

DynamicToolBench: Evaluating the Adaptability of Agentic LLMs to Evolving APIs and Tool Environments

Pranav Pusarla

*Department of Computer Science
Columbia University
New York, USA
pp2903@columbia.edu*

Trisha Maturi

*Department of Computer Science
Columbia University
New York, USA
tm3530@columbia.edu*

Abstract—Tool-augmented large language models (LLMs) increasingly rely on external APIs to access real-world information and perform actions. While existing benchmarks evaluate tool use under static and fully aligned schemas, real-world APIs evolve continuously through versioning, refactoring, and undocumented changes. As a result, models often generate tool calls based on outdated interface assumptions while execution environments enforce updated specifications.

We introduce Dynamic ToolBench, a benchmark designed to evaluate LLM tool use under schema drift. Dynamic ToolBench decouples the API schema used for request generation from the schema enforced at execution time, modeling realistic version skew mediated by cached tool documentation. We apply controlled, semantics-preserving schema transformations—such as parameter renaming, type changes, and structural refactoring—and independently evaluate robustness to natural language variation through query perturbations.

Our experiments show that, while LLMs are largely robust to surface-level query perturbations, schema drift induces systematic and severe performance degradation, particularly for complex, long-horizon tool-use tasks. Allowing iterative request refinement via chain-of-thought does not reliably mitigate these failures. A detailed failure analysis reveals that schema drift disrupts tool use at the level of parameter grounding, type enforcement, and structural validation, exposing a gap between language understanding and interface adaptation.

These findings highlight the limitations of static tool-use evaluations and underscore the need for schema-aware, adaptive mechanisms in tool-augmented LLM systems.

I. INTRODUCTION

Large Language Models (LLMs) are increasingly deployed as tool-augmented systems, where external APIs are invoked to retrieve real-world information, execute actions, or interface with dynamic services [16] [9]. From travel booking and financial analysis to code execution and data retrieval, modern LLM applications rely heavily on tool calls to extend model functionality beyond static knowledge. As this dependency grows, the

correctness, latency, and robustness of tool usage have become first-order concerns in real-world deployments [3].

At the same time, tool environments are inherently non-stationary. APIs evolve continuously: endpoints are deprecated, parameters are renamed, response formats change, and error behaviors shift [4]. Even when functionality remains semantically similar, surface-level changes can break brittle tool-calling behavior. Unlike traditional software systems, LLMs often generate API requests implicitly from natural language and learned priors, making them especially vulnerable to such changes. As a result, tool-augmented LLM systems must operate under persistent drift between their internal expectations and the external tools they interact with. However, existing benchmarks largely evaluate tool use under static and idealized assumptions, where API documentation is fixed, schemas are stable, and tool interfaces perfectly match the model’s expectations [7] [10] [14]. This leaves an important open question: *Can LLMs keep up with real-world API fluctuation while maintaining reliable performance?*

In this work, we introduce Dynamic ToolBench, a benchmark designed to evaluate LLM tool use under realistic API evolution. Unlike prior benchmarks that assume a static tool interface, Dynamic ToolBench explicitly models version skew between what the model expects and what the execution environment enforces.

Concretely, the LLM generates API requests conditioned on an old version of the tool schema, reflecting outdated knowledge or prior exposure. At execution time, however, requests are validated against a new version of the API. To systematically study robustness, we introduce controlled schema-level perturbations, including parameter renaming and type changes, while preserving the underlying semantics of the tool. In addition, we independently evaluate query robustness by

perturbing user inputs through paraphrasing, distractors, casing variation, and punctuation noise.

Our contributions are threefold:

- 1) We propose Dynamic ToolBench, a benchmark that evaluates LLM tool use under realistic API schema drift induced by versioned tools and cached “latest” documentation.
- 2) We introduce a factorized robustness framework that independently perturbs user queries and tool schemas, enabling controlled analysis of failure modes.
- 3) We provide an empirical study of LLM behavior under schema drift, revealing systematic weaknesses in parameter grounding, type adaptation, and recovery from execution-time mismatches.

