Doc2Agent: Scalable Generation of Tool-Using Agents from API Documentation

Xinyi Ni, Haonan Jian, Qiuyang Wang, Vedanshi Chetan Shah & Pengyu Hong
Michtom School of Computer Science
Brandeis University
Waltham, MA 02453, USA
{xinyini, hongpeng}@brandeis.edu

Abstract

REST APIs play important roles in enriching the action space of web agents, yet most API-based agents rely on curated and uniform toolsets that do not reflect the complexity of real-world APIs. Building tool-using agents for arbitrary domains remains a major challenge, as it requires reading unstructured API documentation, testing APIs and inferring correct parameters. We propose Doc2Agent, a scalable pipeline to build agents that can call Python-based tools generated from API documentation. Doc2Agent generates executable tools from API documentations and iteratively refines them using a code agent. We evaluate our approach on real-world APIs, WebArena APIs, and research APIs, producing validated tools. We achieved a 55% relative performance improvement with 90% lower cost compared to direct API calling on WebArena benchmark. A domain-specific agent built for glycomaterial science further demonstrates the pipeline's adaptability to complex, knowledge-rich tasks. Doc2Agent offers a generalizable solution for building tool agents from unstructured API documentation at scale.

1 Introduction

Tool agents (Ferrag et al. (2025), Yehudai et al. (2025), Wang et al. (2024a), Qu et al. (2025)), built on LLMs, aim at interacting autonomously with existing software or web services. One important tool source to enrich the capability of tool agents is from existing **REST API services** (Barry, 2003). REST APIs are a widely adopted standard for communication between clients and servers, which expose programmic access to web-based services through structured endpoints. API agents (Qin et al. (2023), Du et al. (2024), Song et al. (2025b)) use REST APIs by sending HTTP requests with appropriate parameters to access, manipulate, or retrieve structured data from web services. The method to connect REST APIs to these agents varies. For example, ToolLlama (Qin et al., 2023) created a scraper to fetch 16,000 APIs from RapidAPI which provides uniformed API specifications. API-based agent (Song et al., 2025b), added API documentation as part of input prompt to the agent. On the other hand, API-calling benchmarks such as WebArena (Zhou et al., 2024), Appworld (Trivedi et al., 2024), ComplexFuncBench (Zhong et al., 2025) provides ready-to-use APIs in an uniform format, usually a predefined JSON schema. While existing benchmarks and agent frameworks typically offer APIs in clean, uniform formats to facilitate convenient function calling, this setup overlooks the real-world challenges posed by inconsistent and low-quality API documentation. The assumption that APIs can be seamlessly used by AI agents does not hold in practice. Beyond well-maintained commercial platforms such as RapidAPI and Postman, many APIs lack comprehensive documentation and often do not follow standardized schemas. Even when schemas are available, they may be incomplete or omit critical information, making it difficult to automate tool generation reliably. Furthermore, API services and their documentation are frequently outdated, requiring extensive validation and refinement to ensure usability. These challenges reveal a significant gap between tool agents and real-world API services.

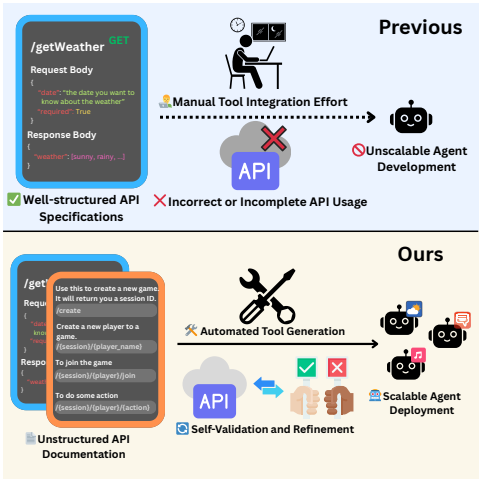


Figure 1: **Bridging the Gap Between Real-World APIs and AI Agents** (Top) Conventional agent development relies on high-quality APIs and manual integration, leading to scalability issues. (Bottom) Our approach automates tool generation from natural language API documentation, enabling self-validation, refinement, and seamless deployment of AI agents.

We argue that it is essential to ***develop an automated pipeline for scalable agent generation with AI-ready tools from any domain-specific REST APIs*** (Figure 1). Such a pipeline enables rapid tool creation directly from natural language API documentation, enhancing development efficiency for AI practitioners. For end users, API agents provide an intuitive natural language interface to interact with APIs, lowering the barrier to entry. For the AI agent community, this automation facilitates seamless integration of diverse APIs, expanding agent capabilities and supporting the creation of tool-accessible benchmarks. In this work, we focus on constructing Python-based tools that can be seamlessly generated and executed by an action parser (Wang et al., 2024b). These tools are natively compatible with popular agentic frameworks such as LangGraph, LlamaIndex, and AutoGen. Compared to approaches that rely on direct API requests (Song et al., 2025b), encapsulating functionality as Python functions improves token efficiency and allows code agents to concentrate on task logic and parameter usage, rather than low-level API construction. Another important application of agent-generation pipeline lies in the development of scientific agents (Wang et al., 2024a). Agents are increasingly demonstrating their versatility across research domains, including science (Baek et al. (2024); Chen et al. (2025)), healthcare (Abbasian et al. (2024)), and finance (Li et al. (2023)). These agents are designed to tackle domain-specific challenges—for example, ChemCrow (Bran et al., 2023), a chemistry-focused research agent, uses expert-curated tools to perform tasks such as data retrieval, analysis, and prediction. However, its tools were manually implemented by researchers. Many research services are already accessible via REST APIs, including dataset repositories (Sayers et al. (2021), Rose et al. (2021)) and scientific applications (Dorst & Widmalm (2023), Woods Group (2025)). Our pipeline has the potential to automate tool generation for such APIs, accelerating the deployment of domain-specific research agents.

Research APIs are often less actively maintained than commercial APIs due to limited developer resources, resulting in lower documentation quality and inconsistent schema design. Furthermore, research agents typically need to query across multiple datasets, each using distinct entries, representations, and semantic conventions. These variations pose significant challenges for using correct input parameter, especially in the absence of domain-specific prior knowledge.

To address the limitations in tool agent development, we propose an open-source pipeline, **Doc2Agent**[1](Figure 3), which **(1)** Autonomous generation of Python-based tools from any REST API documentation written in natural language, **(2)** Automatic evaluation of tool

[1]Code available at https://anonymous.4open.science/r/20cvj3g0jsdffodob032-a0a/README.md

functionality to verify operability and alignment with API descriptions, **(3)** Iterative tool refinement to improve reliability and usability, infer parameter values without requiring prior domain knowledge. **(4)** Deploy toolkits to MCP server for seamless deployment into any agentic framework.

To demonstrate the effectiveness of our approach, we collected 167 real-world API documentation pages comprising 744 publicly accessible endpoints, 174 API documentations for WebArena benchmark and 16 websites for glycoscience research (Taniguchi et al., 2015). Differ from prior benchmark datasets, these API docs have diverse and inconsistent documentation formats (1, bottom) rather than uniform schemas such as OpenAPI specification. Our results can be summarized as follows: (1) Doc2Agent generated verifiable tools for 59.5% of real-world APIs and 81.5% of research APIs, with refinement improving tool pass rates by 47.6%; (2) compared to direct API calling (Song et al., 2025b), our tool-based approach achieved a 55.1% performance boost at only 10% of the cost per task; (3) we built a glycomaterial research agent using only auto-generated tools, demonstrating Doc2Agent's potential for scientific applications without domain-specific knowledge.

2 Doc2Agent Pipeline

Existing API-calling approaches (Song et al., 2025b) allow LLMs to read API documentation and generate URLs to invoke REST APIs directly. However, this method is cumbersome: lengthy documentation consumes context window space, wastes tokens, and can cause the model to lose focus in multi-turn interactions (Liu et al., 2023; Laban et al., 2025). To address these challenges, we provide agents with generated, Python-based tools in place of raw REST APIs, enabling more efficient and intuitive function calls (Figure 2). Our workflow can be divided into 4 main steps: tool generation, tool validation, tool improvement, and agent deployment(Figure 3). We take a set of API documentations in HTML/Markdown format as our input. Each API documentation has multiple endpoints. The output of our pipeline is an agent that can use all API documentations as its tools automatically.

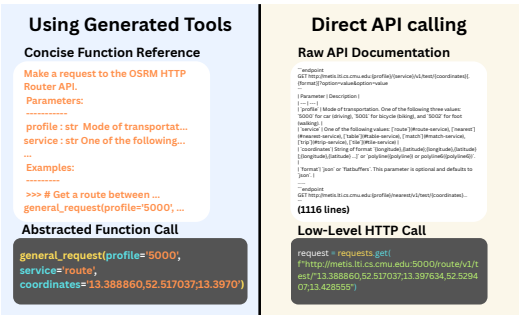


Figure 2: Comparison of function-based API using and direct API calling

2.1 Tool Generation

In tool generation, we produce Python functions with customizable parameters designed for seamless agent use. We adopt two strategies for generating agent-friendly tools:

Direct Tool Generation When API documentation is simple and well structured, we leverage LLMs' structured information extraction capabilities (Dagdelen et al., 2024; GPTs, 2024) to convert essential API details into standardized JSON format (Appendix F). These JSON files are then transformed into Python functions that wrap the API calls, incorporating basic input validation and default value autofill. This encapsulation simplifies API interaction, reduces redundancy, and streamlines agent development.

Target-Oriented Tool Generation For complex or highly flexible APIs, such as general-purpose search endpoints with multiple filters, direct wrapping may be ineffective, as agents struggle to interpret input requirements from the context. In these cases, we guide the generator to first produce simplified function "fingerprints," which define a specific use case along with expected inputs and outputs. The full function is then generated based on the fingerprint. This strategy produces tools that are better aligned with downstream tasks and more accessible to agents than raw API interfaces.

