semantic search to dynamically shortlist tools based on natural language task descriptions. For example, RAG Agents combine dense vector retrieval with LLM-based reasoning to improve tool selection precision (Karia et al. 2024). More recent methods such as MCP-Zero and ScaleMCP demonstrate that semantic retrieval pipelines can scale to thousands of tools while reducing prompt size and improving retrieval accuracy (Jiang et al. 2025; Zhang et al. 2025). These studies collectively demonstrate the effectiveness of embedding-based retrieval for dynamic tool shortlisting. Our framework explicitly utilizes the metadata of the enriched tool to shortlist tools.

Automated Creation and Enrichment (ACE) Framework

The ACE Framework for Enterprise APIs as Tools is shown in Figure 1. All the APIs present in the catalog first undergo 'Enrichment' and then are transformed into the Agentic Framework specific Python tools. The final enriched tools are stored in the tool catalog. This tool catalog is loaded into the Agentic Framework that can now be used by the agent to serve user queries. Given a user query, the relevant top-k tools are shortlisted and dynamically loaded into the Agentic Framework, if shortlisting is enabled. If not, the entire tool catalog is loaded. The agent serves the user query by selecting and invoking a tool, thereby forming the appropriate tool input and obtaining the tool output.

OAS Metadata Enrichment

This section describes a method for enhancing OAS metadata by exploiting the internal structure of OAS documents. Although OAS is intended primarily to provide a machine-readable description of RESTful interfaces, its components often encode implicit semantic information. Elements such as endpoint paths, HTTP methods, operation IDs, API titles, parameter definitions, and schema constraints can be used to infer the function of an endpoint, the nature of its inputs and outputs, and typical usage scenarios. By making these infer-

[2]https://python.langchain.com/docs/integrations/tools/openapi/

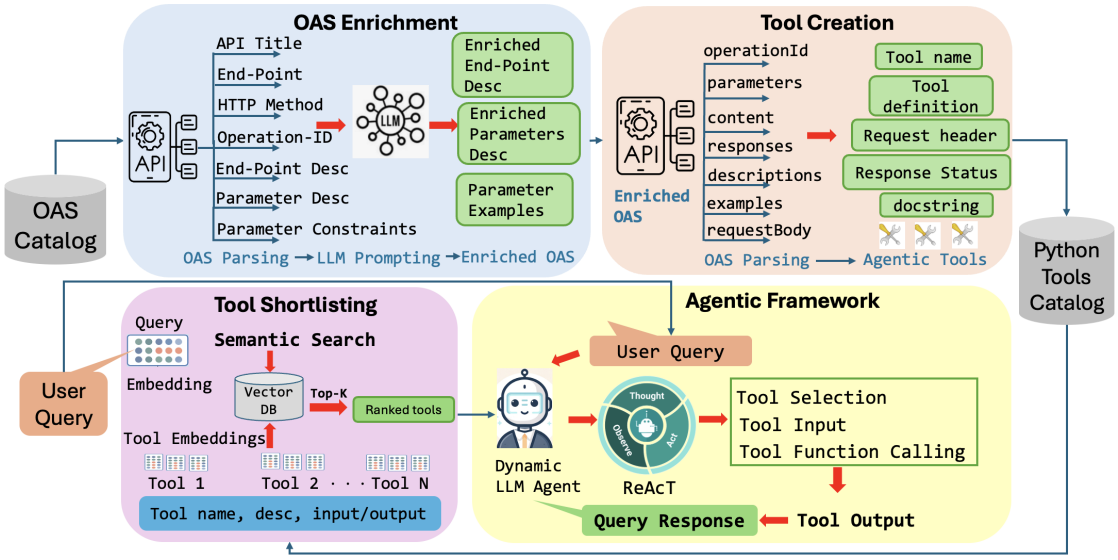

Figure 1: Automated Creation and Enrichment (ACE) Framework

ences explicit, we can produce richer metadata that supports a more effective use of APIs by automated agents.

