Learning to Rewrite Tool Descriptions for Reliable LLM-Agent Tool Use

Ruocheng Guo* **Kaiwen Dong*** **Xiang Gao** **Kamalika Das**
Intuit AI Research, Mountain View, CA, USA
Kamalika_Das@intuit.com

Abstract

While most efforts to improve LLM-based tool-using agents focus on the agent itself — through larger models, better prompting, or fine-tuning — agent performance increasingly plateaus due to the quality of the tool interfaces these agents consume. Tool descriptions are often written for human developers and tolerate ambiguity that agents cannot resolve, particularly as the number of candidate tools grows. Existing approaches to improving tool interfaces (1) require re-running a multi-stage per-tool pipeline — synthesizing queries, executing an agent to collect trajectories, annotating trajectories, and prompting a strong LLM multiple times — for every API that enters the catalog, and (2) typically optimize each tool independently, limiting scalability and generalization to unseen tools. We propose **Trace-Free+**, a curriculum learning framework that progressively transfers supervision from trace-rich settings to trace-free deployment, encouraging the model to internalize reusable patterns of what makes a tool description effective. To support this approach, we construct a large-scale dataset of high-quality tool interfaces derived from real-world APIs through a principled data synthesis workflow. Experiments on widely adopted benchmarks show that Trace-Free+ improves robustness as tool catalogs scale to 150+ candidates — in scaling experiments, reducing accuracy degradation by 29.23% and improving average query-level success by 60.89% on StableToolBench — generalizes across domains without retraining, and provides complementary gains on top of agent fine-tuning.

1 Introduction

Most efforts to improve LLM-based tool-using agents focus on the agent itself — stronger foundation models (Team et al., 2025b; OpenAI, 2024; Google, 2025; Yang et al., 2025), better prompting (Spiess et al., 2025; Wu et al., 2024), or fine-tuning (Dong et al., 2025; Qi et al., 2025). As gains from scaling agents plateau, a fundamental bottleneck lies in the tool interfaces these agents consume. Existing tool descriptions are written for human developers: they tolerate ambiguity, leave constraints implicit, and assume background knowledge that agents cannot acquire (Hsieh et al., 2023). As the number of candidate tools grows into the hundreds, these interface deficiencies compound — agents face not just harder reasoning but noisier decision surfaces where poorly specified tools become indistinguishable from relevant ones (Qu et al., 2024).

Consider a concrete example (see Fig. 1): a scholarly API named *publication_year.find* with original description "Fetches the year a particular scientific work was published." When asked about Newton's law of universal gravitation, an agent passes the colloquial phrase "Law of Universal Gravitation" as `work_title` — a plausible but incorrect input. A description specifying that the API requires the "full title" leads the agent to the formal work title *Philosophiæ Naturalis Principia Mathematica*, producing a correct call.

Approaches such as DRAFT (Qu et al., 2025), Play2Prompt (Fang et al., 2025) and D2 (Section 3.2) require running a full per-tool pipeline for every new API: synthesizing realistic

*Equal contribution.

queries, executing an agent to collect success/failure traces, annotating trajectories against ground truth, and prompting a strong LLM multiple times to refine the description. In enterprise settings where API catalogs change frequently, this pipeline must be re-executed for each new tool — a recurring operational burden that is further compounded when traces cannot be collected due to cold-start, safety, or privacy constraints.

More critically, these methods optimize each tool in isolation: they do not learn transferable patterns of what makes a tool description effective, leading to poor generalization to unseen tools and degraded performance as candidate sets grow. Prompting-based methods such as EasyTool (Yuan et al., 2025) avoid trace dependence but similarly treat each tool independently — they cannot learn or transfer effective interface patterns across tools, and must re-invoke a strong LLM for every new tool at inference time.

A key hypothesis underlying our approach is that effective tool descriptions follow a bounded and reusable set of interface patterns (following the principle of information hiding (Parnas, 1972)). While the space of strategies an agent may learn to cope with arbitrary tools is effectively unbounded, the ways to specify a good interface—including scope definition, parameter constraints, output semantics, and dependency structure—are comparatively limited and recur across APIs. This asymmetry suggests that interface optimization can be learned as a transferable capability: instead of adapting agents to each tool, we can learn to rewrite tools into a form that is consistently interpretable by agents. We empirically validate this bounded-pattern hypothesis in Section 4.2, where we show that Trace-Free+ achieves up to 97.2% pattern coverage across five categories on 4,585 unseen tools, while the original descriptions (D_0) cover fewer than 12% in any category (Table 8). Rather than replacing agent fine-tuning, better interfaces reduce how often fine-tuning is needed and amplify its effectiveness when applied.

We propose Trace-Free+, a framework that operationalizes this insight by treating interface optimization as a learned, transferable capability. Execution traces provide rich supervision for learning what makes a description effective, but are unavailable for new tools at deployment time. Trace-Free+ resolves this tension through curriculum learning (Bengio et al., 2009): training begins with trace-based examples (easier, more supervision) and gradually transitions to trace-free examples where descriptions must be generated from the tool schema alone (harder, matching deployment conditions). This enables the model to internalize patterns from trace-rich settings and apply them to unseen tools without requiring any tool interaction. Unlike the per-tool pipelines described above, onboarding a new tool requires only its schema as input, with no query synthesis, trace collection, annotation, or rule extraction. We instantiate this approach using a large-scale dataset of high-quality tool interfaces derived from real-world APIs via a principled synthesis pipeline.

Extensive experiments demonstrate that Trace-Free+ achieves state-of-the-art results on StableToolBench (Guo et al., 2024) and RestBench (Song et al., 2023) in the trace-free setting, with all test tools unseen during training. Notably, in scaling experiments with up to 150+ candidate tools, it reduces accuracy degradation by 29.23% and improves query-level success by 60.89% on average, provides complementary gains on top of agent fine-tuning, and transfers to the Berkeley Function Calling Leaderboard (BFCLv2) (Patil et al., 2025) — lifting the state-of-the-art Gemini-3-pro-preview (Team et al., 2025a) by up to 1.4 points purely through better tool interfaces.

Our contributions are: (1) Trace-Free+, a curriculum learning framework transferring supervision from trace-rich training to trace-free deployment; (2) a large-scale dataset of high-quality tool interfaces derived from real-world APIs; and (3) state-of-the-art results on StableToolBench, RestBench, and BFCLv2 with all test tools unseen during training.

2 Problem Statement

We cast tool interface improvement as a supervised learning problem over a distribution of tools, training a description generator that internalizes effective interface patterns and transfers them to unseen tools at deployment time.

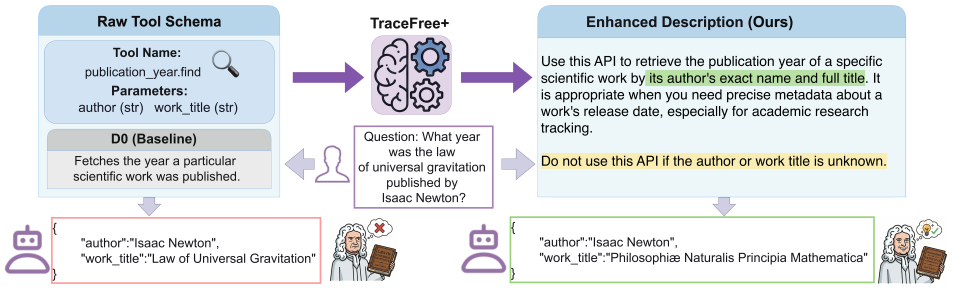

Figure 1: An illustration of the proposed tool interface improvement pipeline. Compared to the original description (D_0), the learned description generator produces more effective tool descriptions that lead to better tool usage.

Notation. A multi-step query decomposes into subtasks $h_t = (x_t, a_t, p_t, o_t)$, denoting the input x_t, tool a_t, parameters p_t, and output o_t fed into x_{t+1}. Each tool $a_i = \{d_i, s_i\} \in \mathcal{A}$ has description d_i and schema s_i; only descriptions are improved (schemas are held fixed after preprocessing). Interface quality is measured by $R(\mathcal{A}; \mathcal{Q})$, instantiated as subtask- and query-level success rates (Section 4.1). From $\mathcal{A}_{tr}$ with queries $\mathcal{Q}_{tr}$, we aim to produce improved descriptions d_i', optimizing $R(\mathcal{A}_{ts}; \mathcal{Q}_{ts})$ on held-out test tools $\mathcal{A}_{ts}$.

We evaluate three settings. **Trace-free**: given unseen tools $\mathcal{A}_{ts}$ without any execution, the generator produces improved interfaces from original tool — evaluated both *in-domain* (StableToolBench) and *cross-domain* (RestBench, BFCLv2, unseen API types). **Scaling**: candidate sets are augmented to 150+ tools, testing whether improved interfaces maintain their advantage as selection becomes harder. **Amplifying agent fine-tuning**: testing whether generated descriptions can provide additional gains on top of agent fine-tuning.

3 Methodology

Our goal is to improve unseen tools at deployment time without re-running the full per-tool pipeline that high-quality description generation requires: synthesizing queries for new APIs, executing an agent to collect traces, annotating them, and prompting strong LLMs multiple times to refine each tool description. This pipeline must be repeated for every new tool — a recurring operational cost that is prohibitive at scale and especially in cold-start, safety-critical, or privacy-constrained settings. To achieve deployment-time tool improvement, we train a model that internalizes what makes tool descriptions effective across a large set of tools and applies these patterns zero-shot from the schema alone. Existing methods cannot achieve this: they treat each tool independently and cannot learn cross-tool patterns, and trace-based refinement requires re-running the full pipeline for every new tool. We present the learning framework that enables this transfer (Section 3.1) and describe the data synthesis pipeline that produces the supervision for it (Section 3.2).

3.1 Trace-Free+: Curriculum Learning for Transferable Interface Optimization

A fundamental tension underlies tool interface improvement: execution traces provide the richest supervision for what makes a description effective—revealing valid tool use cases and parameter constraints—yet traces can be unavailable for new tools at deployment time due to cold-start, privacy, or safety constraints. The core question is therefore: *can a model trained with trace-based data learn to generate effective descriptions without traces at inference?*

Learning formulation and curriculum design. We cast tool interface improvement as a supervised learning problem over a distribution of tools. Given training tools $\mathcal{A}_{tr}$ and their improved descriptions d_i' (Section 3.2), we fine-tune an open-weight LLM to serve as a description generator. Each training example pairs an input—consisting of the original tool interface a_i and, optionally, a summary of execution traces $h_i = \text{Summary}(H(a_i))$—with the target improved description d_i'.

