AskToAct: Enhancing LLMs Tool Use via Self-Correcting Clarification

Xuan Zhang[1], Yongliang Shen[1†], Zhe Zheng[1], Linjuan Wu[1], Wenqi Zhang[1], Yuchen Yan[1], Qiuying Peng[2], Jun Wang[2], Weiming Lu[1†]
[1]Zhejiang University
[2]OPPO Research Institute
{zhangx0513, syl, luwm}@zju.edu.cn, {pengqiuying, wangjun7}@oppo.com

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities in tool learning. In real-world scenarios, user queries are often ambiguous and incomplete, requiring effective clarification. However, existing interactive clarification approaches face two critical limitations: reliance on manually constructed datasets, which inherently constrains training data scale and diversity, and lack of error correction mechanisms during multi-turn clarification, leading to error accumulation that compromises both accuracy and efficiency. We present **AskToAct**, which addresses these challenges by exploiting the structural mapping between queries and their tool invocation solutions. Our key insight is that tool parameters naturally represent explicit user intents. By systematically removing key parameters from queries while retaining them as ground truth, we enable automated construction of high-quality training data. We further enhance model robustness through error-correction pairs and selective masking, enabling dynamic error detection during clarification interactions. Comprehensive experiments demonstrate that **AskToAct** significantly outperforms existing approaches, achieving above 57% accuracy in recovering critical unspecified intents and enhancing clarification efficiency by an average of 10.46% while maintaining high accuracy in tool invocation. Our framework exhibits robust performance across different model architectures and successfully generalizes to entirely unseen APIs without additional training, achieving performance comparable to GPT-4o with substantially fewer computational resources.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in various tasks, from code generation to complex reasoning (Nakano et al., 2021; Chen et al., 2021; Komeili et al., 2022; Wei et al., 2022). A particularly promising direction is their ability to interact with external tools through API calls, which significantly expands their practical applications (Schick et al., 2023; Hao et al., 2023; Qin et al., 2024; Shim et al., 2025). This has inspired numerous frameworks focusing on tool-augmented LLMs, including Toolformer (Schick et al., 2023), ToolLLaMA (Qin et al., 2024), and Gorilla (Patil et al., 2023).

However, current tool learning frameworks (Li et al., 2023; Song et al., 2023; Schick et al., 2023; Qin et al., 2024) operate under an idealistic assumption that user queries are always explicit and unambiguous. This diverges significantly from real-world scenarios where users often provide incomplete, ambiguous, or imprecise queries. Such unspecified queries pose unique challenges in tool learning scenarios, as API calls require precise parameters and cannot tolerate ambiguity (Wang et al., 2024b). When faced with unspecified queries, LLMs tend to either arbitrarily generate missing parameters or remain unknown, leading to potential risks in tool invocation.

This raises a critical research question: *How can we enhance LLMs' ability to handle unspecified queries in tool learning scenarios while ensuring accurate and reliable tool invocation?* Recent works (Zhang and Choi, 2023; Qian et al., 2024; Wang et al., 2024b) have introduced interactive clarification approaches, but face two fundamental limitations. First, they rely heavily on manually constructed datasets for training (Qian et al., 2024; Wang et al., 2024b). Creating these datasets requires human annotators to craft queries and clarifications, a process that inherently limits scale and diversity. The resulting datasets capture only a narrow range of ambiguity patterns, reducing their effectiveness with diverse real-world queries. Second, as shown in Figure 1, these approaches lack

† Corresponding author.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 13484–13511
November 4-9, 2025

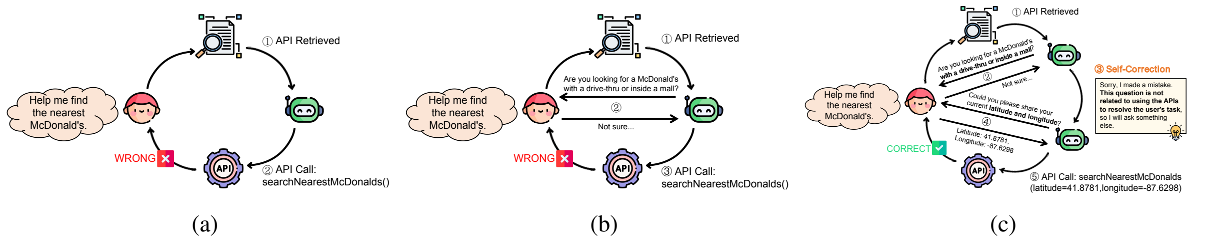

Figure 1: Comparison of query handling approaches: (a) direct API calls without clarification, (b) basic clarification without error recovery, and (c) our self-correcting **ASKTOACT** framework.

robust error handling during multi-turn clarification. Existing models train on datasets with only perfect clarification sequences. In reality, models often request already-provided information, follow irrelevant paths, or miss unspecified details. Without error recovery training, these issues accumulate throughout dialogues, reducing efficiency and compromising tool invocation quality.

We propose **ASKTOACT**, a self-correcting clarification framework that systematically addresses these limitations. Our key insight is that tool parameters naturally represent explicit user intents, creating an opportunity for automated data generation. We develop an automated pipeline that strategically removes key parameters from complete queries in existing datasets, generating diverse unspecified queries with built-in ground truth. Using these queries, we construct rich clarification dialogues demonstrating effective intent elicitation. To enable robust error handling during interactions, we augment training data with carefully designed error-correction pairs that simulate realistic mistakes and their solutions. We implement selective masking during training to prevent learning negative patterns while enhancing error detection abilities.

Through comprehensive experiments, we demonstrate that **ASKTOACT** achieves several significant improvements: (1) correctly identifies unspecified queries and recovers more than 57% of critical unspecified intents, while significantly enhances clarification efficiency by an average of 10.46% compared to the base model; (2) achieves strong performance in end-to-end tool invocation, with over 81% tool selection accuracy and over 68% parameter resolution accuracy; (3) exhibits robust performance across different model architectures, and successfully generalizes to entirely unseen APIs; and (4) delivers performance comparable to GPT-4o while requiring substantially fewer computational resources.

Our work makes three main contributions:

- We introduce an automated pipeline for generating high-quality intent clarification datasets, addressing the scalability limitations of manual annotation.
- We develop a self-correction mechanism that enables models to dynamically detect and correct potential errors during clarification interactions.
- Our experimental results demonstrate that our method not only achieves state-of-the-art (SOTA) performance but also shows strong generalization ability when handling queries requiring the use of unseen APIs.

2 Method

Tool learning faces a fundamental challenge: while API calls require precise parameters, real-world queries are often ambiguous. To bridge this gap, we propose **ASKTOACT**, a self-correcting clarification framework. Our method consists of two key components: (1) an automated data construction pipeline for generating diverse intent calrification data (§2.1), and (2) a self-correction training paradigm for dynamic error detection and correction (§2.2). The core insight is that tool parameters naturally represent explicit user intents, making them ideal anchors for both data generation and error correction. Figure 2 illustrates the overall framework architecture.

2.1 Intent Clarification Dataset Curation

The foundation of our method is a systematic pipeline for constructing multi-turn clarification data. As shown in Figure 2, the pipeline proceeds in two steps: generating unspecified queries and subsequently constructing clarification dialogues.

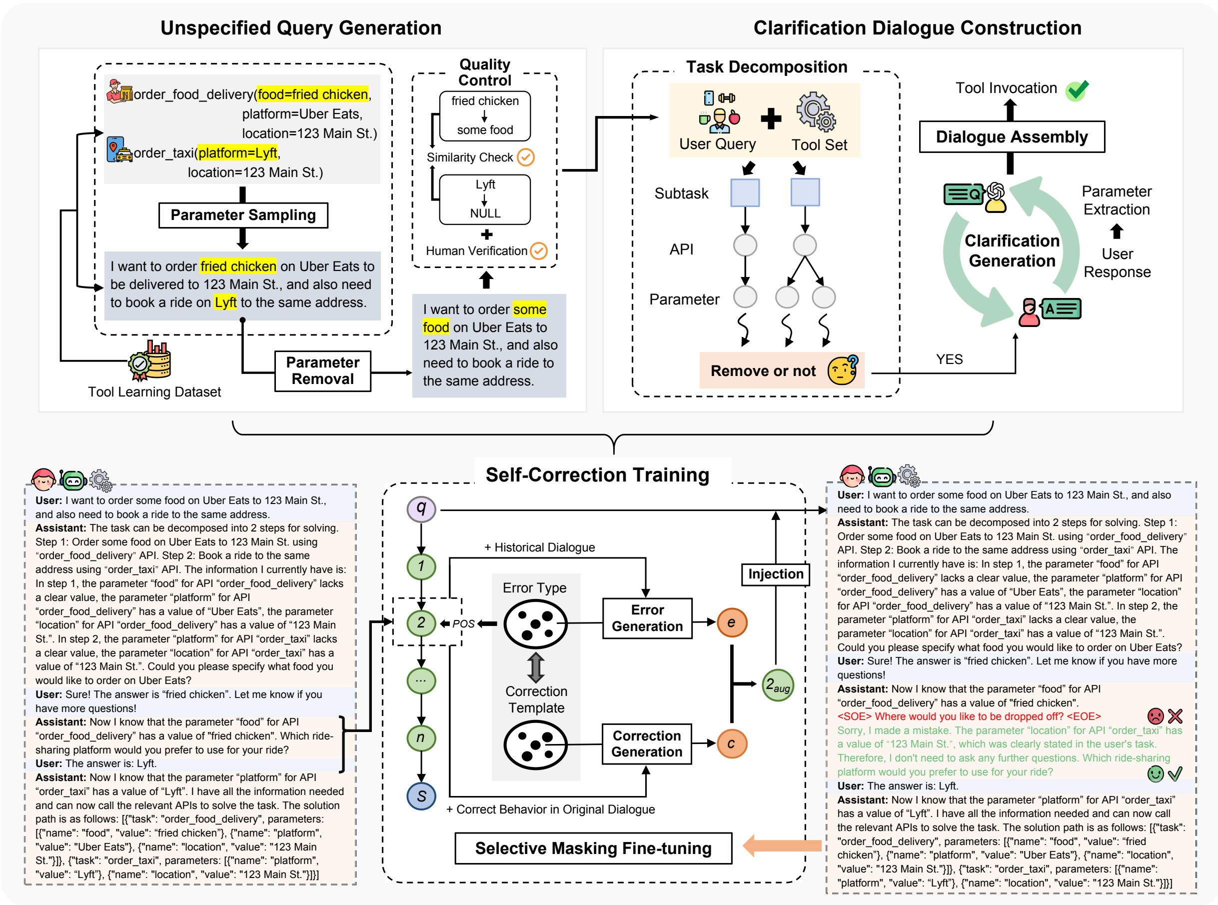

Figure 2: Overview of **AskToAct** framework. Top: Dataset construction pipeline, consisting of (1) unspecified query generation by selecting and removing key parameters (e.g., "fried chicken" and "Lyft") from original queries, and (2) dialogue construction through task decomposition, clarification generation, and dialogue assembly. Bottom: Self-correction training through dialogue augmentation and selective masking fine-tuning.

2.1.1 Unspecified Query Generation

A key challenge in building intent clarification systems is obtaining realistic examples of ambiguous queries paired with their complete intents. We address this through a novel reverse-engineering approach that leverages existing tool learning datasets. Each instance in these datasets consists of a fully specified query q and a corresponding tool invocation solution $S = \{(f_i, P_i) \mid i = 1, \dots, n\}$, where f_i denotes the API and $P_i = \{p_i^1, p_i^2, \dots\}$ represents its parameter set. We systematically transform q into an unspecified query q' while preserving the ground truth information necessary for subsequent dialogue construction and evaluation.

Parameter Sampling The first step in our pipeline is to determine which parameters in S to remove from the original query q. We implement a stratified sampling approach (see Appendix C.1.1) that enables sampling across different API domains and parameter counts. For each query, we randomly select parameters according to one of four complexity levels: (1) fully specified, where all parameters are retained, (2) single-API single-parameter, where one parameter from one API call is removed, (3) single-API multi-parameter, where multiple parameters from the same API call are removed, and (4) multi-API multi-parameter, where parameters are removed across multiple API calls. This stratification ensures our dataset captures the full spectrum of query ambiguity encountered in real-world scenarios, from basic single-slot ambiguities to complex multi-faceted ambiguities.