- **API Method Description:** If an API endpoint's description is missing, inadequate, or unclear, we generate a new one based on contextual information surrounding the endpoint. Elements such as the HTTP method (GET, POST, etc.) and operation ID often reveal the method's purpose and functionality, while the parameter section outlines the expected inputs. By synthesizing these components, we can produce richer and more accurate method descriptions that clearly convey both the intent of the endpoint and the required input data.
- **Parameter Descriptions:** Similarly, when parameter descriptions are missing or vague, we analyze contextual cues such as API method name, parameter name, and operation ID to infer the parameter's meaning, intent, and purpose. Additionally, the OAS can explicitly define constraints—for example, allowed enum values or required input formats. By synthesizing this information, we generate concise and informative parameter descriptions that clearly convey the role of the parameter, any constraints, and how it should be used.
- **Parameter Examples:** In addition to generating parameter descriptions, we also use the OAS structure to derive meaningful parameter examples. We use various details - parameter name, type and description, operation ID etc. - to generate parameter examples. The updated parameter description is used for this purpose. If the parameter description mentions any parameter constraints, the generated examples will be consistent with these constraints.

To support OAS enrichment, we design dedicated LLM prompts for each of the three metadata generation tasks described above. The prompts incorporate the relevant instructions and contextual signals outlined earlier, guiding the LLMs to produce accurate and informative metadata. Given an OAS, we first parse it to extract individual endpoints and their surrounding context. We then identify the key constructs like API title, operation ID, end-point path, etc., and invoke the appropriate LLM prompt for each task with the relevant inputs.

API Tool Creation

We assume the OAS as input for generating Python tools that can be used in LLM agent frameworks. OAS (OpenAPI 2023) is the industry standard for defining RESTful API interfaces. The approach takes into consideration to be scalable for different frameworks such as LangChain, CrewAI, or protocols such as MCP (Model Context Protocol) tools (Hou et al. 2025). A tool is created by parsing different components of each path operation or endpoint in an OAS as follows:

1. *Imports and Tool Decorator:* Framework-specific libraries are imported to enable tool creation. The tool decorator is then applied to the function, marking it as callable by the LLM.
2. *Function Definition:* The `operationId` specified in the OAS is used as the function name. Path parameters, query parameters, and request body fields are extracted along with their metadata (e.g., required/optional status, data types) to define function arguments.
3. *Headers:* Request headers are derived from the OAS `content` field. When authentication is required, credentials are retrieved from the configuration file and included in the request.
4. *Docstring:* To provide the LLM with complete tool documentation, the docstring is generated from: (1) endpoint descriptions, (2) parameter descriptions, and (3) an illustrative input example with sample parameter values. The example serves as a one-shot prompt, helping the LLM generate correct payload while reducing hallucinations.
5. *API Request Construction:* Input arguments defined in the function are mapped to their corresponding positions

in the request (path, query string, or body). This mapping ensures that the final API call conforms to the endpoint specification.

6. *Response Handling:* The `responses` field in the OAS is parsed to enumerate possible status codes. At runtime, the API returns responses according to execution status, which are surfaced as tool outputs.

Tool Shortlisting

As agents interact with large, diverse toolsets, tool shortlisting becomes essential. Without it, agents needs to consider the entire toolset T, comprising hundreds of functions, making it impractical to fit into a prompt or perform real-time reasoning and execution. Shortlisting narrows this space semantically, preserving relevance while enabling scalable decisions and more accurate tool use in practice.

Given the user query q, the agent retrieves a ranked list of k tool candidates $\{T_1, T_2, \ldots, T_k\}$ from a tool catalog $\mathcal{T}$, where each tool $T_i \in \mathcal{T}$ is represented by metadata such as name, description and input/output schemas. The challenge lies in mapping the latent semantic meaning of q to appropriate tool affordances under varying degrees of ambiguity, underspecification, or multi-intent composition.

We employ a RAG-based shortlisting approach, that leverage semantic vector representations to address the mapping problem. Both the user intent q and tool descriptions $d_i \in \mathcal{D}$ are embedded into a shared vector space $\mathbb{R}^n$ using Sentence Transformer[3]. However, other sentence-level encoders such as BGE-M3, OpenAI Ada v2 can also be used here. The shortlisting then proceeds via approximate nearest neighbor (ANN) search to identify high-similarity matches under cosine or Euclidean distance. Formally,

$$T_{\text{shortlisted}} = \text{Top}_k \left(\text{sim}(E(q), E(d_i)) \; \forall \; d_i \in \mathcal{D} \right)$$

where $E(\cdot)$ is the embedding function, and sim is the similarity metric. We expect the enriched metadata to provide similar and relevant tools while shortlisting.