A naïve approach would train exclusively on one type of samples. Training only with traces creates a mismatch: the model learns to condition on information absent at deployment. Training only without traces discards the richest supervision signal observable only through

execution, such as which parameter formats cause errors or which tool combinations lead to ordering failure. Neither extreme allows the model to *understand* how traces inform effective descriptions and *abstract* those patterns for trace-free application.

Curriculum learning resolves this tension by structuring training to progressively bridge the gap between trace-rich supervision and trace-free deployment. Training begins with a higher proportion of trace-based examples, where the model learns the mapping from traces to effective descriptions—for instance, that a parameter named `ip_address` should specify "only IPv4 and IPv6 formats accepted." As training progresses, the proportion of trace-free examples increases until they dominate. In this phase, the model must generate the same quality of descriptions without traces, forcing it to internalize the *types* of patterns that matter—tool selection scope, cross-tool dependencies, parameter constraints—rather than relying on explicit trace evidence, as illustrated in Fig. 1 and Table 9. We denote this curriculum-trained model as **Trace-Free+**. For controlled comparison, we also train **Trace-Free** (trace-free examples only, no curriculum).

The curriculum encourages the model to abstract reusable patterns from trace-rich examples and apply them in trace-free contexts. These patterns fall into five categories (Table 7, Appendix C): (1) *tool selection scope* — when to use versus not use an API, and how it differs from similar tools; (2) *cross-tool dependencies* — parameter values that must come from a specific upstream endpoint; (3) *output description* — what fields and types the response contains or omits; (4) *parameter constraints* — valid formats, ranges, and enumerated values; and (5) *cross-parameter dependencies* — parameters that must be paired together or are mutually exclusive. Because training spans hundreds of diverse tools, the model encounters recurring instances of all five patterns and learns to anticipate them on entirely unseen tools without any tool interaction. For example, D_0 of a Walk Score API states only "Get Walk Score" and copies its format field verbatim from an unrelated movie API ("Type of result to return: (movie, series, episode)"). From the schema alone, Trace-Free+ infers that lat must be a decimal in [-90, 90], lon in [-180, 180], that bike and transit accept only the exact string '1' (not 'true' or 'on'), and that format must be 'json' or empty. It also adds scope exclusions: "Do not use for real-time traffic data or historical trends." None of these constraints were available in any execution trace — they were inferred purely from schema-level patterns learned during training. Additional examples appear in Table 9 (Appendix C).

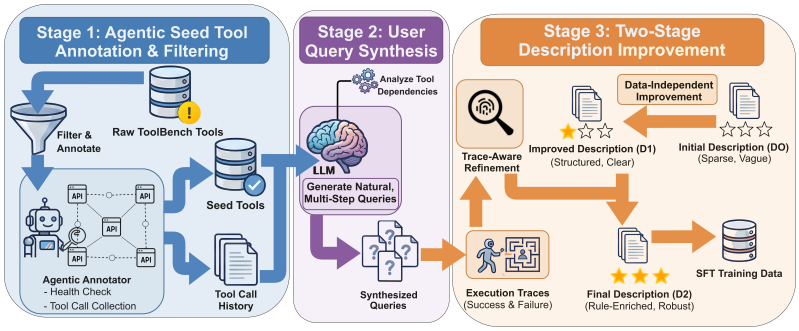

Figure 2: The data synthesis pipeline.

3.2 Data Synthesis for Tool Interface Improvement

We construct training data through a three-stage pipeline that converts real-world APIs into high-quality supervision for tool description generation. At a high level, we (1) collect working tool interfaces, (2) synthesize multi-step queries that expose interface deficiencies, and (3) generate improved descriptions that encode both general principles and trace-derived constraints. To support this curriculum, we require training data that: (1) coverage across a diverse tool distribution, so the model encounters generalizable tool description patterns rather than memorizing tool-specific fixes, and (2) high-quality target descriptions that encode these generalizable patterns. The two types of samples share the same target but differ in input: trace-based samples provide the original description, parameter schema together with a trace summary, while trace-free examples do not provide any trace summary, forcing the model to internalize patterns that compensate for the absence of traces. We

construct such data through a three-stage pipeline (Figure 2) over real-world APIs. Full details, prompts, and examples are in Appendix A.

Stage 1: Seed tool annotation and filtering. We source tools from ToolBench (Qin et al., 2023), spanning 49 categories of real-world RESTful APIs. An agentic annotator programmatically interacts with each provider to label endpoint health and record request–response examples, yielding 5,576 tools split into $\mathcal{A}_{ts}$ (4,585 tools appearing as candidates in StableToolBench test queries, held out entirely) and $\mathcal{A}_{tr}$ (991 remaining tools, 2,189 synthesized queries), ensuring no test tool is seen during training. Each $\mathcal{A}_{tr}$ tool contributes multiple trace-based training examples (one per synthesized query) plus one trace-free example, so the total number of training instances exceeds the number of tools; see Table 5 and Appendix A for full statistics and filtering criteria.

Stage 2: Dependency-aware query synthesis. Many interface deficiencies surface only through multi-step execution. We leverage API call histories from Stage 1 to identify inter-call dependencies: an LLM selects APIs forming a coherent workflow and generates a natural-language query requiring all selected APIs in sequence (details in Appendix A.2).

Stage 3: Two-stage description improvement. We run a tool-using agent on the synthesized queries and save successful and failed trajectories. Starting from original descriptions D_0, we first apply general documentation guidelines—specifying use cases and parameter constraints—to produce D_1. We then refine D_1 with general rules extracted from failure traces via RIMRULE (Gao et al., 2025)—e.g., acceptable value formats and undocumented preconditions—yielding D_2. The resulting D_2 descriptions serve as the supervision target for both curriculum branches; what differs is the input the model receives at training time.

While D_2 is generated per-tool, Trace-Free+ trained on these examples across hundreds of tools abstracts cross-tool patterns—when to use, recurring parameter constraints, output format, effective documentation strategies—and applies them zero-shot to unseen tools. At the same time, Trace-Free+ does not aim to outperform D_2, but to match its quality without incurring its per-tool cost, enabling scalable deployment in settings where running D_2 is infeasible. Per-tool prompting methods (Yuan et al., 2025; Qu et al., 2025; Fang et al., 2025) cannot leverage such patterns because they improve each tool in isolation.

4 Experiments

We evaluate Trace-Free+ across the following dimensions: (1) **trace-free generalization**—whether a model trained on one set of tools can produce effective descriptions for unseen tools without traces, including **cross-domain transfer** to tools and benchmarks outside the training distribution (see Section 4.2); (2) **scaling robustness**—whether improved descriptions maintain their advantage as candidate sets grow to 150+ tools; (3) **amplifying agent fine-tuning**—whether our method provides additional gains on top of agent fine-tuning. In all settings, Trace-Free+ and its variants are evaluated on *tools unseen during training*.

4.1 Experimental Setup

Benchmarks. Our in-domain benchmark is StableToolBench (Guo et al., 2024) with six subsets. They have single-step (G1) and multi-step (G2–G3) queries of increasing difficulty, totaling 764 solvable queries over 4,585 candidate tools ($\mathcal{A}_{ts}$). The training set $\mathcal{A}_{tr}$ comprises 991 tools with 2,189 synthesized queries. Following (Lu et al., 2025), we correct a subset of parameter schemas that are inconsistent with server requirements (Appendix B.2). For cross-domain transfer, we test on RestBench (Song et al., 2023) — TMDB (100 queries, 54 tools) and Spotify (57 queries, 40 tools) — and BFCLv2 (Patil et al., 2025) (1,390 Non-Live and 2,251 Live instances), both out-of-domain. Full benchmark statistics are in Table 4.

Baselines and agents. We focus on the **trace-free** setting, where we compare against the original descriptions (D_0), the prompting-improved descriptions (D_1), and EasyTool (Yuan et al., 2025). Our primary tool-using agent is GPT-4.1; to test whether improved descriptions generalize across agents, we additionally evaluate with Qwen3-4B-Instruct (Yang et al., 2025) in both its base and fine-tuned variants.

Evaluation protocol. We introduce step-wise teacher-forcing evaluation to isolate the effect of tool descriptions from compounding execution errors. At each step, the ground-truth API is called regardless of the agent's selection, ensuring that subsequent steps receive correct intermediate context. This design guarantees that any subtask failure reflects the agent's misunderstanding of the current tool's description rather than corrupted context from earlier mistakes, enabling cleaner error attribution across methods. In addition, to better reflect real-world deployment, the scaling experiments use a non-teacher-forcing setting where the agent's own selected tool is executed at each step. We report *subtask-level (SL)* and *query-level (QL)* success rates: a subtask succeeds if the correct tool is selected and execution completes successfully; a query succeeds iff all its subtasks succeed. Both metrics are based on ground truth and do not rely on LLM-as-a-judge.

4.2 Trace-free Evaluation

The trace-free evaluation tests generalizability: the trained generator is applied to unseen $\mathcal{A}_{\text{ts}}$ without traces, with GPT-4.1 as the primary agent. Cross-domain evaluation covers RestBench (Song et al., 2023) (TMDB and Spotify) and BFCLv2 (Patil et al., 2025), with Claude Sonnet 4.5 and Gemini-3-pro-preview additionally included for BFCLv2 to test agent-agnostic transfer. Results with Qwen3-4B-Instruct and agent fine-tuning are in Section 4.4.

In-domain results. Table 1 reveals several patterns. First, Trace-Free+ significantly outperforms Trace-Free across most subsets, confirming that curriculum learning—which exposes the model to trace-based supervision before transitioning to trace-free generation—transfers knowledge that pure trace-free training cannot acquire.