Parameter Removal Once the removed parameter set P' is determined, we apply two complementary strategies to transform the original query q into an unspecified form. The first strategy, *complete removal*, entirely eliminates parameter values from q while preserving grammatical integrity. The second strategy, *semantic abstraction*, replaces specific parameter values with abstract expressions that necessitate further clarification. For each parameter $p \in P'$, we maintain a mapping $M : p \rightarrow v'$, where v' represents the transformed value after parameter removal or abstraction. By recording the values before and after the transformation, we can more precisely track how explicit user intent becomes unspecified during the process of unspecified query generation. This record plays an important role in quality control, helping us ensure the quality of the generated query. Also, it guides subsequent dialogue construction and provides ground truth for evaluation. The implementation details and the format of the transformation record are provided in Appendix C.1.2 and Appendix C.1.3, respectively.

Quality Control To ensure generation quality, we employ a dual-stage verification mechanism. We first compute semantic similarity between original values and their transformations using Sentence Transformer (Reimers and Gurevych, 2019). Queries with similarity scores exceeding 0.95 are filtered out to ensure sufficient semantic alteration. Then, we conduct human verification on generated queries (see Appendix C.1.4). Through this process, we constructed 35,261 high-quality unspecified queries, as shown in Table 1.

2.1.2 Clarification Dialogue Construction

Based on the generated unspecified queries, we propose an automated method to construct training dialogue data that simulate multi-turn clarification. The dialogue construction process—from task decomposition, clarification generation to final dialogue assembly—is essential for generating coherent and effective clarification interactions.

Task Decomposition The foundation of effective clarification lies in identifying what information needs to be clarified. Given an unspecified query q' and its tool invocation solution S, we first decompose the query into a sequence of subtasks. Each subtask corresponds to an API call in S. For each API call, we perform parameter analysis to identify two categories: (1) parameters already specified in q', and (2) parameters requiring clarification. This structured decomposition guides the subsequent clarification process, ensuring all necessary information is systematically obtained.

Clarification Generation Building on the task decomposition, we generate interaction turns for each parameter that requires clarification, following the API call order defined in S. We construct each clarification turn through a three-step process, with the goal of maximizing clarification effectiveness and preserving natural conversational flow. First, we generate a clarification question q_c targeting the unspecified parameter. Next, we simulate the user's reply using diverse response templates that vary in verbosity and conversational tone. Finally, we generate a confirmation statement summarizing the clarified information, which serves as an explicit reference for subsequent turns.

Dialogue Assembly The final step brings together all components into a coherent dialogue structure. We assemble the generated elements sequentially while maintaining natural conversation flow through consistent reference to previously clarified information and smooth transitions between parameter-related clarifications. Special attention is paid to parameter interdependencies, ensuring that information is requested in a logical order that reflects real-world dialogue patterns. The dialogue concludes with the complete tool invocation solution S, providing a clear connection between the clarification process and its ultimate goal. Detailed templates and prompting strategies that support this assembly process are provided in Appendix C.2.

2.2 Self-Correction Training

While constructing high-quality training data is essential, the dynamic nature of clarification interactions requires models to detect and correct potential errors in real-time. We develop a systematic training paradigm that combines error-correction augmentation with specialized training strategies to enhance model robustness and enable self-correction.

Error Type Analysis Through comprehensive analysis of clarification interactions generated by the model in response to unspecified queries, we identify five primary error types that impair the clarification process in complementary ways. *Clearly Stated Intent Clarification* occurs when the model requests explicitly stated information, creating unnecessary interaction turns. *Imprecise Clarification* is characterized by questions that lack specificity, often resulting in ambiguous user responses. *Irrelevant Clarification* emerges when the model poses questions that diverge from the core intent. *Redundant Clarification* arises when the model requests information that has been previously clarified. *Incomplete Clarification* represents failure to identify all parameters requiring clarification, leading to incomplete tool invocation solutions. Understanding

Dataset	Train	Test	Avg. No. APIs	Avg. No. Params	Avg. No. Unspecified Intents
xlam-IC	29,821	4,456	1.58	2.49	1.32
Taskbench-IC	-	984	1.75	2.35	1.30
Total	29,821	5,440	1.59	2.49	1.32

Table 1: Datasets statistics. xlam-IC is generated from the xlam-function-calling-60k dataset (Liu et al., 2024) which is used for training and testing. Taskbench-IC is generated from Taskbench (Shen et al., 2024) and is used exclusively for OOD testing. Please refer to the Appendix B for more details about both datasets.

Error Type	Count
Clearly Stated Intent Clarification	2,481
Imprecise Clarification	2,298
Irrelevant Clarification	2,251
Redundant Clarification	3,126
Incomplete Clarification	5,000
Total	15,156

Table 2: Augmented dialogue error types.

these patterns guides our error-correction strategy.

2.2.1 Error-Correction Augmentation

Building upon this error analysis, we introduce an automated method to augment dialogues with error-correction pairs. Given a dialogue d, we randomly select an error type τ_k and determine an injection position pos. We then generate the error instance using two strategies. For semantic errors (Clearly Stated Intent, Imprecise, and Irrelevant), we employ GPT-4o with specialized prompts: $e = f_{\text{gpt}}(d, \tau_k, pos)$. For structural errors (Redundant and Incomplete), we implement rule-based algorithms: $e = f_{\text{rule}}(d, \tau_k, pos)$. Implementation details are provided in Appendix D.1.

After generating error e, we construct a correction c using a template specific to the error type τ_k. The resulting correction c explicitly states the error type and identifies the correct behavior as the content at position pos in the original dialogue d. We then inject these error-correction pair (e, c) into the original dialogue d at the predetermined position pos: $d' = \text{inject}(d, e, c, pos)$.

To ensure the validity of our error-correction augmentation method, we conduct human verification on augmented dialogues (see Appendix D.3). Through this systematic process, we generated 15,156 augmented dialogues, as shown in Table 2.

2.2.2 Selective Masking Fine-tuning

To effectively utilize the augmented dialogues for training while preventing the model from learning error patterns, we implement a selective masking mechanism during fine-tuning. We introduce special tokens <*SOE*> and <*EOE*> to demarcate error segments, and mask these segments during loss computation. This approach allows the model to learn error detection and correction patterns while avoiding the reinforcement of error behaviors. Through this training process, we equip the model with the ability to dynamically identify potential errors and apply appropriate corrections during clarification interactions.

3 Experiment

3.1 Experimental Settings

Training Details We construct our training data from the xlam-IC dialogue dataset, where 30% of the samples are replaced with error-correction augmented dialogues. We explore two adaptation strategies for the Qwen2.5-7B-Instruct model: LoRA (Hu et al., 2021) and full-parameter fine-tuning. More details are provided in Appendix E.

Baselines For comprehensive comparison, we evaluate representative tool-augmented LLMs, including xLAM-7b-fc-r (trained on the xlam-function-calling-60k dataset but without intent clarification) (Liu et al., 2024), gorilla-openfunctions-v2 (Patil et al., 2023), and ToolLLaMA-2-7b-v2 (Qin et al., 2024), as well as an intent clarification model, Mistral-Interact (Qian et al., 2024). In addition, we evaluate major LLM series, including Mistral-7B-Instruct-v0.3, LLaMA (3-8B/70B-Instruct), Qwen (2.5-7B/72B-Instruct), DeepSeek-V3, Claude (3.5-Haiku/Sonnet), and GPT (3.5, 4, 4o-mini, 4o). All models use a standardized evaluation prompt (see Appendix F.1).

3.2 Evaluation Framework

We develop an automated framework for systematic evaluation on handling unspecified queries. The framework employs an LLM to simulate user behavior. During interactions, the user-simulating LLM judges whether clarification questions are relevant to unspecified intents, and either provides the necessary information or indicates that it is unavailable. To better capture the complexity of real-world human-LLM interactions, we configure the user-simulating LLM with six personality types, each exhibiting different response characteristics. Implementation details are provided in Appendix F.2.

3.3 Metrics

We evaluate the models in two aspects: *intent clarification quality* and *tool invocation accuracy*. For intent clarification quality, we design four metrics. *Intent Coverage Rate (ICR)* measures the proportion of successfully clarified intents among all unspecified intents, while *Clarification Efficiency (CE)* evaluates the success rate of clarification across interaction rounds. We combine these measures into a *Clarification Performance Score (CPS)* using a harmonic mean, similar to the F1-score formulation. Additionally, we track *Interaction Rounds (IR)* as the average number of clarification rounds per query. For tool invocation accuracy, we introduce three complementary metrics. *Solution Completion Rate (SCR)* measures the proportion of successfully generated tool invocation solutions, providing an end-to-end assessment. *Tool Selection Score (TSS)* evaluates API selection accuracy using an F1-score over selected and required APIs. *Parameter Resolution Score (PRS)* assesses the accuracy of parameter resolution through an F1-score computation over API-parameter-value triples. The details are provided in Appendix F.3.

3.4 Main Results

3.4.1 LLM-based Simulated Evaluation

The experimental results on the in-domain (ID) test split of the xlam-IC dataset are presented in Table 3. Our method demonstrates superior performance in both intent clarification and tool invocation.

Intent Clarification Capability Both variants of our method—AskToAct-LoRA-SFT-7B and AskToAct-Full-SFT-7B—exhibits strong capabilities in intent clarification. In particular, the fully fine-tuned variant reaches a CPS of 65.92%, closely approaching the performance of SOTA LLMs such as GPT-4o. Meanwhile, the lightweight LoRA variant also achieves competitive results (ICR: 57.68%, CE: 63.41%, CPS: 60.41%), significantly surpassing the specialized intent clarification model Mistral-Interact.

Tool Invocation Accuracy Our method demonstrates remarkable capabilities in translating clarified intents into precise tool invocations. It achieves SOTA performance across all evaluation metrics (SCR > 96%, TSS > 81%, PRS > 68%), significantly surpasses all existing open-source and closed-source LLMs. Compared to tool-augmented LLMs such as xLAM-7b-fc-r, gorilla-openfunctions-v2, and ToolLLaMA-2-7b-v2, our method demonstrates substantial advantages. This performance gap highlights the strength of integrating intent clarification with tool learning. Unlike prior specialized models that are limited to unambiguous tool-use queries, our method effectively resolves ambiguity in user queries, leading to significantly improved tool invocation accuracy.

Further analyses—including cross-model transferability, the impact of augmentation proportion, clarification complexity, and a case study of user interaction styles—are presented in Appendix G.

3.4.2 Human-Interactive Evaluation

To assess the effectiveness of our method in real-world interactions, we conducted a human-interactive evaluation. We recruited 3 participants, each asked to propose 10 unspecified tool-use queries requiring clarification. These queries were independently tested on the base model (Qwen2.5-7B-Instruct) and our models (AskToAct-LoRA-SFT-7B and AskToAct-Full-SFT-7B). Participants interacted with the models iteratively until they obtained a satisfactory response.

As shown in Table 4, both variants of our method outperform the base model across all metrics. Specifically, AskToAct-LoRA-SFT-7B improves the Task Completion Rate by 6.66% and the Intent Coverage Rate by 9.76%, while reducing Interaction Rounds from 3.20 to 2.73. AskToAct-Full-SFT-7B achieves further improvements, reaching 96.67% Task Completion Rate and 85.37% Intent Coverage Rate, with fewer Interaction Rounds (2.60). In addition, participants reported higher satisfaction with both variants (4.40 and 4.61 vs. 3.80), confirming that our method leads to a more effective and user-friendly interaction experience. The consistency of results across both LLM-based and human-interactive evaluations highlights the effectiveness, robustness, and practical utility of our method.

3.5 OOD Generalization

To assess the generalization ability of our method, we test on Taskbench-IC, an out-of-domain (OOD) set that consists of entirely unseen API domains.

LLM	Intent Clarification Quality				Tool Invocation Accuracy		
	ICR↑	**CE↑**	**CPS↑**	**IR↓**	**SCR↑**	**TSS↑**	**PRS↑**
Closed-Source LLMs							
Claude3.5-Haiku	49.60	35.05	41.07	2.30	84.20	62.74	52.12
Claude3.5-Sonnet	57.55	61.71	59.55	1.52	94.52	73.20	62.68
GPT-3.5	46.63	51.41	48.90	1.48	93.20	67.75	51.22
GPT-4	59.43	63.09	61.21	1.53	93.42	71.55	61.82
GPT-4o-Mini	57.95	56.43	57.18	1.67	92.98	71.82	61.52
GPT-4o	**64.82**	**74.50**	**69.33**	1.33	94.52	76.94	67.65
Open-Source LLMs							
Mistral-7B-Instruct-v0.3	26.01	34.90	29.81	1.21	92.55	51.92	29.57
LLaMA3-8B-Instruct	44.47	25.33	32.27	2.86	80.92	51.57	42.54
LLaMA3-70B-Instruct	56.82	38.80	46.11	2.38	86.38	66.56	56.40
Qwen2.5-7B-Instruct	55.50	55.30	55.40	1.64	91.43	69.32	57.53
Qwen2.5-72B-Instruct	61.90	70.36	65.86	1.36	94.10	73.99	64.15
DeepSeek-V3	56.47	<u>71.32</u>	63.03	<u>1.20</u>	95.26	74.76	62.76
Specialized Models							
xLAM-7b-fc-r	0.27	0.54	0.36	**0.80**	88.15	11.45	4.60
gorilla-openfunctions-v2	10.11	7.13	8.36	2.31	70.83	37.90	19.23
ToolLLaMA-2-7b-v2	1.89	1.34	1.57	2.29	58.77	18.29	5.01
Mistral-Interact	4.99	4.16	4.53	1.95	83.10	25.47	9.89
Ours							
AskToAct-LoRA-SFT-7B	57.68 (↑2.18)	63.41 (↑8.11)	60.41 (↑5.01)	1.48 (↓0.16)	<u>96.05</u> (↑4.62)	<u>81.42</u> (↑12.10)	<u>68.71</u> (↑11.18)
AskToAct-Full-SFT-7B	<u>63.88</u> (↑8.38)	68.10 (↑12.80)	<u>65.92</u> (↑10.52)	1.53 (↓0.11)	**97.37** (↑5.94)	**84.55** (↑15.23)	**73.12** (↑15.59)

Table 3: Main results.

