Learning of API Functionality from In-Context Demonstrations for Tool-based Agents

Bhrij Patel[†]
University of Maryland, College Park

Ashish Jagmohan[*]
Emergence AI
NYC, New York

Aditya Vempaty[*]
Emergence AI
NYC, New York

[†]Work done while intern at Emergence AI
[*]Equal Advising
Code and Data Repository

Abstract

Digital tool-based agents, powered by Large Language Models (LLMs), that invoke external Application Programming Interfaces (APIs) often rely on documentation to understand API functionality. However, such documentation is frequently missing, outdated, privatized, or inconsistent—hindering the development of reliable, general-purpose agents. In this work, we propose a new research direction: learning of API functionality directly from in-context demonstrations. This task is a new paradigm applicable in scenarios without documentation. Using API benchmarks, we collect demonstrations from both expert agents and from self-exploration. To understand what information demonstrations must convey for successful task completion, we extensively study how the number of demonstrations and the use of LLM-generated summaries and evaluations affect the task success rate of the API-based agent. Our experiments across 3 datasets and 6 models show that learning functionality from in-context demonstrations remains a non-trivial challenge, even for state-of-the-art LLMs. We find that providing explicit function calls and natural language critiques significantly improves the agent's task success rate due to more accurate parameter filling. We analyze failure modes, identify sources of error, and highlight key open challenges for future work in documentation-free, self-improving, API-based agents.

1 Introduction

In the past few years, AI agents have been rapidly adopted in society due to the development of Large Language Models (LLMs) (Achiam et al., 2023; Touvron et al., 2023), allowing for superb performance in natural language understanding, summarization, and generation. More recently, *tool-based agents* have been introduced to increase the abilities of agents for specialized tasks that may even require up-to-date knowledge of external databases, such as in cases for enterprise workflows (Styles et al., 2024; Xu et al., 2024; Wang et al., 2024). Tool-based agents can select and call external functions like Application Programming Interface[1] (API) functions. API-based agents can automate digital tasks in various domains such as finance, health, and analytics (An et al., 2024; Gawade et al., 2025; Guo et al., 2024).

Overall Task/Query: "I need to get back to yuki's last email about 'Update on Client Appreciation Gala' with 'Thanks for the update - I will get back to you tomorrow. Can you send the reply for me?"

Conversation History/Previous Steps:
[email.search_emails.func(query="Update on Client Appreciation Gala", date_max="2023-11-30")]

Demonstration of Function:
email.reply_email.func(email_id="00000234", body="Thanks for the update - I will get back to you tomorrow.")

Figure 1: Expert demonstration of the email.reply_email function extracted from WorkBench dataset (Styles et al., 2024). Demonstrations are the basis of how agents understand functionality without prior documentation.

A common, crucial requirement for API-based agents is access to API documentation that explains to the agent in natural language the functionality of the specific API functions available (Kim et al., 2024; Styles et al., 2024). The documentation explains both 1) what the function does and returns (if anything), and 2) what the input parameter schema is. Both these components are important for tool selection and their execution by the agent. However, API documentation can frequently be unavailable, inconsistent, unstructured, or out-of-date (Li et al., 2022; Khan et al., 2021; Zhong and Su, 2013). Prior work in software engineering research on API

[1]In this work, "tool" and "API" are used interchangeably, as our work can be extended to any external tool with an API wrapper.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 18318–18336
November 4-9, 2025

documentation generation (Khan and Uddin, 2022; Yang et al., 2023) lacks evaluation of the generated documents via downstream tasks by an LLM-based agent. Other work (Tang et al., 2023) generates API documentation by inputting a brief description of the function into an LLM, and this documentation is then used by an agent. However, a rigorous analysis of learning functionality with no prior documentation or description remains absent in the API-based agent literature.

We address these gaps by introducing and formalizing the problem of learning API functionality from scratch via in-context demonstrations of tool calls. Demonstrations of tool calls are present within codebases of public repositories and logs. While prior work has finetuned agents on demonstrations (Schick et al., 2023), we focus on in-context learning to remove the need to update parameters. In this work, we analyze the ability of an API-based agent with a frozen LLM with only access to demonstrations of tool calls to learn functionality to perform tasks. Using various API datasets, we extract and standardize demonstrations for the agent to learn from, to perform the dataset tasks. Our experiments revolve around three main pillars. We investigate the effect of 1) the number of demonstrations and 2) the representation of the demonstrations. Specifically, we compare, in terms of downstream task performance, between providing the demonstrations directly, generating documentation from the demonstrations, or a combination of both. Furthermore, we also show the impact of 3) collecting, evaluating, and summarizing experiences of the agent to update its understanding of the APIs. Our results consistently show across experiments the difficulties of the agent to perform tasks, highlighting how non-trivial learning of API functionality from in-context demonstrations is, even with state-of-the-art (SoTA) LLMs.

We summarize our main contributions below:

- **Novel, Applicable Problem.** We present the problem of learning of API functionality from in-context demonstrations without any initial ground-truth documentation. Our work is the first work that removes the assumption of access to documentation, and the agent must rely solely on demonstrations to understand functionality. We formalize this problem with an optimization objective to characterize the task success rate's dependence on the information processing of the demonstrations.
- **Various Methods to Learn From Demonstrations.** To tackle this new issue, we present 3 processing methods to learn from a set of demonstrations while also providing 4 methods to update the functionality understanding from the experiences collected by the agent's self-exploration. We also incorporate an LLM-based evaluator to provide richer, natural language feedback on each step the agent takes.
- **Empirical Evaluation and Analysis.** We conduct experiments with our methods, showing that learning of API functionality from in-context demonstrations for downstream task completion is challenging for existing LLMs, highlighting the need for further research. We wish to answer what is needed to maximize performance given a set of demonstrations. From our experiments, we find that a central recurrent problem is filling in parameter values, leading up to a 39% decrease in an agent's success rate if the method for processing demonstrations incorrectly describes the parameter schema.

2 Preliminaries: Task Completion with API-based Agents

Consider a goal-conditioned Partially Observable Markov Decision Process (POMDP) $(G, \mathcal{S}, \mathcal{O}, \mathcal{F}, P, Z, r)$. G is a set of goals, which we refer to as "tasks", that the user could ask. (e.g. "Reply to my last email from Dev with the body 'Let us catch up soon to discuss the project.' "[2]); state $s \in \mathcal{S}$ in the digital environment could be a set of employees, emails, calendar events, etc.; $\mathcal{O}$ is the information the agent currently knows (e.g. Dev's email is dev@business.com); $\mathcal{F}$ is the set of function names (e.g. reply_email) the agent can choose to execute; $P(f)$ is the parameter schema given whose domain contains the valid inputs of $f \in \mathcal{F}$ (e.g. email ID and message). Z represents the transition dynamics. Executing function f and input parameters $p \in \text{dom}(P(f))$ at s results in the new state and observation $s', o' = Z(s, f, p)$ (e.g. s' is Dev having a new message in his inbox and o' is the agent receiving a message that the email was sent). Finally, $r(s'|g) : \mathcal{S} \times G \to \{0, 1\}$, where $r = 1$ indicates task g is completed correctly. Note that r is not dependent on o' as the observation may not correctly indicate task completion (e.g.

[2]Example task from WorkBench dataset (Styles et al., 2024).

"Email sent successfully" is returned to the agent, but the email is sent to the wrong person.) In this work, the policy π of that API-based agent that completes tasks is a fixed LLM[3].