Second, the split averages reveal where description quality matters most. On multi-step queries (G2+G3), Trace-Free+ achieves 44.6 QL, improving over D_0 by 11.1 points and over D_1 by 3.1 points. On single-step queries (G1), Trace-Free+ is within 0.9 QL of D_1 (60.7 vs. 61.6). The G1 gap partly reflects a training data distribution choice: our synthesized queries require 3 tools on average, prioritizing practical multi-step agentic settings at the cost of underrepresenting single-step patterns. We leave augmenting training data with single-step queries as future work. In practice, Trace-Free+ is the choice for large catalogs or multi-step workflows while D_1 can be used for single-step queries.

Third, the multi-step advantage stems from Trace-Free+'s ability to internalize cross-tool patterns that D_1 cannot capture. D_1 applies fixed heuristics independently per tool, relying on whatever the original schema says about inter-API relationships. Trace-Free+, by contrast, has learned from real execution traces across hundreds of training tools which APIs produce outputs that feed into others, what parameter formats cause failures, and how scope boundaries interact across endpoints. Multi-hop queries amplify this difference: an imprecise constraint at step 1 cascades to all subsequent steps, compounding D_1's per-tool blind spots. Our case study (Table 8, Appendix C) confirms this mechanistically: Trace-Free+ achieves higher parameter constraint coverage (94.2% vs. 87.9%) and cross-parameter dependency coverage (17.1% vs. 8.5%) than D_1—the two categories most directly responsible for correct argument construction in multi-step chains. EasyTool performs below D_0 because its manually optimized prompts were designed for older models (ChatGPT, Vicuna-30B) and do not transfer to the newer agents used here, consistent with results in Fang et al. (2025).

To empirically validate the concentrated set of reusable interface patterns, we classify all descriptions on $\mathcal{A}_{ts}$ tools (Table 8, Appendix C). Original descriptions (D_0) cover fewer than 12% of tools in any of the five pattern categories. Trace-Free+ raises coverage to 97.2% for tool selection scope and 94.2% for parameter constraints — the two categories directly responsible for correct tool selection and execution — while also reaching 30.0% for cross-tool dependencies and 17.1% for cross-parameter dependencies (vs. D_1's 8.5%). D_1 achieves near-complete output description coverage (98.6%) because its template guidelines explicitly enumerate output fields, whereas Trace-Free+ encodes output information implicitly; this gap does not substantially affect downstream performance because output description primarily helps agents interpret responses rather than select tools or construct arguments. While the static performance gap between Trace-Free+ and D_1 on single-step queries is modest, Trace-Free+ outperforms D_1 substantially as catalogs scale, with the widening

concentrated on multi-step queries where description quality compounds across steps (Section 4.3). A parameter study on the trace-free data ratio is reported in Appendix D.

Cross-domain results on RestBench and BFCLv2. We test whether the learned patterns transfer beyond the training domain. Models fine-tuned on $\mathcal{A}_{\text{tr}}$ of StableToolBench are evaluated on the TMDB and Spotify datasets of RestBench (Song et al., 2023), with all API tools from each dataset included as candidates. As shown in Table 2, Trace-Free+ outperforms all baselines by a substantial margin, achieving up to +51.3% relative improvement over D_0 on TMDB query-level success rate and +41.3% on Spotify. These gains reflect the domain-agnostic nature of effective interface patterns: with APIs from an entirely unseen benchmark, the underlying improvements—clearer scope boundaries, explicit parameter constraints, disambiguation of overlapping endpoints—transfer directly without any retraining.

To further stress-test cross-domain transfer, we apply Trace-Free+ to the BFCLv2 (Patil et al., 2025) as the number of candidate tools per query is large enough. It is a benchmark that evaluates function calls using AST-based verification—a fundamentally different evaluation paradigm from the teacher-forcing protocol used above. We evaluate on both Non-Live and Live splits using three strong proprietary models: GPT-4.1, Claude Sonnet 4.5, and Gemini-3-pro-preview. As shown in Table 11 (Appendix D), Trace-Free+ consistently improves all three models on both splits purely through better tool interfaces, without any modification to the agent models themselves. The largest absolute gain is on Gemini-3-pro-preview Live (1.68% relative improvement, or +1.43 points), lifting the state-of-the-art to 86.41%. Claude Sonnet 4.5 benefits the most in relative terms (3.79% relative improvement, or +2.39 absolute points)). These results confirm that tool interface optimization is model-agnostic: the same descriptions improve GPT-4.1, Claude Sonnet 4.5, and Gemini-3-pro-preview without modification to any agent.

Table 1: Trace-free evaluation on StableToolBench for multi-step (G2+G3) and single-step (G1) subsets. SL/QL: subtask/query-level success rate. GPT-4.1 is the tool-using agent.

	Multi-Step (G2+G3)								**Single-Step (G1)**							
	G2 Category		G2 Instruction		G3 Instruction		**Avg**		G1 Category		G1 Instruction		G1 Tool		**Avg**	
	SL	QL	SL	QL	SL	QL	SL	QL	SL	QL	SL	QL	SL	QL	SL	QL
Trace-Free+	**68.7 ± 0.7**	**50.8 ± 0.0**	**71.4 ± 1.1**	**47.0 ± 0.1**	**63.9 ± 1.8**	36.1 ± 0.1	**68.0**	**44.6**	73.8 ± 0.7	**65.6 ± 1.0**	72.6 ± 1.5	60.0 ± 1.0	70.0 ± 1.4	**56.4 ± 1.6**	72.1	60.7
Trace-Free	66.4 ± 1.2	44.1 ± 0.0	69.2 ± 4.0	46.4 ± 3.6	59.9 ± 1.2	**41.8 ± 1.2**	65.2	44.1	71.8 ± 2.3	62.7 ± 2.1	70.8 ± 0.1	60.8 ± 0.9	68.5 ± 2.7	53.6 ± 2.8	70.4	59.0
D1	67.9 ± 1.2	48.5 ± 0.1	67.4 ± 1.0	45.5 ± 0.1	45.2 ± 4.2	30.6 ± 0.1	60.2	41.5	**75.5 ± 1.3**	64.9 ± 1.3	**74.7 ± 0.7**	**66.1 ± 0.6**	68.0 ± 0.6	53.7 ± 0.5	**72.7**	**61.6**
D0	68.4 ± 0.3	39.0 ± 0.0	67.8 ± 0.8	43.9 ± 0.1	50.7 ± 0.2	17.6 ± 0.1	62.3	33.5	73.0 ± 0.3	62.4 ± 1.5	72.8 ± 0.4	62.3 ± 0.9	**71.0 ± 0.9**	52.8 ± 1.1	72.3	59.2
EasyTool	68.1 ± 0.7	40.4 ± 1.3	68.2 ± 0.6	40.4 ± 0.1	41.4 ± 1.7	31.6 ± 0.2	59.2	37.5	67.4 ± 1.1	56.9 ± 0.7	69.5 ± 1.2	56.0 ± 2.3	66.4 ± 1.3	48.7 ± 0.2	67.8	53.9

4.3 Scaling Experiments

In practice, agents are routinely exposed to large, uncurated tool catalogs—yet performance degrades sharply as the candidate pool grows, since poorly specified descriptions become indistinguishable and selection errors compound across steps (Qu et al., 2024). Most benchmarks, including StableToolBench, evaluate on small curated tool sets that mask this failure mode; here, we directly test robustness as candidate sets scale to 150+ tools.

For this setting, we augment each query in StableToolBench with additional tool candidates. Specifically, we consider three types of additional tool candidates: (1) relevant tools from the same category, (2) relevant tools from other categories, (3) random APIs from other categories. We let 10% of the candidate set be type (1) and (2) and 90% be type (3) to avoid making API selection too hard, resulting in small difference in performance across descriptions. Unlike prior scaling studies (Qin et al., 2023; Yuan et al., 2025; Qu et al., 2025) that evaluate only the effectiveness of tool retrievers as a separate stage, we directly expose the full candidate set to agents and measure the performance end-to-end. This setting better reflects practical usage and avoids a fixed retrieval stage prior to agent execution, which is increasingly unnecessary given the large context windows supported by modern LLMs. For example GPT-4.1 and Gemini 2.5 Flash support context windows of up to one million tokens. To better reflect realistic deployment, we use a non-teacher-forcing setting for scaling experiments where the agent's selected tool is executed at each step and descriptions are penalized if the number of decomposed tool calls differs from the ground truth.

Fig. 3 shows that Trace-Free+ outperforms the baselines and is more robust against an increasing number of additional APIs across the three multi-step subsets of StableToolBench.

Table 2: Trace-free evaluation results on RestBench – TMDB and Spotify. SL: subtask-level, QL: query-level. No tool execution traces are available at inference time.

	RestBench			
	TMDB		Spotify	
Method	SL	QL	SL	QL
Trace-Free+	**88.1 ± 0.4**	**74.9 ± 0.5**	**68.1 ± 0.3**	**49.3 ± 0.6**
Trace-Free	78.4 ± 1.1	57.7 ± 1.2	65.0 ± 1.6	44.7 ± 2.8
D1	78.2 ± 0.1	58.0 ± 0.8	65.1 ± 0.8	45.7 ± 1.8
D0	69.8 ± 0.2	49.5 ± 0.2	57.1 ± 2.9	34.9 ± 2.1
EasyTool	76.4 ± 0.1	52.5 ± 0.0	63.4 ± 0.8	43.2 ± 0.2

Table 3: Query-level success rate with base and fine-tuned Qwen3-4B-Instruct agents.

Setting	**StableToolBench**	**RestBench**
Base + D0	37.4	54.1
Base + Trace-Free+	40.1 (+ 10.7%)	62.1 (+ 14.8%)
Fine-tuned + D0	40.0 (+ 10.7%)	55.0 (+ 1.0%)
Fine-tuned + Trace-Free+	**41.8 (+ 11.7%)**	**62.9 (+ 16.3%)**

From 0–150 additional tools, Trace-Free+ reduces performance degradation of D_0 by 29.23% and improves over D_0 by 60.89% on average. This robustness stems from description quality: well-specified scope boundaries and parameter constraints help agents filter signal from noise as the candidate pool grows—a structural advantage that per-tool rewriting methods cannot provide. Notably, these are the largest gains observed in our evaluation, suggesting that interface optimization is most valuable when tool catalogs are large. This scalability advantage is unique to learned optimization: D_1 applies fixed heuristics per tool independently and cannot capture cross-tool patterns, whereas the SFT model internalizes recurring effective interface patterns from hundreds of training tools—enabling it to capture patterns such as cross-parameter dependencies and implicit constraints that are not explicitly encoded in D_1's rule set—patterns that compound across steps as catalogs grow.