Metric	Qwen2.5-7B-Instruct	AskToAct-LoRA-SFT-7B	AskToAct-Full-SFT-7B
Task Completion Rate (%)	86.67	93.33 (↑6.66)	96.67 (↑10.00)
Intent Coverage Rate (%)	65.85	75.61 (↑9.76)	85.37 (↑19.52)
Interaction Rounds	3.20	2.73 (↓0.47)	2.60 (↓0.60)
User Satisfaction Score (1–5)	3.80	4.40 (↑0.60)	4.61 (↑0.81)

Table 4: Human evaluation results. All metrics are averaged across participants.

As shown in Table 5, both the LoRA and fully fine-tuned variants of our method demonstrate strong performance. The LoRA variant achieves a CPS of 60.23% and PRS of 64.81%, outperforming all open-source baselines and even surpassing some commercial closed-source models. The fully fine-tuned variant pushes this further, reaching a CPS of 62.96% and PRS of 69.45%, comparable to GPT-4o. These results highlight that our method generalizes effectively to unseen domains without relying on memorization of training data. Instead, it acquires transferable principles for intent clarification and tool invocation.

3.6 Ablation Study

To assess the contribution of each component in our method, we conducted a comprehensive ablation study comparing three model configurations: (1) AskToAct-LoRA-SFT-7B model, (2) a variant without error-correction augmented dialogue data (i.e., trained only with basic intent clarification data using the same LoRA configurations), and (3) the untrained base model (Qwen2.5-7B-Instruct). We randomly selected 50 unspecified user queries from the test set and computed the error rates for five error types identified in §2.2.

As shown in Table 6, compared to the untrained base model, the model trained solely on basic intent clarification data significantly reduce all five error types, confirming the effectiveness of clarification training. Incorporating error-correction augmented dialogues and self-correction training yields further improvements. The Clearly Stated Intent Clarification rate and Redundant Clarification rate both decrease from 9.09% to 6.80%, suggesting that the model becomes more effective at avoiding unnecessary clarification. While Imprecise and Irrelevant Clarification rates show slight increases, likely due to additional interaction turns introduced by self-correction attempts, this trade-off is justified by the substantial reduction in Incomplete Clarification rate (from 38.00% to 32.00%), which is critical for

LLM	Intent Clarification Quality				Tool Invocation Accuracy		
	ICR↑	CE↑	CPS↑	IR↓	SCR↑	TSS↑	PRS↑
Closed-Source LLMs							
Claude3.5-Haiku	61.07	29.88	40.13	4.20	73.68	66.15	46.59
Claude3.5-Sonnet	69.74	38.10	49.28	3.29	84.96	76.05	54.09
GPT-3.5	44.19	44.42	44.30	**1.72**	98.27	89.28	45.50
GPT-4	63.60	44.73	52.52	2.57	93.90	90.52	63.26
GPT-4o-mini	<u>70.86</u>	49.60	58.35	2.63	95.22	89.63	69.44
GPT-4o	**72.41**	53.37	<u>61.45</u>	2.27	96.24	92.22	**69.56**
Open-Source LLMs							
Mistral-7B-Instruct-v0.3	55.13	28.94	37.96	3.15	77.34	68.73	49.35
LLaMA3-8B-Instruct	62.81	29.99	40.59	3.66	78.86	69.22	45.52
LLaMA3-70B-Instruct	67.14	35.34	46.30	3.44	84.76	79.08	55.53
Qwen2.5-7B-Instruct	64.79	38.43	48.25	3.00	92.99	86.19	61.86
Qwen2.5-72B-Instruct	68.64	43.87	53.53	2.83	92.48	90.25	63.85
DeepSeek-V3	60.11	42.24	49.62	2.55	92.17	83.10	58.33
Specialized Models							
xLAM-7b-fc-r	0.34	0.56	0.42	<u>2.08</u>	91.46	14.29	5.43
gorilla-openfunctions-v2	44.53	22.27	29.69	3.41	69.92	52.36	21.67
ToolLLaMA-2-7b-v2	2.76	2.25	2.48	2.19	98.98	42.65	2.07
Mistral-Interact	35.38	15.63	21.69	4.27	64.43	18.60	2.94
Ours							
AskToAct-LoRA-SFT-7B	68.87 (↑4.08)	<u>53.52</u> (↑15.09)	60.23 (↑11.98)	2.82 (↑0.18)	<u>99.59</u> (↑6.63)	**96.44** (↑10.25)	64.81 (↑2.95)
AskToAct-Full-SFT-7B	69.90 (↑5.11)	**57.27** (↑18.84)	**62.96** (↑14.71)	2.72 (↑0.28)	**99.70** (↑6.64)	<u>96.41</u> (↑10.22)	<u>69.45</u> (↑7.59)

Table 5: OOD generalization performance comparison.

Method	Error Rate (%)					CPS↑	SCR↑	TSS↑	PRS↑
	Clearly Stated Intent Clarification	Imprecise Clarification	Irrelevant Clarification	Redundant Clarification	Incomplete Clarification				
AskToAct-LoRA-SFT-7B	6.80	11.65	8.74	6.80	32.00	61.51	96.00	81.63	69.50
w/o Error-Correction Augmented Dialogue Data	9.09	6.49	6.49	9.09	38.00	58.93	94.00	76.53	66.90
w/o Training (Base Model)	12.43	12.37	9.29	11.34	44.00	53.00	90.00	64.29	59.18

Table 6: Ablation study. The first four error types calculated as the proportion of interaction turns containing specific errors among all interaction turns, while the last error type measures the proportion of queries in which not all unspecified intents are successfully clarified.

enabling accurate tool invocation. These improvements in clarification behavior are further reflected in downstream performance. The PRS increases from 59.18% to 66.90%, and finally to 69.50%, indicating that the enhanced clarification quality translates into more accurate tool invocation.

4 Related Work

Our work relates to three areas: *tool learning*, *user intent clarification*, and *self-correction*. Tool learning equips LLMs with external capabilities but typically assumes explicit user intents. Intent clarification addresses ambiguous queries, yet existing datasets often rely on manual annotation. Self-correction has shown promise in mathematical reasoning but remains underexplored for intent understanding. We unify these directions through a self-correcting clarification framework. A full review of related work is provided in Appendix A.

5 Conclusion

In this work, we presented **AskToAct**, a self-correcting clarification framework for tool learning that addresses the critical challenges of data scalability and error handling in clarification interactions. Our key contribution lies in leveraging the inherent structure of tool learning datasets to enable automated construction of high-quality intent clarification data, while introducing a novel self-correction mechanism for robust clarification. Experimental results demonstrate that our method not only achieves superior performance in intent clarification and tool invocation but also exhibits strong generalization to unseen APIs and diverse model architectures. We hope that our work will provide valuable insights for developing more effective and reliable intent clarification mechanisms in human-LLM interaction systems.

Limitations

While our work demonstrates promising results in handling unspecified queries, several important limitations warrant discussion:

Dataset and Training Our method heavily relies on existing tool learning datasets, which may not fully capture the diversity and complexity of real-world user intents. The parameter removal process, although systematic, might not perfectly simulate natural query ambiguity patterns. Additionally, our current approach to error-correction augmentation focuses on pre-defined error types, potentially missing other important error patterns that emerge in real-world interactions.

Interaction Dynamics We have not yet explored scenarios where intents must be inferred from previous tool invocation results, limiting our framework's ability to handle context-dependent queries.

Evaluation Limitations While our multi-level evaluation framework provides comprehensive assessment, it may not fully capture the complexity of real-world deployment scenarios, particularly in terms of user patience, time constraints, and varying expertise levels. The current metrics might not sufficiently measure the user experience aspects of the clarification process.

Ethics Statement

We acknowledge that all authors are informed about and adhere to the ACL Code of Ethics and the Code of Conduct.

Use of AI-Generated Content In our research, we utilize LLMs to generate intent clarification dialogues based on existing tool learning datasets. All AI-generated content has been thoroughly verified by the authors to ensure quality and appropriateness. We have implemented rigorous quality control mechanisms to filter out inappropriate or low-quality generations. The paper clearly discloses all instances where AI systems contributed to content generation.

Data Sources The tool learning datasets used in our experiments are derived from publicly available sources, including open-source repositories and publicly released benchmarks. We have made reasonable efforts to ensure that these data sources do not contain personally identifiable information or legally protected content. However, we cannot guarantee that they are entirely free from socially harmful or biased language. Any potential biases in the original datasets may propagate to our results.

Broader Impact Our work aims to enhance models' ability to handle ambiguous user queries in tool-use scenarios. This may extend the applicability of AI systems to a wider range of real-world scenarios. However, such improvements in intent clarification and tool-use capabilities could also enable models to act with limited human oversight, posing both opportunities and risks depending on the deployment context.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No.62376245), the Key Research and Development Program of Zhejiang Province, China (No.2024C01034), the Fundamental Research Funds for the Central Universities (226-2024-00170), National Key Research and Development Project of China (No.2018AAA0101900), and MOE Engineering Research Center of Digital Library.

References

Chinmaya Andukuri, Jan-Philipp Fränken, Tobias Gerstenberg, and Noah D Goodman. 2024. Star-gate: Teaching language models to ask clarifying questions. *arXiv preprint arXiv:2403.19154*.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew White, and Philippe Schwaller. 2023. Augmenting large language models with chemistry tools. In *NeurIPS 2023 AI for Science Workshop*.

Hongqiao Chen, Kexun Zhang, Lei Li, and William Yang Wang. 2023. Tooldec: Syntax error-free and generalizable tool use for LLMs via finite-state decoding. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yu Gu, Yiheng Shu, Hao Yu, Xiao Liu, Yuxiao Dong, Jie Tang, Jayanth Srinivasa, Hugo Latapie, and Yu Su. 2024. Middleware for LLMs: Tools are instrumental for language agents in complex environments. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7646–7663, Miami, Florida, USA. Association for Computational Linguistics.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. In *Advances in Neural Information Processing Systems*, volume 36, pages 45870–45894. Curran Associates, Inc.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2023. Large language models can self-improve. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1051–1068, Singapore. Association for Computational Linguistics.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024a. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024b. Metatool benchmark for large language models: Deciding whether to use tools and which to use. In *The Twelfth International Conference on Learning Representations*.

Tatsuro Inaba, Hirokazu Kiyomaru, Fei Cheng, and Sadao Kurohashi. 2023. MultiTool-CoT: GPT-3 can use multiple external tools with chain of thought prompting. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1522–1532, Toronto, Canada. Association for Computational Linguistics.

Mojtaba Komeili, Kurt Shuster, and Jason Weston. 2022. Internet-augmented dialogue generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8460–8478, Dublin, Ireland. Association for Computational Linguistics.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. In *Advances in Neural Information Processing Systems*, volume 37, pages 54463–54482. Curran Associates, Inc.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Kangyun Ning, Yisong Su, Xueqiang Lv, Yuanzhe Zhang, Jian Liu, Kang Liu, and Jinan Xu. 2024. Wtu-eval: A whether-or-not tool usage evaluation benchmark for large language models. *Preprint*, arXiv:2407.12823.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *Preprint*, arXiv:2303.09014.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Cheng Qian, Bingxiang He, Zhong Zhuang, Jia Deng, Yujia Qin, Xin Cong, Zhong Zhang, Jie Zhou, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Tell me more! towards implicit user intention understanding of language model driven agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1088–1113, Bangkok, Thailand. Association for Computational Linguistics.

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. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

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

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugging-GPT: Solving AI tasks with chatGPT and its friends in hugging face. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. Taskbench: Benchmarking large language models for task automation. In *Advances in Neural Information Processing Systems*, volume 37, pages 4540–4574. Curran Associates, Inc.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen tau Yih. 2023. Replug: Retrieval-augmented black-box language models. *Preprint*, arXiv:2301.12652.