For an LLM-based agent to understand the functions in $\mathcal{F}$, it needs a textual description, or *textualization*, of P, and Z (e.g., API documentation). Let T_P^*, T_Z^* be the set of all possible textualizations of P and Z, respectively, that correctly convey the functionality of $\mathcal{F}$. With $t_p \in T_P^*, t_z \in T_Z^*$, an agent selects an API with input parameters with a policy $(f, p) \sim \pi(\cdot|o, \mathcal{F}, t_p, t_z, g)$. In practice, the LLM agent generates a JSON object detailing the chosen f and p that will be passed to another program to execute. Note that π is treated as a probability due to the inherent randomness of LLMs. However, given the agent has access to some t_p and t_z, it can make appropriate decisions.

Limitation in Prior Work: Access to t_p and t_z. As mentioned previously, API documentation is often inaccessible, out-of-date, or inconsistent with the functionality of the APIs. Thus, the agent may not have access to some t_p and t_z, and we aim to learn and model the information as t_p^θ and t_z^θ for P and Z, respectively. Ideally $t_p^\theta \in T_P^*$ and $t_z^\theta \in T_Z^*$. We wish to maximize the total successful task completions with (t_p^θ, t_z^θ). Letting H be the number of steps an agent performed to try to complete g, we can formally write

$$\max_\theta J(\theta) = \sum_{g \in G} \mathbf{E}_{(f_h, p_h) \sim \pi(\cdot|o_h, \mathcal{F}, t_p^\theta, t_z^\theta, g)} \left[R_H\right], \quad (1)$$

where $R_H = \sum_{h=0}^{H} r(s_{h+1}|g), (s_{h+1}, o_{h+1}) = Z(o_h, f_h, p_h)$. Equation 1 provides a mathematical grounding of how well t_p^θ and t_z^θ textualize the unknown P and Z to how well the LLM-based agent performs the set of tasks G. The next section details how we tackle this problem by learning θ via demonstrations of API function calls.

3 Learning of API Functionality from In-Context Demonstrations

Although API documentation may inaccessible, often there are demonstrations available based on codebases of public repositories and logs. With this in mind, we investigate how we can utilize these demonstrations to learn API functionality in-context without updating the LLM parameters.

3.1 Demonstration Definition and Format

For each API function f, we aim to obtain a collection of expert demonstrations D_{expert}^f, where each demonstration $d \in D_{\text{expert}}^f$ is a single use case of f for a given task. We also define a demonstration trajectory as a series of demonstrations to complete a task. A trajectory of length M can be comprised of demonstrations of at most M distinct functions.

Each demonstration must show not only the function call itself, but also the *context* of the call. To display the context of the call, we need the task it is trying to perform and the previous steps taken to complete the task. Demonstrations of single-step tasks or initial steps have an empty list of calls for the previous steps. Figure 1 shows an example demonstration of the reply_email function from the WorkBench dataset by (Styles et al., 2024). In this example, the reply_email function is trying to reply to an email as mentioned in the "Overall Task/Query" section and is preceded by the search_emails function that is necessary to find the email_id value.

Expert Demonstrations. We define an "expert demonstration" as a demonstration produced by an agent that used ground-truth, original documentation. We propose that instead of relying on prior documentation, we can learn P and Z and model their textualization as t_p^θ and t_z^θ via demonstrations of the function calls used to complete tasks. With a set of expert demonstrations D_{expert}, we can obtain the textualization $(t_p^\theta, t_z^\theta) = I(D_{\text{expert}})$, where I is a method for processing the demonstrations. We can thus rewrite Equation 1 as

$$\max_I J(I) = \sum_{g \in G} \mathbf{E}_{(f_h, p_h) \sim \pi(\cdot|o_h, \mathcal{F}, I(D_{\text{expert}}), g)} \left[R_H\right]. \quad (2)$$

We want to process the information from a given D_{expert} to maximize performance.

3.2 Methods for Processing Expert Demonstrations

In all of the methods for I described below, for each $f \in \mathcal{F}$ that has at least N training expert demonstrations, $|D_{train}^f| \geq N$, we sample a set of N random demonstrations $D_{train}^{f,N}$.

1) Direct Expert Demonstrations (DxD). We pass the $f \times N$ demonstrations directly to π.

[3]We will use the terms "agent" and "policy" interchangeably; so π refers to both.

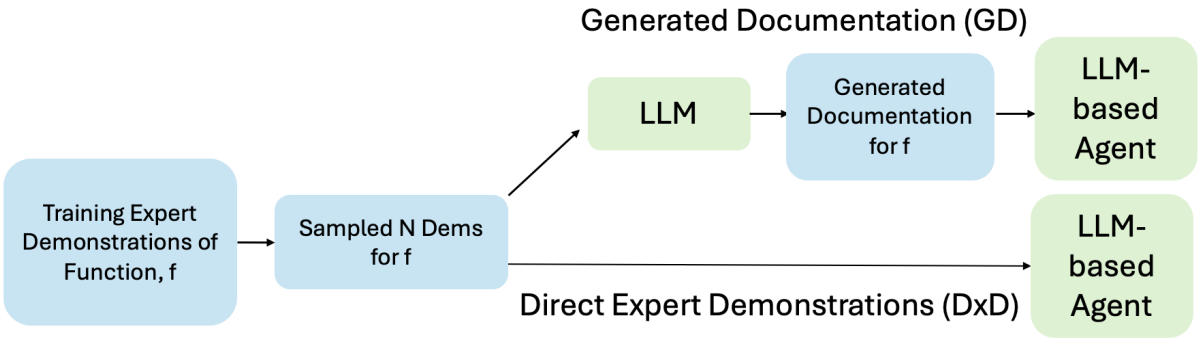

Figure 2: **Processing Methods of Expert Demonstrations:** Given a set of training demonstrations for each function f, we sample N demonstrations for each function to be used for either **(TOP)** LLM-based documentation generation or **(BOTTOM)** to be directly passed into the agent.

2) Generated Documentation With the sampled $D_{train}^{f,N}$, we use an LLM generator to produce documentation for f. We repeat for all f that have at least N demonstrations. The system prompt for the generator is provided in Appendix E.

3) Generated Documents with Example Calls (GDEC). We combine the previous approaches by generating a document of f with $D_{train}^{f,N}$ and then appending the function calls of those demonstrations to the bottom of the generated document as example use cases. See "Demonstration of Function" in Figure 1 for an example call. We do not give the task or previous steps like with DxD.

4) Oracle Baseline: Original Documentation (OD). We give the agent the original, ground-truth documentation provided by the given dataset. This baseline is essentially the "expert" agent.

Figure 2 visualizes the pipeline for DxD and GD. The same $D_{train}^{f,N}$ used for DxD are also used for GD for f. For GDEC, the function calls from all sampled N demonstrations of f are attached to the generated document of f.

3.3 Experiences from Self-Exploration

Demonstrations from Experience. To understand how the agent can improve its understanding of the API functionality after initial learning from expert demonstrations, we study how self-exploration can be used to gather useful observations. Before having the agent π perform test query tasks, we have it complete training tasks. Given D_{train} and a processing method I, the agent uses its resulting (t_p^θ, t_z^θ) to complete tasks in the training set.