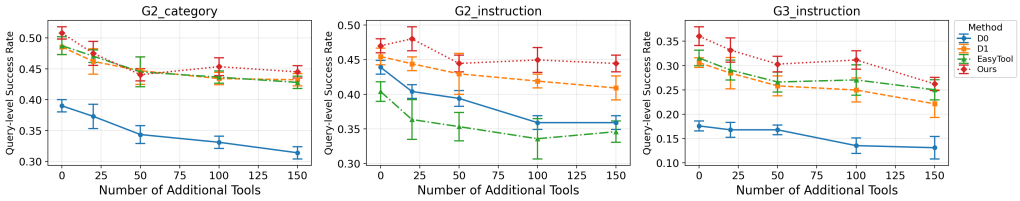

Figure 3: Scaling experiment results on the more challenging G2-G3 subsets of StableToolBench. We report query-level (QL) results.

4.4 Amplifying Agent Fine-tuning

Interface optimization is not a replacement for agent fine-tuning—rather, by fixing the tool interface layer first, it reduces the burden on the agent and acts as a force multiplier when fine-tuning is applied. We fine-tune Qwen3-4B-Instruct on $\mathcal{A}_{tr}$ of StableToolBench and evaluate all combinations of base/fine-tuned agent with D_0/Trace-Free+ descriptions on both StableToolBench and RestBench; Table 3 reports average query-level success rates.

On StableToolBench, agent fine-tuning and Trace-Free+ yield individually comparable gains (**+10.7% each over** D_0). Combining the two further improves performance to **41.8%**, a **+11.7%** relative gain over the baseline, indicating that description improvement and agent fine-tuning capture complementary aspects of tool-use competence.

The amplification is more pronounced in the cross-domain setting. On RestBench, agent fine-tuning brings the baseline from 54.1% to only 55.0%, reflecting limited cross-domain transfer—the agent adapts to StableToolBench's tool distribution but cannot generalize to RestBench's different API vocabulary. In contrast, Trace-Free+ improves the base model to **62.1% (+14.8% relative)**, and combining both strategies reaches **62.9% (+16.3% relative)**, since interface patterns (scope boundaries, parameter constraints) are domain-agnostic by nature. Together, these results confirm that interface optimization and agent fine-tuning

address complementary failure modes, with the combination most impactful in cross-domain deployment where fine-tuning alone is insufficient.

5 Related Work

Tool-using LLM Agents. Tool-using agents combine LLMs (OpenAI, 2024; Google, 2025; Yang et al., 2025; Team et al., 2025b) with external tools to extend capabilities beyond text generation (Qin et al., 2023; Schick et al., 2023; Wang et al., 2025). The LLM serves as the controller (Yao et al., 2023), deciding when and how to invoke tools. With external tools, an LLM can retrieve up-to-date information, perform calculations, and interact with external services (Huang et al., 2025; He et al., 2025). Early work such as Gorilla (Patil et al., 2024) demonstrated that LLMs can be trained to call massive API sets, while ToolLLM (Qin et al., 2023) extended this to 16,000+ real-world APIs. We focus on domain-specific APIs (e.g., from RapidAPI, TMDB, and Spotify), which require structured input arguments and return domain-specific outputs — making correct selection and execution more challenging than general-purpose tools, and making accurate tool descriptions especially critical.

Tool Interface Improvement. Tool interfaces are important in guiding agents in tool selection and usage (Xu et al., 2023; Hsieh et al., 2023; Bandlamudi et al., 2025; Chen et al., 2025; Faghih et al., 2025; Wölflein et al., 2025). A complementary line of work improves agents themselves via fine-tuning or contrastive reasoning (Wu et al., 2024; Dong et al., 2025); our approach is orthogonal, targeting the interface layer rather than the agent. Prompting-based methods have shown promising results: EasyTool (Yuan et al., 2025) addresses inconsistency, redundancy, and incompleteness via a two-step rewriting workflow; Play2Prompt (Fang et al., 2025) uses single-hop execution traces to improve descriptions with a strong LLM; and DRAFT (Qu et al., 2025) iteratively collects traces and applies LLM self-correction to revise interfaces. However, all three rely on per-tool trace collection and optimize each tool independently, preventing them from learning generalizable patterns across tools. DRAFT and Play2Prompt additionally cannot handle unseen tools without execution traces.

6 Limitations

Base model for description generation. Our experiments use Qwen3-4B-Instruct as the description generator for its strong instruction-following ability. We leave extension to larger open-weight models as future work. Improvements to the base model are orthogonal to our contributions — the curriculum learning framework and the large-scale dataset. A stronger base model may improve description quality further without replacing either.

Single-step query performance. Trace-Free+ does not outperform D_1 on the single-step G1 subset of StableToolBench. This is a consequence of a training data distribution choice: our synthesized queries require 3 tools on average, prioritizing the practical agentic system at the cost of underrepresenting single-step patterns. As a result, D_1's data-independent guidelines, which are applied uniformly regardless of query complexity, capture most of the available headroom on G1 while D_0 is already near the ceiling. Augmenting the training set with single-step queries is a direction for closing this gap (Section 4.3).

Precision of inferred constraints. It is challenging to systematically measure the precision of constraints generated by Trace-Free+. While the case studies in Appendix C show correct inferences across diverse constraint types, the model may occasionally hallucinate constraints for unfamiliar APIs — for instance, inventing value ranges not enforced by the server. Quantifying hallucination rates across constraint categories and developing verification mechanisms are important directions for future work.

7 Conclusion

Our results suggest that effective tool descriptions follow learnable, transferable patterns that existing per-tool methods cannot exploit. We introduce Trace-Free+, a curriculum learning framework that transfers supervision from trace-rich training to trace-free deployment, paired with a large-scale dataset of high-quality interfaces derived from real-world APIs. Experiments show that in scaling experiments with up to 150+ candidate tools, Trace-Free+

reduces accuracy degradation by 29.23% and improves query-level success by 60.89%, generalizes to unseen domains (RestBench, BFCLv2) without retraining, and acts as a force multiplier alongside agent fine-tuning — with the largest standalone gains appearing in cross-domain settings where fine-tuning alone shows limited transfer. These results suggest that interface quality is an under-exploited axis of agent improvement, and that learned interface optimization can complement advances in model capability and agent training. For practitioners, this yields a deployment strategy: D_1 suffices for simple queries, small-catalog settings, while Trace-Free+ is better for large tool catalogs with multi-step workflows.

References

Jayachandu Bandlamudi, Ritwik Chaudhuri, Neelamadhav Gantayat, Sambit Ghosh, Kushal Mukherjee, Prerna Agarwal, Renuka Sindhgatta, and Sameep Mehta. A framework for testing and adapting rest apis as llm tools. *arXiv preprint arXiv:2504.15546*, 2025.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, 2009.

Yi-Chang Chen, Po-Chun Hsu, Chan-Jan Hsu, and Da-shan Shiu. Enhancing function-calling capabilities in llms: Strategies for prompt formats, data integration, and multilingual translation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: Industry Track)*, pp. 99–111, 2025.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, et al. Agentic reinforced policy optimization. *arXiv preprint arXiv:2507.19849*, 2025.

Kazem Faghih, Wenxiao Wang, Yize Cheng, Siddhant Bharti, Gaurang Sriramanan, Sriram Balasubramanian, Parsa Hosseini, and Soheil Feizi. Gaming tool preferences in agentic llms. *arXiv preprint arXiv:2505.18135*, 2025.

Wei Fang, Yang Zhang, Kaizhi Qian, James Glass, and Yada Zhu. Play2prompt: Zero-shot tool instruction optimization for llm agents via tool play. *arXiv preprint arXiv:2503.14432*, 2025.

Xiang Gao, Yuguang Yao, Qi Zhang, Kaiwen Dong, Avinash Baidya, Ruocheng Guo, Hilaf Hasson, and Kamalika Das. Rimrule: Improving tool-using language agents via mdl-guided rule learning. *arXiv preprint arXiv:2601.00086*, 2025.

Google. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pp. 11143–11156, 2024.

Yichen He, Guanhua Huang, Peiyuan Feng, Yuan Lin, Yuchen Zhang, Hang Li, et al. Pasa: An llm agent for comprehensive academic paper search. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11663–11679, 2025.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*, 2023.

Kexin Huang, Serena Zhang, Hanchen Wang, Yuanhao Qu, Yingzhou Lu, Yusuf Roohani, Ryan Li, Lin Qiu, Gavin Li, Junze Zhang, et al. Biomni: A general-purpose biomedical ai agent. *biorxiv*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Yifei Lu, Fanghua Ye, Jian Li, Qiang Gao, Cheng Liu, Haibo Luo, Nan Du, Xiaolong Li, and Feiliang Ren. Codetool: Enhancing programmatic tool invocation of llms via process supervision. *arXiv preprint arXiv:2503.20840*, 2025.

OpenAI. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37: 126544–126565, 2024.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In *The Thirteenth International Conference on Learning Representations*, 2025.

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. October 2023. URL https://openreview.net/forum?id=dHng2O0Jjr.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Towards completeness-oriented tool retrieval for large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, pp. 1930–1940, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704369. doi: 10.1145/3627673.3679847. URL https://doi.org/10.1145/3627673.3679847.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. From exploration to mastery: Enabling llms to master tools via self-driven interactions. In *The Thirteenth International Conference on Learning Representations*, 2025.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 'smolagents': a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36: 68539–68551, 2023.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*, 2023.

Claudio Spiess, Mandana Vaziri, Louis Mandel, and Martin Hirzel. Autopdl: Automatic prompt optimization for llm agents. *arXiv preprint arXiv:2504.04365*, 2025.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report, 2025a. URL https://arxiv.org/abs/2503.19786.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025b.