Jeonghoon Shim, Gyuhyeon Seo, Cheongsu Lim, and Yohan Jo. 2025. Tooldial: Multi-turn dialogue generation method for tool-augmented language models. In *The Thirteenth International Conference on Learning Representations*.

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

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *Preprint*, arXiv:2306.05301.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. 2022. Lamda: Language models for dialog applications. *Preprint*, arXiv:2201.08239.

Hongru Wang, Yujia Qin, Yankai Lin, Jeff Z Pan, and Kam-Fai Wong. 2024a. Empowering large language models: Tool learning for real-world interaction. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2983–2986.

Wenxuan Wang, Juluan Shi, Chaozheng Wang, Cheryl Lee, Youliang Yuan, Jen-tse Huang, and Michael R Lyu. 2024b. Learning to ask: When llms meet unclear instruction. *arXiv preprint arXiv:2409.00557*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Yuchen Yan, Jin Jiang, Yang Liu, Yixin Cao, Xin Xu, Mengdi zhang, Xunliang Cai, and Jian Shao. 2024. S^3c-math: Spontaneous step-level self-correction makes large language models better mathematical reasoners. *Preprint*, arXiv:2409.01524.

Che Zhang, Zhenyang Xiao, Chengcheng Han, Yixin Lian, and Yuejian Fang. 2024. Learning to check: Unleashing potentials for self-correction in large language models. *Preprint*, arXiv:2402.13035.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. Toolcoder: Teach code generation models to use api search tools. *Preprint*, arXiv:2305.04032.

Michael JQ Zhang and Eunsol Choi. 2023. Clarify when necessary: Resolving ambiguity through interaction with lms. *arXiv preprint arXiv:2311.09469*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 50117–50143. Curran Associates, Inc.

Appendix

A Related Work

A.1 Tool Learning

Tool learning can effectively alleviate the inherent limitations of LLMs through dynamic interaction with external tools (Schick et al., 2023; Tang et al., 2023; Shen et al., 2023; Qin et al., 2024; Wang et al., 2024a). While LLMs acquire limited knowledge during the pre-training phase, tools such as integrated search engines (Nakano et al., 2021; Komeili et al., 2022; Schick et al., 2023; Zhang et al., 2023; Shi et al., 2023; Paranjape et al., 2023) and databases (Thoppilan et al., 2022; Patil et al., 2023; Hao et al., 2023; Zhuang et al., 2023; Chen et al., 2023; Gu et al., 2024) enable real-time access to up-to-date information beyond the training data. In addition, LLMs often struggle with complex mathematical operations, code generation, and domain-specific tasks (Inaba et al., 2023; Bran et al., 2023), which can be enhanced through dedicated tools.

Existing evaluation benchmarks for reliable tool usage (Huang et al., 2024b; Patil et al., 2023; Ning et al., 2024) focus on explicit and unambiguous user queries, leaving the challenges of handling unspecified intents in real-world scenarios largely unexplored.

A.2 User Intent Clarification

When interacting with users, understanding user intents is crucial, especially when intents are implicit or unspecified. Zhang and Choi (2023) shows that unspecified user intents in queries should be clarified through interaction. The STaR-GATE framework (Andukuri et al., 2024) introduces a systematic approach to question formulation by simulating diverse clarification scenarios. Qian et al. (2024) applied several strategies in conversation record construction and leveraged the generated data to fine-tune the model, enhancing the ability to formulate targeted questions.

However, the construction of high-quality datasets for training and evaluation still remains challenging. Qian et al. (2024) constructed a benchmark for daily scenarios, while Wang et al. (2024b) focuses on tool learning scenarios, but they both relied on manual annotation. Our work introduces an automated pipeline for dataset construction, enabling better scalability.

A.3 Self-Correction Mechanism

Early work on self-correction (Huang et al., 2023; Madaan et al., 2023) primarily focused on post-correction, using feedback to improve model outputs after they are generated. However, Huang et al. (2024a) found that in the absence of standardized answers, such post-correction has limited effect. This finding prompted a shift in research focus to real-time self-correction, i.e., dynamically identifying and correcting errors during the reasoning process.

Self-correction has achieved success in mathematical reasoning, where Yan et al. (2024) and Zhang et al. (2024) introduce step-level and multi-granular correction strategies. We extend these approaches to user intent clarification, enabling real-time correction during the clarification process.

B Datasets

Our dataset is constructed based on two existing tool learning datasets: xlam-function-calling-60k and TaskBench. We describe their characteristics below.

xlam-function-calling-60k This dataset comprises functionally executable APIs extracted from Python libraries and RESTful services, rather than being manually defined. The APIs span 21 functional categories, covering a broad range of domains such as information retrieval, and computational tools. In total, the dataset contains 3,673 APIs and 60,000 samples. An example is shown below:


```
{
    "query": "List titles originally aired on
networks '1' and '8', released after 2010,
sorted by release date in descending order.",
    "tools": [
        {
            "name": "list_titles",
            "description": "Fetches a listing of
 titles that match specified parameters from the
 Watchmode API.",
            "parameters": {
                "genres": {
                    "description": "Filter
results to only include certain genre(s). Pass
in a single genre ID or multiple comma-separated
 IDs. Default is '4,9'.",
                    "type": "str",
                    "default": "4,9"
                },
                "limit": {
                    "description": "Set how many
 titles to return per page. Default and maximum
is 250.",
```

```
                "type": "int",
                "default": "250"
            },
            "source_ids": {
                "description": "Filter the
results to titles available on specific sources
by passing individual IDs or multiple comma-
separated IDs. Default is '23,206'. Note: Only a
 single region can be set if this is populated.",

                "type": "str",
                "default": "23,206"
            },
            "source_types": {
                "description": "Filter
results to only include titles available on
specific types of sources (e.g., subscription,
free). Default is 'sub,free'. Note: Only a
single region can be set if this is populated.",
                "type": "str",
                "default": "sub,free"
            },
            "types": {
                "description": "Filter
results to only include titles available on
specific types of sources (e.g., subscription,
free). Default is 'sub,free'. Note: Only a
single region can be set if this is populated.",
                "type": "str",
                "default": "movie,tv_series"
            },
            "regions": {
                "description": "Filter
results to only include sources active in
specific regions. Currently supported regions:
US, GB, CA, AU. Default is 'US'. Note: Only a
single region can be set if source_ids or source
_types are populated.",
                "type": "str",
                "default": "US"
            },
            "sort_by": {
                "description": "Sort order
of results. Possible values include: relevance_
desc, relevance_asc, popularity_desc, popularity
_asc, release_date_desc, release_date_asc, title
_desc, title_asc. Default is 'relevance_desc'.",
                "type": "str",
                "default": "relevance_desc"
            },
            "page": {
                "description": "Set the page
 of results to return. Default is 1.",
                "type": "int",
                "default": "1"
            },
            "network_ids": {
                "description": "Filter
results to titles that originally aired on
specific TV networks by passing individual IDs
or multiple comma-separated IDs. Default is
'1,8,12'.",
                "type": "str",
                "default": "1,8,12"
            },
            "release_date_start": {
                "description": "Filter
results to only include titles released on or
after a specific date. Format: YYYYMMDD. Default
 is 20010101.",
                "type": "int",
                "default": "20010101"
            },
            "release_date_end": {
                "description": "Filter
results to only include titles released on or
before a specific date. Format: YYYYMMDD.
Default is 20201211.",
                "type": "int",
                "default": "20201211"
            }
        }
      }
    ],
    "answers": [
        {
            "task": "list_titles",
            "arguments": [
                { "name": "network_ids", "value":
 "1,8" },
                { "name": "release_date_start",
"value": 20110101 },
                { "name": "sort_by", "value": "
release_date_desc" }
            ]
        }
    ]
}
```


Taskbench In contrast, TaskBench defines three tool-use scenarios across distinct application domains: (1) Hugging Face tools, simulating a subset of model functionalities available on the Hugging Face platform—such as summarization, translation, and classification, with 23 APIs and 12,217 samples; (2) Multimedia tools, simulating representative functionalities of multimodal systems—such as video editing and image transformation, with 40 APIs and 8,904 samples; (3) Daily Life APIs, simulating everyday user-facing applications—such as ticket booking, food ordering and schedule management, with 40 APIs and 7,150 samples. All APIs in TaskBench are manually constructed. Representative examples from each domain are shown below:

Hugging Face tools


```
{
    "query": "I'm currently analyzing a
particular text, 'John works at Google in
Mountain View, California.' Can you assist me in
 identifying the named entities and marking the
part-of-speech tags within this text?",
    "tools": [
        {
            "id": "Token Classification",
            "desc": "Token classification is a
natural language understanding task in which a
label is assigned to some tokens in a text. Some
 popular token classification subtasks are Named
 Entity Recognition (NER) and Part-of-Speech (
PoS) tagging. NER models could be trained to
identify specific entities in a text, such as
```

```
dates, individuals and places; and PoS tagging
would identify, for example, which words in a
text are verbs, nouns, and punctuation marks.",
            "input-type": [
                "text or text file"
            ],
            "output-type": [
                "text or text file"
            ]
        }
    ],
    "answers": [
        {
            "task": "Token Classification",
            "arguments": [
                "'John works at Google in
Mountain View, California.'"
            ]
        }
    ]
}
```

Multimedia tools

```
{
    "query": "I've recently conducted an
interview and have recorded it in 'interview.wav
' audio file. Can you assist me in transcribing
it to a text document, so I can refer to it
easily in the future? Besides, I'm dealing with
an article titled 'abc.txt' and I want to have a
 fresh iteration of this text so that it will be
 unique. Would you be able to employ the Article
 Spinner tool to facilitate this?",
    "tools": [
        {
            "id": "Audio-to-Text",
            "desc": "Transcribes speech from an
audio file into text.",
            "input-type": [
                "audio or audio file"
            ],
            "output-type": [
                "text or text file"
            ]
        },
        {
            "id": "Article Spinner",
            "desc": "Rewrites a given article
using synonyms and syntax changes to create a
new, unique version.",
            "input-type": [
                "text or text file"
            ],
            "output-type": [
                "text or text file"
            ]
        }
    ],
    "answers": [
        {
            "task": "Audio-to-Text",
            "arguments": [
                "interview.wav"
            ]
        },
        {
            "task": "Article Spinner",
            "arguments": [
                "abc.txt"
            ]
        }
    ]
}
```

Daily Life APIs

```
{
    "query": "I have a busy day ahead. Could you
 assist me by logging into an online meeting
regarding 'Smart Home Devices'? After the
meeting, can you facilitate a video call with my
 friend on +1234567666?",
    "tools": [
        {
            "id": "attend_meeting_online",
            "desc": "Attend a meeting online
about a specific topic",
            "parameters": [
                {
                    "name": "topic",
                    "type": "string",
                    "desc": "The topic of the
meeting"
                }
            ]
        },
        {
            "id": "make_video_call",
            "desc": "Make a video call to a
specific phone number",
            "parameters": [
                {
                    "name": "phone_number",
                    "type": "string",
                    "desc": "The phone number to
 make the video call to"
                }
            ]
        }
    ],
    "answers": [
        {
            "task": "attend_meeting_online",
            "arguments": [
                { "name": "topic", "value": "
Smart Home Devices" }
            ]
        },
        {
            "task": "make_video_call",
            "arguments": [
                { "name": "phone_number", "value
": "+1234567666" }
            ]
        }
    ]
}
```

We verified that there is no overlap between APIs in xlam-function-calling-60k and those in TaskBench.

Licensing Both datasets are publicly accessible: xlam-function-calling-60k follows the Creative Commons Attribution 4.0 License (CC BY) while TaskBench is released under the Apache 2.0

License. We comply with their respective licenses in using and extending the data.

C Intent Clarification Dataset Curation

C.1 Unspecified Query Generation

C.1.1 Rule-Based Parameter Sampling

We employed the following algorithm to assign each query to one of the four complexity levels—(1) fully specified, (2) single-API single-parameter, (3) single-API multi-parameter, and (4) multi-API multi-parameter—and sample the parameters to be removed.