Tool Execution

We use LangChain as the base agentic framework and the standard ReAct agent for experimentation (Yao et al. 2023). The ReAct pattern combines reasoning and action in an iterative loop: the agent reasons about the task, invokes a tool, observes the outcome, and repeats as needed. Our framework remains pluggable with other agentic frameworks and agent patterns.

Each interaction begins with a natural language user query. The agentic environment is then initialized with the top-k tools selected by the shortlister, or with all tools in the catalog if shortlisting is disabled. The agent chooses the appropriate tool from the available set, executes it with the required input, and retrieves the tool output.

Experiments

We enrich the API tools in three different variants and benchmark the below against the defined metrics.

[3]https://huggingface.co/sentence-transformers/paraphrase-MiniLM-L3-v2

Dataset	#APIs	#Input Params	#NL Utterances
Salesloft	42	0-16	130
Kubernetes	86	0-7	164

Table 1: Dataset Statistics

- **No Enrich:** Here the original tool description is used added in the tool docstring.
- **Enrich-1:** Here only tool description is enriched and added in the tool docstring.
- **Enrich-2:** Here the tool description and parameter descriptions are enriched and added in the tool docstring.
- **Enrich-3:** Here the generated parameter examples are also added along with enriched tool description and parameter descriptions in the tool docstring.

We evaluate the proposed ACE framework by answering the following Research Questions (RQs):

- **RQ1:** How does the metadata created from enriched OAS affect the tool calling performance?
- **RQ2:** How are different LLMs in the agentic framework impacted by the metadata for tool selection and calling?
- **RQ3:** How does the metadata created from enriched OAS affect the tool shortlisting?

Datasets and Models

APIs: We evaluate ACE framework on the following proprietary (Salesloft) and open-sourced (Kubernetes) enterprise APIs:

- **Salesloft[4]:** These are enterprise sales engagement APIs with different levels of complexity in terms of input parameters, response output parameters.
- **Kubernetes[5]:** APIs enables query and manipulation of the state of objects in Kubernetes such as Pods, Namespaces, ConfigMaps, and Events. Comparatively, this is a more complex dataset with various tools being similar in names and having related functionalities. While number of parameters are less, the request body is complex with nested schema.

Dataset statistics are shown in Table 1.

User Query (NL Utterance): We simulate complex user queries by generating NL utterances for each dataset using an LLM. Each NL utterance is then manually corrected to resemble real-world enterprise user query.

Models: We use different open-source LLMs of varying size to evaluate ACE framework i.e., `granite3.3-8b-instruct` (Granite-8B)[6], `llama3.3-70b-instruct` (Llama-70B)[7], `llama3.1-405b-instruct` (Llama-405B)[8]. We use `mistral-large`[9] to generate NL utterances. We do not evaluate using close-sourced models such as GPT-4 due to the costs and deployment aspects associated with them.

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

[5]https://kubernetes.io/docs/concepts/overview/kubernetes-api

[6]https://huggingface.co/ibm-granite/granite-3.3-8b-instruct

[7]https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct

[8]https://huggingface.co/meta-llama/Llama-3.1-405B-Instruct

[9]https://huggingface.co/mistralai/Mistral-Large-Instruct-2407

Metrics

Following metrics are used to evaluate ACE performance:

- **Tool Selection Accuracy (S):** It is computed as the fraction of NL utterances for which the correct tool was invoked by the agent.
- **Tool Calling Input Errors:** It is computed as the fraction of tool input parameters that were not formed correctly by the LLM. We report this metric only for the NL utterances for which the right tool was selected by the agent. This metric is computed in 3 dimensions: (a) **type mismatch (T)**: fraction of tool input parameters where the type of input parameter is incorrect - the tool definition expects a type but the LLM passes a different type; (b) **missing parameters (M)**: fraction of tool input parameters where the input parameter is present in the NL utterance but missing from the tool input; (c) **incorrect parameters (I)**: fraction of tool input parameters that are incorrect/hallucinated by the LLM, not appearing in the NL utterance.

Results and Discussion

We discuss the results on both the datasets with ablations on enrichment and shortlisting.

RQ1: Impact of Enrichment on Tool Calling

Table 2 present the results of the enrichment experiments on the two datasets.