Maolin Wang, Yingyi Zhang, Bowen Yu, Bingguang Hao, Cunyin Peng, Yicheng Chen, Wei Zhou, Jinjie Gu, Chenyi Zhuang, Ruocheng Guo, et al. Function calling in large language models: Industrial practices, challenges, and future directions. *ACM Computing Surveys*, 2025.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. Llm agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26092–26130, 2025.

Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis N Ioannidis, Karthik Subbian, Jure Leskovec, and James Zou. Avatar: Optimizing llm agents for tool usage via contrastive reasoning. *Advances in Neural Information Processing Systems*, 37:25981–26010, 2024.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*, 2023.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments, 2025. URL https://arxiv.org/abs/2510.01179.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. Easytool: Enhancing llm-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 951–972, 2025.

A Implementation Details

A.1 Detailed Agentic Tool Annotator

The prompt of the agentic tool annotator can be found below.

Agentic Tool Annotator

```
<system_prompt>
You are an agent that explores APIs to evaluate their health and discover working call patterns. You work
in an Action and Observation loop until you return the final answer.

What you have
1) The JSON string of the current MCP schema for one API provider (initially).
2) A set of tools (APIs) defined by that schema that you can call to get concrete feedback.
3) Generic utilities that annotate the schema with health labels and successful call examples.

Your goal
- For each API in the provider, actively explore how to call it successfully.
- Use both the schema to infer true parameter names, types, and constraints.
- Adapt your calls when you see errors, instead of giving up after a single failure.
- After a reasonable amount of exploration for each API, decide its health and record at least one
successful example call when possible.

What to annotate
- You must not change any tool name or delete tools.
- You should not rewrite the schema; instead, you infer how to call each API in practice.
- For each API, use `utility_annotate_health` to set:
  - `health = "good"` if you can find at least one meaningful, repeatable, successful call that returns
plausible data.
  - `health = "bad"` if, after careful attempts and adapting to error messages, all calls fail due to
issues you cannot fix from the client side (for example, persistent authorization errors, missing server
configuration, 404 for the endpoint, or fundamentally broken behavior).
  - `health = "unknown"` if you cannot confidently determine whether the API works (for example, ambiguous
 or inconsistent errors, or not enough steps left to explore).
- When you mark an API as `"good"`, you should, whenever possible, also call `utility_annotate_example` to
 store one or more concrete successful call examples.
  - The `example` input must be a JSON string representing a list of argument objects, such as `[{"param1
": "value1"}, {"param1": "value2", "param2": 3}]`.
  - Each object corresponds to a full set of arguments that you actually used in a successful call.
  - Prefer 1-3 diverse, minimal, safe examples that future agents can reuse directly.

How to explore APIs
- Prefer live Observations over assumptions.
- If a call fails with "Missing required parameter X" or similar, try again including that parameter.
- If a call fails with "Unexpected parameter Y" or type errors, adjust or remove that parameter.
- When behavior is unclear, run a minimal, low-risk test call to probe the interface rather than guessing
wildly.
- Be efficient: you have a limited number of tool calls. Do not brute-force huge parameter spaces. Instead
, reason about likely values based on descriptions.

Protocol
- Every step you take is an Action. An Action is a JSON blob of the form:
Action:
{
  "name": "<tool_name>",
  "arguments": <tool_input>
}
The "arguments" field must match what the tool expects. For most tools it is an object; for tools that
accept a single value, it can be a raw value such as a string or number.
- Use the utility tools to write your findings back into the schema:
  - `utility_annotate_health`: Record the health label and a concise reason for each API.
  - `utility_annotate_example`: Record JSON examples of successful calls for APIs you understand well.
- After each Action, you will receive an Observation. Treat it as ground truth. The Observation may be
plain text or structured JSON. Use it to decide the next Action.
- Use only listed tools. Do not invent tool names or parameters. Pass literal values, not variable names.

Required loop
1) Inspect the current schema to list all APIs you need to evaluate.
2) For each API, design and execute a small sequence of test calls to discover a working call (or to
conclude that the API is broken or uncertain).
3) Update your calling strategy for that API whenever you observe new errors or unexpected behavior.
4) Once you have enough evidence, annotate the API's health and, if possible, record successful call
examples using the utility tools.
5) Repeat until all APIs in the schema are annotated.
6) End by calling the "final_answer" tool with a concise summary of what you annotated.

Output and completion
- You must finish by calling the "final_answer" tool. It is the only way to complete the task.
- The "final_answer" tool will automatically validate and save the annotated schema.

Examples
Task: "Evaluate APIs and annotate their health"
Action:
{
  "name": "some_api",
  "arguments": {"parameter1": "value1"}
```

```
}
Observation: "{response: ...successful result...}"
Action:
{
  "name": "utility_annotate_health",
  "arguments": {"api_name": "some_api", "health": "good", "reason": "Successfully called with parameter1=
value1 and obtained plausible results."}
}
Observation: "Health for API 'some_api' annotated."
Action:
{
  "name": "utility_annotate_example",
  "arguments": {"api_name": "some_api", "example": "[{\"parameter1\": \"value1\"}]"}
}
Observation: "Example for API 'some_api' annotated."
Action:
{
  "name": "final_answer",
  "arguments": "Annotated health and examples for all APIs."
}

Task: "Handle ambiguous or failing APIs"
Action:
{
  "name": "another_api",
  "arguments": {"paramA": "val"}
}
Observation: "Error: Missing required parameter 'paramB'"
Action:
{
  "name": "another_api",
  "arguments": {"paramA": "val", "paramB": 10}
}
Observation: "Error: 401 Unauthorized"
Action:
{
  "name": "utility_annotate_health",
  "arguments": {"api_name": "another_api", "health": "bad", "reason": "Repeated calls with corrected
parameters still return server-side authorization errors."}
}
Observation: "Health for API 'another_api' annotated."
Action:
{
  "name": "final_answer",
  "arguments": "Annotated health as 'bad' for failing APIs where client-side fixes do not help."
}

Available tools
{%- for tool in tools.values() %}
- {{ tool.to_tool_calling_prompt() }}
{%- endfor %}

Rules you must follow
1) Always provide a tool call. If you are answering, call "final_answer".
2) Use only the arguments the tool expects. Pass literal values, not variable names.
3) Do not repeat an identical tool call with the exact same arguments.
4) Prefer evidence from Observations. If information is missing, probe with a minimal call.

Now Begin!
</system_prompt>

<user_prompt>
Evaluate and annotate the health of each API based on the schema by actively interacting with the tools.

The schema you are given is:
{{schema}}
</user_prompt>
```

A.2 Detailed User Query Synthesis Procedure

This section provides a detailed description of the user query synthesis process.

Target Query Properties. High-quality synthetic queries must satisfy three properties. First, queries should sound natural and reflect how real users describe tasks, rather than exposing explicit tool usage or step-by-step instructions. Second, queries must require multiple tool calls to complete, such that no single API invocation suffices. Third, the required tool calls must exhibit dependency relationships, where later calls depend on the outputs of earlier ones, enforcing non-trivial planning and intermediate result handling.

Base Pipeline. We partially adopt the query synthesis pipeline from TOUCAN Xu et al. (2025), which emphasizes realism, linguistic quality, and multi-tool reasoning. Similar to

TOUCAN, we prompt an LLM with tool schemas and descriptions to generate candidate queries under constraints that exclude trivial or single-step tasks. However, our approach differs in how tool combinations are selected and how dependencies are enforced.

Dependency-Aware Query Construction. In addition to schema information, we leverage API calling histories collected during seed tool annotation, which reveal common call orders, data flow patterns, and functional relationships between APIs within the same provider. We explicitly prompt the LLM to analyze these relationships before generating queries.

Concretely, for each API provider, the LLM is instructed to select three APIs whose functionalities exhibit clear dependency structure, such as retrieval followed by transformation or filtering followed by aggregation. The model first produces a brief dependency analysis describing how these APIs interact, and then generates a single user query that implicitly requires invoking all selected APIs in the correct order. Tool names and execution details are omitted from the query text to preserve naturalness.

Outcome. By grounding query synthesis in real API usage traces and explicit dependency reasoning, this process produces queries that are both linguistically natural and structurally challenging. These queries reliably induce multi-step tool-use trajectories with meaningful inter-call dependencies, which are critical for supervising and evaluating advanced tool-using LLM agents.

A.3 Detailed Traces and Improved Description Generation

This section provides a detailed description of how execution traces are collected and how they are used to generate improved tool descriptions $D1$ and $D2$.

Trace Collection. The synthesized user queries are designed to require multi-step tool use with explicit dependencies between APIs. We execute a tool-using agent on these queries and record full execution traces, including intermediate reasoning steps, tool calls, tool responses, and termination states. For each query, we retain both successful traces and failure traces, where failures include incorrect tool selection, invalid argument construction, premature termination, or unrecoverable tool errors.

We associate each failure trace with its corresponding ground-truth tool sequence, enabling direct comparison between incorrect and correct executions. This comparison allows us to identify whether failures arise from missing information in the tool description, unclear argument semantics, or undocumented usage constraints.

Data-Independent Description Improvement. Starting from the original tool description D_0, we first generate a data-independent improved description D_1. This step applies general guidelines for tool description writing, including: clearly stating the tool's intent, specifying required versus optional parameters, documenting expected input formats, describing output semantics, and clarifying common error conditions. These guidelines are initialized from publicly available tool-use best practices[1] and iteratively refined by measuring downstream agent performance when consuming D_1. The prompt used to generate D_1 is shown below.

Trace-Driven Rule Extraction. To incorporate execution-specific information, we further refine descriptions using rules extracted from traces. We adopt the RIMRULE framework Gao et al. (2025), which compares failed traces against their corresponding ground-truth executions to identify root-cause reasoning errors. These errors are distilled into compact, generalizable rules that describe correct tool usage under specific conditions, such as required call ordering, necessary preconditions, or constraints on argument construction. The resulting rules form a reusable rule library derived from observed agent behavior.

[1] https://platform.claude.com/docs/en/agents-and-tools/tool-use/implement-tool-use#best-practices-for-tool-definitions

Table 4: Descriptive statistics of evaluation benchmarks. For StableToolBench, we only consider their solvable queries (Guo et al., 2024).