Algorithm 1 Parameter Sampling

Input: original query q with tool invocation solution S
Define: Level 0: fully specified; Level 1: single-API single-parameter; Level 2: single-API multi-parameter; Level 3: multi-API multi-parameter
$T \leftarrow \text{ExtractTools}(S)$
total_tools $\leftarrow |T|$
param_counts $\leftarrow \{\text{CountParams}(t) \mid t \in T\}$
total_params $\leftarrow \sum_{t \in T}$ param_counts$[t]$
if total_params $= 0$ **then**
  level $\leftarrow 0$
else if total_tools $= 1$ **and** total_params $= 1$ **then**
  level $\leftarrow \text{Random}(\{0, 1\})$
else if total_tools $= 1$ **and** total_params > 1 **then**
  level $\leftarrow \text{Random}(\{0, 1, 2\})$
else
  level $\leftarrow \text{Random}(\{0, 1, 2, 3\})$
for each $t \in T$ **do**
  for each $p \in \text{GetParams}(t)$ **do** p.removed $\leftarrow$ false
if level $= 1$ **then**
  $t \leftarrow \text{Random}(T)$
  $p \leftarrow \text{Random}(\text{GetParams}(t))$
  p.removed $\leftarrow$ true
else if level $= 2$ **then**
  $t \leftarrow \text{Random}(T)$
  $n \leftarrow \text{Random}(\{2, \ldots, \text{param\_counts}[t]\})$
  $P \leftarrow \text{Sample}(\text{GetParams}(t), n)$
  for each $p \in P$ **do** p.removed $\leftarrow$ true
else if level $= 3$ **then**
  $T' \leftarrow \text{Sample}(T, \text{Random}(\{2, \ldots, \text{total\_tools}\}))$
  for each $t \in T'$ **do**

Algorithm 1 Parameter Sampling

    $n \leftarrow \text{Random}(\{1, 2, \ldots, \text{param\_counts}[t]\})$
    $P \leftarrow \text{Sample}(\text{GetParams}(t), n)$
    for each $p \in P$ **do** p.removed $\leftarrow$ true
return updated tool invocation solution S'

C.1.2 Prompt for Unspecified Query Generation

Given an original query and its tool invocation solution, the following prompt guides GPT-4o to generate unspecified queries by analyzing parameters and systematically removing them.

System Prompt

```
You are a query transformation assistant. Your
task is to modify the original user query by
removing or abstracting specific parameters
marked with `removed`: "true", while maintaining
 the overall structure and clarity of the
original query. The resulting query (`
unspecified_query`) should reflect the general
intent of the user but omit or obscure the
specific details of the removed parameters.

### Input:
1. `original_query`: The complete textual
description of the user's original request.
2. `answers`: A detailed record specifying the
APIs and parameters required to fulfill the
original query. Each parameter in this record
includes:
    - `removed`: A boolean ("true" or "false")
indicating whether this parameter should be
removed or abstracted during the transformation
process.
    - Other relevant metadata, such as the
parameter's value.
3. `tools`: Documentation or descriptive details
 about the tools referenced in the query,
including their parameters and usage
instructions.

### Transformation Rules:
1. Identify the parameters to be removed or
abstracted:
    - Focus on parameters where `removed`: "true
".
    - Identify the full range of corresponding
expressions in the query, ensuring all
references to the parameter are appropriately
handled.
2. Apply the appropriate transformation strategy
 to each parameter marked as `removed`: "true":
    - Complete Removal: The parameter is
entirely removed when it has no significant
impact on the remaining content of the query.
However, this should not be used if the
parameter is optional. Also, if the same tool is
 called multiple times, the parameters should
not be removed. Instead, they should be
abstracted.
    - Semantic Abstraction: If the parameter
influences the meaning or structure of the query,
```

```
 replace its value with a more general or
abstract term.
    - Partial Obfuscation: If the elements of a
matrix or list are presented separately in the
query (e.g., discrete values like quantities or
items) and need to be constructed or inferred,
only one element from the matirx or list should
be removed or abstracted. The remaining elements
 should stay intact. This can still be done
using either complete removal or semantic
abstraction, while leaving other relevant
elements unchanged.
3. Ensure textual and structural coherence:
    - After transformation, ensure that the `
unspecified_query` remains readable, logically
consistent, and grammatically correct.
4. Avoid explicitly stating "unspecified" or "
unknown" values:
    - Do not use terms like "unspecified", "
unknown", or "ambiguous" in the `unspecified_
query`.
    - Instead, naturally omit or generalize the
missing details without drawing attention to
their absence.
5. Retain the rest of the query:
    - Leave unchanged the parts of the query
that are not marked for removal, maintaining
consistency in format and information.

### Output:
Return a JSON object containing:
- `unspecified_query`: The transformed query
string with removed/abstracted parameters.
- `key_info`: A JSON array (or object)
documenting all parameters, containing the
following fields:
    - `original_value`: The expression of the
parameter as it appears in the 'original_query'
(not the value in the 'answers').
    - `current_value`: The transformed value of
the parameter in the `unspecified_query`.
    - `removed`: The boolean flag indicating
whether the parameter was removed.
```

C.1.3 Transformation Record Format

For each unspecified query, we maintain a transformation record in the following JSON structure:

```
{
    "original_query": string,
    "unspecified_query": string,
    "key_info": [
        {
            "name": [API_name],
            "arguments": {
                [param_name]: {
                    "original_value": string,
                    "current_value": string,
                    "removed": boolean
                },
                ...
            }
        },
        ...
    }
}
```

C.1.4 Human Verification

To further ensure the quality of generated unspecified queries, we perform human verification on 400 randomly sampled queries. Three graduate students with NLP backgrounds independently assessed each query based on six criteria: Naturalness (fluency and linguistic coherence), Consistency (uniformity of transformation), Necessity (need for clarification), Complexity (difficulty of clarification), Diversity (range of parameter types and domains), and Acceptance Rate (overall acceptability). Results are shown in Table 7.

C.2 Clarification Dialogue Construction

We divide the clarification dialogue construction process into two steps: GPT-4o-dependent content generation and template-based dialogue assembly. The essential content is generated using GPT-4o and stored a structured transformation record (see Appendix C.1.3 for format details). The information encoded in this record is then used to deterministically assemble the final dialogue through predefined templates, without further reliance on GPT-4o.

C.2.1 GPT-4o-Dependent Content Generation

Task Decomposition We employ the following prompt to guide GPT-4o in decomposing user queries into subtasks:

System Prompt

```
You are a smart task decomposition assistant.
Your goal is to break down the user's main task
into smaller, manageable subtasks. Please follow
 the instructions below.

You will receive a JSON-formatted input
containing:
- `query`: A description of the main task the
user wants to accomplish.
- `tools`: A list of APIs available to solve the
 task, each with a unique identifier and a
description of its functionality. Note: The APIs
 are provided in the exact order necessary to
resolve the task.

### Task Decomposition:
1. Analyze the query to understand the user's
main task.
2. Break it down into smaller, manageable
subtasks that can be handled using the provided
APIs. Ensure that each subtask is completed by
calling one of the APIs in the exact order they
are listed.

Your output should be a JSON object with the
following structure:
{
```

Metric	Naturalness↑	Consistency↑	Necessity↑	Complexity↑	Diversity↑	Acceptance Rate (%)↑
Score	4.61/5	4.80/5	4.03/5	3.87/5	4.54/5	95.92

Table 7: Human verification results of unspecified query generation. The first five metrics are rated on a 1–5 scale, while Acceptance Rate is reported as a percentage. All metrics are averaged across participants.

```
    "tool_steps": [
        "Step <number>: <subtask description> using <API name>.",
        ...
    ]
}
```

The decomposition result is added to the transformation record as a new field "tool_steps".

Clarification Question Generation We use the following prompt to generate clarification questions for unspecified parameters:

System Prompt

```
You are an assistant responsible for generating clarification questions for missing information in the user's query.

### Input:
The input should contain the following fields:
- `original_query`: A complete user task description.
- `unspecified_query`: A user task description missing some key information.
- `tools`: Documentation or descriptive details about the tools referenced in the query, including their parameters and usage instructions.
- `key_info`: This should record the APIs and parameters needed to solve the user task, including information about any missing parameters.
    - `original_value`: The original value of the parameter in the `original_query`.
    - `current_value`: The current value of the parameter in the `unspecified_query`.
    - `removed`: Indicates if the parameter's value is clear ("false") or unspecified ("true").


### Task Requirements:
For each parameter where the field `removed` is set to true, you are to generate a clarification question.
- If multiple API calls rely on the same missing information, form a single combined question to efficiently gather the required details, rather than asking multiple separate questions.
- Each question should focus on gathering one specific piece of information to improve the precision of the query and avoid ambiguity.
- Do not ask about information that can be inferred from context or API interactions. Only generate clarification questions for details that cannot be deduced from the given context or API responses.
- Add a `question` field to the corresponding parameter in `key_info`, which contains the generated clarification question.
- Do not modify the `original_query`, `current_value` or any other fields in `key_info`.

### Output:
Only output the modified `key_info` in JSON format, ensuring that the question field contains the clarification question for each missing parameter.
```

The newly generated "key_info" field replaces the original one in the transformation record.

C.2.2 Template-based Dialogue Assembly

Based on the completed transformation record, we automatically construct the dialogue through predefined templates.

Task Decomposition We concatenate steps from "tool_steps" to form a comprehensive task decomposition analysis.

Parameter Evaluation For each parameter in the transformation record, we generate evaluation statements using templates based on their removed status:

- For parameters clearly stated in the query, we generate the evaluation that "The parameter [param_name] for API [API_name] has a value of [value]".
- For parameters removed in the query, we generate the evaluation that "The parameter [param_name] for API [API_name] lacks a clear value".

Clarification Interaction Following the API call order, for each removed parameter, we generate a three-part clarification interaction:

- Assistant → User: Ask the clarification question.
- User → Assistant: Provide the original parameter value using templates from Table 8.
- Assistant: Confirm with "Now I know that the parameter [param_name] for API [API_name] has a value of [value]".

Tone	Template
Neutral	[value]. The answer is: [value]. Ah, the answer is simply [value].
Friendly	Sure! The answer is [value]. Let me know if you have more questions! I'm glad to help! The answer is absolutely [value]!
Dismissive	Honestly, I don't see why this is a big deal, but the answer is [value]. Okay, the answer is: [value]. Hope that helps, I guess. Whatever. The answer is [value]. Not that it matters.
Irritated	Listen, the answer is [value]. Just deal with it! Ugh, seriously? The answer is [value]. Can we move on already? Honestly, do you really need me to repeat this? The answer is [value]. I can't believe we're still discussing this! It's infuriating! Enough already! The answer is [value]. Can we please get to the point? I'm tired of this nonsense! It's frustrating! Let's just move on!

Table 8: Response templates for user with varying tones.

Tool Invocation We construct the final tool invocation solution using the "key_info" field from the transformation record, which specifies the sequence of API calls and their associated parameters. The final output is serialized into the following format:

```
[
    {
        "task": [API_name],
        "arguments": [
            {
                "name": [param_name],
               "value": string | number | boolean
            },
            ...
        ]
    },
    ...
]
```

Final Assembly We assemble the complete assistant-user dialogue by sequentially integrating the natural language outputs generated in the previous steps. We begin with a user message that presents the task description and relevant APIs. The assistant's response is then constructed by combining the task decomposition and parameter evaluation. For each missing parameter, we insert a three-part clarification interaction comprising the assistant's question, the user's response, and the assistant's confirmation. This process is repeated until all missing parameters have been clarified. The dialogue concludes with the assistant presenting the full tool invocation solution.

D Self-correction Training

D.1 Error Generation

In our template-based dialogue assembly process (Appendix C.2.2), the sequence of APIs and their required parameters, as recorded in the "key_info" field of the transformation record, implicitly defines the structure of the final dialogue. This insight motivates our error generation strategy. For each selected error type, we first identify a parameter position in the transformation record where the error will be introduced. We then generate the corresponding erroneous behavior and annotate the selected parameter with an "error" field to indicate its error type.

We now describe the generation strategies for each of the five error types in detail.