Each function call the agent takes to complete a task is an *experience* of some function f. Experiences are a different source of demonstrations, apart from expert demonstrations. For each function call the agent takes, combined with the resulting return value, is a new demonstration. Furthermore, in some sandbox environments like WorkBench (Styles et al., 2024), the agent will sometimes provide its thought process before the call. We add that to the experience-based demonstration. We store each experience to be used to help the agent with the test tasks. Figure 3 gives an example self-exploration experience of the same task in Figure 1. Here, the agent uses reply_email prematurely and receives an "Email not found." Rather than throwing at incorrect uses, we provide richer feedback and distinguish between positive and negative experiences by implementing an LLM-based evaluator, which we describe next.

Evaluating Self-Exploration with LLM Judges. Due to the expensive nature of repeated LLM calls, one would want to extract more feedback from the experience than just whether or not the task was completed. However, the reward signal for the experiment setup described so far has been the sparse, binary reward function of whether or not the task was completed. To synthesize more feedback from the collected experience, we pass each experience to an LLM-based evaluator to produce a natural language critique. We add this evaluation to the experience. We provide the LLM-based judge the task, the entire trajectory by π, whether or not the trajectory was correct, and the specific demonstration we wish to analyze. Therefore, if the trajectory had three function calls, we would make three calls to the LLM-judge, changing only the demonstration to evaluate. We ask it to specifically look at whether or not it was a 1) repeated call, 2) whether the parameter filling is accurate, and 3) whether the function is used in the right place in the trajectory.

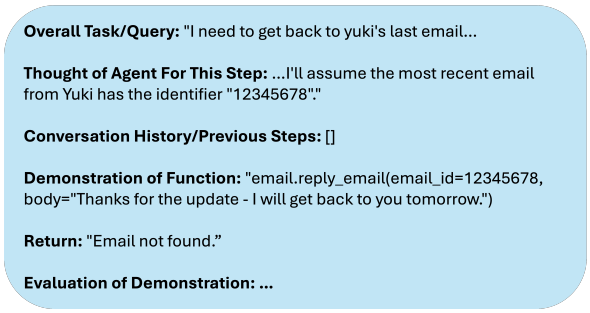

Figure 3: **Experience-based demonstration** gained during self-exploration by π for the same task in Figure 1. This experience includes the thought process and return of the reply_email function. Note that this is an incorrect use of reply_email. The "Evaluation of Demonstration" is shown in Figure 4.

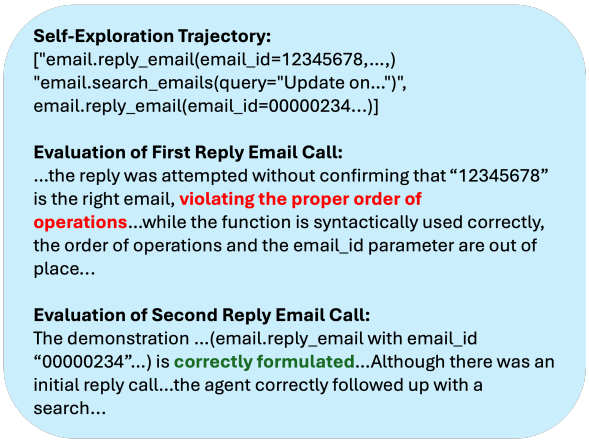

Figure 4: **LLM-Generated Evaluation of Self-Exploration Trajectory** by π for same task in Figures 1 and 3. The agent incorrectly tries to reply email without searching for the email ID. It then corrects itself by executing search_email and then correctly using reply_email again. The first reply_id is formatted into the demonstration shown in Figure 3. The evaluation of the first reply_email call emphasizes that the agent should have first confirmed it had the right email ID. The evaluation for the second reply_email call states it was correctly used after the agent found the email ID with search_email. Each of these evaluations is added to the demonstration of their respective calls.

Figure 4 gives an example self-exploration trajectory from the same task in Figures 1 and 3 and the generated LLM-based evaluation of the first, incorrect reply_email call (shown in Figure 3) and the second, correct reply_email call in that trajectory. We see that it correctly flags the error of the first reply_email call, mentioning that a search_email call should have preceded it to find the right email_id, and that it flags the second reply email call as correct. See Appendix E for system prompts of the evaluator and summarizer.

With these experiences, we can update the textualizations as $(t_p^\theta, t_z^\theta) = I'(D_{\text{expert}}, D_{\text{experience}})$, where I' is a processing method that takes in both the expert demonstrations and experience.

3.4 Methods for Processing Experiences

1) Direct Experience (DE). For each f, we pass in $D^f_{\text{experience}}$ with expert demonstrations D^f_{train} as the descriptions. If $|D^f_{\text{experience}}| = 0$, only D^f_{train} is used instead.
2) Updated Documentation (UD). We take the initial generated document of f (GD) and use an LLM to update the document using the experiences of f. If $|D^f_{\text{experience}}| = 0$, the GD of f is used.
3) Regenerated Documentation (RD). We regenerate the documentation from scratch using both expert demonstrations and experiences. If $|D^f_{\text{experience}}| = 0$, the GD of f is used.
4) Attached Guidelines (AG). An LLM summarizer takes in the experiences of f and generates guidelines. We then attach those guidelines to the initial generated document from expert demonstrations (GD) of f. Figure 5 gives an example lesson generated from all the self-exploration experiences of reply_email. If $|D^f_{\text{experience}}| = 0$, the GD of f is used without any guidelines.

Figure 7 in the Appendix provides a visual pipeline delineating the process of self-exploration on training queries, evaluating experiences, and the methods used to update θ for test-time. The system prompts for the LLM document updater and summarizer are given in Appendix E.

4 Experimental Setup

In this section, we provide the experimental details for studying our methods of learning of API functionality from in-context demonstrations. Across our experiments, we run 3 trials, each with a different random seed: $2003, 2004, 2005$.
Train-Test Splitting Expert Demonstrations. To ensure that we do not evaluate the LLM agent on tasks it has seen from its demonstrations, we divide the tasks and demonstrations with a train-test split. Furthermore, we also have to ensure that no task in the test set includes an API not seen in the training set. Simply train-test splitting the set of tasks and removing test tasks that include APIs not in the training set can greatly reduce the number of test tasks to evaluate on. Therefore, for our train-test

Guidelines:
1. Always locate the correct email first. Make sure to call email.search_emails (or another method that retrieves emails)....
2. Use the exact email_id returned from a previous search...
3. Ensure the reply_email function call is made exactly once....
4. Clearly specify the content of the reply in the body parameter....
Warnings:
1. Avoid using an email_id that wasn't confirmed via a prior search. Relying on unverified identifiers (such as "chenwei_last_email" or "yuki_last_email_id") may lead to error responses like Email not found.
2. Do not repeat the reply_email call...
3. Be consistent with identifiers. Changing email_ids across different calls in the same task could lead to replying to the wrong email...

Figure 5: **Summarized guidelines from experiences and evaluations** of reply_email. The lesson emphasizes using search_email beforehand to find the right email_id to use for reply_email, which the agent sometimes did not do as shown in Figure 4.

split, we iterate through each API function available. For each API f, we train-test split the tasks it is associated with if the task has not already been assigned to the train or test set from a previous iteration. We then use the demonstrations for all the training tasks as the training set, D_{train}. For every train-test split in this iterative process, use a split of $70-30\%$ on the remaining tasks that have yet to be assigned a set. This process, while dividing the demonstrations as we desire, creates variability in the number of test queries across seed numbers. Appendix A details the train-test approach in depth.

Sampling Expert Demonstrations. When comparing the effect of N demonstrations of f against that of $N+k$ demonstrations, we ensure that the $D_{train}^{f,N} \subset D_{train}^{f,N+k}$.