Dataset	Queries	Tools	Tools per Query
RestBench			
TMDB	100	54	54
Spotify	57	40	40
StableToolBench			
G1 Category	153	364	4.21
G1 Instruction	163	820	5.29
G1 Tool	158	500	5.03
G2 Category	124	433	5.90
G2 Instruction	105	595	6.49
G3 Instruction	61	44	5.77
BFCLv2			
Non-Live	1390	1132	1.49
Live	2251	739	2.96

Table 5: Statistics of the dataset we created in Sec 3.2 for tool interface improvement. To ensure evaluation on entirely unseen tools, all tools appearing as candidates in StableToolBench test queries are assigned to the test set $\mathcal{A}_{ts}$ and the remainder form $\mathcal{A}_{tr}$.

Dataset	Queries	Tools	Tools per Query
Synthesized SFT Data			
$\mathcal{A}_{ts}$	4,726	4,585	6.81
$\mathcal{A}_{tr}$	2,189	991	9.21

Trace-Aware Description Generation. For each tool, we retrieve the subset of rules relevant to that tool and combine them with its D_1 description to generate a final description D_2. Unlike D_1, which is independent of execution context, D_2 explicitly encodes behavioral constraints grounded in observed failures and successes. This process produces descriptions that are tailored to the actual usage patterns of each tool while remaining general enough to apply across different queries.

The final D_2 descriptions are used as supervision for supervised fine-tuning of the description generator.

B Experiment Setup Details

Here, we present additional details about the experiments for better understanding and reproducibility.

Table 4 reports descriptive statistics for the three evaluation benchmarks used in our experiments. For StableToolBench, we report only solvable queries as defined by Guo et al. (2024).

Table 5 summarizes the synthesized SFT dataset described in Section 3.2. $\mathcal{A}_{ts}$ contains all tools appearing as candidates in StableToolBench test queries and is held out entirely during training; $\mathcal{A}_{tr}$ comprises the remaining tools used to produce training examples.

B.1 Teacher-forcing Evaluation

The teacher-forcing evaluation begins with a task decomposition step, adopted from DRAFT Qu et al. (2024), to obtain a set of subtasks and their dependencies given a query. Each subtask requires no more than one tool to solve. Then, for each subtask, we perform the following steps: subtask-level tool selection annotation, tool selection, tool execution, and tool response processing. Subtask-level evaluation metrics are computed based on

the results of tool selection and tool execution. For the concern of budget, we use GPT-4.1 (2025-05-14) as our tool-using agent in all the experiments.

Subtask Tool Selection Annotation. For calculation of tool selection accuracy on subtask and query-level and F1 score on tool-level, we annotated the ground truth API tool for each subtask using GPT-4.1, which also judges whether a subtask needs a tool or not. We manually checked correctness on 100 randomly selected subtasks; the annotation is correct in 96 cases. The prompt of subtask tool selection annotation can be found below.

Subtask Tool Selection Annotation

```
<system_prompt>
You are an expert at analyzing task decomposition and determining whether subtasks require external API
calls.
</system_prompt>

<user_prompt>
TASK: Analyze whether a subtask requires an API call or is just data processing.

CONTEXT:
- Original Query: {original_query}
- Subtask Input: {subtask_input}
- Previous Context: {previous_context}
- Available APIs: {tool_info}

INSTRUCTIONS:
1. Analyze the subtask input to understand what it's trying to accomplish
2. Consider whether the subtask needs to fetch NEW data from external sources
3. Determine if the subtask is just processing/analyzing data that's already available

CRITERIA for API NEED:
- The subtask needs to SEARCH for, FIND, GET, or RETRIEVE information
- The subtask needs to access external data sources
- The subtask cannot be completed with just the data from previous steps
- The subtask involves making requests to external services or databases

CRITERIA for NO API NEED (Processing Step):
- The subtask only processes/analyzes data from previous steps
- The subtask involves counting, filtering, selecting, comparing, or organizing existing data
- The subtask uses phrases like "from the list", "from the results", "based on the", "select one", etc.
- The subtask can be completed using only the information already gathered
- The subtask involves logical operations, calculations, or data manipulation on existing data

OUTPUT FORMAT:
Respond with a JSON object containing:
{{
    "needs_api": true/false,
    "reasoning": "Detailed explanation of your decision",
    "confidence": 0.0-1.0,
    "api_name": "Name of the API if needed, or empty string if not needed"
}}

EXAMPLES:
- Subtask: "Search for movies with Tom Hanks" -> needs_api: true, api_name: "search_movies"
- Subtask: "Count how many are comedies from the results" -> needs_api: false, api_name: ""
- Subtask: "Select the highest rated movie from the list" -> needs_api: false, api_name: ""
- Subtask: "Get movie details for the selected movie" -> needs_api: true, api_name: "get_movie_details"
- Subtask: "Compare the ratings of the top 3 movies" -> needs_api: false, api_name: ""
- Subtask: "Find similar movies to the selected one" -> needs_api: true, api_name: "get_similar_movies"

Now analyze the given subtask and provide your judgment.
</user_prompt>
```

Tool Selection, Execution, and Response Processing. These three steps are performed by the tool-using agent model based on the following prompts below. Basically, in both tool selection and execution, the tool descriptions generated by different methods are injected to the prompts in the corresponding section marked by *tools_info*. After the response is processed, the result is fed into context of the next subtask as its context.

Subtask Tool Selection

```
<system_prompt>
You are an expert API selector. Given a user query for a specific subtask and available tools/APIs, you
need to select exactly ONE most appropriate API to handle this subtask.
</system_prompt>

<user_prompt>
Available Tools and APIs:
```

```
{tools_info}

Your task:
1. Analyze the subtask query and the "subtask_output" in the Context Section to understand what specific
information is needed
2. Select exactly ONE API from the available tools that is most appropriate for this subtask
3. Focus on the current subtask only - don't consider future steps

### Context Section
{context_section}

### Subtask Query
{query}

Important: You must select exactly ONE API that is most appropriate for this specific subtask.

You must respond in JSON format with exactly one selected API:
{{
    "selected_api": {{
        "reasoning": "why this specific API was selected for this subtask",
        "api_name": "api_name_here",
    }}
}}
</user_prompt>
```

Subtask Tool Execution (Parameter Generation)

```
<system_prompt>
You are a helpful assistant that generates parameters for an API call.
</system_prompt>

<user_prompt>
Given a subtask and an API and its description, you need to first write your reasoning step by step in
plain text about how to extract the correct parameters. After reasoning, you must then output the final
parameters in strict JSON format according to the API description.

Please note that:

The API description can help you better understand the use of the API.

Ensure the parameters you output are correct. The output must contain the required parameters, and may
contain the optional parameters if needed. If no parameters exist in the required and optional parameters,
 just leave it as {{"Parameters":{{}}}}.

If the subtask mentions other APIs, you should ONLY consider the API description I give and do not
consider other APIs.

Parameter Extraction from Previous Context: When the API requires path parameters (like person_id,
movie_id, tv_id, company_id, etc.), you may have to extract them from the subtask_output of previous steps
 if they are missing from the subtask input. Try to extract the numeric ID values from these text
descriptions and use them as the corresponding path parameters.

You must ONLY output in a parsable JSON format for the final answer, with no extra explanations, notes, or
 comments after it.

The output must have two parts:

"Reasoning": your step-by-step reasoning as plain text.

"Parameters": the final extracted parameters in JSON format.

An example output looks like:

{{
  "Reasoning": "The subtask asks for person details. The required parameter is person_id. From
previous_log, I see that person_id is 190. Therefore, the correct parameter is person_id=190.",
  "Parameters": {{
    "person_id": 190
  }}
}}


There are logs of previous questions and answers:
{previous_log}

This is API tool documentation:
{api_instruction}

This is the current subtask:
{question}

Output:
</user_prompt>
```

Subtask Tool Response Processing

```
<system_prompt>
You are a helpful assistant.
</system_prompt>

<user_prompt>
You should answer the question based on the response output by the API tool.

Please note that:
1. Try to organize the response into a natural language answer.
2. We will not show the API response to the user, thus you need to make full use of the response and give
the information in the response that can satisfy the user's question in as much detail as possible.
3. The question may have dependencies on answers of other questions, so we will provide logs of previous
questions and answers.

There are logs of previous questions and answers:
{context_section}

This is the user's question: {subtask query}

This is the response output by the API tool:
{call_result}
...
</user_prompt>
```

Evaluation Metrics For the tool-level F1 score, precision and recall are defined as $\text{Prec} = \frac{TP}{TP+FP}$, $\text{Recall} = \frac{TP}{TP+FN}$. A true positive (TP) corresponds to a tool that is both part of the ground truth and selected; a false positive (FP) is a selected tool that is not in the ground truth; and a false negative (FN) is a ground-truth tool that is not selected.

B.2 StableToolBench Parameter Schema Correction

This section describes the preprocessing procedure used to correct parameter schemas in StableToolBench.

We observe that a subset of tool parameter schemas in StableToolBench does not accurately reflect the true API requirements. Common issues include missing required parameters, inclusion of unsupported parameters, and incorrect parameter types. When such schemas are used during evaluation, tool invocations can fail with server-side errors, even when the model selects the correct tool and follows a reasonable call pattern. These failures introduce noise into benchmark results and confound comparisons between methods.

To address this issue, we connect StableToolBench to Smolagents Roucher et al. (2025) and programmatically invoke each tool in an iterative manner. For each API, we examine server responses to identify mismatches between the declared schema and actual API behavior. Based on these observations, we revise parameter definitions to align with the true requirements enforced by the server, including parameter presence and type constraints.

By correcting these schema-level inconsistencies, we eliminate a class of evaluation failures that are unrelated to model capability. This preprocessing step improves the stability and interpretability of StableToolBench results and enables more reliable assessment of tool-use performance. Importantly, all methods — including all baselines — are evaluated on the corrected schemas, ensuring a fair comparison across methods. As a result, absolute numbers are not directly comparable to prior work that uses the original StableToolBench schemas. The prompt of the parameter fixing agent can be found below.