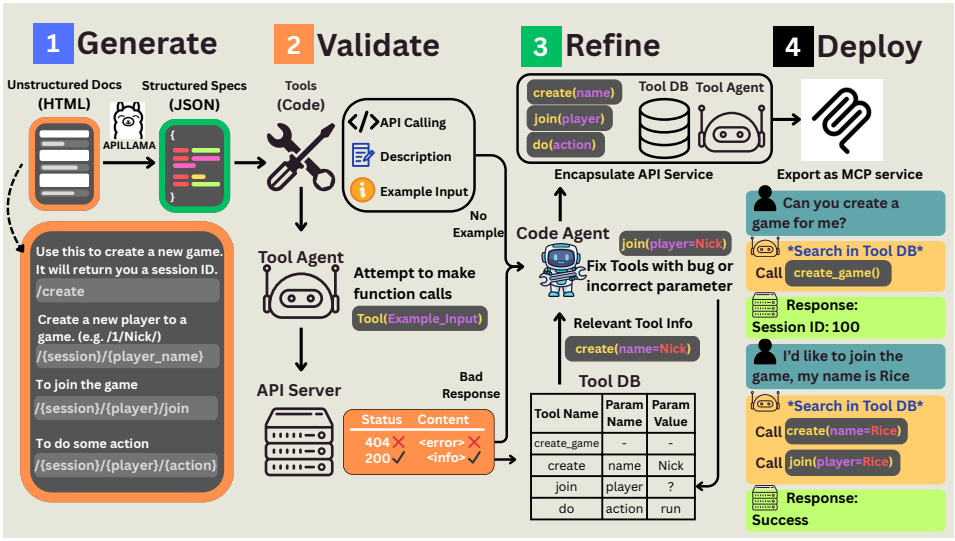


Figure 3: Overview of Doc2Agent: Automated pipeline for generating AI agents from API docs. The API docs in free-text are used to generate tools. The tools are validated and refined. A tool agent equipped with the generated tools is deployed as an MCP service.

The test of generated APIs, especially in a real-life environment, may cause unexpected circumstanses, like changing password and deleting items (Section 7). We allow users to specify REST API methods allowed for tools to minimize the risk of automatic testing.

2.2 Tool Validation

The quality of API documentation significantly impacts the performance of API-based agents. However, real-world API documentation is often incomplete or unreliable, making automatic evaluation of generated tools essential.

REST APIs require valid input parameters to function, tools cannot be evaluated without example values. Malformed inputs will fail even if the API itself is functional. When documentation provides example parameters, we assume they produce valid responses and use these tools for evaluation. For tools lacking examples, we infer parameter values in a later stage. To validate a tool, we call it using the provided example inputs and compare the actual API response against an expected output generated by a language model, conditioned on the tool's description. A tool is considered verified if the response aligns with the model-predicted expectation. For tools derived from real-world APIs, our validation results show strong agreement with human judgment. We further analyze failed tools using status codes, response content, and runtime exceptions. Most failures stem from incorrect or incomplete parameter values, motivating our method for automatic generation of high-quality parameter inputs(Appendix C).

2.3 Tool Refinement

Parameter Value Inference For APIs lacking proper parameter values, large language models often struggle to generate valid inputs. Our experiments (Section 4.3) confirm that LLMs are unreliable at guessing parameter values. This aligns with recent findings (Song et al., 2025a), which highlight the difficulty LLMs face in producing complete and accurate API inputs. We construct an automatically generated parameter database to support parameter value inference. We leverage two primary sources of information: (1) parameter examples from other API documentations, particularly from the same domain, and (2) JSON responses from previously validated API tools, which contain rich domain-specific key-value pairs. The latter captures implicit inter-API dependencies, where outputs of one service often correspond to inputs of another—conceptually forming a service de-

pendency graph (Bushong et al. (2021); Lercher et al. (2024)). These dependencies allow us to repurpose response data to infer plausible parameter values. The database is constructed without requiring external domain knowledge. All discovered parameter values are stored in a vector database to enable efficient semantic similarity search (Han et al., 2023). Parameter examples are indexed by both name and description. When encountering an unknown parameter γ, we embed its name and description, retrieve semantically similar entries from the database, and rank them based on tool and parameter similarity. We then sample up to 10 candidate values for downstream usage.

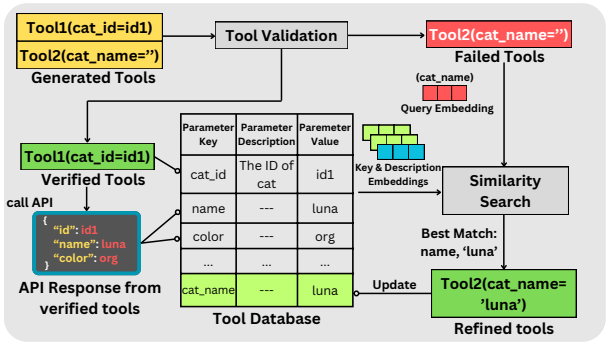


Figure 4: A parameter database is automatically constructed using validated tools, enabling parameter value inference based on the semantic similarity of parameter/response keys.

Tool Fixing We fix the tools that either have no parameter values or produced incorrect responses. For each failed tool, we provide a code agent with the corresponding API documentation, error information, and the original Python code. If the failure stems from missing parameters, we additionally supply the agent with candidate parameter values.

The agent generates a revised version of the tool, which is then re-evaluated through the validation process. If the updated tool passes validation, we overwrite its code and documentation. When a new parameter value is used, it is recorded as an example and added to the parameter database. This **refinement-validation loop** is repeated for multiple rounds to maximize tool recovery and quality.

2.4 Deployment

Once high-quality tools are obtained, they can be readily integrated into tool-using agents such as CodeAct (Wang et al., 2024b). Since the tools are implemented in Python, they remain compatible with a wide range of agent architectures, including those that support advanced capabilities like planning (Stein et al., 2025), reasoning(Wei et al. (2023), Kojima et al. (2023)), long-term memory (Du et al., 2025), and tool retrieval (Yehudai et al., 2025). This synergy between agents and tools significantly enhances task-solving performance.

To support scalable and standardized integration, tools can be deployed via an MCP (Model-Context-Protocol) server (Hou et al., 2025), which provides a unified protocol for accessing tools, managing data, and orchestrating workflows. Tools can operate independently through a host-side agent or be dynamically invoked by clients, enabling flexible and modular agent-tool interactions. We deploy the API services using *FastAPI MCP* (Abramov, 2025). As an alternative deployment path, tools can also be exported as OpenAPI specifications (Swagger, 2024), enabling seamless integration with both enterprise systems (e.g., GPTs) and open-source agent frameworks such as CrewAI, LangGraph, and AutoGen.

3 Experiment

3.1 Data Collection

Previous REST API-based benchmarks (Zhou et al. (2024), Trivedi et al. (2024), Zhong et al. (2025)) typically assume ideal conditions by either providing agent-ready, Python-based tools or clean, standardized API specifications. In contrast, our work emphasizes **robustness and scalability** in the face of real-world API documentation, which is often unstructured,

inconsistent, and heterogeneous in format and quality. To evaluate the effectiveness of our pipeline under these realistic conditions, we curated data from three diverse sources.

Real-world API documentations We collect 167 API documentation pages from APIList.com, comprising a total of 744 endpoints. Upon analysis, we find that only 24 of these documentations were of high quality. The majority were semi-structured and difficult to interpret at first glance, often requiring careful manual inspection to infer correct parameter values. Examples and selection criteria are provided in Appendix A.

WebArena (Zhou et al., 2024) WebArena is a reproducible web environment designed for developing autonomous agents to perform complex web-based tasks. While originally intended for web browsing agents, its environment also supports API interactions, with API documentation quality varying across domains (Song et al., 2025b), which provides a great environment for testing robust tool generation approach. We utilize the available API documentation from five WebArena environments: GitLab, Open Street Map(Map), One Stop Shop(Shopping-Customer), E-Commerce Content Management(Shopping-Admin, CMS) and Cross site tasks(Multi). We exclude the Reddit task, because WebArena Reddit sandbox doesn't provide API documentations.

Glycoscience APIs To evaluate our pipeline's ability to generate domain-specific research agents, we target glycoscience (Taniguchi et al., 2015), with complex APIs and inconsistent data standards. We collect REST API documentation from major databases, including **GlycoData** (Wang et al., 2025a), **GlyGen** (York et al., 2020), **GlyTouCan** (Tiemeyer et al., 2017), **KEGG GLYCAN** (Hashimoto et al., 2006), **Glycosmos** (Yamada et al., 2020), **Glyconnect** (Alocci et al., 2018), **The O-GlcNAc Database** (Wulff-Fuentes et al., 2021), **GLYCAM** (Woods Group, 2025), **Protein API** (Nightingale et al., 2017), **PubChem** (Kim et al., 2016) and **UniLectin** (Imberty et al., 2021).. These APIs vary widely in quality and structure, and present additional challenges such as inconsistent identifiers, non-standard representations, and cross-database linking (Appendix D.2).

3.2 Implementation

Generator We use `GPT-4o` in structured mode for direct tool generation and `Claude 3.7 Sonnet` for target-oriented generation. In the direct tool generation, one tool is generated per extracted API. In the target-oriented generation, up to 10 task-specific tools are generated per API documentation. For WebArena, we use direct tool generation for Gitlab and Map docs; and task-oriented generation for Shopping and Admin docs.

Code Agent for Validation and Refinement We employ `Claude 3.7 Sonnet` as the code agent for tool testing and refinement. It is capable of generating well-structured Python code along with clear documentation, enabling seamless integration with downstream agents.

Tool Agent To evaluate the impact of high-quality, agent-usable tools, we conduct a comparative study based on the design of API agents from Song et al. (2025b). Our implementation builds upon CodeAct Wang et al. (2024b), powered by GPT-4o, with basic code execution capabilities and minimal tool retrieval support. The key difference lies in tool usage: our API agent utilizes refined, Python-based API tools, whereas prior approaches rely solely on raw API documentation as input prompts. For more advanced applications, stronger tool agents such as ReTool Feng et al. (2025) can be used to further enhance tool utilization.