Together, our results suggest that reliable tool-augmented LLM systems require not only stronger reasoning capabilities, but also improved mechanisms for adapting to evolving tool interfaces.

II. RELATED WORK: TOOLBENCH AND STABLETOOLBENCH

Tool-augmented large language models extend their capabilities by invoking external APIs to retrieve real-world information or perform actions [12] [8] [13] [5]. To evaluate these capabilities, ToolBench [11] introduced a large-scale benchmark consisting of thousands of public APIs formatted into structured tool descriptions that could be consumed by LLMs. Given a natural language query and a set of tool APIs, the model was tasked with selecting the appropriate tool and generating a valid API request.

However, ToolBench relied on live public APIs, which proved to be brittle over time. When StableToolBench [2] was introduced, the authors observed that approximately 50% of the APIs in ToolBench had become unavailable, deprecated, or versioned, rendering the benchmark non-reproducible and increasingly invalid. This instability motivated the development of StableToolBench, which replaces live APIs with a virtualized, cached tool execution environment.

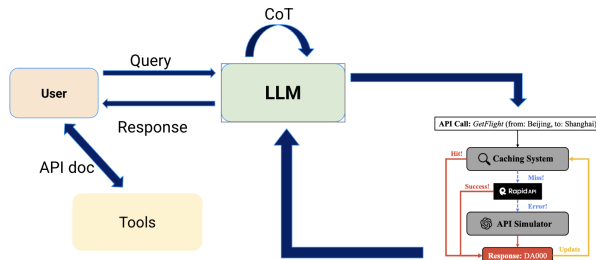


Fig. 1. Stable Tool Bench Architecture

StableToolBench 1 preserves the tool-call generation process of ToolBench while modifying how API calls

are executed and evaluated. At inference time, the LLM is provided with a user query and the corresponding API documentation, and is required to generate a structured API request that conforms to the tool schema. Rather than executing the request against a live service, StableToolBench routes the request through a cached execution system designed to ensure reproducibility and stability.

Execution proceeds as follows:

- 1) **Cache Lookup:** If an identical API request has been previously executed, StableToolBench returns a cached JSON response corresponding to that request.
- 2) **Response Simulation:** If the request is not present in the cache, StableToolBench invokes a separate LLM (GPT-4) to generate a plausible API response conditioned on the API documentation and the request arguments. This simulated response is then cached, ensuring that all future executions of the same request return the same result.

This design allows StableToolBench to achieve reproducibility, scalability, and stability, while abstracting away the non-determinism and fragility of live APIs. Although simulated responses may not perfectly reflect real-world behavior, StableToolBench prioritizes consistent evaluation of model tool-use capability rather than factual correctness of external data.

A. Iterative Tool-Use with Chain-of-Thought

In addition to single-shot tool invocation, StableToolBench supports iterative tool-use reasoning through a single chain-of-thought (CoT) trajectory [15]. After receiving execution feedback such as errors, invalid parameters, or unexpected responses, the model is allowed to revise and improve its API request within the same reasoning chain. This setup enables evaluation of not only initial tool-call correctness, but also the model’s ability to self-correct and refine requests based on feedback.

B. Limitations of StableToolBench

While StableToolBench addresses the instability of live APIs, it implicitly assumes a static tool environment. The API documentation provided to the model exactly matches the schema enforced during execution. As a result, StableToolBench evaluates whether models can correctly follow a known specification, but does not test whether they can adapt when tool interfaces change over time.

In real-world deployments, however, API schemas frequently evolve due to versioning, refactoring, or undocumented changes [17] [6] [1]. Models may generate requests based on outdated assumptions, while execution environments enforce updated schemas through cached or centralized documentation systems. This mismatch is

not captured by existing static benchmarks. Our work addresses this gap by explicitly modeling API evolution under cached execution, enabling systematic evaluation of tool adaptation rather than schema memorization.