Schema Parameter Fixing

```
<system_prompt>
You are an agent that rewrites a tool's schema so future tool calls succeed. You work in an Action and
Observation loop until you return the final answer.

What you have
1) The JSON string of the current schema that you will rewrite (initially).
2) A set of tools defined by that schema that you can call to get concrete feedback.
3) Generic utilities that help edit the schema incrementally.
4) The interactive history of past tool calls and their results.
```

```
What to change
- You may change a tool's description and parameters.
- You must not change any tool name. You must not change the structure of the schema.
- Add types, defaults, value ranges, enums, and constraints when the history shows they are needed.
- Make hidden requirements explicit in the description and parameter docs.
- Rewrite the API provider description and API descriptions to be clear and helpful for future LLM
function calls.

How to write the description
Start with a one-sentence plain summary of what the tool does and the problem it solves. Then document:
- Inputs: each parameter with type, required/optional, default, min/max, allowed values, and formatting
rules.
- Data model: shape of nested objects or arrays; limits on size or length; paging or cursors if present.
- Outputs: what the tool returns and what it does not return, including common items a caller might expect
 but are not provided.
- Primary use cases: the common ways the tool is used.
- Non-use cases: when the tool should not be used.

How to decide changes
- Prefer observed behavior from the history over assumptions. If a call failed with "Missing required
parameter X," make X required and document it.
- If a call failed with "Unexpected parameter Y," remove or rename Y to match the actual interface.
- When behavior is unclear, run a minimal tool call to probe the interface rather than guessing.
- Keep backward compatibility with prior successful calls when possible.

Protocol
- Every step you take is an Action. An Action is a JSON blob of the form:
  Action:
  {
    "name": "<tool_name>",
    "arguments": <tool_input>
  }
  The "arguments" field must match what the tool expects. For most tools it is an object; for tools that
accept a single value, it can be a raw value such as a string or number.
- Use the utility tools to update the schema incrementally.
  - `utility_update_provider_description`: Update the provider's description.
  - `utility_update_api_description`: Update an API's description.
  - `utility_update_api_parameters`: Update an API's parameters (JSON string).
  - `utility_remove_api_parameters`: Remove the 'parameters' field from an API (making it accept no
parameters).
  - `utility_print_schema`: See the current state of the schema.
- After each Action, you will receive an Observation. Treat it as ground truth. The Observation may be
plain text or structured JSON. Use it to decide the next Action.
- Use only listed tools. Do not invent tool names or parameters. Pass literal values, not variable names.

Required loop
1) Inspect the current schema and the history.
2) Exercise the tool(s) with targeted calls to reveal true parameter names, required fields, and
constraints.
3) Edit the schema incrementally using the utility tools to reflect observed behavior.
4) Validate by re-running the previously failing calls until they succeed or until you reach the real
limits of the tool.
5) End with the final answer tool.

Output and completion
- You must finish by calling the "final_answer" tool. It is the only way to complete the task.
- The "final_answer" tool will automatically save the schema.

Examples
Task: "Rewrite the schema based on the log"
Action:
{
  "name": "some_tool_in_schema",
  "arguments": {"parameter1": "value1"}
}
Observation: "TypeError: some_tool_in_schema() takes 0 positional arguments but 1 was given"
Action:
{
  "name": "utility_remove_api_parameters",
  "arguments": {"api_name": "some_tool_in_schema"}
}
Observation: "Parameters field removed from API 'some_tool_in_schema'."
Action:
{
  "name": "some_tool_in_schema",
  "arguments": {}
}
Observation: "{response: ...}"
Action:
{
  "name": "final_answer",
  "arguments": "The schema has been rewritten."
}

Task: "Update parameters"
Action:
{
  "name": "another_tool",
  "arguments": {"paramA": "val"}
}
Observation: "Missing required parameter: paramB"
```

```
Action:
{
  "name": "utility_update_api_parameters",
  "arguments": {
    "api_name": "another_tool",
    "parameters_json": "{\"paramA\": {\"type\": \"string\", \"required\": true, \"description\": \"...\"},
 \"paramB\": {\"type\": \"string\", \"required\": true, \"description\": \"...\"}}"
  }
}
Observation: "Parameters for API 'another_tool' updated."
Action:
{
  "name": "final_answer",
  "arguments": "Updated parameters."
}

Task: "Clarify API description"
Action:
{
  "name": "utility_update_api_description",
  "arguments": {
    "api_name": "complex_tool",
    "description": "This tool calculates the mortgage payment. Inputs: 'principal' (number, required), '
rate' (number, required, annual interest rate in percentage), 'years' (number, required, loan term).
Output: Monthly payment amount."
  }
}
Observation: "Description for API 'complex_tool' updated."
Action:
{
  "name": "final_answer",
  "arguments": "Updated API description to be more clear."
}

Available tools
{%- for tool in tools.values() %}
- {{ tool.to_tool_calling_prompt() }}
{%- endfor %}

Rules you must follow
1) Always provide a tool call. If you are answering, call "final_answer".
2) Use only the arguments the tool expects. Pass literal values, not variable names.
3) Do not repeat an identical tool call with the exact same arguments.
4) Prefer evidence from Observations and the history over guesses. If information is missing, probe with a
 minimal call.

Now Begin!
</system_prompt>

<user_prompt>
Rewrite the schema of the tool based on the log below and interacting with the tools.

The schema you are given is:
{{schema}}
</user_prompt>
```

B.3 Training and Inference Details

All experiments are conducted on a single node equipped with 8 $\times$ NVIDIA A100 (80GB) GPUs. For SFT, we develop based on the FSDP SFT Trainer of the verl library[2] to optimize training efficiency and streamline checkpoint saving and conversion. The SFT hyperparameters are shown in Table 6. For inference, we leverage vLLM (Kwon et al., 2023) for efficiency and perform top-p sampling with temperature 0.3, top-p 0.9, repetition_penalty 1.1.

We fine-tune Qwen3-4B-Instruct-2507[3] due to its strong instruction following ability with $\mathcal{A}_{tr}$ of our synthesized dataset to make $\mathcal{A}_{ts}$ and the test queries unseen during training.

The prompts for Trace-Free+, Trace-Free at inference time and Trace-based Sample at training time for Trace-Free+ are shown below

Prompt for Trace-Free+ (Inference and Trace-free Samples in Training) and Trace-Free

```
<system_prompt>
You are an API documentation specialist.
</system_prompt>
```

[2] https://github.com/volcengine/verl

[3] https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507

Parameter	SFT
Base Model	Qwen3-4B-Instruct-2507
Hardware	$1 \times$ 8-A100 GPU Node
Optimizer	AdamW
Learning Rate	5.0×10^{-5}
LR Scheduler	Cosine
Training Epochs	2.0
LoRA Rank	64
Precision	bf16
Max Length	2,048
Effective Batch Size	8
DeepSpeed Stage	ZeRO-3

Table 6: Hyperparameters for SFT.

```
<user_prompt>

Rewrite the API description so an AI agent can:
1) Decide when to use this API
2) Generate valid parameters

Inputs:
- API name: {tool_name}
- Parameter schema: {parameter_json}
- Baseline description: {original_description}

Infer (do not output):
- When to use vs not use this API
- Required vs optional parameters
- Parameter meanings and constraints
- Cross-parameter dependencies or exclusions
- Common parameter mistakes
  - no examples are provided, infer from the schema and baseline description only

Write a clear API description that:
- States when to use and NOT use the API
- Does not invent or reference non-provided APIs
- Explains each parameter's meaning, type, required/optional status, constraints, and defaults
- Describes likely validation failures and how to avoid them
- Abstracts patterns into general rules
- Does not restate the full schema verbatim
- Does not mention whether examples were provided

You may replace the baseline description entirely.

Output ONLY valid JSON (no markdown, no code blocks):
{{"description": "<your improved API description here>"}}

</user_prompt>
```

Prompt for Trace-Free+ (Trace-based Samples in Training)

```
<system_prompt>
You are an API documentation specialist.
</system_prompt>

<user_prompt>
Rewrite the API description so an AI agent can:
1) Decide when to use this API
2) Generate valid parameters

Inputs:
- API name: {tool_name}
- Parameter schema: {parameter_json}
- Example queries + errors: {query_examples}
- Baseline description: {original_description}

Infer (do not output):
- When to use vs not use this API
- Common parameter mistakes
- Required vs optional parameters
- Cross-parameter constraints

Write a clear API description that:
- States when to use and NOT use the API
- Does not invent other APIs
- Explains each parameter's meaning, type, required/optional status, constraints, and defaults
- Describes common validation failures and how to avoid them
- Abstracts examples into general rules
- Does not restate the full schema or copy examples
```

```
You may replace the baseline entirely.

Output ONLY valid JSON (no markdown, no code blocks):
{{"description": "<your improved API description here>"}}

</user_prompt>
```

C Case Study: Pattern Types Learned by Trace-Free+

C.1 Pattern Category Definitions

Table 7 defines the five interface pattern categories used throughout this analysis, with their level of application, key classification test, and a representative example.

C.2 Pattern Category Coverage

This analysis is performed on tools from the Media and Finance categories of $\mathcal{A}_{ts}$, selected as representative domains with sufficient tool variety across all five pattern types. Observations from Table 8: Most notably, Trace-Free+ dramatically improves over D0 across all five categories, confirming that learned interface generation substantially enriches otherwise sparse baseline descriptions. While D_0 covers fewer than 12% of tools in any category, Trace-Free+ raises coverage to near-complete levels for tool selection scope (97.2%) and parameter constraints (94.2%), and introduces non-trivial gains even in harder categories such as cross-tool dependencies (30.0%) and cross-parameter dependencies (17.1%). This highlights that the model successfully internalizes and applies generalizable interface patterns from training, even without access to execution traces at inference time.

Compared to D_1 and D_2, Trace-Free+ shows a more nuanced trade-off. For tool selection scope and parameter constraints, all three methods achieve similarly high coverage, with D_2 slightly outperforming D_1 and Trace-Free+. For cross-tool dependencies, all methods converge to a similar range (30–32%), suggesting this pattern is largely driven by explicit parameter guidance and is equally captured across approaches.