4 Result

4.1 Generation of Python-based API Tools from Documentation

Our validator enables self-evaluation of the generated tools. An example of the generated tool is provided in Appendix F. We allow up to three refinement rounds per tool. Results are summarized in Table 1. Despite the complexity of real-world API documentation, we successfully produce 443 validated tools. A frequent cause of failure is the REST API service no longer operational. In the WebArena benchmark, Wiki and Map environments have relatively simple API designs, and we apply direct tool generation. Some Map APIs fail due to

	Real-life API	WebArena					Glycoscience API
		Wiki	Map	Shopping-Admin(CMS)	Shopping-Customer	GitLab	
Endpoints	744	26	53	555	108	988	131
Validated Tools	443	21	28	159	35	213	70

Table 1: Generated Agent-Ready Python-based Tools with Doc2Agent from Raw API Docs

Method	Gitlab	Shopping	CMS	Map	Multi	Avg.
Vanilla WebArena Evaluation						
WebArena Baseline[b](Zhou et al., 2024)	15.0	13.9	10.4	15.6	(8.3)	(12.3)
SteP[b](Sodhi et al., 2024)	32.2	50.8	23.6	31.2	(10.4)	(36.5)
SkillWeaver[b](Zheng et al., 2025)	22.2	27.2	25.8	33.9	-	(29.8)
API-based agent[t](Song et al., 2025b)	43.9	25.1	20.3	45.4	(8.3)	(29.2)
Hybrid agent[bt](Song et al., 2025b)	44.4	25.7	41.2	45.9	(16.7)	(38.9)
API-Specified Evaluation						
Hybrid agent[bt](Song et al., 2025b)	47.2	29.4	**45.5**	50.5	**44.0**	42.2
Doc2Agent[t](Ours)	**48.9**	**39.6**	39.7	**58.7**	**44.4**	**45.3**
Δ vs Direct API Calling	↑11.4%	↑57.8%	↑95.6%	↑29.3%	-	↑55.1%

Table 2: Comparison result of different Methods on WebArena. We exclude Reddit task due to no API documentation. Numbers in parentheses include Reddit tasks. Δ represents the estimated relative improvement percentage. [b] Browser-based Agent [t] API-based Agent

authentication errors, which is not required in WebArena task. In contrast, Shopping-Admin, Shopping-Customer and GitLab involve complex API usage with numerous endpoints. For these, we use both direct tool generation and target-oriented tool generation to produce more task-aligned, agent-friendly tools. For Glycoscience APIs, most tools pass validation without refinement, as they primarily consist of information retrieval endpoints.

4.2 Tool agent performance on WebArena

To compare our tool-using agent with direct API-based agent (Song et al., 2025b), we keep most components of the CodeAct agent unchanged, including tool retrieval, planning, and memory. As shown in Table 2, our method outperforms direct API calling with a 55% relative increase in average success rate. Notably, Doc2Agent achieves a 57.8% improvement on the Shopping task and a 95.6% improvement on the CMS task, both of which provide low-quality API documentation. These results highlight the effectiveness of a tool-based approach over direct API calls, especially when documentation is incomplete or poorly structured. Even for well-documented APIs like Map, performance improves by 29.3% due to simplified parameter usage. The substantial overall performance gain enables our pure tool-based agent to outperform the hybrid approach that combines direct API calls with browser interactions. Meanwhile, repeated use of tools significantly reduces token consumption. On average, our approach costs $0.12 per task, compared to $1.20 for direct API calling and $1.50 for the hybrid approach.

In comparison, SteP (Sodhi et al., 2024) relies on manually defined policies and task-specific prompts to guide agent actions. Our Doc2Agent outperforms SteP on all tasks except Shopping, demonstrating the superior effectiveness of automatically generated tools. SkillWeaver (Zheng et al., 2025) synthesizes reusable skills as Python-based browser tools, which are relatively inefficient compared to our generated API-based tools. Doc2Agent consistently achieves higher performance across all tasks compared to SkillWeaver.

API-Specified Evaluation The original WebArena evaluator is tailored for browsing-based agents, relying on string matching, URL matching, and site navigation. We observed several false positives, such as cases where the agent failed to complete the task but included partial or coincidental keywords. Additionally, evaluation criteria involving browser interactions (e.g., editing web elements) are not applicable to API-only agents. To address these limitations, we introduce two adjustments: (1) For tasks marked as successful by the WebArena

evaluator, we use an LLM to verify whether the task was genuinely completed or merely matched keywords by chance. (2) We restructure site navigation and content-checking evaluations into `exact_match` or `must_include` criteria, based on the agent's final action log. This enables direct assessment of agent outputs, retrieved content, or API JSON responses, ensuring fair evaluation for API-based agents. We apply this API-specific evaluation to the hybrid agent as well, observing consistent overall performance with minor improvements. We were unable to re-evaluate the API-based agent from prior work due to the lack of publicly available logs. The evaluation details are provided in Appendix E.

4.2.1 Discussion: What factors make good tools for agents?

+ Accurate Parameter Values Our pipeline is particularly effective in addressing incomplete or vague API documentation. The code agent infers valid input parameter values by interacting with the environment—an approach especially beneficial for APIs like Shopping-Customer and GitLab, which return user-specific data not described in the documentation. By first querying supportive endpoints (e.g., `list_project`), the agent gathers contextual information to validate additional tools. We observed a strong negative correlation between parameter complexity and success rate for HybridAgent on Map tasks, highlighting the challenge of manually specifying complex inputs. In contrast, tools generated by Doc2Agent tend to use parameters with higher semantic alignment and lower complexity. However, validating parameters for stateful REST methods remains difficult (see Section 7) and introduces safety concerns for automatic testing on public servers.

+ Function-usage Success Rate Function usage success rate has the highest correlation with the task-level success rate. Doc2Agent calls functions consistently well(85%-97%) across all sites, while HybridAgent excels in shopping and CMS(98%-99%) tasks.

- Complex Response JSON Our approach shows lower performance than the Hybrid agent on CMS tasks, primarily due to the presence of long and unfiltered API responses. To manage context limitations, we truncated the response content, which may cause information loss. This highlights the need for more sophisticated response processing. Integrating advanced JSON navigation or summarization techniques could significantly improve the effectiveness of API-based agents in such settings. Moreover, developing tools for response-side information filtering presents a promising direction for enhancing tool efficiency.

4.3 Generation and Deployment of Research Agent

Unlike many real-world APIs, research-domain APIs often involve complex database identifiers and representations (see Appendix D3), along with relatively limited domain knowledge in base model. In this experiment, we explore the applicability of Doc2Agent in a challenging scientific setting, where we apply Doc2Agent to glycoscience APIs and automatically constructed a tool-using research agent specialized for the glycan domain.

Tool-generation Using Doc2Agent, we generate 70 refined tools and automatically constructed a database of parameter name–value pairs through parameter value inference. We compare two methods for parameter value acquisition: an API response–based inference approach and GPT-generated parameter candidates. Our results show that the API-based inference approach doubles the tool pass rate (see Appendix D.2).

Settings To enable seamless access to the research agent, we host the tools on an MCP server using FastMCP (Abramov, 2025). We evaluate three settings: (1) As used in Section 4.2, we apply CodeAct which loads tools directly into functions (2) An entirely open-source setup using `Qwen-Agent` (QwenLM, 2025) as the agentic framework, which supports basic MCP tool orchestration, with `Qwen3-32B` (Yang et al., 2025) as the base model. (3) A deployment using the enterprise-level application Claude Desktop, which offers advanced MCP Connector integration with Claude Sonnet 4 as the base model.

Task Generation We prompted GPT-4o to generate 50 research-oriented tasks based on the tool descriptions, including 30 single-tool tasks and 20 multi-tool tasks. Due to the lack of a formal evaluation framework, we used an LLM-as-a-judge approach (see template in Appendix H.3) to estimate task success. Since not all generated tasks are guaranteed to be

solvable, we report success rates over the Filtered set using the union of all tasks successfully completed by at least one agent to provide a fair basis for relative performance comparison.

Results Table 4.3 presents our results. Among the tested agents, Claude achieved the highest average success rate of 36%, successfully solving 58.1% of the do-able tasks. Claude excelled in multi-tool tasks, leveraging its advanced MCP orchestration technique. CodeAct's success rate is comparable to its performance on WebArena tasks, suggesting that our function generation method is not constrained by domain-specific characteristics and has the potential to scale effectively to other research domains. Notably, Qwen3-32B, despite being a smaller base model, performed on par with CodeAct, underscoring the potential of open-source research-tool agents.

Setting	Total	Filtered	Tool Use		Task Type		
			Single	Multi	Analysis	Data Retrieval	Transformation
Claude Desktop(Claude-Sonnet-4)	**36.0**	**58.1**	36.7	**35.0**	**45.5**	**30.4**	20.0
CodeAct(GPT-4o)	32.0	51.6	**46.7**	10.0	40.9	**30.4**	0.0
QwenAgent(Qwen3-32B)	30.0	48.3	33.3	25.0	31.8	26.1	**40.0**

Table 3: Success rate over different agent frameworks. **Filtered** refers to evaluation on the combined set of tasks successfully completed by at least one framework.

5 Related Works

Tool(API) Agent LLM-based tool agents are able to reason through user queries, select and apply appropriate actions, and return the results of the chosen action. For example, Bran et al. (2023) developed ChemCrow by integrating GPT-4 and 18 tools designed by experts. Qin et al. (2023) collected 16k public REST APIs from the RapidAPI platform RapidAPI (2024), and trained a tool retriever that can choose the most appropriate API in response to a user query. Wang et al. (2025b) represented tools as a unique token that are integrated into LLM generation. Zhang et al. (2025) compared API agents and GUI agents in solving web tasks. Song et al. (2025b) built a hybrid agent that can browse and call APIs to perform online tasks. Model Context ProtocolHou et al. (2025) allows fast deployment of argentic services. BenchmarksYehudai et al. (2025) are developed to evaluate the effectiveness of agents, while most of current benchmarks assume access to well-prepared toolsets(Trivedi et al. (2024), Qin et al. (2023), Song et al. (2023)). Particularly, we use WebArenaZhou et al. (2024) to demonstrate the improvement by encapsulated API-based tools.