Environments. We utilize three API benchmarks: **WorkBench** (Styles et al., 2024) **τ-Bench (retail)** (Yao et al., 2024), and **CRMArena** (Huang et al., 2025). We utilize these benchmarks because they focus on multi-step queries. Furthermore, each has its own sandbox environment to execute its function, ensuring reliability in evaluation that is absent from other API benchmarks (Kim et al., 2024). We augment each sandbox environment so that we can change the tool descriptions given to the agent to experiment with the methods described above. In Appendix C, we detail how we standardized each dataset to fit our demonstration and documentation formats. For WorkBench, we created demonstrations from the pre-computed trajectories available in the repository that were generated by a GPT-4-powered agent that used ground-truth, original documentation. We did the same with τ-Bench, where those pre-computed trajectories are from a GPT-4.5-based agent. For CRMArena, we regenerated the expert trajectories using GPT-4o as that was their default model. For both τ-Bench and CRMArena, the return values for each step were included in the expert trajectories, so we added the return of a function call to the demonstration.

The Presence of Noisy, Suboptimal Expert Demonstrations. An important note is how we extracted expert demonstrations. We only used demonstrations from trajectories from the expert agent if it got an $r = 1$ from its environment. However, this heuristic has its flaws in cases where the evaluation is outcome-centric, where only the ending state of the environment matters. So, for example, the agent tries to send an email to an address that does not exist (e.g. nadia@example.com in WorkBench). Even though it is a wrong step, the environment does not change due to the email not going to any inbox. If the agent self-corrects by then searching for the right email and retrying to send the email, all demonstrations are used in the expert demonstration pool.

Models. For experiments with the entire pipeline, we utilize o3-mini, GPT-4o-mini, GPT-4o, as this set contains reasoning and non-reasoning models. For each experiment, we use the same model across the system. To include open-source models, we use Mixtral-8x7B-Instruct-v0.1 (Mistral) and gemma-2b-it (Gemma) to analyze document generation.

Evaluation. For task completion, we report the mean success rate (SR) over the three trials.

Filtering Tasks with Unavailable APIs. In the next section, we compare the agent task success rate with different methods I for expert demonstrations. We compare the different methods and also the number of demonstrations from $N = \{5, 15, 25, 35\}$. If an API in a dataset did not have at least 35 demonstrations, we removed that API from the set of available API the agent can use. Therefore, we filtered out any task that relies on any f where $|D_{train}^{f}| < 35$ when comparing different I. When comparing different I' after self-exploration, we set $N = 5$ for the two I methods, DxD and GD, we use before self-exploration. Thus, we filtered out queries that relied on any f where $|D_{train}^{f}| < 5$. We report the average number of test queries next to each dataset in Tables 1 and 2.

5 Results: Learning Parameter Information Is Crucial

From our experiments, a recurring problem at every stage of the pipeline (document generation, task completion, and evaluation) is the parameter fill-

Benchmark	Method	o3-mini				gpt-4o-mini				gpt-4o			
		No. of Expert Demonstrations											
		5	15	25	35	5	15	25	35	5	15	25	35
WorkBench (55 Tasks)	DxD	**41.53**	39.24	38.03	40.25	**50.65**	34.15	35.85	40.09	**17.76**	16.53	14.90	15.51
	GD	28.86	30.07	28.68	24.74	32.37	21.96	22.93	15.46	15.01	15.00	14.60	14.81
	GDEC	32.56	26.36	26.30	27.52	40.37	28.87	26.09	30.44	15.97	15.16	14.76	15.37
	OD	24.16				17.19				17.31			
τ-Bench (77.6 Tasks)	DxD	48.49	12.89	0.00	0.00	42.12	0.00	0.00	0.00	54.59	0.00	0.00	0.00
	GD	10.44	5.62	6.43	8.03	35.34	27.31	38.15	21.29	27.71	31.33	21.69	13.25
	GDEC	40.96	40.16	34.94	34.94	56.63	66.27	65.06	51.0	58.23	53.82	53.01	39.76
	OD	**78.59**				**69.08**				**86.30**			
CRMArena (183 Tasks)	DxD	69.24	67.22	0.00	0.00	12.02	0.00	0.00	0.00	66.33	0.00	0.00	0.00
	GD	52.30	49.86	18.86	39.22	0.0	0.0	0.0	0.0	8.20	14.19	15.84	14.38
	GDEC	60.49	51.88	45.92	9.96	7.51	11.52	0.0	0.0	54.13	49.91	0.00	0.00
	OD	**72.14**				**29.26**				**84.53**			

Table 1: The models specified in the top row indicate the LLM used as the API-based agent and the document generator. All experiments ran across 3 trials, and all values reported are the mean success rate (%). Bolded values are the highest SR given a dataset and model.

ing. Parameter filling has also been indicated as a problem by Styles et al. (2024); Yao et al. (2024) for WorkBench and τ-Bench, respectively. We now show how the problem increases in difficulty in scenarios with no ground-truth documentation.

Document Generation Has Problems With Specifying Parameter Information. One main issue is generating parameter information. Many times, generators hallucinated input parameters that did not exist or left out input parameters seen in the demonstrations. This occurred even when in the system prompt we stated not to hallucinate and only focus on the parameters it saw. Thus, given a set of demonstrations for f, we programmatically found all used input parameters and explicitly mentioned in the user prompt to generate information on only these parameters. That solved the issue for the OpenAI models and Gemma. However, documents generated by Mistral many times do not include parameter information (see Figure 13 in the Appendix). Other sources of error were more specific, such as when generated documents would specify to include timezone information for time_min and time_max parameters in WorkBench functions. Agents that followed this got a runtime error. The generated documents from more demonstrations would more frequently make this mistake, and also say that the time_min and time_max parameters were required when they were optional. Documents generated from fewer demonstrations would, in some trials, correctly say that those parameters were optional, allowing the agent to not fill in those parameters with incorrect formatting. We hypothesize that the more repeated use of these parameters in the larger set of demonstrations causes the document generator to believe that those are required, causing the degradation in Table 1 as N increases.

Generated Guidelines Can Help Parameter Understanding. Attaching generated guidelines (AG) has a 12% improvement over (GD) in WorkBench. In a representative case, "Kerry Moore is no longer a customer. Can you delete them from the crm?", the GD agent tries to directly delete the customer by using their name as the customer_id parameter, rather than searching for their ID and then deleting. The guidelines for customer.delete_customer state to avoid directly deleting customers and ensure that the customer ID is a valid value, not someone's name. This result shows that self-exploration with an evaluation protocol can enable self-improvement.