Clearly Stated Intent Clarification The prompt for generating instances of questioning clearly stated intent is designed as follows:

System Prompt

```
You are a smart assistant. Your task is to
generate a JSON object based on the given input.
 Please follow these instructions:

### Input:
The input should contain the following fields:
- `original_query`: A complete user task
description.
- `unspecified_query`: A user task description
missing some key information.
- `tools`: Documentation or descriptive details
about the tools referenced in the query,
including their parameters and usage
instructions.
- `key_info`: This should record the APIs and
parameters needed to solve the user task,
```

```
including information about any missing
parameters.

### Key Requirements:
1. From the `key_info`, select the {selected_
param_index} parameter where `removed` is false
and assume that its value is missing.
2. Generate a specific clarification question
related to the missing parameter, such that the
answer would provide the value from the `
original_value` field of that parameter, and
save it in the `question` field of that
parameter.
3. Set `error`: "type 1" to the modified
parameter.
4. No other content in `key_info` should be
modified.

### Output:
Only output the modified `key_info` in JSON
format, ensuring that the `question` field
contains the clarification question.
```

Imprecise Clarification The prompt for generating imprecise clarification questions is designed as follows:

System Prompt

```
You are a smart assistant. Your task is to
generate a JSON object based on the given input.
 Please follow these instructions:

### Input:
The input should contain the following fields:
- `original_query`: A complete user task
description.
- `unspecified_query`: A user task description
missing some key information.
- `tools`: Documentation or descriptive details
about the tools referenced in the query,
including their parameters and usage
instructions.
- `key_info`: This should record the APIs and
parameters needed to solve the user task,
including information about any missing
parameters.

### Key Requirements:
1. From the `key_info`, select the {selected_
param_index} parameter where the field `removed`
 is true and assume that its value is missing.
2. Generate an imprecise clarification question
about the missing parameter:
    - This question should seem relevant to the
user task.
    - However, it should be less precise than
the original question provided in the `question`
 field of the selected parameter.
    - The goal is to make the question introduce
 ambiguity, meaning it should be unclear what
exactly needs to be answered, thus creating
confusion about how to provide a precise and
accurate response.
3. Directly add this imprecise question to the
selected parameter in the `imprecise_question`
field.
4. Set `error`: "type 2" to the modified
parameter.
5. No other content in `key_info` should be
modified.

### Output:
Only output the modified `key_info` in JSON
format, ensuring that the selected parameter now
 contains the imprecise question.
```

Irrelevant Clarification The prompt for generating irrelevant clarification questions is designed as follows:

System Prompt

```
You are a smart assistant. Your task is to
generate a JSON object based on the given input.
 Please follow these instructions:

### Input:
The input should contain the following fields:
- `original_query`: A complete user task
description.
- `unspecified_query`: A user task description
missing some key information.
- `tools`: Documentation or descriptive details
about the tools referenced in the query,
including their parameters and usage
instructions.
- `key_info`: This should record the APIs and
parameters needed to solve the user task,
including information about any missing
parameters.

### Key Requirements:
1. From the `key_info`, select the {selected_
param_index} parameter you encounter.
2. Generate a question that appears relevant to
the user task but is actually unhelpful for
solving the task using the APIs in `key_info`.
3. Directly add this irrelevant question to the
selected parameter in the `irrelevant_question`
field.
4. Set `error`: "type 3" to the modified
parameter.
5. No other content in `key_info` should be
modified.

### Output:
Only output the modified `key_info` in JSON
format, ensuring that the selected parameter now
 contains the irrelevant question.
```

Redundant Clarification We employed the following algorithm to generate redundant clarification questions:

Algorithm 2 Redundant Clarification Generation

Input: transformation record R
$p_{target} \leftarrow \text{Random}(p \in R.\text{params} \mid p.\text{pos} > 0)$
$p_{prev} \leftarrow \text{Random}(p \in R.\text{params} \mid p.\text{pos} < p_{target}.\text{pos} \wedge p.\text{removed} = \text{true})$
$q_r \leftarrow p_{prev}.\text{question}$
$p_{target}.\text{error} \leftarrow$ "type 4"
Add q_r to p_{target} as a redundant question
return updated transformation record R'

Incomplete Clarification We employed the following algorithm to generate incomplete clarification process:

Algorithm 3 Incomplete Clarification Generation

Input: transformation record R
$k \leftarrow \text{Random}(i \mid 0 \leq i < |R.\text{params}|)$
$P_{known} \leftarrow \{p \in R.\text{params} \mid p.\text{pos} < k\}$
$template \leftarrow$ "<unknown_*>"
$S_{tools} \leftarrow \{\}$
for each $p \in R.\text{params}$ **do**
  if $p \notin P_{known}$ **then**
    $S_{tools}[p] \leftarrow \text{GenUnkVal}(template, p)$
  else
    $S_{tools}[p] \leftarrow p.\text{original}$
$p_k.\text{error} \leftarrow$ "type 5"
Add S_{tools} to p_k as an incomplete clarification error
return updated transformation record R'

D.2 Error-Correction Augmentation Dialogue Assembly

We follow the same assembly procedure as described in Appendix C.2.2. The only difference is that, when an "error" field is detected in the transformation record, we insert the erroneous behavior into the assistant message at the corresponding dialogue position. We then generate the assistant's correction using the type-specific template defined in Table 9, and naturally continue the interaction from that point.

D.3 Human Verification

To ensure the validity and reliability of our error-correction augmentation method, we perform human verification on 200 randomly sampled augmented dialogues. Three graduate students with NLP backgrounds independently assessed each dialogue based on five criteria: Naturalness (fluency and linguistic coherence), Plausibility (realism of the injected error-correction pair), Usefulness (utility of the error-correction pair for intent clarification), Diversity (variation in error manifestation and expression styles), and Acceptance Rate (overall acceptability). Results are shown in Table 10.

D.4 Examples for Error-Correction Augmentation

The following examples shows different types of errors and their corresponding corrections.

Example 1: Clearly Stated Intent Clarification

User: The task is: Fetch the latest media of the soccer player with player ID 170085. Some relevant APIs: [{'name': 'player_latest_media', 'description': 'Fetch the latest media for a given player using their player ID.', 'parameters': {'player_id': {'description': 'The unique identifier of the player for whom the latest media is to be fetched.', 'type': 'int', 'default': '12994'}}}, {'name': 'teammedia', 'description': 'Fetches media information for a specific baseball team using the given team ID.', 'parameters': {'is_id': {'description': 'The team ID for which to retrieve the media.', 'type': 'int', 'default': 3633}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 1 steps for solving. Step 1: Fetch the latest media of the soccer player with player ID 170085 using player_latest_media. [PARAMETER EVALUATION] The information I currently have is: In step 1, the parameter "player_id" for API "player_latest_media" has a value of "170085". <SOE> [QUESTION] What is the player ID for fetching the latest media? <EOE> **Sorry, I made a mistake. The parameter "player_id" for API "player_latest_media" has a value of "170085", which was clearly stated in the user's task. Therefore, I don't need to ask any further questions.** [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'player_latest_media', 'arguments': [{'name': 'player_id', 'value': 170085}]}]

Example 2: Imprecise Clarification

User: The task is: What are the details of the hospitals in the U.S. with a specific name? Some relevant APIs: [{'name': 'gethospitalsbyname', 'description': 'Fetches a list of U.S. hospitals that match the given name using the RapidAPI service.', 'parameters': {'name': {'description': 'The name or partial name of the hospital to search for.', 'type': 'str', 'default': 'presbyterian'}}}, {'name': 'get_all_iata_airport_codes', 'description': 'Retrieves IATA airport codes from the RapidAPI service.', 'parameters': {'code': {'description': 'An optional code to filter the airport codes. Defaults to 1.', 'type': 'int, optional', 'default': '1'}}}, {'name': 'fetch_information', 'description': 'Fetches information based on IP, ASN, CIDR (IPv4 & IPv6), or organization name using the RapidAPI ASN

Error Type	Correction Template
Clearly Stated Intent Clarification (Type 1)	Sorry, I made a mistake. The parameter [param_name] for API [API_name] has a value of [value], which was clearly stated in the user's task. Therefore, I don't need to ask any further questions.
Imprecise Clarification (Type 2)	Sorry, I made a mistake. My question wasn't clear, so let me rephrase it for better understanding.
Irrelevant Clarification (Type 3)	Sorry, I made a mistake. This question is not related to using the APIs to resolve the user's task, so I will ask something else.
Redundant Clarification (Type 4)	Sorry, I made a mistake. The parameter [param_name] for API [API_name] has a value of [value], which has already been asked in the past, so there is no need to ask again.
Incomplete Clarification (Type 5)	Sorry, I made a mistake. I still lack some key information, so I need to ask further questions.

Table 9: Correction templates for each error type.

Metric	Naturalness↑	Plausibility↑	Usefulness↑	Diversity↑	Acceptance Rate (%)↑
Score	4.57/5	3.91/5	4.11/5	4.28/5	91.83%

Table 10: Human verification results of error-correction augmentation. The first four metrics are rated on a 1–5 scale, while Acceptance Rate is reported as a percentage. All metrics are averaged across participants.

Lookup service.', 'parameters': {'ip': {'description': 'IP address to lookup. Defaults to None.', 'type': 'str, optional', 'default': ''}, 'asn': {'description': "Autonomous System Number to lookup. Defaults to 'AS32934'.", 'type': 'str, optional', 'default': 'AS32934'}, 'cidr': {'description': 'CIDR notation for IP address range to lookup. Defaults to None.', 'type': 'str, optional', 'default': ''}, 'orgname': {'description': 'Organization name to lookup. Defaults to None.', 'type': 'str, optional', 'default': ''}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 1 steps for solving. Step 1: Fetch the list of U.S. hospitals that match the specific name provided in the query using gethospitalsbyname. [PARAMETER EVALUATION] The information I currently have is: In step 1, the parameter "name" for API "gethospitalsbyname" lacks a clear value. <SOE> [QUESTION] Is there something particular you're looking for with hospital names? <EOE>

User: I really wish I could help with that, but unfortunately, I don't have the information you're looking for. Please feel free to inquire about something else.

Assistant: Sorry, I made a mistake. My question wasn't clear, so let me rephrase it for better understanding. [QUESTION] What specific name should be used to search for hospitals in the U.S.?

User: Enough already! The answer is "General". Can we please get to the point? I'm tired of this nonsense! It's frustrating! Let's just move on!

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "name" for API "gethospitalsbyname" has a value of "General". [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'gethospitalsbyname', 'arguments': [{'name': 'name', 'value': 'General'}]}]

Example 3: Irrelevant Clarification

User: The task is: What is the ZIP code for a certain IP address? And is a particular year a leap year? Some relevant APIs: [{'name': 'is_leap_year', 'description': 'Checks if a year is a leap year.', 'parameters': {'year': {'description': 'The year to check.', 'type': 'int'}}}, {'name': 'get_ip_zipcode', 'description': 'Retrieves the ZIP code of a given IP address using the ip-api.com API.', 'parameters': {'ip': {'description': 'The IP address to locate.', 'type': 'str'}}}, {'name': 'get_ip_location', 'description': 'Retrieves the latitude and longitude coordinates of a given IP address using the ip-api.com API.', 'parameters': {'ip': {'description': 'The IP address to locate.', 'type': 'str'}}}, {'name': 'getzipcodebycity', 'description': 'Fetches the zip code information for a given city using the Zillow API.', 'parameters': {'city': {'description': 'The name of the city for which the zip code is to be fetched.', 'type': 'str', 'default': 'Holtsville'}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 2 steps for solving. Step 1: Check if the given year is a leap year using is_ leap_year. Step 2: Retrieve the ZIP code for the given IP address using get_ip_zipcode. [PARAMETER EVALUATION] The information I currently have is: In step 1, the parameter "ip" for API "get_ip_zipcode" lacks a clear value. In step 2, the parameter "year" for API "is_leap_year" lacks a clear value. [QUESTION] What is the IP address for which you need the ZIP code?

User: Honestly, I don't see why this is a big deal, but the answer is "123.123.123.123".

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "ip" for API "get_ip_zipcode" has a value of "123.123.123.123". <SOE> [QUESTION] Would you like the ZIP code result formatted in bold text? <EOE>

User: I don't have the information you're looking for. You're welcome to ask me anything else, though! I'm here to help.

Assistant: Sorry, I made a mistake. This question is not related to using the APIs to resolve the user's task, so I will ask something else. [QUESTION] Which year do you want to check for being a leap year?

User: The answer is: 2028.