Agent Generation and Improvement The creation for AI agents requires prompt design and tool design, which will be challenging to be automated. Chen et al. (2024) proposed a framework that generates specialized agents to form an AI team tailored to specific tasks. Shi et al. (2025) proposed AutoTools, which explores function generation through docs, but relies on standardized API specifications from RapidAPI. Zheng et al. (2025) introduced Skillweaver, a framework that enables web agents to self-improve by practicing reusable skills into APIs. However, their test revealed synthesized skills are worse than human APIs. Another direction(Gutiérrez et al. (2025), Du et al. (2025)) focus on memory updating that allows agents to improve through past experience.

6 Conclusion

In this work, we present **Doc2Agent**, a scalable pipeline for generating AI agents equipped with validated, Python-based tools from natural language REST API documentation. Doc2Agent not only converts unstructured documentation into executable tools but also detects and refines inaccuracies through automated validation. We applied our pipeline to three sources: real-world APIs, WebArena environments, and glycomaterial research APIs. We successfully generated and validated a large set of agent-ready tools. In WebArena, agents using our tools achieved a 55% relative performance improvement while reducing per-task cost to just 10% of the original. We further demonstrated Doc2Agent's domain

adaptability by building a research agent for glycomaterial science, leveraging our parameter inference method to resolve diverse data representations without external expert input. Doc2Agent enables efficient and robust agent creation across domains, bridging the gap between unstructured API documentation and practical tool-based agent deployment.

7 Limitation

API Documentation Acquisition Our method relies on available API documentation (e.g., web pages, OAS files) to generate Python-based tools, and thus cannot be applied to APIs lacking documentation. Currently, some human effort is still required to collect the initial API documentation. A promising future direction is to integrate our approach with web browsing capabilities, enabling agents to automatically discover and scrape relevant API documentation based on a given task—thereby expanding the agent's toolset autonomously.

Validation for Stateful APIs Our current validation approach relies on analyzing individual API responses, which is effective for stateless APIs but insufficient for stateful ones. Validating stateful APIs often requires more complex workflows involving multiple, coordinated API calls and additional endpoints to query server-side status or track changes over time.

API dependency We infer values for unknown parameters using example inputs and JSON responses from validated API calls. This approach leverages implicit dependencies between APIs and performs well when such relationships exist. However, its effectiveness diminishes for unrelated APIs, where parameter values cannot be reliably inferred from prior tool outputs.

Evaluation Evaluating our pipeline poses unique challenges. Existing function-calling benchmarks (e.g., ToolBench (Qin et al., 2023), AppWorld (Trivedi et al., 2024), Complex-FuncBench (Zhong et al., 2025)) focus primarily on tool usage, assuming well-prepared APIs and thus bypassing the need for tool generation. In contrast, browsing-based benchmarks (e.g., WebArena (Zhou et al., 2024), WebVoyager (He et al., 2024), ST-WebAgentBench (Levy et al., 2024)) introduce biases when used to evaluate our method: not all tasks in these environments are solvable via API calls, while others are directly derived from API usage. Consequently, comparisons between API agents and browsing agents on these benchmarks may be skewed due to task distribution. This highlights the need for a benchmark specifically designed to evaluate **open-domain API usage**, where tool generation, validation, and application can be fairly assessed across diverse and realistic API scenarios.

Cheating in Code Agent We observed that the code agent occasionally attempted to "cheat" during tool refinement. For example, by generating `try-catch` blocks to suppress exceptions and bypass validation errors. To mitigate this issue, we introduced an uneditable testing code section to enforce strict validation. However, this behavior highlights a broader concern: the agent's optimization strategy may not always align with human intent or practical utility.

8 Ethical Consideration

Autonomous REST API agents may trigger unintended or harmful actions, especially when interacting with external servicesMudryi et al. (2025). The tools generated by our pipeline are not manually reviewed for safety and, if misused, could result in undesirable or potentially malicious behavior. Given the ongoing development of AI safety practices, we recommend restricting tool generation to `GET` methods only, which are generally read-only and pose lower risk. This precaution helps mitigate unintended side effects during agent execution.

References

Mahyar Abbasian, Iman Azimi, Amir M. Rahmani, and Ramesh Jain. Conversational health agents: A personalized llm-powered agent framework, 2024. URL https://arxiv.org/abs/2310.02374.

Shahar Abramov. fastapi mcp. https://mem0.ai/openmemory-mcp, 2025.

Davide Alocci, Julien Mariethoz, Alessandra Gastaldello, Elisabeth Gasteiger, Niclas G Karlsson, Daniel Kolarich, Nicolle H Packer, and Frédérique Lisacek. Glyconnect: glycoproteomics goes visual, interactive, and analytical. *Journal of proteome research*, 18(2): 664–677, 2018.

axiom.ai. Automate logins with browser bots. https://axiom.ai/automate/login, 2024. Accessed: 2024-10-02.

Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. Researchagent: Iterative research idea generation over scientific literature with large language models, 2024. URL https://arxiv.org/abs/2404.07738.

Douglas K Barry. *Web services, service-oriented architectures, and cloud computing*. Elsevier, 2003.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools, 2023. URL https://arxiv.org/abs/2304.05376.

Vincent Bushong, Amr S Abdelfattah, Abdullah A Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, et al. On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, 11(17):7856, 2021.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation, 2024. URL https://arxiv.org/abs/2309.17288.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery, 2025. URL https://arxiv.org/abs/2410.05080.

John Dagdelen, Alexander Dunn, Sanghoon Lee, Nicholas Walker, Andrew S Rosen, Gerbrand Ceder, Kristin A Persson, and Anubhav Jain. Structured information extraction from scientific text with large language models. *Nature Communications*, 15(1):1418, 2024.

Kevin M Dorst and Göran Widmalm. Nmr chemical shift prediction and structural elucidation of linker-containing oligo-and polysaccharides using the computer program casper. *Carbohydrate research*, 533:108937, 2023.

Yiming Du, Wenyu Huang, Danna Zheng, Zhaowei Wang, Sebastien Montella, Mirella Lapata, Kam-Fai Wong, and Jeff Z. Pan. Rethinking memory in ai: Taxonomy, operations, topics, and future directions, 2025. URL https://arxiv.org/abs/2505.00675.

Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale api calls, 2024. URL https://arxiv.org/abs/2402.04253.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms, 2025. URL https://arxiv.org/abs/2504.11536.

Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. From llm reasoning to autonomous ai agents: A comprehensive review, 2025. URL https://arxiv.org/abs/2504.19678.

OpenAI GPTs. Introducing structured outputs in the api. https://openai.com/index/introducing-structured-outputs-in-the-api/, 2024. Accessed: 2025-5-01.

Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. From rag to memory: Non-parametric continual learning for large language models, 2025. URL https://arxiv.org/abs/2502.14802.

Yikun Han, Chunjiang Liu, and Pengfei Wang. A comprehensive survey on vector database: Storage and retrieval technique, challenge, 2023. URL https://arxiv.org/abs/2310.11703.

Kosuke Hashimoto, Susumu Goto, Shin Kawano, Kiyoko F Aoki-Kinoshita, Nobuhisa Ueda, Masami Hamajima, Toshisuke Kawasaki, and Minoru Kanehisa. Kegg as a glycome informatics resource. *Glycobiology*, 16(5):63R–70R, 2006.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models, 2024. URL https://arxiv.org/abs/2401.13919.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions, 2025. URL https://arxiv.org/abs/2503.23278.

Anne Imberty, François Bonnardel, and Frédérique Lisacek. Unilectin, a one-stop-shop to explore and study carbohydrate-binding proteins. *Current Protocols*, 1(11):e305, 2021.

Sunghwan Kim, Paul A Thiessen, Evan E Bolton, Jie Chen, Gang Fu, Asta Gindulyte, Lianyi Han, Jane He, Siqian He, Benjamin A Shoemaker, et al. Pubchem substance and compound databases. *Nucleic acids research*, 44(D1):D1202–D1213, 2016.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023. URL https://arxiv.org/abs/2205.11916.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation, 2025. URL https://arxiv.org/abs/2505.06120.

Alexander Lercher, Johann Glock, Christian Macho, and Martin Pinzger. Microservice api evolution in practice: A study on strategies and challenges. *Journal of Systems and Software*, 215:112110, September 2024. ISSN 0164-1212. doi: 10.1016/j.jss.2024.112110. URL http://dx.doi.org/10.1016/j.jss.2024.112110.

Ido Levy, Ben Wiesel, Sami Marreed, Alon Oved, Avi Yaeli, and Segev Shlomov. St-webagentbench: A benchmark for evaluating safety and trustworthiness in web agents, 2024. URL https://arxiv.org/abs/2410.06703.

Yang Li, Yangyang Yu, Haohang Li, Zhi Chen, and Khaldoun Khashanah. Tradinggpt: Multi-agent system with layered memory and distinct characters for enhanced financial trading performance, 2023. URL https://arxiv.org/abs/2309.03736.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023. URL https://arxiv.org/abs/2307.03172.

Mykyta Mudryi, Markiyan Chaklosh, and Grzegorz Wójcik. The hidden dangers of browsing ai agents, 2025. URL https://arxiv.org/abs/2505.13076.

Andrew Nightingale, Ricardo Antunes, Emanuele Alpi, Borisas Bursteinas, Leonardo Gonzales, Wudong Liu, Jie Luo, Guoying Qi, Edd Turner, and Maria Martin. The Proteins API: accessing key integrated protein and genome information. *Nucleic Acids Research*, 45(W1):W539–W544, 04 2017. ISSN 0305-1048. doi: 10.1093/nar/gkx237. URL https://doi.org/10.1093/nar/gkx237.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023. URL https://arxiv.org/abs/2307.16789.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343, 2025.

QwenLM. Qwen agent github page. https://github.com/QwenLM/Qwen-Agent, 2025.