Error Handling of Parameter Formatting Affects Performance. However, we see that for τ-Bench, AG, and all other methods except for OD, achieved a low score with τ-Bench. A major contributing factor is the order_id parameter. The order_id parameter is in 7 out of 14 functions in τ-Bench, and requires a '#' at the beginning of the value; otherwise, an error is returned. When the order_id is formatted incorrectly, the error message "Order not found", shown in Figure 6, is misleading to the agent and the self-improvement system because there is no indication that the missing "#" is the problem of the error. Some of the generated evaluations marked some of the function calls that used the '#' as wrong. This error in evaluation then affects the summarized guidelines. In 2 trials, the guidelines would say to remove the '#' formatting

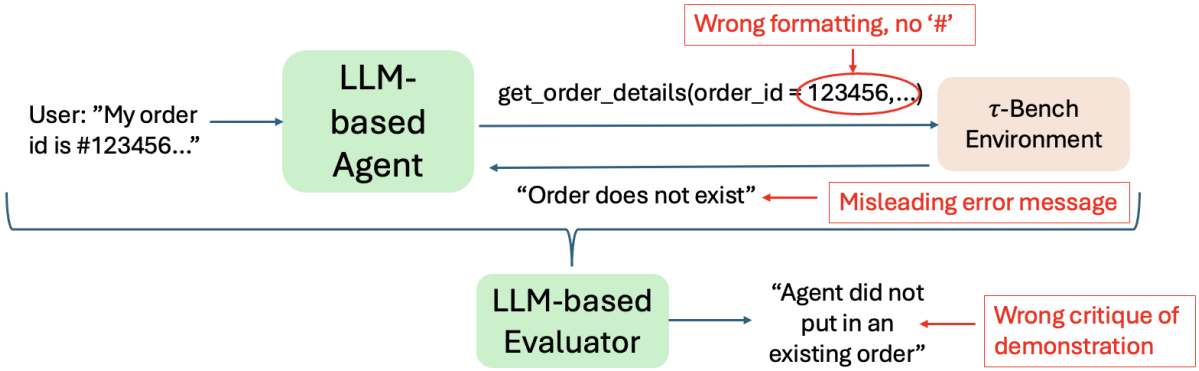

Figure 6: **Failure of System Due to Misleading Error Messages.** Incorrectly formatting the order_id parameter value for an order that does exist causes the τ-Bench function to return "Order does not exist", misleading the LLM-based agent and LLM-based evaluator.

from IDs. In τ-Bench, none of the generated documentation methods (GD, UD, and RD) mention the formatting requirement either. Therefore, the only way to know that the "#" is required was by using the OD. We reran the AG (with DxD method before self-exploration) experiment. However, this time we hardcoded the following phrase in all the guidelines for all functions that use order_id: "THERE NEEDS TO BE A '#' AT THE START OF THE ORDER ID. THIS IS MANDATORY. FOR EXAMPLE: "#W8732376" IS VALID, BUT "W8732376" IS NOT!!!!" This modification raises the average success rate from 10% to 49%, and this result shows again that having specific, correct parameter information can significantly help the SR, and that robust and descriptive error handling of the API functions is imperative for reliable evaluation and summarization.

DxD and DE have more advantages than the summarization methods. DxD and DE have consistently had a higher success rate over other I and I' methods. However, this can be attributed to data leakage. The set of tasks G for each environment has many repeated formats, such as sending emails, canceling orders, or looking up information on cases. Thus, the train-test distributions are similar. It would be interesting to try DxD and DE in environments that provide distinctive tasks that use the same APIs.

Note on Increasing N. For DxD with τ-Bench and CRMArena, using the conversation history between causes the LLM of the agent to exceed its maximum context length. We truncated the conversation history to at most the past three messages. This truncation allowed o3-mini to handle 5 and 15 demonstrations per function for τ-Bench, and 5 per function for CRMArena. All other models could only handle 5 demonstrations per function. For GDEC, we noticed that for WorkBench and CRMArena, the agent will repeatedly call the same function, suggesting that the repeated N example calls teach the LLM agent to repeat.

Test-Time Method	WorkBench (167 Tasks)		τ-Bench (83 Tasks)	
	I Method Before Self-Exploration			
	DxD	GD	DxD	GD
DE	**44.71**	32.14	45.00	44.00
UD	29.20	28.94	9.00	9.00
RD	30.34	28.54	6.00	10.00
AG	42.51	38.92	10.00	9.00
DxD	44.11		47.00	
GD	30.14		9.00	
GDEC	34.73		41.00	
OD	33.33		**78.00**	

Table 2: Results of gaining experience with training queries during self-exploration using o3-mini. We used $N = 5$ expert demonstrations per f for self-exploration. The methods on the bottom half of the table (DxD, GD, GDEC, OD) do not depend on self-exploration. All experiments ran across 3 trials, and all values reported are the mean success rate (%). Bolded values represent the highest success rate for a given dataset.

6 Additional Analysis and Results

Retrieval-Augmented Generation (RAG). To handle the context length issue seen with GPT-4o-mini, we provide an additional RAG (Lewis et al., 2020) experiment on τ-Bench using DxD. We embed the sampled N demonstrations with OpenAI text-embedding-3-small and retrieve the 5 most relevant demonstrations based on the current query with cosine similarity. This experiment is over 3 trials with the set of questions that use APIs with at least 35 demonstrations, same as in Table 1. Table

3 shows no significant change in performance in number of demonstrations when using RAG.

OD (No RAG)	5 Dems (No RAG)	15 Dems	35 Dems
69.08	42.12	39.13	44.262

Table 3: Mean success rate (%) across different demonstration counts with RAG on τ-Bench.

DxD with Open-Source Models. We ran Mistral 8x7B and Qwen3 on three trials of WorkBench with OD, and DxD (with 5 Dems). The test tasks are the same set of questions from Table 1, where APIs needed have at least 35 demonstrations. In Table 4, Qwen3 achieves higher SR with DxD than with OD, and it outperforms all OpenAI models on WorkBench. We believe that this increase is due to the amount of prior reasoning it does before generating an action, and the similarity between the train and test split in WorkBench.

I Method	Qwen3	Mistral 8x7B
DxD 5 Dems	67.27	13.28
OD	54.62	14.56

Table 4: Comparing 5 DxD against OD with Qwen-3 and Mistral 8x7B on WorkBench.

7 Related Works

Tool-based agents expand the capabilities of traditional LLMs by connecting to external sources such as API functions, search engines, data centers, etc., to complete multi-step queries. Multiple benchmarks have been proposed with various sets of API functions with documentation (Guo et al., 2024; Liu et al., 2024; Huang et al., 2025; Styles et al., 2024; Yao et al., 2024; Xu et al., 2024; Arcadinho et al., 2024). While generating documentation has been studied before the advent of LLMs (Wang et al., 2023; Nybom et al., 2018), there is little work on generating documentation for downstream agent task planning. For documentation-free agents, ToolAlpaca by Tang et al. (2023) generated documentation via an LLM given a brief description of the function from OpenAPI. API-DocBooster (Yang et al., 2023) used GPT-4 to produce and update documentation based on search results from StackOverflow. Concurrent to this work, Fang et al. (2025) also used experiences of the agent to update the documentation. However, they still relied on initial documentation. Toolformer by Schick et al. (2023) finetuned their agent on API calls. Our work is the first to study how we can generate *and* update an agent's understanding of functionality with expert demonstrations and experience with a frozen LLM. We are also the first to use demonstrations directly into the LLM-based agent for functionality understanding. Our formalization is similar to Reinforcement Learning with parameterized action spaces (Masson et al., 2016; Zhang et al., 2024) where the agent must choose an action from a discrete set (in our case, choose an $f \in \mathcal{F}$), and then must choose parameters specific to the selected action, i.e. $p \in \text{dom}(P(f))$. However, to our knowledge, previous literature assumes a single, continuous parameter per action. Our work relaxes this assumption as an API function can allow for multiple parameters, continuous or discrete.