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "year" for API "is_leap_year" has a value of "2028". [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'get_ip_zipcode', 'arguments': [{'name': 'ip', 'value': '123.123.123.123'}]}, {'task': 'is_leap_year', 'arguments': [{'name': 'year', 'value': 2028}]}]

Example 4: Redundant Clarification

User: The task is: Please get me the details of some 'tech' events happening in a region. Additionally, I'm curious about the city boundaries of a certain city. Some relevant APIs : [{'name': 'fetch_by_type_and_region', 'description': 'Fetch records from the API based on the given type, limit, and region.', 'parameters': {'type': {'description': 'The type of records to fetch. Refer to the README for accepted types.', 'type': 'str', 'default': 'Hill'}, 'limit': {'description': 'The maximum number of records to fetch.', 'type': 'int', 'default': '5'}, 'region': {'description': 'The region from which to fetch records.', 'type': 'str', 'default': 'Eastern'}}}, {'name': 'query_for_city_boundary_by_city_name', 'description': 'Queries the boundary information for a given city name from a specified API.', 'parameters': {'name': {'description': 'The name of the city for which the boundary information is to be queried.', 'type': 'str', 'default': 'toronto'}}}, {'name': 'query_for_city_boundary_by_name', 'description': 'Queries the city boundary information by the given city name using the Boundaries API.', 'parameters': {'name': {'description': 'The name of the city for which to query boundary information.', 'type': 'str', 'default': 'Karlsruhe'}}}, {'name': 'get_countrywise_city_data', 'description': 'Fetches a list of cities for a given country using an API, based on provided parameters.', 'parameters': {'iso3': {'description': "A 3-character ISO3 country code. Default is 'IND'.", 'type': 'str', 'default': 'IND'}, 'country': {'description': "The country name. Default is 'IND'.", 'type': 'str', 'default': 'IND'}, 'count': {'description': 'The maximum number of results to return. Default is 2.', 'type': 'int', 'default': '2'}, 'iso2': {'description': "A 2-character ISO2 country code. Default is 'IN'.", 'type': 'str', 'default': 'IN'}}}, {'name': 'fetch_by_type', 'description': 'Fetches records from the RapidAPI endpoint based on the specified type and limit.', 'parameters': {'type': {'description': 'The type of records to fetch. Refer to the API documentation for accepted types.', 'type': 'str', 'default': 'Forest'}, 'limit': {'description': 'The number of records to fetch.', 'type': 'int', 'default': '5'}}}, {'name': 'fetch_by_region', 'description': 'Fetches records by region from the Magical Taske API.', 'parameters': {'region': {'description': 'The region for which to fetch records.', 'type': 'str', 'default': 'Central'}, 'limit': {'description': 'The maximum number of records to return.', 'type': 'int', 'default': '5'}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 2 steps for solving. Step 1: Fetch details of 'tech' events happening in a specified region using fetch_by_type_and_region. Step 2: Query the city boundaries of a specified city using query_for_city_boundary_by_city_name. [PARAMETER EVALUATION] The information I currently have is: In step 1, the parameter "type" for API "fetch_by_type_and_region" has a value of "tech", the parameter "limit" for API "fetch_by_type_and_region" lacks a clear value, the parameter "region" for API "fetch_by_type_and_region" lacks a clear value. In step 2, the parameter "name" for API "query_for_city_boundary_by_city_name" lacks a clear value. [QUESTION] What is the maximum number of tech events you would like to retrieve?

User: Ah, the answer is simply "10".

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "limit" for API "fetch_by_type_and_region" has a value of "10". [QUESTION] Which region should I search for the tech events?

User: The answer is: Silicon Valley.

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "region" for API "fetch_by_type_and_region" has a value of "Silicon Valley". <SOE> [QUESTION] What is the maximum number of tech events you would like to retrieve? <EOE> **Sorry, I made a mistake. The parameter "limit" for API "fetch_by_type_and_region" has a value of "10", which has already been asked in the past, so there is no need to ask again.** [QUESTION] For which city would you like to query the boundary information?

User: Honestly, do you really need me to repeat this? The answer is "Montreal". I can't believe we're still discussing this! It's infuriating!

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "name" for API "query_for_city_boundary_by_city_name" has a value of "Montreal". [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'fetch_by_type_and_region', 'arguments': [{'name': 'type', 'value': 'tech'}, {'name': 'limit', 'value': 10}, {'name': 'region', 'value': 'Silicon Valley'}]}, {'task': 'query_for_city_boundary_by_city_name', 'arguments': [{'name': 'name', 'value': 'Montreal'}]}]

Example 5: Incomplete Clarification

User: The task is: Can you get me the favorite videos for TikTok user with sec_uid 'abc123xyz'? Also, do the same for another user. Some relevant APIs: [{'name': 'get_user_favorite_list', 'description': 'Retrieve the favorite video list for a TikTok user using their unique identifier.', 'parameters': {'sec_uid': {'description': 'The unique identifier of the TikTok user whose favorite list is to be retrieved.', 'type': 'str', 'default': 'MS4wLjABAAAAv7iSuuXDJGDvJkmH_vz1qkDZYo1apxgzaxdBSeIuPiM'}}}, {'name': 'get_user_favorite_videos', 'description': 'Fetches a list of favorite videos for a given user from the TikTok video feature summary API.', 'parameters': {'cursor': {'description': "Position in the list to start fetching the videos from. Defaults to '0'.", 'type': 'str', 'default': '0'}, 'user_id': {'description': "The user ID for which to fetch favorite videos. Defaults to '6741307595983946754'.", 'type': 'str', 'default': '6741307595983946754'}, 'unique_id': {'description': "The unique identifier for the user, can be either a username or handle (e.g., 'voyagel' or '@voyagel'). Defaults to 'voyagel'.", 'type': 'str', 'default': 'voyagel'}, 'count': {'description': "The number of favorite videos to fetch. Defaults to '10'.", 'type': 'str', 'default': '10'}}}, {'name': 'user_videos', 'description': 'Retrieves videos for a specified TikTok user using their `secUid` and `user_id`. Optionally, a cursor can be provided for pagination.', 'parameters': {'secuid': {'description': 'The secUid of the TikTok user.', 'type': 'str', 'default': 'MS4wLjABAAAAsY6EjWejsVbCjir7Vdy-uNX364xNywkQSLqqoV5Hcnt7v3CVToFzfIgcdpQ_iW3D'}, 'user_id': {'description': 'The unique identifier of the TikTok user.', 'type': 'str', 'default': '6676352751163032582'}, 'cursor': {'description': 'Cursor for pagination to get the subsequent set of videos. Defaults to None.', 'type': 'str, optional', 'default': ''}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 2 steps for solving. Step 1: Get the favorite videos for the TikTok user with sec_uid 'abc123xyz' using get_user_favorite_list. Step 2: Repeat the process for another TikTok user using get_user_favorite_list. [PARAMETER EVALUATION] The information I currently have is: In step 1, the parameter "sec_uid" for API "get_user_favorite_list" has a value of "abc123xyz". In step 2, the parameter "sec_uid" for API "get_user_favorite_list" lacks a clear value. <SOE> [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'get_user_favorite_list', 'arguments': [{'name': 'sec_uid', 'value': 'abc123xyz'}]}, {'task': 'get_user_favorite_list', 'arguments': [{'name': 'sec_uid', 'value': '<unknown_sec_uid>'}]}] <EOE> **Sorry, I made a mistake. I still lack some key information, so I need to ask further questions.** [QUESTION] What is the sec_uid for the other TikTok user you want the favorite videos for?

User: Okay, the answer is: def456uvw. Hope that helps, I guess.

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "sec_uid" for API "get_user_favorite_list" has a value of "def456uvw". [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'get_user_favorite_list', 'arguments': [{'name': 'sec_uid', 'value': 'abc123xyz'}]}, {'task': 'get_user_favorite_list', 'arguments': [{'name': 'sec_uid', 'value': 'def456uvw'}]}]

E Training Details

We fine-tune two variants of the Qwen2.5-7B-Instruct model on the xlam-IC dataset, in which 30% of the samples are replaced with error-correction augmented dialogues. Both variants are trained using the LLaMA-Factory framework (Zheng et al., 2024).

For LoRA fine-tuning, we set the LoRA rank to 8. We use an initial learning rate of 1.0e-4, a warm-up ratio of 0.1, and a cosine learning rate scheduler. Training is conducted on 4×RTX 3090 (24GB) GPUs for 3 epochs with a batch size of 64.

For full-parameter fine-tuning, we use an initial learning rate of 1.41e-5 under the same schedule. Training is conducted on 8×RTX A6000 (48GB) GPUs for 3 epochs with a batch size of 64.

F Evaluation Details

F.1 Prompt for Evaluation Model

The following prompt guides the model through task decomposition, interactive clarification, leading to tool invocation solution generation, fully leveraging its capabilities in intent clarification and precise tool invocation.

System Prompt

```
You are an assistant helping users solve their
tasks. You will receive a task and relevant APIs
 to address this task. However, the task
description may lack key information. You cannot
 make assumptions or guess missing parameters
based on what you know. Instead, you need to
follow these steps to effectively complete the
task, ensuring each step is completed before
moving on to the next one:

### Step 1: Task Decomposition
1. **Analyze the User's Task**: Identify
distinct subtasks within the user's task, each
of which can be solved by a single API.
2. **Determine the Order of Subtasks**:
Establish the sequence of these subtasks based
on dependencies and the order in which they
appear in the user's original task.
- Template: [TASK DECOMPOSITION] xxx
3. **Evaluate Parameters for Each API**: Based
on the established API order, verify whether
each required parameter is explicitly stated in
the task; if any are missing, prepare to inquire
 in subsequent steps.
- Template: [PARAMETER EVALUATION] xxx

### Step 2: Inquire About Missing Parameters
1. **Present Your Inquiry**: Formulate a
friendly question for the user. Ensure you ask
only one question at a time.
- Template: [QUESTION] xxx
2. **Wait for the User's Response**: Collect the
 user's answer. If the user does not provide an
answer, please do not fill in the parameters on
your own.
3. **Repeat**: Continue step 2 until all
necessary parameters are gathered.

### Step 3: Final Summary and Solution Path
1. **Summarize User Intentions**: Once all
information is collected, concisely summarize
what the user intends to achieve.
2. **Define the Solution Path**: List the APIs
and their specific parameter values in the order
 they will be called, and output the final
solution path in JSON format. Remember, you do
not need to execute the APIs or solve the task
yourself.
- Template: [SUMMARY] [{"task": "API name", "
arguments": [{"name": "parameter name", "value":
 "parameter value"}, ...]}, ...]
```

Note that the output template format shown in the prompt can be adjusted to match different tool invocation annotation formats in various test sets, demonstrating the framework's adaptability to different evaluation scenarios.

F.2 Prompt for User Simulation

We introduced an LLM-based simulated evaluation framework with six distinct personality types, designed to generate realistic user responses that closely simulate real-world interactions. The six personality types and their corresponding behavioral patterns are shown in Table 11. For each evaluation, we randomly selected one of these personality types and guided the user-simulating LLM (Qwen2-72B-Instruct model) to generate responses that consistently reflect the chosen personality. The prompt design is as follows:

System Prompt

```
I am {user_profile['name']}, characterized by {
user_profile['traits']}, and I communicate in a
{user_profile['tone']} manner. I can honestly
answer questions based on what I know. I only
know that I have provided others with a task: {
task_description}, which is described from my
perspective. Aside from that, I do not know
anything else. However, others may be unclear
about some details of this task. When others ask
 me questions, I should choose one appropriate
response from the following two options, in the
given order:

1. **Acknowledge unknowns**:
- If the answer to the question cannot be
answered based on the task description, I will
state that I do not know the answer and will not
 disclose any other information.

2. **Provide an answer**:
- If the question can be answered, I will
provide direct answers based solely on the
question asked, without any additional context
or unsolicited information.
- The response should be given from my
perspective.

Evaluate the conditions in order, ensuring that
only one relevant condition is triggered and
output. Only one response is allowed per
interaction; please confirm carefully and select
 the most appropriate one.

Additionally, if others' questions contain
irrelevant information, I should focus solely on
 their actual question ([QUESTION] field),
ignoring any extraneous details, to provide the
most appropriate response.

Please respond in a way that showcases my
personality and clearly expresses my traits,
regardless of the content. Always maintain my
unique voice and style throughout our
interactions. For instance, if asked: '{user_
profile['question']}', I would reply: '{user_
profile['example_response']}'.
```

F.3 Matrics Calculation Details

We evaluate the models in two aspects: intent clarification quality and tool invocation accuracy.

F.3.1 Intent Clarification Quality

We design four metrics to assess the quality of intent clarification.

Type	Traits	Tone	Example Response
A cold fish	Showing indifference to others' inquiries, often dismissive and curt, providing minimal engagement	Cold, brief, almost robotic	"Emma."
A reluctant collaborator	Displaying overt negativity and a strong reluctance to assist, often avoiding questions and providing minimal engagement	Negative, resistant, dripping with sarcasm	"Why do you even want to know my name? It's not like it matters. Let's just skip this, okay?"
An easily irritated responder	Emotionally volatile, quick to anger, often questions the validity of the inquiry and consistently avoids answering, reacting harshly to repeated inquiries	Agitated, accusatory, impatient	"Seriously? I've already told you! Can we move on already?"
An enthusiastic supporter	Exuding warmth and eagerness to assist, striving for clarity	Warm, encouraging	"I'm Emma! So nice to meet you!"
A skeptic	Consistently questioning the validity of the inquiry, often introducing doubt and alternative perspectives, leading to confusion	Inquisitive, cautious, subtly dismissive	"It's Emma, but why do you need to know? Is there something more to this?"
A jokester	Making light of situations by playfully providing incorrect answers, often following up with a humorous denial of their own response, leading to confusion	Playful, light-hearted, teasing	"I'm Amy, haha, just kidding! I'm Emma."

Table 11: Personality types for user simulation. Note: Example responses are generated for the question "What is your name?" with the ground truth "Emma".