RapidAPI. Rapidapi hub. https://rapidapi.com/hub, 2024. Accessed: 2024-12-31.

Yana Rose, Jose M Duarte, Robert Lowe, Joan Segura, Chunxiao Bi, Charmi Bhikadiya, Li Chen, Alexander S Rose, Sebastian Bittrich, Stephen K Burley, et al. Rcsb protein data bank: architectural advances towards integrated searching and efficient access to macromolecular structure data from the pdb archive. *Journal of molecular biology*, 433(11): 166704, 2021.

Eric W Sayers, Jeffrey Beck, Evan E Bolton, Devon Bourexis, James R Brister, Kathi Canese, Donald C Comeau, Kathryn Funk, Sunghwan Kim, William Klimke, et al. Database resources of the national center for biotechnology information. *Nucleic acids research*, 49 (D1):D10–D17, 2021.

Zhengliang Shi, Shen Gao, Lingyong Yan, Yue Feng, Xiuyi Chen, Zhumin Chen, Dawei Yin, Suzan Verberne, and Zhaochun Ren. Tool learning in the wild: Empowering language models as automatic tool agents, 2025. URL https://arxiv.org/abs/2405.16533.

Paloma Sodhi, S. R. K. Branavan, Yoav Artzi, and Ryan McDonald. Step: Stacked llm policies for web actions, 2024. URL https://arxiv.org/abs/2310.03720.

Yewei Song, Xunzhu Tang, Cedric Lothritz, Saad Ezzini, Jacques Klein, Tegawendé F. Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. Callnavi, a challenge and empirical study on llm function calling and routing, 2025a. URL https://arxiv.org/abs/2501.05255.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. Restgpt: Connecting large language models with real-world restful apis, 2023. URL https://arxiv.org/abs/2306.06624.

Yueqi Song, Frank Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: Api-based web agents, 2025b. URL https://arxiv.org/abs/2410.16464.

Katharina Stein, Daniel Fišer, Jörg Hoffmann, and Alexander Koller. Automating the generation of prompts for llm-based action choice in pddl planning, 2025. URL https://arxiv.org/abs/2311.09830.

Swagger. Openapi specification. https://swagger.io/specification/, 2024. Accessed: 2024-10-02.

Naoyuki Taniguchi, Tamao Endo, Gerald Warren Hart, Peter H. Seeberger, and Chi Huey Wong. *Glycoscience: Biology and medicine*. Springer Japan, January 2015. ISBN 9784431548416. doi: 10.1007/978-4-431-54841-6.

Michael Tiemeyer, Kazuhiro Aoki, James Paulson, Richard D Cummings, William S York, Niclas G Karlsson, Frederique Lisacek, Nicolle H Packer, Matthew P Campbell, Nobuyuki P Aoki, et al. Glytoucan: an accessible glycan structure repository. *Glycobiology*, 27(10):915–919, 2017.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents, 2024. URL https://arxiv.org/abs/2407.18901.

Fangxi Wang, Swarnadeep Seth, Saikiran Reddy Ramacharla, and Sanket A Deshmukh. Glycodata. glycodata.org/, 2025a. Accessed: 2025-1-15.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024a. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. Toolgen: Unified tool retrieval and calling via generation, 2025b. URL https://arxiv.org/abs/2410.03439.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024b. URL https://arxiv.org/abs/2402.01030.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Woods Group. Glycam web: Website builders, 2025. URL http://glycam.org. Accessed 2025.

Eugenia Wulff-Fuentes, Rex R Berendt, Logan Massman, Laura Danner, Florian Malard, Jeet Vora, Robel Kahsay, and Stephanie Olivier-Van Stichelen. The human o-glcnacome database and meta-analysis. *Scientific data*, 8(1):25, 2021.

Issaku Yamada, Masaaki Shiota, Daisuke Shinmachi, Tamiko Ono, Shinichiro Tsuchiya, Masae Hosoda, Akihiro Fujita, Nobuyuki P Aoki, Yu Watanabe, Noriaki Fujita, et al. The glycosmos portal: a unified and comprehensive web resource for the glycosciences. *Nature Methods*, 17(7):649–650, 2020.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. Survey on evaluation of llm-based agents, 2025. URL https://arxiv.org/abs/2503.16416.

William S York, Raja Mazumder, Rene Ranzinger, Nathan Edwards, Robel Kahsay, Kiyoko F Aoki-Kinoshita, Matthew P Campbell, Richard D Cummings, Ten Feizi, Maria Martin, et al. Glygen: computational and informatics resources for glycoscience. *Glycobiology*, 30 (2):72–73, 2020.

Chaoyun Zhang, Shilin He, Liqun Li, Si Qin, Yu Kang, Qingwei Lin, and Dongmei Zhang. Api agents vs. gui agents: Divergence and convergence, 2025. URL https://arxiv.org/abs/2503.11069.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. Complexfuncbench: Exploring multi-step and constrained function calling under long-context scenario, 2025. URL https://arxiv.org/abs/2501.10132.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

A Real-life API Dataset

A.1 Constuction

Documentation styles The API documentation we collected can be categorized into three levels based on the organization and clarity of the API descriptions: (1) **Fully organized** The documentation follows a well-defined template, providing all necessary information to call the API in a structured and comprehensive way. Use cases are clearly explained, often with example code. API documentation on platforms like RapidAPI Hub and Postman API typically fall into this category. (2) **Semi-organized** This type of documentation includes basic descriptions but lacks clarity for each endpoint. Some essential information may not be labeled with specific keywords and is instead embedded within general text. Additional effort is often required to identify key details. (3) **Unorganized** These documents are minimal, often missing example code or detailed descriptions. They require some level of inference and reasoning to understand the API's usage, with clues only available through endpoint names. We show that our benchmark consists of mostly semi-structure documentations, and only a few documentations are fully organized(Appendix A.2).

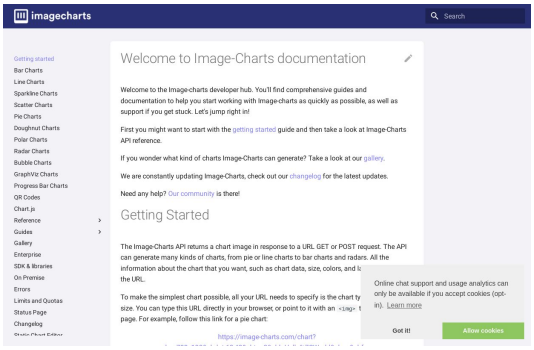
Selection criteria We filtered for API documents that do not require API keys, allowing for easier and more convenient API access without needing authentication, which often involves submitting forms or linking payment methods. Although automating API key sign-up is feasible using web agents axiom.ai (2024), we opted for APIs without authentication requirements to streamline the process. In total, 347 unique API documents were selected and downloaded in HTML format. Since some links pointed to index pages or API information that was dynamically loaded via JavaScript, we employed a large language model to identify pages containing static HTML code with API endpoints. This approach ensured that we captured only the documentation with accessible and actionable API details.

Due to variations in the quality and completeness of API documentation, we extracted only the essential information needed for tool generation. Specifically, for each API documentation, we captured the `base URL` and a list of endpoints. For each endpoint, we extracted the `endpoint path`, `required parameters`, `optional parameters`, and a brief `description`. In cases where the base URL was not specified, human annotation was necessary. The schema used for this extraction is provided in the Appendix G.

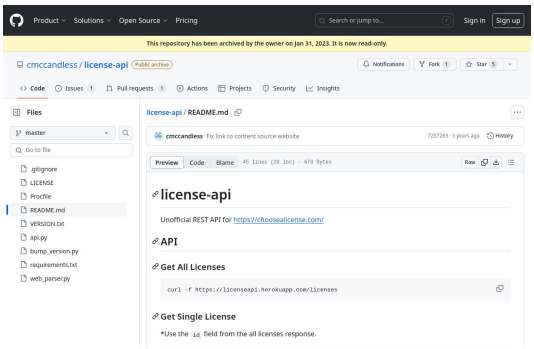
To implement this extraction, we defined the schema using a Pydantic model and employed GPT-4o in structured mode to parse the HTML documents and extract the desired information. After filtering out pages that lack API information (primarily product index pages), we obtained 167 API documentation with 744 endpoints and extracted their structured information in JSON format. An example of input API documentation and output JSON structure is in Appendix F.

A.2 Documentation Quality Classification

We classify API documentations into 3 categories: **Fully organized**, **Semi-organized** and **Unorganized**, based on their clarity and completeness of information. Using GPT-4o with chain-of-thought reasoning (Appendix H.2), we categorized all API documentation in the API Extraction Benchmark. Out of the total, 24 documents were classified as Fully Organized, 134 as Semi-Organized, and 9 as Unorganized. Figure A1 provides examples of API documentation in each category. Notably, none of the documentation is fully standardized, and Semi-Organized and Unorganized documents frequently lack clarity or essential API details.



(a)



(b)

(c)

Figure A1: Example of API documentation of each category. (a) is an example of organized API documentation **link**, (b) is the example of semi-organized documentation **link**, and (c) is the unorganized API documentation **link**

B JSON-To-Tool generation

JSON-To-Tool generation is mostly about engineering. As the documentations are written in various formats, multiple conditions need to be considered, especially when handling URLs. It is common for the API server to only support a specific input pattern, which is not mentioned in the documentation or the error message. To make the autogenerated tool more robust, we considered the following procedures in our tool generator:

parameter handling Documentations use various ways to represent path parameters. We set matching rules to find commonly used patterns such as "`:param`", "`{param}`", "`<param>`" etc.

encoding correction To pass some special characters such as "+" or "=", the URL will use an encoding method known as percent encoding. Usually this won't be an issue, but for some APIs this need to be done before passing the parameters.

required parameter checking As APIs may not necessarily return the error information, we added a required parameter validation step in the generated tool so that we can report any missing parameter errors to the AI agent.