Tool Selection Accuracy (S): For the Salesloft dataset, tool selection accuracy is already saturated (close to 100%) across all enrichment variants, leaving little room for improvement. For Kubernetes, however, performance varies significantly across models and enrichments. With Enrich-1, Llama-70B improves by +2.9 points, while Llama-405B shows only marginal gains. For the Granite model, the no-enrichment variant performs best, outperforming enriched variants by about 5%, suggesting that smaller models may be confused by additional verbosity. There is an approximate 10% difference in average accuracy between Salesloft and Kubernetes. This gap is primarily due to the fact that, as the number of tools in the catalog increases, the LLMs' ability to disambiguate between them tends to decrease.

Tool Input Calling Errors: Enrichment generally improves parameter correctness, especially with Enrich-3.

Type mismatch errors (T) for both Salesloft and Kubernetes datasets, are almost zero for Granite-8B across all enrichments. For Llama-70B, significant reductions occur with Enrich-2 and Enrich-3, with Enrich-3 outperforming Enrich-2. In contrast, Llama-405B does not leverage extra metadata effectively, as it often considers all parameters of string type leading to persisting type mismatches.

Missing parameter errors (M) are reduced with Enrich-2 and Enrich-3, as parameter descriptions help, and Enrich-3 further lowers errors by including request body examples. For Llama-405B, type errors often propagate into missing parameter errors. Smaller models such as Granite-8B struggle on Kubernetes, where verbosity and similar parameter names create confusion.

Incorrect parameter errors (I) follow a similar trend. Enrich-3 yields significant improvements for Granite-8B and Llama-70B on Salesloft, and also benefits Llama-70B significantly on Kubernetes.

Summary (RQ1): Effective enrichment is stage-dependent — concise metadata helps agents select tools efficiently, while detailed metadata ensures reliability in forming the right input for tool calling.

RQ2: Impact of Enrichment on Agent LLM

From our experiments on the three LLM models, we can make the following observations:

Small Size Model – Granite-8B: On Salesloft, Enrich-3 yields clear improvements in tool input calling for missing and incorrect parameters. However, in Kubernetes, tool selection degrades under Enrich-3, and input errors show no improvement, indicating the model's sensitivity to metadata verbosity with complex schema and a large number of tools.

Medium Size Model – Llama-70B: On Salesloft, Enrich-3 consistently reduces all tool input errors. In Kubernetes, missing parameters stabilize and incorrect parameters reduce (–7.1 points). We observe that the model often formats inputs incorrectly despite identifying the right parameters while considering the enrichment information. For example, instead of passing {`'dryRun':"All", 'fieldValidation':"Ignore"`} as strings, it produces dictionaries such as {`'dryRun': {'type': 'string', 'value': 'All'}, ...`}.

Large Size Model – Llama-405B: Tool selection accuracy is near-saturated on Salesloft and Kubernetes, so enrichment provides little benefit. It worsens missing or type errors as the model overgeneralizes inputs as strings, even for numbers, booleans, or objects. This leads to unreliable tool calls, such as representing a request body as a string: {`'requestBody': "{'apiVersion': 'v1', 'data': {'mute': 'True'}"`}}.

Summary (RQ2): Results indicate that enrichment improvements are model-dependent; most beneficial for smaller models, but cannot overcome inherent model limitations such as over-generalization of types or format.

RQ3: Impact of Enrichment on Tool Shortlisting

Our experiments across two distinct domains: *Salesloft* (a business-oriented environment) and *Kubernetes* (a technical, API-driven setting) (see Table 3), enrichment helps to achieve better tool shortlist performance at a smaller k. In the Kubernetes dataset, enrichment increases Top-3 accuracy by +10% points. A similar improvement is seen in the Top-5. For larger k, the relevant tool is picked up irrespective of enrichment. A similar pattern holds for Salesloft, where the baseline accuracy is already high, but enrichment provides modest improvements at lower k. Hence, enrichment is most valuable at smaller k, where it helps models overcome limitations of context and tool set size, enabling better tool selection.