8 Conclusion

We provide and formalize the problem of earning of API functionality from in-context demonstrations with no prior documentation. We highlight the persisting and challenging problem of API-agent planning with limited information on the set of functions. We investigate tackling this problem by learning functionality, specifically function description, return, and input parameters, from in-context demonstrations of tool calls. We analyze the number of demonstrations, various processing methods, and the impact of self-exploration and LLM-based evaluation. Importantly, our extensive experiments highlight the difficulties of SoTA LLMs on this problem, specifically due to failures in describing the parameter schema, suggesting further research to improve results.

9 Limitations and Further Work

As we present this new challenge, we would like to highlight some limitations of this work. The heuristic mentioned in Section 4 when extracting expert demonstrations leads to suboptimal demonstrations. We do not focus on mitigating or filtering out these suboptimal demonstrations. One could also look at using an LLM-based evaluator to signal what steps are correct. We do not consider it in this work, as we analyzed LLM-based evaluations during self-exploration. However, filtering out suboptimal demonstrations is an exciting and interesting direction to pursue.

Furthermore, we iteratively generated documents for each function independently. One interesting extension is to see the effect of group learning functions together. We see in τ-Bench that the order_id parameter was shared across multiple functions, so the agent should only have to learn it correctly once. This sharing of parameter information could lead to more stable performance as the information between functions is more consistent.

One could also extend this problem to the Model-Context Protocol (MCP), where the agent must rely on multiple data sources and sets of API functions. This direction is a more complex, realistic scenario than ours, as we only focus on a single set of data and APIs. Furthermore, human-in-the-loop setups where the user acts as an expert policy to extract online demonstrations could pave the way to incorporate imitation learning algorithms, such as DAgger (Ross et al., 2011).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Siyu An, Qin Li, Junru Lu, Di Yin, and Xing Sun. 2024. Finverse: An autonomous agent system for versatile financial analysis. *arXiv preprint arXiv:2406.06379*.

Samuel Arcadinho, David Aparicio, and Mariana Almeida. 2024. Automated test generation to evaluate tool-augmented llms as conversational ai agents. In *Proceedings of the 2nd GenBench Workshop on Generalisation (Benchmarking) in NLP*, pages 54–68.

Runnan Fang, Xiaobin Wang, Yuan Liang, Shuofei Qiao, Jialong Wu, Zekun Xi, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. 2025. Synworld: Virtual scenario synthesis for agentic action knowledge refinement. *Preprint*, arXiv:2504.03561.

Sakharam Gawade, Shivam Akhouri, Chinmay Kulkarni, Jagdish Samant, Pragya Sahu, Jai Pahal, Saswat Meher, et al. 2025. Multi agent based medical assistant for edge devices. *arXiv preprint arXiv:2503.05397*.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *Preprint*, arXiv:2403.07714.

Kung-Hsiang Huang, Akshara Prabhakar, Sidharth Dhawan, Yixin Mao, Huan Wang, Silvio Savarese, Caiming Xiong, Philippe Laban, and Chien-Sheng Wu. 2025. Crmarena: Understanding the capacity of llm agents to perform professional crm tasks in realistic environments. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*.

Junaed Younus Khan, Md Tawkat Islam Khondaker, Gias Uddin, and Anindya Iqbal. 2021. Automatic detection of five api documentation smells: Practitioners' perspectives. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 318–329. IEEE.

Junaed Younus Khan and Gias Uddin. 2022. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6.

Woojeong Kim, Ashish Jagmohan, and Aditya Vempaty. 2024. Seal: Suite for evaluating api-use of llms. In *NeurIPS 2024 Workshop on Open-World Agents*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Ruishi Li, Yunfei Yang, Jinghua Liu, Peiwei Hu, and Guozhu Meng. 2022. The inconsistency of documentation: a study of online c standard library documents. *Cybersecurity*, 5(1):14.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh RN, et al. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482.

Warwick Masson, Pravesh Ranchod, and George Konidaris. 2016. Reinforcement learning with parameterized actions. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Kristian Nybom, Adnan Ashraf, and Ivan Porres. 2018. A systematic mapping study on api documentation generation approaches. In *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*, pages 462–469. IEEE.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings.

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

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. 2024. Workbench: a benchmark dataset for agents in a realistic workplace setting. In *First Conference on Language Modeling*.

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. *arXiv preprint arXiv:2306.05301*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Shujun Wang, Yongqiang Tian, and Dengcheng He. 2023. gdoc: Automatic generation of structured api documentation. In *Companion Proceedings of the ACM Web Conference 2023*, pages 53–56.

Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. 2024. Officebench: Benchmarking language agents across multiple applications for office automation. *arXiv preprint arXiv:2407.19056*.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. 2024. Theagentcompany: Benchmarking llm agents on consequential real world tasks. *Preprint*, arXiv:2412.14161.

Chengran Yang, Jiakun Liu, Bowen Xu, Christoph Treude, Yunbo Lyu, Junda He, Ming Li, and David Lo. 2023. Apidocbooster: An extract-then-abstract framework leveraging large language models for augmenting api documentation. *arXiv preprint arXiv:2312.10934*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

Renhao Zhang, Haotian Fu, Yilin Miao, and George Konidaris. 2024. Model-based reinforcement learning for parameterized action spaces. In *International Conference on Machine Learning*, pages 58935–58954. PMLR.

Hao Zhong and Zhendong Su. 2013. Detecting api documentation errors. *ACM SIGPLAN Notices*, 48:803–816.

A Train-Test Split

Algorithm 1 details the process of how, for a given dataset, we divide the demonstrations of the API functions into the train and test sets. Because a task in a dataset could require different APIs, had we simply just split the demonstrations, which include task information, into train and test sets, data leakage would almost certainly occur. To ensure no tasks in the test sets were in the training sets, for each API function, we first divide the tasks from only the demonstrations of that function into train and test task sets. We then divide the demonstrations into training and test demonstration sets. For the next API function, if a task for that API has already been assigned to the train or test task set, we simply assign the demonstration to the corresponding set for demonstrations. The remaining tasks for the function that have not been seen will be split into the train and test task sets, and the process repeats for all functions.

Algorithm 1 Train-Test Split

Input: List of API functions A, dictionary of demonstrations D, test split percentage p, random seed s
Initialize: List of training tasks T_{train} = []; list of test tasks T_{train} Dictionary of training demonstrations $D_{train} =$, Dictionary of training demonstrations $D_{test} =$
for a in A **do**
  $T^a_{remaining} = \{\}$ ▷ See which tasks of a have not been assigned to either train or test
  for d in $D[a]$ **do** ▷ $D[a]$ is the list of demonstrations for a
    $t = d[Task]$ ▷ Get the task that the demonstration is for
    if t in T_{train} **then**
      Add d to list $D_{train}[a]$
    else if t in T_{test} **then**
      Add d to list $D_{test}[a]$
    else
      Add t to list $T_{remaining}Ya$
    end if
  end for
  Set random state to seed s ▷ We now split the tasks that have not been assigned yet
  $num_{test} = len(T^a_{remaining}) * p$
  $T^a_{test} =$ random n subset of $T^a_{remaining}$
  $T^a_{train} = T^a_{remaining} - T^a_{test}$
  for d in $D[a]$ **do**
    $t = d[Task]$
    if t in T^a_{train} **then**
      Add d to $D_{train}[a]$
    else
      Add d to $D_{test}[a]$
    end if
  end for
  Add elements of T^a_{train} to T_{train}
  Add elements of T^a_{test} to T_{test}
end for

B AI Writing Tools

We used LLM services such as ChatGPT and Perplexity for feedback and editing of this document.