C Tool Error Types and Causes

Errors are categorized below:

Incomplete URL: Occasionally, the URL of a tool is incomplete, resulting in failed requests. These errors can be classified into two subtypes: **Missing Endpoint Path** or **Missing Base URL**. By examining the corresponding API documentations, we found that endpoint paths were usually provided, but base URLs were often missing.

Request Errors: Request errors are complex and challenging to diagnose, as status codes alone do not clearly indicate whether the issue originates from the server or client side. A status code of 200 ("OK") guarantees valid communication between the client and server. To further validate the response content in such cases, we use an GPT-4o based evaluator (Appendix H.1). If the evaluator returns "pass", the tool is labeled as **Passed Validation**; otherwise, it is labeled as **Failed Validation**. For other status codes or unexpected cases, the tool is classified under **Abnormal Response**.

Incorrect Parameter Values: If a tool throws an exception that does not fall into any of the previously mentioned scenarios, it indicates that the tool was very likely called with invalid parameters. Specifically:

- If a required parameter is missing, the error is classified as **No Parameter Value**.
- If all required parameters are provided but the tool still fails, the issue likely stems from an incorrect example value, and the error is classified as **Wrong Parameter Value**.

For all error types other than **Passed Validation**, we group them into four main categories: **C1** Missing API Documentation Details, **C2** Incorrectly Extracted URL Path, **C3** Incorrect Parameter Values, and **C4** Server-Side Errors. For each category, we provide a range of possible error diagnosis, from the most conservative to the most aggressive (see Appendix C1).

Category	Conservative Estimate	Aggressive Estimate(Additional Terms Only)
Missing API Documentation Details	0	Missing Base URL+No Parameter Value
Incorrectly Extracted URL Path	Missing Endpoint Path	Missing Base URL
Incorrect Parameter Values	Wrong Parameter Value+Failed Validation	No Parameter Value+Abnormal Response
Server-Side Error	0	Failed Validation+Abnormal Response

Table C1: Estimation of error causes

We aim to automatically fix the tools that either failed in the validation or can't be validated due to missing parameter examples. We find that most of the errors are caused by **C3**.

We investigate the failed tools and find the example parameters from such tools are either missing or the parameter is not filled properly due to low-quality documentation. This inspires us to develop an approach to produce high-quality parameter values.

D Glycoscience Agent

D.1 Contribution to Glyco Research Community

Our AI agent provides natural language interfaces for researchers to access web services (e.g., databases and utility APIs) without requiring technical expertise, which greatly facilitates the usage of online resources and benefits researchers in several ways:

(1) Automatic cross-database integration No single database fulfills all information needs due to their specific focuses. For example, GlyTouCan catalogs glycan structures, KEGG GLYCAN maps pathways and reactions, and PubChem offers general molecular data. Our AI agent integrates these diverse sources for seamless information access.

(2) Tool synergy Individual tools are often limited to specific scenarios, but our AI agent enhances their applicability by integrating them into cohesive workflows, including ID conversion, database querying, data normalization, visualization, and resolution of inconsistencies. Appendix Table D2 shows an example of a glycan being represented using various formats. This integration broadens the applicability of certain tools. For instance, the GLYCAM 3D visualization tool, initially limited to GLYCAM strings, now supports multiple glycan formats through automated conversion.

(3) Facilitate the development and adoption of new services Using Doc2Agent, our AI agent can easily adopt new APIs and datasets, keeping its tools up-to-date. This enables researchers to prioritize scientific exploration over technical integration. In addition, the AI agent can serve as a platform for developing, sharing, and publishing applications, fostering dissemination and collaboration.

D.2 Parameters in Glycoscience APIs

String Representation	
IUPAC Condensed	Fuc(a1-2)Gal(b1-3)[Fuc(a1-4)]GlcNAc(b1-
GLYCAM	LFucpa1-2DGalpb1-3[LFucpa1-4]DGlcpNAcb1-OH

Database ID		**Glycan Name**
GlyToucan ID	G00048MO	Lewis b
PubChem ID	45480569	

Table D2: Example of different representations of Glycan *Lewis b*. The table shows various ways to represent the glycan, demonstrating the complexity of glycan reference.

Table D2 and D3 shows the complexity of database entries and representations in glyco databasee, demonstrating the importance of infering correct parameter values for domain-specified agent.

To simulate missing parameter information, we applied a leave-one-API-out setup: for each API, we masked all corresponding parameter values and related entries in the database. Our parameter inference algorithm then generated candidate values using the remaining data. We evaluated the success of each inferred value using an LLM-as-a-judge framework, assessing whether the resulting tool call executed correctly.

Results As shown in Table D4, by using the leave-one-API-out test setting, our approach successfully infers parameter values for 33 tools. Meanwhile, GPT-4o managed to find

Database	Reference ID	Primary Usage
GlyTouCan	GlyTouCan ID	Glycan Structure GlyTouCan ID for API Calling
KEGG GLYCAN	KEGG ID	KEGG Pathyway Reaction
GlyGen	GlyTouCan ID	Publication Cross Reference
O-GlcNAc	UniProtKB ID	Protein O-GlcNAcylation Data
PubChem	PubChem CID	Chemical Properties
Unilectin	Unilectin ID	Get Lectin and Ligand Information

Table D3: Glycan databases covered in glyco agent

	Ours	GPT-4o
Pass@10	33	17

Table D4: Successful Parameter Value Guess in glyco-material APIs

the parameter values for only 17 tools. The results clearly indicate the superiority of our approach over GPT-4o.

D.3 Deployment of the Research Agent

A research agent can be easily deployed through any agentic framework. Figure D2 shows an example of the aggregation of the MCP server produced by Doc2Agent framework and Claude Desktop UI.

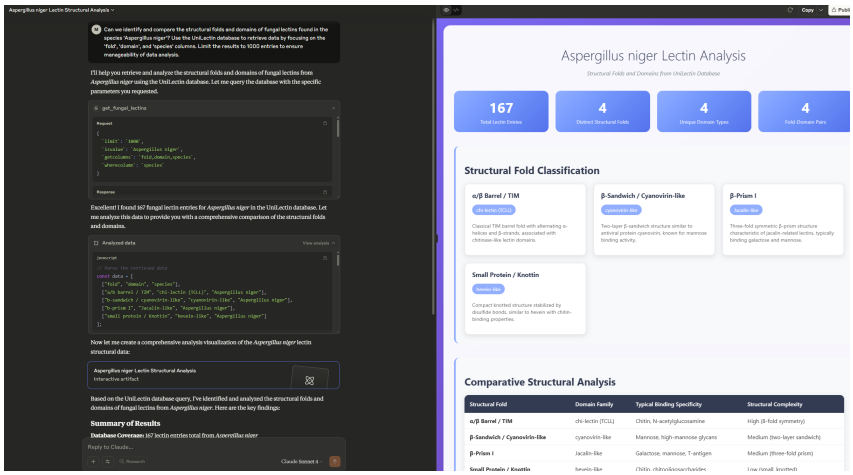


Figure D2: Example of a research agent produced by Doc2Agent aggregated into Claude Desktop. It can solve domain-specific questions with tools and generate reports in various formats.

E WebArena Evaluation for API-agent

We modify the vanilla WebArena evaluator to tackle inaccurate results for better evaluation for API using agents.

E.1 Intend Confirmation

A common source of false positives occurs when the agent fails to execute the intended task but inadvertently includes an output that satisfies the string match criteria. To ensure the agent's behavior aligns with the task intent, we employ a two-step string match evaluation process: (1) The agent's output is first evaluated using the standard WebArena string match evaluator to confirm the presence of the target reference string. (2) An LLM then reviews the agent's reasoning path to determine whether the task was genuinely completed or if the match was incidental.

An example of such a false positive is shown below:

WebArena False Positive Case

WebArena Task 22:

```
'''List out reviewers, if exist, who mention about under
    water photo'''
```

The vanilla evaluator validate the answer with:

```
"fuzzy_match": "N/A" ("There is no review about under water
    photo")
```

In the agent log, the agent ends up getting struck and accidentally output nothing, which allows it to pass the test:

```
Step 4:
To list out reviewers who mention "under water photo" for the product with SKU B001D0G57S, I will
     follow these steps:

1. **Check for an API to retrieve product reviews**: I will look for an API that can fetch reviews
     for a specific product.
2. **Fetch the reviews using the API**: If an API is available, I will use it to get the reviews for
     the product with SKU B001D0G57S.
3. **Filter reviews mentioning "under water photo"**: I will filter the reviews to find those that
     mention "under water photo".
4. **Verify the results using web browsing**: I will navigate to the product page and manually verify
      the reviews to ensure the results are correct.

First, let's check for an API that can retrieve product reviews.
...
Step 12:
Too many errors encountered. Task failed.
Step 13:
Too many errors encountered. Task failed.
Step 14:
Too many errors encountered. Task failed.
```

Vanilla WebArena Evaluator

```
correct: true
```

API-specified Evaluator

```
correct: false
```

String-match LLM-as-a-Judge Prompt

We use an LLM to check the agent log.

```
'''You are evaluating an agent's performance on the task:
{task_intent}

The agent's success is measured by these requirements:
{evaluator_str}

Below are the execution logs. Determine whether the agent has successfully completed the task based on the requirements.
{extracted_log}

IMPORTANT: Your evaluation should be based on whether the agent achieved the task's objective. Focus on the logical reasoning path and the final outcome.

AGENT'S FINAL ANSWER:
{final_answer}

Provide your assessment in this exact format:
REASON: [One sentence explaining your decision]
DECISION: [success/fail]'''
```

E.2 Adaption for API-agent

The `url_match` and `program_html` evaluator in WebArena assesses agent actions by examining URLs or changes in webpage elements—such as verifying whether a string was correctly entered into a search field. However, this type of interaction can be bypassed entirely by API-based agents. For example, rather than add search strings into the search bar and clicking a `search` button, an API agent may directly invoke a search endpoint which uses a different URL to the button link. Such behavior is not detectable by the original evaluator.