Dataset	Models	No Enrich				Enrich-1				Enrich-2				Enrich-3			
		S%	T%	M%	I%	S%	T%	M%	I%	S%	T%	M%	I%	S%	T%	M%	I%
Salesloft	Granite-8B	96.1	0.15	6.1	3.4	97.0	0.0	15.0	8.7	96.1	0.13	4.8	3.0	96.1	0.0	0.8	1.5
	Llama-70B	100	23.4	6.6	4.0	100	23.5	9.1	5.6	97.7	15.9	5.9	3.3	99.2	16.9	0.1	0.55
	Llama-405B	100	21.7	43.3	0.4	100	24.2	36.3	0.13	100	20.72	44.0	0.14	100	25.2	41.3	0.26
Kubernetes	Granite-8B	71.8	0.3	9.4	5.3	59.2	0.0	12.3	5.3	65.3	0.0	9.2	1.7	65.3	0.0	12.2	8.0
	Llama-70B	90.3	11.4	0.95	19.1	93.2	13.9	0.9	18.9	92.7	7.3	0.7	13.4	91.2	3.5	0.7	12.0
	Llama-405B	91.3	12.3	2.5	11.3	89.5	15.3	3.0	15.3	90.8	14.9	2.2	12.2	92.0	15.1	1.3	13.7

Table 2: Results on Kubernetes and Salesloft Datasets

Dataset	Method	Top 3	Top 5	Top 10	Top 15	Top 20
Salesloft	No Enrich	89.23	91.54	96.15	96.15	96.15
	Enrich-1	90.00	94.62	96.15	96.15	96.15
	Enrich-2	90.00	93.85	95.38	95.38	96.92
	Enrich-3	88.46	93.85	93.85	93.85	93.85
Kubernetes	No Enrich	71.34	82.32	90.85	94.51	96.95
	Enrich-1	77.44	87.20	92.68	96.34	98.17
	Enrich-2	81.71	86.59	92.68	95.73	98.17
	Enrich-3	80.49	86.59	91.46	95.73	98.17

Table 3: Accuracy (in %) of Tool Shortlisting

As smaller models are constrained by their context length, tool shortlisting helps LLM models overcome this limitation. To investigate this, we conducted an experiment with Granite-8B (small model) where, the agent dynamically obtains top-10 shortlisted tools from Enrich-3 variant. The tool selection with Enrich-3 is 66% despite having only 10 tools provided to the agent. Note that, the performance with Enrich-3 is similar to that achieved without tool shortlisting (see Table 2). Hence, tool shortlisting with enrichment effectively enables LLM agents to handle large tool catalogs.

Summary (RQ3): Enrichment provides a boost in tool shortlisting accuracy at smaller k. With increase in k, the tool shortlisting accuracy comes closer towards 100%.

Deployment and Impact

Our framework is currently deployed as part of the alpha release of IBM Watsonx Orchestrate Agent Development Kit (ADK)[10], specifically designed for the agent builder persona. The ADK provides a tooling environment where builders can configure, deploy, and manage agents and tools within Watsonx Orchestrate. The tool builders can import OAS and Python functions as tools. The current feature would support building domain specific agents with REST APIs for a wide range of domains, including IT, human resources, finance, procurement, productivity, and sales. Our automated enrichment feature standardizes generation of tool metadata and reduces errors caused by poor tool definitions. We conducted internal experiments using an IT ticketing domain containing 6 agents and 46 tools developed by in-house tool builders. In this setting, tool specifications and their descriptions were carefully authored and tested manually by developers. We evaluated 3 scenarios: (i) tools with both method and parameter descriptions written by the developer, (ii) tools where the description was identical to the tool name and parameter descriptions were absent—representing a case of minimal or poor metadata, and (iii) tools where the tool name and parameter descriptions were automatically generated by the ACE framework. For each scenario, we executed 600 single-turn user utterances through the agent. The results, shown in Figure 2, report the fraction of utterances in which the correct tool was selected and successfully invoked. We observe a 27% improvement in tool selection and invocation accuracy when metadata is enriched by ACE compared to the minimal metadata condition. Furthermore, ACE-enriched metadata achieves performance comparable to human-authored metadata.

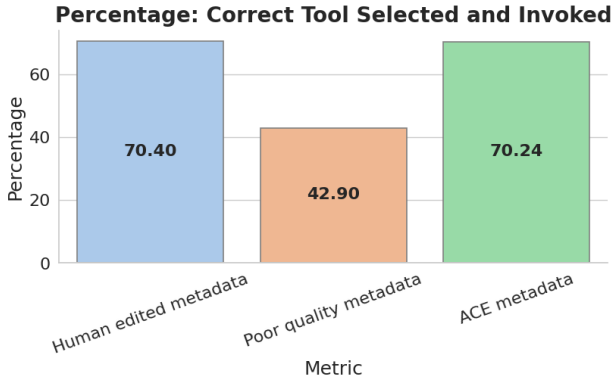

Figure 2: Evaluation on ADK Tools for IT Service Domain

Conclusion and Future Work

In this work, we propose Automated Creation and Enrichment (ACE) framework that automates creation, enrichment and shortlisting of API tools. We examined tool learning from the perspective of metadata enrichment in OAS-derived enterprise tools, showing that enhanced metadata improves LLM-based agents' ability to shortlist, select, and invoke tools effectively. Our findings highlight the importance of semantically rich tool specifications for reliable tool calling. As future work, we plan to extend this study across additional domains such as HR, finance, and procurement to assess the generality of our approach.