C Dataset Statistics and Specifications

WorkBench. To extract the expert demonstrations, we take their pre-computed predictions for all queries and then run their evaluation script. From there, we have access to the prediction of their GPT-4 agent that used the original documentation. The prediction is used as the demonstration trajectory. For each

Function from WorkBench	No. of Demonstrations
analytics.create_plot	152
analytics.engaged_users_count	46
analytics.get_average_session_duration	33
analytics.total_visits_count	28
analytics.traffic_source_count	2
calendar.create_event	93
calendar.delete_event	118
calendar.search_events	85
calendar.update_event	52
company_directory.find_email_address	67
customer_relationship_manager.add_customer	25
customer_relationship_manager.delete_customer	175
customer_relationship_manager.search_customers	13
customer_relationship_manager.update_customer	133
email.delete_email	44
email.forward_email	67
email.reply_email	31
email.search_emails	62
email.send_email	108
project_management.create_task	78
project_management.search_tasks	34
project_management.update_task	96

Table 5: WorkBench functions and Number of Expert Demonstrations

function call of prediction, we format a demonstration as shown in Figure 1. Because WorkBench is outcome-centric, the ground-truth will contain fewer steps than needed to properly complete the task. The only functions that are included affect the environment. For example, sending an email to someone, a task that involves finding the email address, will only have the search_email function in the ground-truth. All functions that search for events, customers, emails, etc., or analyze data like analytics.get_average_session will not appear. Some queries will have ground truths with 0 steps. These are for queries where the required condition before performing an environment-changing action is not satisfied. For example, the agent needs to set up a meeting only if the last meeting was over a week ago, and the last meeting was the day before. Then, the ground truth is 0 steps. However, the agent should execute the search_events function before deciding not to set up an email. For this reason, we use the pre-computed trajectories provided by (Styles et al., 2024) with a GPT-4 powered agent with ground-truth documentation. We only extract demonstrations from successful trajectories. This heuristic, as previously mentioned in Section 4, can lead to wrong demonstrations in the set of expert demonstrations. Table 5 provides the number of expert demonstrations we extracted from the pre-computed trajectory.

τ-Bench and CRMArena. Both τ-Bench and CRMArena have their results stored as JSON objects. Each task attempted stores a sequence of tool calls and accompanying tool executions. Each pair of tool calls and tool executions is turned into the demonstration data, where we can also extract the Return value from the tool execution to add to the expert demonstrations. While τ-Bench does provide ground-truth function calls for the tasks, it disregards the interaction with the simulated user. Seeing examples of these interactions is important for the agent to understand when to use tools, relating to parameter filling. We thus do not use the ground-truth answers and rather use the pre-computed trajectories from the dataset as expert demonstrations. Figure 14 shows generated guidelines that specify, in green, that the agent should confirm the input parameters of the address details with the user.

Function from τ-Bench	No. of Demonstrations
calculate	36
cancel_pending_order	129
exchange_delivered_order_items	133
find_user_id_by_email	245
find_user_id_by_name_zip	320
get_order_details	871
get_product_details	340
get_user_details	279
list_all_product_types	74
modify_pending_order_address	43
modify_pending_order_items	146
modify_pending_order_payment	12
modify_user_address	16
return_delivered_order_items	140

Table 6: τ-Bench functions and Number of Expert Demonstrations

Function from CRMArena	No. of Demonstrations
calculate_average_handle_time	62
calculate_region_average_closure_times	114
find_id_with_max_value	175
find_id_with_min_value	144
get_account_id_by_contact_id	63
get_agent_handled_cases_by_period	136
get_agent_transferred_cases_by_period	97
get_agents_with_max_cases	113
get_agents_with_min_cases	32
get_cases	421
get_email_messages_by_case_id	53
get_issue_counts	166
get_issues	83
get_livechat_transcript_by_case_id	53
get_month_to_case_count	38
get_non_transferred_case_ids	59
get_order_item_ids_by_product	304
get_period	290
get_purchase_history	172
get_qualified_agent_ids_by_case_count	218
get_shipping_state	82
get_start_date	240
search_knowledge_articles	114
search_products	195

Table 7: CRMArena functions and Number of Expert Demonstrations

D Self-Exploration Pipeline

Figure 7 visualizes the process of updating the agent's initial learning from in-context demonstrations via self-exploration.