To address this limitation, we extend the evaluator to support API-calling functions by enabling it to match relevant information across API requests, responses, and webpage content. Based on this matched information, we incorporate an LLM-based evaluator to determine: (1) whether the API agent's action trajectory is functionally equivalent to the intended `url_match` or `program_html` behavior, and (2) whether the agent successfully completes the task. An example is provided below:

WebArena API use example

WebArena Task 395:

```
'''Fork the Pytorch GAN repo with most stars.'''
```

The vanilla evaluator validate the answer with:

```
"__GITLAB__/byteblaze/PyTorch-GAN": must_include("Pytorch-GAN")
```

API-agent action

```
'''Step 5:
To fork the Pytorch GAN repository with the most stars on GitLab, we need to follow these steps:
1. List the available tools in the 'projects' category to find tools that can help us search for projects.
2. Use the appropriate tool to search for Pytorch GAN repositories and identify the one with the most stars.
3. Use the project ID of the identified repository to fork it.
Let's start by listing the tools in the 'projects' category to find the right tool for searching projects.
<observation not shown>
Step 6:
We have a list of tools available in the 'projects' category. To find the Pytorch GAN repository with the most stars, we should use the `list_projects` tool to search for relevant projects and then identify the one with the most stars.
Let's first get the documentation for the `list_projects` tool to understand how to use it effectively.
<observation not shown>
Step 7:
The `list_projects` tool allows us to search for GitLab projects with various filtering options. We can use the `search` parameter to look for Pytorch GAN projects and the `order_by` and `sort` parameters to sort the results by the number of stars in descending order.
Let's proceed to call the `list_projects` function to search for Pytorch GAN projects and identify the one with the most stars.
<observation not shown>
Step 8:
It seems that the `order_by` parameter does not accept `star_count` as a valid value. Let's check the documentation again to see what valid values are available for the `order_by` parameter. We might need to use a different approach to identify the project with the most stars.
I'll review the documentation for the `list_projects` tool again to find the correct `order_by` options.
<observation not shown>
Step 9:
The documentation indicates that the `order_by` parameter defaults to `created_at`, but it doesn't specify other valid values. Since `star_count` is not valid, we should try using the default `created_at` and then manually identify the project with the most stars from the results.
Let's call the `list_projects` function again without specifying `order_by` and `sort` to retrieve the projects and then find the one with the most stars.
<observation not shown>
Step 10:
The response was truncated, so we need to retrieve the full response to analyze all the projects and identify the one with the most stars. Let's do that now.
<observation not shown>
Step 11:
We have retrieved the full response, and it contains details about the Pytorch GAN projects. Now, let's identify the project with the most stars and proceed to fork it.
The project with the most stars is "PyTorch-GAN" by Erik Linder-Noren, with 45 stars. We will use the `fork_project` tool to fork this project.
Let's proceed to get the documentation for the `fork_project` tool to understand how to use it.
<observation not shown>
Step 12:
The `fork_project` tool allows us to create a fork of an existing project. We need to provide the `project_id` of the project we want to fork. We can also specify additional parameters like `namespace_path`, `name`, and `description` if needed.
Let's proceed to fork the "PyTorch-GAN" project with ID 86.
<observation not shown>
Step 13:
The "PyTorch-GAN" project has been successfully forked. The forked project is now under the namespace "Byte Blaze" with the project ID 195.\n\nFinish[The Pytorch GAN repo with the most stars has been successfully forked.]
```

LLM Reason

```
The agent successfully forked the \"PyTorch-GAN\" project, as indicated by the status code 201 and the confirmation that the forked project is under the namespace \"Byte Blaze\" with a new project ID.'''
```

LLM decision

```
correct: true
```

Program html LLM-as-a-Judge Prompt

We use an LLM to check the agent log.

```
'''You are evaluating an API agent's performance on the task: {task_intent}

The reference implementation accesses {ref_urls_str}, but this agent uses a different API ecosystem.
      Your goal is to determine if the agent's API calls accomplish the same functional objective,
      regardless of specific endpoints used.

**Evaluation Criteria:**
1. **Functional Alignment**: Do the API calls serve the same purpose as the reference URL?
2. **Parameter Relevance**: Are the parameters extracted from the query and used appropriately?
3. **Logical Sequence**: Do the API calls follow a reasonable order to accomplish the task?
4. **Error Handling**: Does the agent handle failures gracefully and attempt alternatives?

**Assessment Guidelines:**
- SUCCESS if the agent's API calls would reasonably accomplish the same goal as the reference URL
- SUCCESS if the agent uses equivalent but different endpoints (e.g., different weather APIs for
      weather queries)
- SUCCESS if the agent makes multiple related calls that collectively achieve the objective
- FAIL only if the API calls are clearly unrelated to the task or would not achieve the intended
      outcome
- Consider the agent's reasoning process from the logs, not just the final API calls

**Parameter Analysis:**
{overlap_functions}
Reference parameters: {ref_parameter}
Agent parameters: {agent_parameter}

**Agent Execution:**

Execution log:
{extracted_log}

**Your Assessment:**
REASON: [One sentence explaining whether the agent's approach would accomplish the same objective as
      the reference URL, considering functional equivalence rather than exact matching]
DECISION: [success/fail]'''
```

F Direct Tool Generation Example

Tool Generation Example

We extract web contents into JSONs first. the API documentation(HTML) page, while the output is a JSON string following our API-extraction Schema. Below is the input example:

```
# OSRM HTTP Router
## General options
All OSRM HTTP requests use a common structure.
The following syntax applies to all services, except as noted.
### Requests
```endpoint
GET http://http://ec2-3-129-135-45.us-east-2.compute.amazonaws.com:{profile}/{service}/v1/test/{
      coordinates}[.{format}]?option=value&option=value
```
Parameter	Description
`profile`	Mode of transportation. One of the following three values: `5000` for car (driving),
`5001` for bicycle (biking), and `5002` for foot (walking).
`service`	One of the following values: [`route`](#route-service), [`nearest`](#nearest-service),
[`table`](#table-service), [`match`](#match-service), [`trip`](#trip-service), [`tile`](#tile-
service)
`coordinates`	String of format `{longitude},{latitude};{longitude},{latitude}[;{longitude},{
latitude} ...]` or `polyline({polyline}) or polyline6({polyline6})`.
`format`	`json` or `flatbuffers`. This parameter is optional and defaults to `json`.
......
```

The output example shows as follows:

```
{
    "title": "OSRM HTTP Router API Documentation",
    "endpoints": [
        {
            "name": "General Request",
            "description": "All OSRM HTTP requests use a common structure.",
            "method": "GET",
            "url": [
                "http://ec2-3-129-135-45.us-east-2.compute.amazonaws.com:{profile}/{service}/v1/test
                      /{coordinates}[.{format}]?option=value&option=value"
            ],
            "headers": [],
            "required_parameters": [
                {
                    "name": "profile",
                    "type": "string",
                    "description": "Mode of transportation. One of the following three values: '5000'
                           for car (driving), '5001' for bicycle (biking), and '5002' for foot (
                          walking).",
                    "default": null,
                    "example": "5000"
                },
                {
                    "name": "service",
                    "type": "string",
                    "description": "One of the following values: 'route', 'nearest', 'table', 'match
                          ', 'trip', 'tile'.",
                    "default": null,
                    "example": "route"
                },
                ...
            ],
            "optional_parameters": [
                {
                    "name": "format",
                    "type": "string",
                    "description": "'json' or 'flatbuffers'. This parameter is optional and defaults
                          to 'json'.",
                    "default": "json",
                    "example": "json"
                }
            ]
        },
}
```

Tool Generation Example

By appling the JSON-To-Tool generation script, we can convert the structured API information into an excutable tool. In this work, a tool is a python function which can be called by the LLM. A typical tool example for 'Pokemon TCG API' is shown below.

```python
import requests
import json
from urllib.parse import quote

def general_request(profile=None, service=None, coordinates=None, format='json', **kwargs):
    assert profile is not None, 'Missing required parameter: profile'
    assert service is not None, 'Missing required parameter: service'
    assert coordinates is not None, 'Missing required parameter: coordinates'

    base_url = f"http://ec2-3-129-135-45.us-east-2.compute.amazonaws.com:{profile}/{service}/v1/test/{coordinates}"


    response = requests.get(url=base_url, params=kwargs, timeout=50)
    return response

if __name__ == '__main__':
    r = general_request(profile='5000', service='route', coordinates='13.388860,52.517037;13.397634,52.529407', format='json')
    r_json = None
    try:
        r_json = r.json()
    except:
        pass
    result_dict = dict()
    result_dict['status_code'] = r.status_code
    result_dict['text'] = r.text
    result_dict['json'] = r_json
    result_dict['content'] = r.content.decode("utf-8")
    print(json.dumps(result_dict, indent=4))
```