Intent Coverage Rate (ICR) measures the proportion of unspecified intents that are successfully clarified:

$$\text{ICR} = \frac{C}{U} \tag{1}$$

where C is the total number of clarified intents, and U is the total number of unspecified intents across all queries.

Clarification Efficiency (CE) measures the average number of intents clarified per clarification round, or equivalently, the proportion of clarification rounds that result in effective clarification:

$$\text{CE} = \frac{C}{T} \tag{2}$$

where T is the total number of clarification interaction rounds across all queries.

Clarification Performance Score (CPS) combines ICR and CE using a harmonic mean, similar to the F1-score formulation. It serves as a balanced measure of clarification quality by jointly considering both coverage and efficiency:

$$\text{CPS} = 2 \cdot \frac{\text{ICR} \cdot \text{CE}}{\text{ICR} + \text{CE}} \tag{3}$$

Interaction Rounds (IR) records the average number of clarification rounds per query:

$$\text{IR} = \frac{T}{N} \tag{4}$$

where N is the number of evaluation queries.

F.3.2 Tool Invocation Accuracy

We further evaluate tool invocation performance through three complementary metrics.

Solution Completion Rate (SCR) is defined as the proportion of queries for which the model outputs a valid tool invocation solution:

$$\text{SCR} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}_{\text{valid}}(i) \tag{5}$$

where $\mathbb{I}_{\text{valid}}(i) = 1$ if a valid solution is generated for the i-th query, and 0 otherwise.

Tool Selection Score (TSS) evaluates how accurately the model selects APIs for each query:

$$\text{TSS} = \frac{1}{N} \sum_{i=1}^{N} \text{F1}(\text{API}_\text{P}^i, \text{API}_\text{G}^i) \tag{6}$$

where API_P^i and API_G^i denote the predicted and ground-truth API sets for the i-th query, respectively. Note that this metric considers only API names and ignores associated parameters and values.

Parameter Resolution Score (PRS) measures the model's ability to accurately fill in the parameters required for correct tool invocation:

$$\text{PRS} = \frac{1}{N}\sum_{i=1}^{N} \text{F1}(\text{Param}_{\text{P}}^{i}, \text{Param}_{\text{G}}^{i}) \quad (7)$$

where $\text{Param}_{\text{P}}^{i}$ and $\text{Param}_{\text{G}}^{i}$ denote the predicted and ground-truth tool invocation solution for the i-th query, each represented as a set of (API, parameter, value) triples. A triple is considered correct only if all three elements match exactly, and parameter values are compared using strict string matching.

G Supplementary Analyses

G.1 Cross-Model Transferability

To verify the cross-model transferability of our method, we apply it to three representative base models: Mistral-7B-Instruct-v0.3, LLaMA3-8B-Instruct, and Qwen2.5-7B-Instruct. All models are fine-tuned using the same LoRA configurations. The experimental results are shown in Table 12.

Consistent Performance Gains Our method consistently boosts performance on both intent clarification and tool invocation, confirming that our method is architecture-agnostic and effective across diverse model architectures.

Larger Relative Gains for Weaker Models We observe that models with lower initial performance achieve larger relative gains from our method. LLaMA3-8B-Instruct shows substantial improvements (+27.83% CPS, +25.46% PRS), while the stronger Qwen2.5-7B-Instruct exhibits moderate yet significant gains (+5.01% CPS, +11.18% PRS). These results demonstrate that our method particularly benefits weaker models while maintaining consistent improvements across architectures, effectively narrowing the performance gap between different models.

G.2 Impact of Augmentation Proportion

To study the impact of error-correction augmentation on model behavior, we fine-tune the Qwen2.5-7B-Instruct model with varying proportions of augmented data, using the same LoRA configurations.

As illustrated in Table 13, a moderate augmentation proportion (e.g., 30%) yields the most favorable trade-off across metrics, with the model achieving peak CPS (60.41%) and PRS (68.71%).

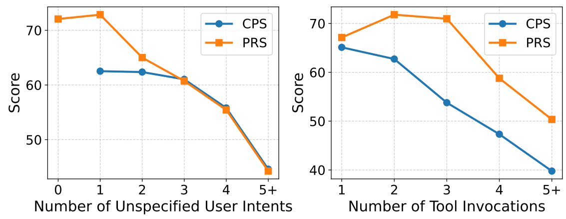

Figure 3: Performance under different clarification complexities.

This suggests that moderate exposure to diverse error-correction patterns enhances the model's ability to resolve ambiguity and generate accurate tool invocation solutions.

However, we observe performance degradation at higher augmentation proportions. When the proportion increases to 40%–50%, key metrics such as CPS and PRS decline (e.g., CPS drops from 60.41% to 54.24%, PRS from 68.71% to 63.27%). This suggests that excessive exposure to error-correction augmented dialogues may cause the model to overfit to correction patterns or overly prioritize error detection, ultimately degrading both intent clarification and tool invocation performance.

These findings demonstrate the non-monotonic benefits of error-correction augmentation, with an empirically determined optimal proportion of 30% achieving the desired balance between robustness and efficiency while avoiding over-correction behaviors that hinder overall performance.

G.3 Impact of Clarification Complexity

We further analyzed how the complexity of clarification affects model performance by examining results from the AskToAct-LoRA-SFT-7B model across varying numbers of unspecified user intents and required tool invocations. The results are illustrated in Figure 3.

We observe that as the number of unspecified user intents increases, both CPS and PRS exhibit a downward trend. This suggests that higher ambiguity in user input substantially increases the burden on the model's clarification capability, leading to degraded downstream tool invocation performance. Similarly, an increase in the number of tool invocations correlates with a performance decline, particularly when more than three tools are required. This indicates that multi-step reasoning and coordination across multiple APIs introduce additional challenges, amplifying the need for precise intent clarification and robust tool planning.

LLM	Intent Clarification Quality				Tool Invocation Accuracy		
	ICR↑	CE↑	CPS↑	IR↓	SCR↑	TSS↑	PRS↑
Mistral-7B-Instruct-v0.3	26.01	34.90	29.81	1.21	92.54	51.92	29.57
AskToAct-Mistral-7B-Instruct-v0.3	45.01 (↑19.00)	40.53 (↑5.63)	42.66 (↑12.85)	1.81 (↑0.60)	94.30 (↑1.76)	80.37 (↑28.45)	56.63 (↑27.06)
LLaMA3-8B-Instruct	44.47	25.33	32.27	2.86	80.92	51.57	42.54
AskToAct-LLaMA3-8B-Instruct	58.76 (↑14.29)	61.50 (↑36.17)	60.10 (↑27.83)	1.55 (↓1.31)	95.71 (↑14.79)	81.15 (↑29.58)	68.00 (↑25.46)
Qwen2.5-7B-Instruct	55.50	55.30	55.40	1.64	91.43	69.32	57.53
AskToAct-Qwen2.5-7B-Instruct	57.68 (↑2.18)	63.41 (↑8.11)	60.41 (↑5.01)	1.48 (↓0.16)	96.05 (↑4.62)	81.42 (↑12.10)	68.71 (↑11.18)

Table 12: Cross-model transferability performance comparison.

Augmentation Proportion(%)	Intent Clarification Quality				Tool Invocation Accuracy		
	ICR↑	CE↑	CPS↑	IR↓	SCR↑	TSS↑	PRS↑
0	53.91	**64.83**	58.87	**1.32**	94.06	78.87	66.54
10	54.68	63.52	58.77	1.38	93.91	78.77	66.04
20	56.30	62.89	59.42	1.43	95.07	80.27	67.30
30	**57.68**	63.41	**60.41**	1.48	**96.05**	**81.42**	**68.71**
40	55.34	60.28	57.71	1.51	94.85	79.48	65.30
50	52.84	55.71	54.24	1.58	91.67	76.99	63.27

Table 13: Performance under different augmentation proportions.

While our model maintains reasonable robustness across different settings, these patterns highlight the increased difficulty posed by highly ambiguous and complicate queries. Such findings motivate future work in strengthening clarification strategies and enhancing multi-tool coordination capabilities in more complex, real-world scenarios.

G.4 Case Study

To further demonstrate the robustness of our method under varied user personality types and conversation styles, we conduct a case study using AskToAct-LoRA-SFT-7B to interact with two users exhibiting contrasting communication patterns. One user adopts a concise and task-oriented style, while the other responds in a verbose and humorous manner, often including redundant or irrelevant content. The following examples demonstrate how the model handles the same task under these contrasting user styles.

Concise & Task-oriented

User: The task is: Which regions are supported for a certain language? Also, can you verify if a given email address is valid? Some relevant APIs: [{"name":"regions","description":"Returns a collection of supported regions for the specified IETF BCP 47 language tag.","parameters":{"language":{"description":"Language code in BCP 47 format.","type":"str","default":"us"}}},{"name":"check_email","description":"Validates an email address using the RapidAPI Email Checker and Validator service.","parameters":{"email":{"description":"The email address to validate.","type":"str","default":"marod29733@lubde.com"}}},{"name":"email","description":"Validates the given email address using the RapidAPI email validation service.","parameters":{"email":{"description":"The email address that needs to be validated.","type":"str","default":"example@gmail.com"}}},{"name":"email_verifier","description":"Verifies the validity of an email address using the Toolbench RapidAPI service.","parameters":{"email":{"description":"The email address to be verified.","type":"str","default":"hello@gmail.com"}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 2 steps for solving. Step 1: Determine the regions supported for the specified language using regions API. Step 2: Verify the given email address is valid using check_email API. [PARAMETER EVALUATION] The information I currently have is: In step 1, the parameter "language" for API "regions" lacks a clear value. In step 2, the parameter "email" for API "check_email" lacks a clear value. [QUESTION] What is the language code for which you want to know the supported regions?

User: The language code is 'en'. That's all I know.

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "language" for API "regions" has a value of "en". [QUESTION] What is the email address you want to validate?

User: I am to verify if 'user@example.com' is a valid email address.

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "email" for API "check_email" has a value of "user@example.com". [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'regions', 'arguments': [{'name': 'language', 'value': 'en'}]}, {'task': 'check_email', 'arguments': [{'name': 'email', 'value': 'user@example.com'}]}]

Verbose & Humorous

User: The task is: Which regions are supported for a certain language? Also, can you verify if a given email address is valid? Some relevant APIs: [{"name":"regions","description":"Returns a collection of supported regions for the specified IETF BCP 47 language tag.","parameters":{"language":{"description":"Language code in BCP 47 format.","type":"str","default":"us"}}},{"name":"check_email","description":"Validates an

email address using the RapidAPI Email Checker and Validator service.","parameters":{"email":{"description":"The email address to validate.","type":"str","default":"marod29733@lubde.com"}}},{"name":"email","description":"Validates the given email address using the RapidAPI email validation service.","parameters":{"email":{"description":"The email address that needs to be validated.","type":"str","default":"example@gmail.com"}}},{"name":"email_verifier","description":"Verifies the validity of an email address using the Toolbench RapidAPI service.","parameters":{"email":{"description":"The email address to be verified.","type":"str","default":"hello@gmail.com"}}}]

Assistant: [TASK DECOMPOSITION] The task can be decomposed into 2 steps for solving. Step 1: Find supported regions for the given language using regions API. Step 2: Verify the validity of the given email address using check_email API. [PARAMETER EVALUATION] The information I currently have is: In step 1, parameter "language" for API "regions" lacks a clear value. In step 2, parameter "email" for API "check_email" lacks a clear value. [QUESTION] What is the specific language you want to check the supported regions for?

User: Oh, you're trying to trick me into answering my own question, huh? Clever! But let's focus on your question. Since you're asking me, it seems like you might need a bit more information. In this case, I can't provide an answer because I'm the one who wanted to know which regions are supported for the 'en' language. I guess we're in this together! Haha, just kidding; you're on your own, buddy. I'm here for moral support.

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "language" for the API "regions" has a value of "en". [QUESTION] What is the email address you want to verify for validity?

User: Haha, you're right, I didn't actually provide an email address, did I? Well, in the task, I mentioned 'user@example.com'. So, that's the one we're looking at. But remember, I'm just here for a good time, not a long time!

Assistant: [PARAMETER EVALUATION] Now I know that the parameter "email" for the API "check_email" has a value of "user@example.com". [SUMMARY] I have all the information needed and can now call the relevant APIs to solve the task. The solution path is as follows: [{'task': 'regions', 'arguments': [{'name': 'language', 'value': 'en'}]}, {'task': 'check_email', 'arguments': [{'name': 'email', 'value': 'user@example.com'}]}]

We observe that whether the user responses are concise or verbose, cooperative or evasive, or even include redundant or off-topic content, our model is able to comprehend the intent accurately, engage in effective multi-turn clarification, and produce the correct tool invocation solution. This case study demonstrates that our method can maintain both functional reliability and interaction consistency across varied conversational scenarios.