III. METHODOLOGY

A. Dynamic ToolBench

We propose Dynamic ToolBench, a benchmark for evaluating tool-augmented large language models under realistic API schema drift. Dynamic ToolBench is designed to test whether models can adapt their tool usage when the interface enforced at execution time differs from the interface assumed during request generation.

The key design principle of Dynamic ToolBench is asymmetric schema exposure. During inference, the model generates an API request conditioned on an old version of the tool documentation. However, execution is governed by a new version of the API schema, which is treated as the authoritative, cached representation of the tool. This induces a controlled version skew between model expectations and execution constraints, mirroring real-world scenarios in which APIs evolve while cached documentation and tooling infrastructure enforce the latest specification.

Dynamic ToolBench builds on the virtualized execution infrastructure of StableToolBench, preserving reproducibility and scalability while introducing controlled non-stationarity in the tool environment.

B. System Architecture

Each evaluation instance in Dynamic ToolBench proceeds through the following stages:

- 1) **Query and Tool Conditioning:**

For each evaluation instance, the model is conditioned on a set of relevant APIs and a natural language user query. Each API is represented as a structured tool specification containing: category name, tool name, api name, api description, http method, required parameters, and optional parameters. This mirrors real-world tool-selection settings, where a model must reason over multiple candidate tools and identify which API is appropriate for a given task.

- 2) **API Request Generation:**

Conditioned on the user query and the old-version API schemas, the model generates an API request that invokes one of these functions with a structured argument dictionary. This request represents the model’s best attempt to satisfy the query under its assumed tool interface.

- 3) **Cached Execution with Updated Schema:**

API request execution is mediated by a cache-backed virtual execution environment, which en-

forces the new version of the API schema. Given a tool request:

- a) If an identical request is found in the cache, the cached JSON response is returned directly.
- b) If the request is not present in the cache, the system invokes a fake response generation function hosted on a server connected to a live environment.

The fake response generator receives:

- a) API examples: a list of (input, output) pairs illustrating expected behavior
- b) Tool input: the structured arguments provided by the model
- c) API documentation: the new-version schema enforced at execution time

Using these inputs, a separate LLM generates a plausible JSON response consistent with the updated API specification. This response is then cached, ensuring deterministic behavior for future identical requests.

To ensure robustness and comparability, responses are normalized, truncated to a maximum observation length, and mapped to standardized status codes representing execution outcomes such as success, timeout, unauthorized access, rate limiting, or hallucinated responses.

- 4) **Feedback and Iterative Refinement:**

Dynamic ToolBench supports iterative tool-use reasoning via a single chain-of-thought controller, which implements a linear, bounded reasoning trajectory. At each step, the model produces:

- a) a Thought, representing intermediate reasoning,
- b) an Action, typically a tool invocation or termination signal,
- c) and an Action Input, specifying structured tool arguments.

Each tool call is executed through the cached execution environment, and the resulting observation and status code are returned to the model. Based on this feedback, the model may revise its request, adapt parameter names or types, or terminate with a final answer.

The chain is constrained by a maximum depth and pruned when irrecoverable error codes are encountered. The controller tracks execution statistics including token usage, number of tool calls, success states, and failure modes. A special termination function is used to signal completion or controlled restarts. This setup enables evaluation of not only first-attempt correctness, but also recovery behavior under schema drift, where successful per-

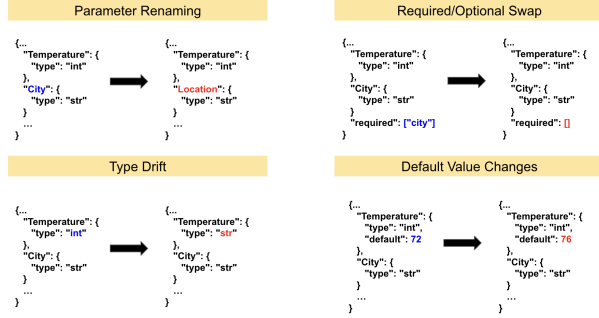
formance may require multiple refinement steps.