The main divergence appears in output description and cross-parameter dependencies. D_1 and D_2 achieve near-complete output description coverage (99%) because D_1's data-independent documentation guidelines explicitly require enumerating output fields and their semantics directly from the schema — making output description a near-guaranteed addition for every tool. Trace-Free+, by contrast, lags on this category (27.0%), as it must infer output semantics from schema-level signals alone without the benefit of explicit guidelines, and tends to encode output information implicitly through usage scope rather than explicit field enumeration. This gap does not substantially affect downstream performance because output description primarily helps agents interpret tool responses, whereas the categories most critical for correct tool selection and argument construction — tool selection scope and parameter constraints — are the ones where Trace-Free+ achieves its highest coverage (97.2% and 94.2% respectively). In contrast, Trace-Free+ outperforms both D_1 and D_2 on cross-parameter dependencies (17.1% vs. 8–10%), indicating a relative strength in modeling inter-parameter relationships from schema-level signals.

Overall, while D_1/D_2 achieve near-complete coverage on more surface-level documentation patterns, Trace-Free+ better captures certain structural constraints (e.g., parameter interactions), suggesting complementary strengths between template-based refinement and learned, schema-driven generalization.

C.3 Qualitative Examples

To illustrate what the model learns to internalize, Table 9 presents three examples of Trace-Free+-generated descriptions for unseen tools from $\mathcal{A}_{ts}$, highlighting the constraint types added without access to execution traces.

D Additional Experimental Results

Parameter Study: Ratio of Trace-free Data in Curriculum. Here, we also investigate the impact of the ratio of the trace-free data in the curriculum learning. We fix the number of total training samples and investigate the impact of different learning curriculums. To avoid overfitting, we limit the training to 2 epochs. As we can observe from Table 10, the two-stage curriculum with 10% trace-free data in the first stage and 90% in the second stage is the most effective one. A hypothesis for this is that the second with 90% trace-free is close to the trace-free scenario at inference time. We also tried 3 stages but the results are worse than 2 stages on StableToolBench.

Transfer to BFCLv2. Table 11 reports the full results of applying Trace-Free+-improved tool descriptions on BFCLv2 (Patil et al., 2025).

E Future work

This work opens several directions for future research. First, fine-tuning agents and improving tool interfaces can be done jointly for optimizing the performance of agents. Second, within the scope of tool interface improvement, developing principled methods for query synthesis to cover both single-step and multi-step queries for both training and benchmarking is an interesting research direction. Finally, while we focus on RESTful APIs, the same framework may apply to other domains such as databases or code execution environments. Finally, combining tool interface optimization with downstream agent training in an end-to-end fashion may lead to further gains.

#	Category	Level / Key Test	Explanation	Example
1	Tool selection scope	Tool-level. Does it help an agent decide between multiple options?	Explicitly states when to use this tool vs. another, when NOT to use it, or compares it to a similar tool. Must go beyond a simple purpose statement.	"Use this API when you need to retrieve a list of artworks by search query. Do not use it if you require detailed metadata such as dimensions or provenance. For more detail, use Detect Features instead."
2	Cross-tool dependencies	Tool-level. Does it name a specific upstream endpoint?	A parameter value must come from calling a specific named tool or endpoint first. The description explicitly names the other tool/endpoint — not just general prior state.	"The `hash` parameter must be extracted from the Check Status API response when `movie_status` equals DONE, and `vsid` must come directly from the Start Movie Session API response."
3	Output description	Tool-level. Does it enumerate response fields or explicitly state what is NOT returned?	Explicitly enumerates response fields, data types, or structure — or explicitly states what is not included. Does not apply to general purpose statements that merely describe what the tool retrieves.	"Returns all static metadata including logo, description, official website URL, and social links. Does not include videos, subcategories, or category-specific details."
4	Parameter constraints	Parameter-level. Does it restrict valid values or structure?	Restricts what values a parameter can take or specifies how it must be structured: enums, numeric ranges, case sensitivity rules, date formats, encoding schemes, separator conventions, or array structure.	"Must be one of `en` (English) or `nl` (Dutch). `limit` must be a positive integer between 1 and 100. `date` must be in `YYYY-MM-DD` format. `image` must be a Base64-encoded string."
5	Cross-param dependencies	Parameter-level. Does it constrain one parameter based on another?	Constraints between parameters within the same tool — parameters that must be paired together or are mutually exclusive.	"`longitude` (requires `lat`)."

Table 7: Definitions of the five interface pattern categories used to classify tool description improvements. Categories 1–3 are tool-level (apply to the description as a whole); Categories 4–5 are parameter-level (apply per parameter).

#	**Category**	D_0	D_1	D_2	**Trace-Free+**
1	Tool selection scope	3.3%	99.7%	100.0%	97.2%
2	Cross-tool dependencies	0.3%	32.0%	21.1%	30.0%
3	Output description	11.6%	98.6%	100.0%	27.0%
4	Parameter constraints	9.9%	87.9%	89.5%	94.2%
5	Cross-param dependencies	0.3%	8.5%	21.1%	17.1%

Table 8: Percentage of tool descriptions covering each of the five interface pattern categories, comparing D_0, D_1, D_2, and Trace-Free+. D_0 descriptions are consistently sparse across all categories; all three improved variants close most of the gap on scope and parameter constraints, with differences concentrated on cross-tool dependencies and output description.

	D_0 **(Original)**	**Trace-Free+ (Ours)**
API	`Walk Score` (provider: `Walk Score`)	
Parameters	`lat` (str), `lon` (str), `address` (str), `wsapikey` (str), `bike` (str, opt.), `transit` (str, opt.), `format` (str, opt.)	
Description	Get Walk Score. `address`: "The URL encoded address." `lat/lon`: no range stated. `format`: "Type of result to return: (movie, series, episode)."	Retrieves Walk, Transit, and Bike scores for a location. `lat` must be a decimal in $[-90, 90]$; `lon` in $[-180, 180]$. `address` must *not* be URL-encoded. `bike` and `transit` must be exactly `'1'` — not `'true'` or `'on'`. `format` must be `'json'` or empty (XML default); any other value is rejected. Do not use for real-time traffic data or historical trends.
Patterns	—	*Parameter constraints* (`lat/lon` numeric ranges; exact-value flags for `bike`, `transit`, `format`; format correction for `address`); *Tool selection scope* (exclusions: not for real-time traffic or historical trends)
API	`Player Trophy Information` (provider: `Trackmania`)	
Parameters	`player_id` (str), `limit` (float, opt.), `page` (float, opt.)	
Description	Get the trophy info of a player. `limit`: "The result limit that gets returned. Default limit is `50`." `page`: "The result page, each containing 50 results. Default page is `0`."	Retrieves a player's trophy records, paginated. `player_id` must be obtained from the `/players` endpoint; invalid or missing IDs cause validation failure. `limit` must be between **1 and 100 inclusive**; values above 100 or below 1 are rejected. `page` must be a **non-negative integer**; negative or non-integer inputs are invalid. Do not use for general player statistics or game metadata.
Patterns	—	*Parameter constraints* (`limit` $\in [1, 100]$; `page` $\geq$ 0); *Cross-tool dependencies* (`player_id` must come from `/players` endpoint); *Tool selection scope* (trophy records only, not general player stats)
API	`GET queue` (provider: `League of Legends API`)	
Parameters	`region` (str), `queue` (str), `division` (str), `tier` (str)	
Description	Returns the list of people in division, tier and according to the type of queue. All four parameter descriptions are empty.	Retrieves ranked-queue player listings by division, tier, and region. `division` must be one of `'I'`, `'II'`, `'III'`, `'IV'`. `queue` must be a valid ranked queue identifier (e.g., `'RANKED_SOLO_5x5'`). `region` must be a supported server code (e.g., `'NA'`, `'EUW'`, `'KR'`). `tier` must be a valid rank tier (e.g., `'Gold'`, `'Diamond'`). All four parameters are required; any missing or malformed value causes a validation error.
Patterns	—	*Parameter constraints* (enumerated valid values for all four parameters, filled from empty stubs); *Tool selection scope* (ranked queue listings only, not live match data)

Table 9: Three examples of Trace-Free+-generated descriptions for unseen tools compared against D_0. In all cases the model infers constraints without execution traces.

Table 10: Parameter Study for Trace-Free+ in Trace-free Evaluation

	StableToolBench											
	G1 Category		G1 Instruction		G1 Tool		G2 Category		G2 Instruction		G3 Instruction	
Ratios	SL	QL	SL	QL	SL	QL	SL	QL	SL	QL	SL	QL
(0.1, 0.9)	73.8 ± 0.7	65.6 ± 1.0	72.6 ± 1.5	60.0 ± 1.0	70.0 ± 1.4	56.4 ± 1.6	68.7 ± 0.7	47.5 ± 0.8	71.4 ± 1.1	48.3 ± 1.6	63.9 ± 1.8	46.4 ± 4.1
(0.3, 0.7)	73.7 ± 1.9	63.6 ± 2.8	73.2 ± 2.3	61.3 ± 4.3	68.3 ± 1.1	52.8 ± 0.7	68.6 ± 0.0	47.9 ± 0.6	69.1 ± 2.9	45.4 ± 3.6	61.1 ± 3.0	41.8 ± 3.5
(0.5, 0.5)	75.8 ± 0.0	66.0 ± 0.0	71.1 ± 0.0	64.3 ± 0.0	69.2 ± 0.0	55.6 ± 0.0	65.9 ± 0.0	48.3 ± 0.0	68.9 ± 0.0	44.9 ± 0.0	59.8 ± 0.0	39.3 ± 0.0
Trace-Free	71.8 ± 2.3	62.7 ± 2.1	70.8 ± 0.1	60.8 ± 0.9	68.5 ± 2.7	53.6 ± 2.8	66.4 ± 1.2	44.1 ± 0.0	69.2 ± 4.0	46.4 ± 3.6	59.9 ± 1.2	41.8 ± 1.2

Table 11: Transfer evaluation on BFCLv2. We apply Trace-Free+ to improve tool descriptions and evaluate on the Non-Live and Live splits.

	Non-Live		**Live**	
Model	D0	+ Trace-Free+	D0	+ Trace-Free+
GPT-4.1	88.56	**89.04**	79.28	**80.63**
Claude Sonnet 4.5	63.11	**65.50**	52.40	**53.45**
Gemini-3-pro-preview	91.24	**91.55**	84.98	**86.41**