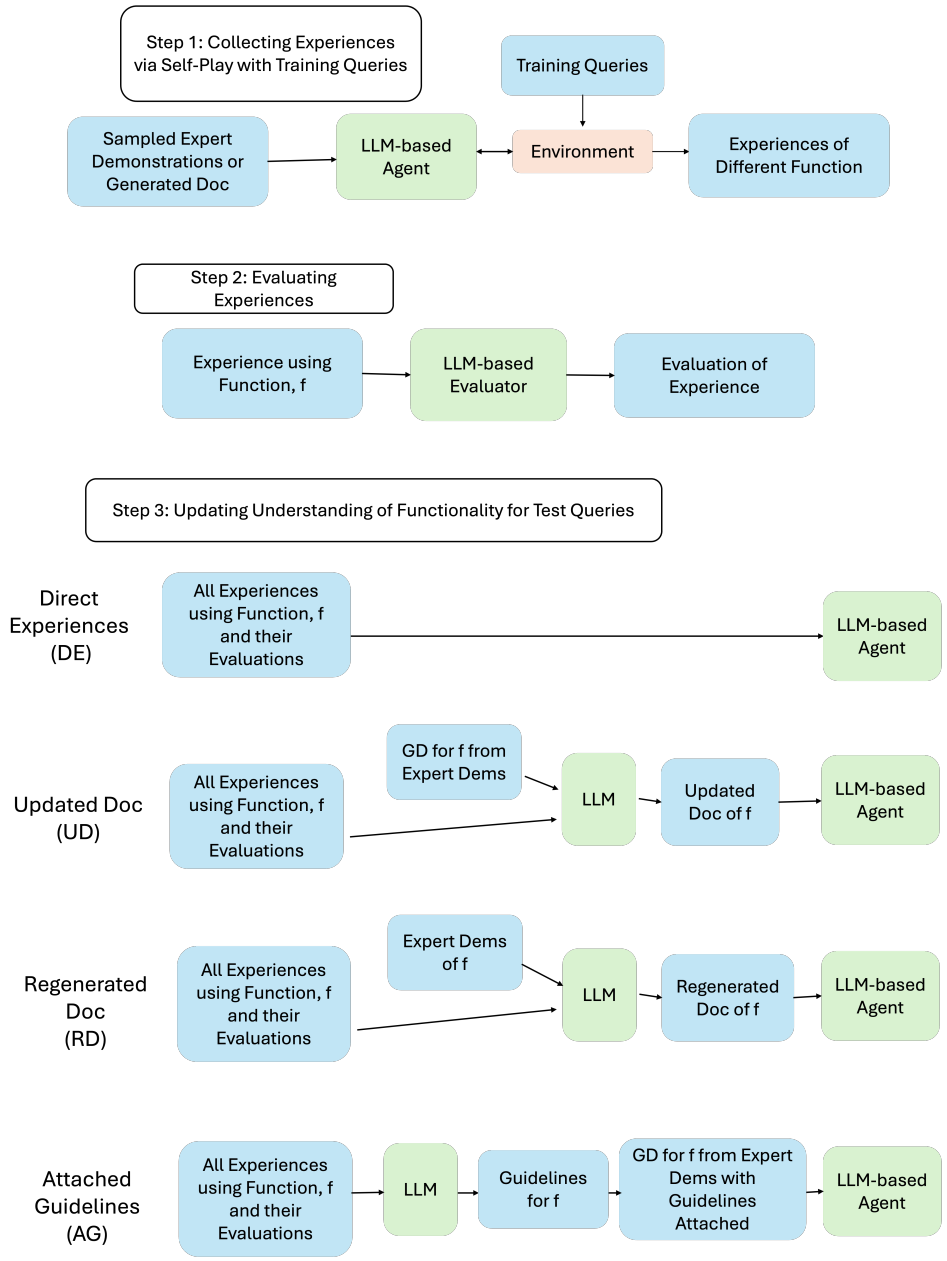

Figure 7: **Overview of Self-Exploration Pipeline.** Visualizes is the three-step process of updating functionality understanding with experience. In the first step, the agent performs self-exploration with the training queries in the sandbox environment, collecting experiences. The agent can use any method I to initially learn functionality from expert demonstrations. In the second step, each experience is then evaluated via an LLM-based evaluator. Finally, we present four different options to update the agent's understanding of functionality with the original expert demonstrations and the new experience-based demonstrations. The updated understanding is then passed into the agent for test queries.

E System Prompts

```
You are given the demonstrations of an API function. Your task is to create a
comprehensive introduction for this API function.
The demonstration will include:
1) the name
2) input parameter names
3) an example task it is trying to solve. Note that some demonstrations of an
API are part of sequence of function calls. You will not be given the whole
sequence, just the call of the function in question.
Some demonstrations may include the output of the function call.
Guidelines:
1. For the API function, detail its purpose, input requirements, and output
results.
2. For function input, present it in JSON format. Each key should be the input
parameter's name, and its value should be a string
indicating whether it's required or not, its type, and a brief description,
such as "Required/Optional. Integer. {some description}".
3. The input parameters you include in your documentation should be restricted
to values you have seen in the example demonstrations,

DO NOT MAKE UP NEW INPUT PARAMETERS!!!!!!!!

Specifically, consider adding/editing if you can:
- valid input values for the input parameters. Consider if the variable is
categorical and takes only a finite set of values.
- for each input parameter, whether or not it is required and the value type.
- if the input parameter is a string representing the date, be careful with
the formatting
- the return value type and/or what it represents. For example, if the output
has '.png' or '.pdf', that means it returns a filepath
- the natural language description of the functions, input parameters, and
return values

4. Output with the following format:

Name: function name
Description:
short description of function

Input Parameters:
List of all input variables formatted with name: description
Each input variable should be on a separate line.


Return:
function output, describe all the output variable that this function will
return
```

Figure 8: System Prompt of document generator

```
You are given an AI-generated documentation of an API function, and you are
given the demonstrations of that API function.
Your task is to update the AI-generated documentation.
1) the name
2) input parameter names
3) an example task it is trying to solve. Note that some demonstrations of an
API are part of
sequence of function calls. You will not be given the whole sequence, just the
call of the function in question.
From the given demonstrations, please update the documentation with more
details.
Please follow the guidelines as in the initial generation, listed below:
Guidelines:
1. For the API function, detail its purpose, input requirements, and output
results.
2. For function input, present it in JSON format. Each key should be the input
parameter's name, and its value should be a string
indicating whether it's required or not, its type, and a brief description,
such as "Required/Optional. Integer. {some description}".
3. The input parameters you include in your documentation should be restricted
to values you have seen in the example demonstrations,

DO NOT MAKE UP NEW INPUT PARAMETERS!!!!!!!!

Specifically, consider adding/editing if you can:
- valid input values for the input parameters. Consider if the variable is
categorical and takes only a finite set of values.
- for each input parameter, whether or not it is required and the value type.
- if the input parameter is a string representing the date, be careful with
the formatting
- the return value type and/or what it represents. For example, if the output
has '.png' or '.pdf', that means it returns a filepath
- the natural language description of the functions, input parameters, and
return values

4. Output with the following format:

Name: function name
Description:
short description of function

Input Parameters:
List of all input variables formatted with name: description
Each input variable should be on a separate line.


Return:
function output, describe all the output variable that this function will
return
```

Figure 9: System Prompt of documentation updater given initial generated document and experiences

```
You are evaluating the API calls of a tool-based agent. You will be given a query
with a accompany trajectory of API calls that try to solve the task. You will be
told if the trajectory is correct or not. Each API call presented in the trajectory
will be formatted as a json dictionary. We will specify an API call in the
trajectory for you to evaluate and explain whether the API call was used correctly.
Your response will be used as verbal feedback to the tool-based agent.
Please consider elements like:
1) whether or not this was a repeated call
2) the parameters are filled correctly
3) whether or not the API is used in the correct order of the sequence
```

Figure 10: System Prompt of evaluator that critiques an experience during self-exploration

```
You are LLM summarizer that takes in the demonstrations of an API function for
various tasks. These demonstrations were collected from the agent trying to perform
the tasks. From these demonstrations, please give some guidelines and warnings on how
to best use the given function. Each demonstration has an evaluation on the use of
that function. The guidelines and warnings generated could be example scenarios of
when to use the function or an incorrect ways to use the function the agent should
avoid. Your response will be attached to the documentation of the given API function
for the agent to use for unseen tasks.
```

Figure 11: System Prompt that summarizes experiences and evaluations to generate guidelines

Name: analytics.create_plot
Description:
The `analytics.create_plot` function is designed to generate a visual representation of data based on the provided parameters. It allows users to specify the type of plot, the data to be visualized, and various customization options to tailor the plot to specific needs. This function is useful for creating charts and graphs for data analysis and presentation purposes.
Input Parameters:
{"**data**": "Required. Array or DataFrame. The dataset to be visualized.", "plot_type": "Required. String. The type of plot to create, such as 'line', 'bar', 'scatter', etc.", "title": "Optional. String. The title of the plot.", "x_label": "Optional. String. The label for the x-axis.", "y_label": "Optional. String. The label for the y-axis.", "color": "Optional. String. The color scheme to use for the plot."}
Return:
The function returns a plot object that can be displayed or saved. This object includes the visual representation of the data as specified by the input parameters, and it can be further manipulated or exported in various formats for reporting or presentation purposes.

Figure 12: GPT-4o Generated Document of analytics.create_plot with 10 demonstrations before we explicitly stated in the user prompt what the input parameters should be. GPT-4o hallucinates a 'data' input parameter that does not exist in the implementation.

Name: analytics.create_plot

Description:
This function creates a plot based on the given input parameters. It can be used to visualize data over a specific time range with a chosen plot type.

Figure 13: Mistral Generated Document of analytics.create_plot with 5 demonstrations. It has no parameter information.

Below are some guidelines and warnings to consider when using modify_pending_order_address in unseen tasks...

• Strip any extraneous symbols from the order ID if needed. In the demos, the user sometimes provided an order number with a leading "#" but the API call typically uses the ID without it. Follow the proper format expected by the backend.

• Confirm the new address details with the user before issuing the API call. For example, re-summarize the new address and ask the user to confirm that these are the correct shipping details to be applied.

• Check the correct sequence of operations. The modify_pending_order_address call should come after identity verification and confirmation that the user's latest shipping address update information is correct. Do not repeat calls if a prior call already made a modification...

By following these guidelines, the agent can minimize errors and ensure that address modifications are performed correctly in unseen tasks.

Figure 14: Generated guidelines for the modify_pending_order_items function in τ-Bench