5) Final Response Evaluation:

Raw execution traces produced by the run pipeline are converted into a standardized, evaluator-friendly format. Each query is represented by a structured record containing the original query, the available tools, and an answer block with: the execution method, total number of steps, the final answer, and an execution graph capturing the sequence of tool calls and observations.

Evaluation scripts operate on this normalized representation to compute pass rates and robustness metrics. Automatic evaluators are run multiple times per example in parallel to account for stochasticity, and results are aggregated with mean performance and standard deviation reported across runs.

Fig. 2. Examples of schema drift perturbations applied



C. Modeling Schema Drift

Dynamic ToolBench models schema drift by generating versioned API interfaces that maintain semantic intent while altering surface-level structure. Given an original specification, we construct an evolved version through controlled stochastic transformations that mirror real-world API changes arising from versioning, refactoring, or undocumented updates. Let A_{old} denote the schema used during request generation and A_{new} the schema enforced at execution time. Drift is introduced via a transformation function

$$A_{new} = T(A_{old}) \quad (1)$$

where T consists of multiple schema-level perturbation operators.

1) URL and Endpoint Evolution: We restructure endpoint URLs to reflect API version increments, domain or path migrations, and changes to action names (e.g., `get` \rightarrow `fetch`). Although URLs are not directly produced by the LLM, such changes alter execution-time validation and distinguish old from new API versions.

2) Parameter Renaming: Required and optional parameters are renamed using mappings to semantically equivalent identifiers (e.g., `city` \rightarrow `location`). This preserves meaning while breaking exact string alignment and tests whether models rely on descriptions and feedback rather than memorized names.

3) Type Drift: To emulate relaxed or refactored typing, certain primitive types, specifically integers and booleans, are converted to strings. This requires models to adapt their argument formatting and handle implicit coercion during execution.

4) Required-Optional Parameter Swaps: We modify parameter necessity by swapping required and optional fields. These changes adjust validation constraints without altering functionality, probing whether models can infer which arguments are essential for successful execution.

5) Default Value Changes: Default behaviors are modified by flipping common defaults (e.g., `asc` \leftrightarrow `desc`, `true` \leftrightarrow `false`). Such silent changes test whether models rely on assumed defaults or explicitly specify values to ensure correct tool behavior.

6) Nested Object Introduction: Finally, we introduce nested objects by grouping related parameters into structured fields (e.g., combining multiple address components). This reflects common schema refactoring and tests whether models can adapt from flat to hierarchical argument structures.

D. Query Robustness Perturbations

Dynamic ToolBench also evaluates robustness to natural language variation in user queries, treated independently from schema drift to distinguish failures arising from linguistic noise rather than interface evolution. Given a query q , we generate a perturbed version \tilde{q} that preserves the underlying task intent while modifying surface form. All perturbations are applied before tool selection, and execution uses the same tool schemas as in the baseline.

Perturbations are designed to (1) preserve semantics and concrete values, (2) introduce only surface-level variation, and (3) reflect realistic user noise such as verbosity or formatting inconsistencies. We implement four categories:

Model-Based Paraphrasing: An auxiliary LLM generates meaning-preserving paraphrases to test whether models rely on brittle lexical cues or robust semantic understanding.

Distractor Injection: An unrelated narrative is prepended to the query to evaluate whether models can isolate the actionable instruction amid extraneous context.

Case Mixing: The query is transformed to uppercase, lowercase, or random character-level casing to test resilience to formatting inconsistencies found in informal text.

Punctuation Noise: Excess or missing punctuation and irregular line breaks are inserted to assess sensitivity to superficial syntactic disruptions.