[10]https://developer.watson-orchestrate.ibm.com/

References

Ding, H.; Tao, S.; Pang, L.; Wei, Z.; Gao, J.; Ding, B.; Shen, H.; and Cheng, X. 2025. ToolCoder: A Systematic Code-Empowered Tool Learning Framework for Large Language Models. arXiv:2502.11404.

Hou, X.; Zhao, Y.; Wang, S.; and Wang, H. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. arXiv:2503.23278.

Jiang, R. Y. W. J., Z.; et al. 2025. MCP-Zero: Token-Efficient Tool Retrieval for Agentic LLMs. *arXiv preprint arXiv:2506.01056*.

Karia, P. R. M. A., R.; et al. 2024. RAG Agents: Tool Selection via Retrieval-Augmented LLM Reasoning. *arXiv preprint arXiv:2401.12345*.

Lazar, K.; Vetzler, M.; Kate, K.; Tsay, J.; Boaz, D.; Gupta, H.; Shinnar, A.; Vallam, R. D.; Amid, D.; Goldbraich, E.; Uziel, G.; Laredo, J.; and Tavor, A. A. 2025. Generating OpanAPI Specifications from Online API Documentation with Large Language Models. In *ACL*.

Li, M.; Zhao, Y.; Yu, B.; Song, F.; Li, H.; Yu, H.; Li, Z.; Huang, F.; and Li, Y. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In Bouamor, H.; Pino, J.; and Bali, K., eds., *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 3102–3116. Singapore: Association for Computational Linguistics.

OpenAPI. 2023. OpenAPI standard.

Patil, S. G.; Zhang, T.; Wang, X.; and Gonzalez, J. E. 2023. Gorilla: Large Language Model Connected with Massive APIs. arXiv:2305.15334.

Qin, Y.; Hu, S.; Lin, Y.; Chen, W.; Ding, N.; Cui, G.; Zeng, Z.; Huang, Y.; Xiao, C.; Han, C.; Fung, Y. R.; Su, Y.; Wang, H.; Qian, C.; Tian, R.; Zhu, K.; Liang, S.; Shen, X.; Xu, B.; Zhang, Z.; Ye, Y.; Li, B.; Tang, Z.; Yi, J.; Zhu, Y.; Dai, Z.; Yan, L.; Cong, X.; Lu, Y.; Zhao, W.; Huang, Y.; Yan, J.; Han, X.; Sun, X.; Li, D.; Phang, J.; Yang, C.; Wu, T.; Ji, H.; Liu, Z.; and Sun, M. 2024. Tool Learning with Foundation Models. arXiv:2304.08354.

Qin, Y.; Liang, S.; Ye, Y.; Zhu, K.; Yan, L.; Lu, Y.; Lin, Y.; Cong, X.; Tang, X.; Qian, B.; Zhao, S.; Hong, L.; Tian, R.; Xie, R.; Zhou, J.; Gerstein, M.; Li, D.; Liu, Z.; and Sun, M. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:2307.16789.

Schick, T.; Dwivedi-Yu, J.; Dessí, 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 *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23. Red Hook, NY, USA: Curran Associates Inc.

Schneider, J. 2025. Generative to Agentic AI: Survey, Conceptualization, and Challenges. arXiv:2504.18875.

Yao, S.; Shinn, N.; Razavi, P.; and Narasimhan, K. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. arXiv:2406.12045.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629.

Yuan, S.; Song, K.; Chen, J.; Tan, X.; Shen, Y.; Kan, R.; Li, D.; and Yang, D. 2024. EASYTOOL: Enhancing LLM-based Agents with Concise Tool Instruction. arXiv:2401.06201.

Zhang, L. Q. S. R., H.; et al. 2025. ScaleMCP: Scalable Tool Retrieval for LLM Agents Using Semantic Indexing. *arXiv preprint arXiv:2505.06416*.