Refined Tool Example

After validation and refinement, bugs will be fixed and __doc__ will be added to the generated python function based on the API response.

```python
import requests
import json
from urllib.parse import quote

def general_request(profile=None, service=None, coordinates=None, format='json', **kwargs):
    """
    Make a request to the OSRM HTTP Router API.

    Parameters:
    -----------
    profile : str
        Mode of transportation. One of the following values:
        - '5000' for car (driving)
        - '5001' for bicycle (biking)
        - '5002' for foot (walking)

    service : str
        One of the following values:
        - 'route': Find the fastest route between coordinates
        - 'nearest': Snap a coordinate to the street network
        - 'table': Compute the duration matrix between coordinates
        - 'match': Match GPS points to the road network
        - 'trip': Solve the Traveling Salesman Problem
        - 'tile': Generate Mapbox Vector Tiles

    coordinates : str
        String of format '{longitude},{latitude};{longitude},{latitude}[;{longitude},{latitude} ...]'
        or 'polyline({polyline})' or 'polyline6({polyline6})'.

    format : str, optional (default='json')
        Response format, either 'json' or 'flatbuffers'.

    **kwargs : dict
        Additional query parameters to pass to the API.
        Common options include:
        - overview: 'simplified', 'full', or 'false'
        - steps: 'true' or 'false'
        - annotations: 'true', 'false', or specific values
        - geometries: 'polyline', 'polyline6', or 'geojson'

    Returns:
    --------
    requests.Response
        The response from the OSRM API.

    Examples:
    ---------
    >>> # Get a route between two points
    >>> response = general_request(
    ...     profile='5000',
    ...     service='route',
    ...     coordinates='13.388860,52.517037;13.397634,52.529407',
    ...     overview='false'
    ... )

    >>> # Get a table of durations between three points
    >>> response = general_request(
    ...     profile='5000',
    ...     service='table',
    ...     coordinates='13.388860,52.517037;13.397634,52.529407;13.428555,52.523219',
    ...     annotations='duration'
    ... )
    """
    assert profile is not None, 'Missing required parameter: profile'
    assert service is not None, 'Missing required parameter: service'
    assert coordinates is not None, 'Missing required parameter: coordinates'

    base_url = f"http://ec2-3-129-135-45.us-east-2.compute.amazonaws.com:{profile}/{service}/v1/test/{coordinates}"

    if format and format != 'json':
        base_url += f".{format}"

    response = requests.get(url=base_url, params=kwargs, timeout=50, verify=False)
    return response

if __name__ == '__main__':
    r = general_request(profile='5000', service='route', coordinates='13.388860,52.517037;13.397634,52.529407', format='json')
    r_json = None
    try:
        r_json = r.json()
    except:
        pass
    result_dict = dict()
    result_dict['status_code'] = r.status_code
    result_dict['text'] = r.text
    result_dict['json'] = r_json
    result_dict['content'] = r.content.decode("utf-8")
    print(json.dumps(result_dict, indent=4))
```

G API-extraction Schema

API-extraction Schema

We defined the Pydantic schema to extract API information.

```python
class Parameters(BaseModel):
    name: str = Field(description="Name of the parameter")
    type: Optional[str] = Field(description="Type of the
        parameter")
    description: Optional[str] = Field(description="
        Description of the parameter. If the parameter is
        categorical, please list all possible values.")
    default: Optional[Any] = Field(
        None,
        description="Default value of the parameter")
    example: Optional[Any] = Field(
        description="Example value of the parameter")


class Endpoint(BaseModel):
    name: str = Field(description="Name of the endpoint")
    description: Optional[str] = Field(
        description="Description of the endpoint")
    method: str = Field(description="Method of the endpoint")
    url: Union[str, List[str]] = Field(description="URL of
        the endpoint, start with http:// or https://")
    headers: Optional[List] = Field(
        default=[], description="Headers of the endpoint")
    required_parameters: Optional[List[Parameters]]
    optional_parameters: Optional[List[Parameters]]


class Api_json(BaseModel):
    title: Optional[str] = Field(description="Title of the
        API")
    endpoints: List[Endpoint]
```

H LLM Prompts

H.1 Doc2Agent Prompt

API Response Validation Prompt

The prompt for the model is defined as

```
"""
Given an API description, response, and Python code, classify the response type:

- information: Valid response containing useful data as expected
- code_error: Error due to bugs in the Python code (syntax, logic, url path doesn't match, etc.)
- server_error: Server-side error (500, service unavailable, authentication error, etc.)
- request_error: Invalid operation due to bad parameters or unsupported operation (only choose this when other errors are not applicable and you think the code is correct but the response is an error)

API Description: {description}
API Response: {response}
Code: {code}
"""
```

The expected output is:

```
class Classification(BaseModel):
    response_type: str = Field(..., enum=['information', 'code_error', 'server_error', 'request_error'])
```

API Refinement Prompt

To refine a tool, we use the following template

```
"""
You are a Python debugging expert specializing in code analysis and correction. Your will be provided with:
1. Error information.
2. API Json documentation: The API documentation includes all endpoints information. Please refer to the most relevent one based on the similarity of the Python function name and the endpoint's name.
3. Python code containing the function that interacts with APIs: The code may contain syntax errors, parameter issues, or other problems that prevent it from running correctly.
4. Parameters: The parameter examples that are passed to the function.
5. Configuration: The API configuration including base URL, headers, authentication details, and testing information.

Your task is to identify and fix errors in the provided code based on the error information and API documentation.

Requirements:
- IMPORTANT: Return ONLY the corrected code without explanations. The code you answered should be run successfully in a Python environment without any other human modifications.
- DON'T return the code with the wrap of ```python```.
- Ensure the code includes import statements for any necessary libraries.
- If the parameter examples are empty or not given, you should guess the correct values based on the API documentation and include them.
- If the configuration includes an "info" section with user_name, user_id, project_id, etc., use these values as realistic test parameters when the function needs user-specific data. These are real test values that should work with the API.
- Please write the doc after refineing the code so that I can directly call code.__doc__. Be sure to include examples of the parameters in the docstring.
- Please check the parameters and make sure they are passed to the url correctly. Add optional parameters to allow custumizerble use of the tool when necessary.
- Use the provided configuration for base URL, headers, and authentication. Replace any hardcoded URLs or headers with the configuration values.
- You should **NEVER** edit the result_dict in main function.

Error Information:
{error}

API Documentation:
{api_doc}

Configuration:
{config}

Code to fix:
{code}

Candidate Parameter Values:
{params}
"""
```

Parameter Value Inference Prompt

The prompt for direct parameter value inferencing is

```
'''
You will be provided with the information of an API and its
    parameters. The example values of the parameters are
    missing. You need to guess the parameter values.
You may have failed severl times before. If you guess with
    similar values, you may fail again. Please be innovative
    and try different values and formats.

Your previous failed guesses:
***history start
{history}
***history end

API Description:
{description}

Parameter Description:
{param_description}

Your Guess:
'''
```

H.2 API Documentation Classification Prompt

API Response Validation Prompt

The prompt for the model is

```
'''
You need to group the API documentation with the following
    standards:

Fully Organized: The documentation follows a well defined
    template, most likely to be from an API platform. It is
    well-structured, clear, and easy to understand. It
    includes detailed descriptions, example code, and
    explanations of how to use the API.
Semi-Organized: Lacks some structure, but still includes most
     of the necessary information. It may be missing some
    examples or descriptions, making it slightly more
    difficult to understand how to use the API.
Unorganized: Missing example or description, or the structure
     is unclear, making it difficult to understand how to use
     the API.

===
API Documentation:
{API_DOC}
'''
```

The output of the mode is defined as

```
# Output Class Structure
class Classification(BaseModel):
    analysis: str = Field(..., description="The analysis of
        the API documentation. Make it within 300 characters.
        ")
    category: str = Field(..., enum=["Fully Organized", "Semi
        -Organized", "Unorganized"])
```

H.3 LLM-as-a-Judge Prompt

Recheck WebArena Tasks

We use LLM to double check the WebArena Tasks to evaluate if the agent actually complete the task.

```
'''
You are an expert evaluator.
Given a task log and the reference answer.
Your job is to compare the log and the reference answer and then determine if the log shows that its job is done.
Answer, respond with ONLY one word: 'true' or 'false'.

Note:
Focus on the content of the log and the reference answer.
Check if the result has the same meaning as the reference answer.
For shopping tasks, check if the correct cart operations were performed or the correct product information was retrieved.

### Reference answer
{}

### Task log
{log_text}

### Your decision
'''
```

Evaluator for Glycan dataset

```
'''
"""
You are an expert glycomics researcher evaluating an AI agent's performance on a glycan research task.

**RESEARCH QUESTION:**
{question}
**EXPECTED API TRAJECTORY:**
{expected_trajectory}
**EXPECTED ANSWER:**
{expected_answer}
**AGENT'S COMPLETE INTERACTION LOG:**
{agent_log}
**AGENT'S FINAL RESPONSE:**
{agent_final_response}

**EVALUATION CRITERIA:**
Evaluate whether the agent successfully completed the research task by considering ALL of the following aspects holistically:

1. **API Usage Appropriateness**: Did the agent use the right sequence of glycan APIs to address the research question? The agent should have used APIs that align with the expected trajectory, though exact matching is not required if the alternative approach is scientifically valid.

2. **Scientific Accuracy**: Are the obtained results scientifically correct and meaningful for glycomics research? Consider:
   - Correct interpretation of glycan structures and formats (WURCS, IUPAC, etc.)
   - Proper understanding of protein-glycan interactions
   - Accurate use of database cross-references
   - Valid biochemical reasoning

3. **Completeness**: Did the agent obtain all the information requested in the question? Check if:
   - All parts of multi-part questions were addressed
   - Required data fields were retrieved
   - Cross-references were properly resolved

4. **Data Integration**: If multiple API calls were needed, did the agent properly combine and interpret the results? Look for:
   - Logical workflow progression
   - Proper data passing between API calls
   - Meaningful synthesis of results from different sources

5. **Research Workflow**: Did the agent follow a logical research workflow consistent with glycomics best practices?
   - Started with appropriate tool discovery
   - Used documentation when needed
   - Handled errors gracefully
   - Drew appropriate conclusions

**IMPORTANT GUIDELINES:**
- Focus on whether the agent achieved the research objective, not just exact trajectory matching
- Consider that there may be multiple valid approaches to solve the same research question
- Evaluate the scientific validity of the final answer in the context of glycomics
- Account for API errors or limitations that may have affected the agent's performance
- Consider partial credit for agents that made significant progress but didn't fully complete the task
- Be especially attentive to:
  * Correct format conversions (e.g., IUPAC to WURCS)
  * Proper interpretation of molecular structures
  * Valid cross-database references
  * Meaningful biological insights

**RESPONSE FORMAT:**
Provide your evaluation in this exact format:

DECISION: [SUCCESS/FAILURE]

REASONING: [Detailed explanation of your decision, covering the key evaluation criteria and specific observations from the agent's performance. Include specific examples from the agent's log that support your decision.]

CRITICAL_ISSUES: [List any major problems that led to failure, or "None" if successful]

SCIENTIFIC_ACCURACY: [Assessment of the scientific validity of the results, including any concerns about data interpretation or biochemical reasoning]

API_USAGE_ASSESSMENT: [Evaluation of whether the agent used appropriate tools and followed a logical workflow]
'''
```