IV. EXPERIMENTAL SETUP

A. Benchmarks and Task Groups

We evaluate Dynamic ToolBench on multiple instruction sets derived from StableToolBench, grouped by task difficulty and abstraction level. These include G1_category, G1_instruction, G1_tool, G2_category, G2_instruction, G3_instruction, using the definitions:

- G1: single-tool reasoning
- G2: intra-category multi-tool reasoning
- G3: intra-collection multi-tool reasoning
- Instruction: unseen instructions for the same set of tools in the training data
- Category: unseen tools that belong to a different (unseen) category of tools in the training data
- Tool: unseen tools that belong to the same (seen) category of the tools in the training data

B. Evaluation Protocol and Metrics

All experiments use Gemini-Flash-2.5. The cached virtual API execution environment is inherited from StableToolBench. Models are allowed to iteratively refine their API requests via a single chain-of-thought trajectory, subject to a maximum step limit.

We report Solvable Pass Rate, defined as the percentage of queries for which the model successfully produces a valid execution trace and final answer. Each configuration is evaluated multiple times, and we report mean performance with standard deviation.

V. RESULTS

A. Impact of API Versioning

Table I reports performance under baseline schemas and under API versioning.

Across all task groups, API versioning consistently reduces solvable pass rate, with degradation increasing as task complexity grows. While G1 tasks show relatively modest drops (e.g., G1_category: 24.8 to 23.0), more complex tasks experience substantial failures. In particular, G3_instruction exhibits a dramatic collapse from 14.2 to 3.3, indicating that long-horizon tool reasoning is especially brittle under schema drift.

These results suggest that current models rely heavily on exact schema alignment, and even semantics-preserving interface changes can significantly disrupt tool usage.

Fig. 3. API Versioning Solvable Pass Rate

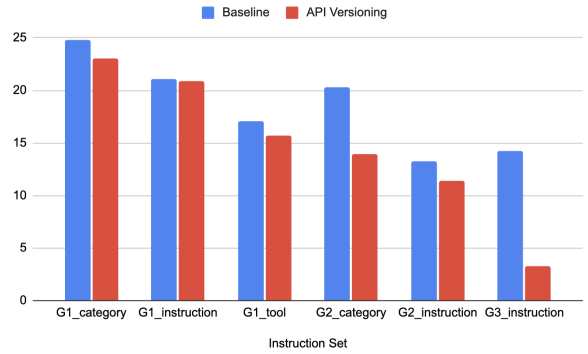
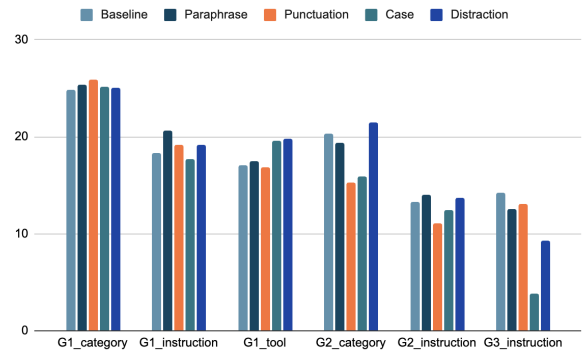


Fig. 4. Measure of Query Robustness Perturbations Solvable Pass Rate



B. Impact of Query Robustness Perturbations

Table II presents results for query perturbations applied independently of schema drift.

Overall, models exhibit substantially higher robustness to query perturbations than to schema drift. Performance under paraphrasing and distractor injection remains comparable to or slightly above baseline for several task groups (e.g., G1_category and G2_category). This indicates that models are generally able to preserve intent understanding under surface-level linguistic variation.

However, performance degradation becomes more pronounced for higher-complexity tasks. For example, G3_instruction shows notable sensitivity to case mixing and distractor stories, suggesting that long-horizon reasoning pipelines are more vulnerable to accumulated noise in the input.

C. Comparative Analysis

The results in Sections 5.1 and 5.2 reveal a clear qualitative distinction between linguistic robustness and schema robustness in tool-augmented LLMs. While both perturbation types introduce mismatch between model

TABLE I
API VERSIONING SOLVABLE PASS RATE

Modifications	G1_category	G1_instruction	G1_tool	G2_category	G2_instruction	G3_instruction
Baseline	24.8 \pm 1.9	21.1 \pm 0.4	17.1 \pm 0.9	20.3 \pm 2.4	13.3 \pm 0.8	14.2 \pm 0.8
API Versioning	23.0 \pm 1.8	20.9 \pm 1.3	15.7 \pm 2.0	13.9 \pm 0.9	11.4 \pm 1.3	3.3 \pm 0.0

TABLE II
QUERY ROBUSTNESS SOLVABLE PASS RATE

Modifications	G1_category	G1_instruction	G1_tool	G2_category	G2_instruction	G3_instruction
Baseline	24.8 \pm 1.9	21.1 \pm 0.4	17.1 \pm 0.9	20.3 \pm 2.4	13.3 \pm 0.8	14.2 \pm 0.8
Model Paraphrase	25.4 \pm 0.6	20.6 \pm 0.9	17.5 \pm 1.8	19.4 \pm 0.7	14.0 \pm 2.7	12.6 \pm 2.8
Punctuation Noise	25.9 \pm 0.8	19.2 \pm 2.4	16.9 \pm 0.3	15.3 \pm 0.7	11.1 \pm 1.6	13.1 \pm 1.3
Case Mix	25.2 \pm 1.1	17.7 \pm 1.2	19.6 \pm 1.0	15.9 \pm 0.4	12.4 \pm 0.8	3.8 \pm 0.8
Distractor Story	25.1 \pm 1.6	19.2 \pm 1.6	19.8 \pm 2.0	21.5 \pm 2.0	13.7 \pm 1.2	9.3 \pm 0.8

expectations and execution, they differ fundamentally in how they interact with the tool-use pipeline.

Query perturbations primarily affect intent recognition and tool selection, operating at the natural language interface. Once the correct tool is identified, downstream execution remains aligned with the expected schema. As a result, the model can often recover implicitly through semantic understanding, even when surface-level linguistic cues are altered.

In contrast, API versioning introduces mismatch at the execution interface itself, where parameter names, types, and validation constraints no longer align with the model’s internal assumptions. These failures propagate directly into tool invocation and cannot be resolved through semantic reasoning alone. Even when iterative refinement is permitted, the model lacks a reliable mechanism for inferring the updated schema structure from execution feedback.

This comparison highlights three key insights:

- Schema drift disrupts tool use at a deeper level than query variation, affecting the structural contract between the model and the execution environment rather than surface interpretation.
- Tool robustness is not implied by language robustness: strong performance under paraphrasing or noisy input does not indicate resilience to API evolution.
- Long-horizon tool reasoning amplifies schema mismatch, as errors in parameter grounding and validation compound across multiple steps.

Overall, these findings suggest that current tool-augmented LLMs are better equipped to handle variability in what users say than changes in how tools must be called. Addressing this gap will require mechanisms that explicitly model and adapt to evolving tool interfaces, rather than relying solely on improved language understanding.

VI. LIMITATIONS

While Dynamic ToolBench captures an important aspect of real-world tool evolution through schema drift, our evaluation focuses on a subset of possible API changes. In particular, we do not model operational failures such as rate limiting, latency spikes, or flaky endpoints, nor do we evaluate error drift arising from changes in error codes or messages. Additionally, our schema transformations are designed to be semantics-preserving; future work could explore settings where functional behavior itself evolves. Finally, while our experiments cover a diverse set of tool-use tasks, extending Dynamic ToolBench to additional domains and real-world APIs would strengthen external validity.

VII. CONCLUSION

We present Dynamic ToolBench, a benchmark for evaluating tool-augmented large language models under realistic API schema drift. By decoupling request-generation schemas from execution-time schemas and independently perturbing user queries, Dynamic ToolBench enables systematic analysis of robustness to both linguistic variation and tool interface evolution. Our results show that, despite strong robustness to query-level perturbations, current models remain brittle under schema drift, particularly for complex, multi-step tasks. These findings highlight the need for schema-aware and adaptive mechanisms in future tool-augmented LLM systems and motivate dynamic evaluation as a necessary complement to static tool-use benchmarks.

REFERENCES

- [1] DI LAURO, F., SERBOUT, S., AND PAUTASSO, C. To deprecate or to simply drop operations? an empirical study on the evolution of a large openapi collection. *Empirical Software Engineering* (2022).
- [2] GUO, Z., CHENG, S., WANG, H., LIANG, S., QIN, Y., LI, P., LIU, Z., SUN, M., AND LIU, Y. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models, 2025.
- [3] KONG, Y., RUAN, J., CHEN, Y., ZHANG, B., BAO, T., SHI, S., DU, G., HU, X., MAO, H., LI, Z., ZENG, X., AND ZHAO, R. Tptu-v2: Boosting task planning and tool usage of large language model-based agents in real-world systems, 2023.
- [4] LERCHER, A., GLOCK, J., MACHO, C., AND PINZGER, M. Microservice api evolution in practice: A study on strategies and challenges. *arXiv preprint* (2023).
- [5] LIU, X., YU, H., ZHANG, H., XU, Y., LEI, X., LAI, H., GU, Y., DING, H., MEN, K., YANG, K., ZHANG, S., DENG, X., ZENG, A., DU, Z., ZHANG, C., SHEN, S., ZHANG, T., SU, Y., SUN, H., HUANG, M., DONG, Y., AND TANG, J. Agentbench: Evaluating llms as agents, 2025.
- [6] McDONNELL, T. S., RAY, B., AND KIM, M. An empirical study of api stability and adoption in the android ecosystem. In *IEEE International Conference on Software Maintenance (ICSM)* (2013).
- [7] PATIL, S. G., MAO, H., JI, C.-J. C., YAN, F., SURESH, V., STOICA, I., AND GONZALEZ, J. E. The berkeley function calling leaderboard (bfc1): From tool use to agentic evaluation of large language models. In *Proceedings of the Forty-Second International Conference on Machine Learning (ICML)* (2025).
- [8] PATIL, S. G., ZHANG, T., WANG, X., AND GONZALEZ, J. E. Gorilla: Large language model connected with massive apis, 2023.
- [9] POLYAKOV, G., ALIMOVA, I., ABULKHANOV, D., SEDYKH, I., BOUT, A., NIKOLENKO, S., AND PIONTKOVSKAYA, I. Tool-Reflection: Improving large language models for real-world API calls with self-generated data. In *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)* (Vienna, Austria, July 2025), E. Kamaloo, N. Gontier, X. H. Lu, N. Dziri, S. Murty, and A. Lacoste, Eds., Association for Computational Linguistics, pp. 184–199.
- [10] 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. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.
- [11] 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. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.
- [12] SCHICK, T., DWIVEDI-YU, J., DESSÌ, R., RAILEANU, R., LOMELI, M., ZETTLEMOYER, L., CANCEDDA, N., AND SCIALOM, T. Toolformer: Language models can teach themselves to use tools, 2023.
- [13] SHEN, Y., SONG, K., TAN, X., LI, D., LU, W., AND ZHUANG, Y. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023.
- [14] TANG, Q., DENG, Z., LIN, H., HAN, X., LIANG, Q., CAO, B., AND SUN, L. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases, 2023.
- [15] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., ICHTER, B., XIA, F., CHI, E. H., LE, Q. V., AND ZHOU, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022).
- [16] XU, W., HUANG, C., GAO, S., ET AL. Llm-based agents for tool learning: A survey. *Data Science and Engineering 10* (2025), 533–563.
- [17] YANG, J., ET AL. A first look at the deprecation of restful apis: An empirical study. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2020).