Beyond Perfect APIs: A Comprehensive Evaluation of LLM Agents Under Real-World API Complexity

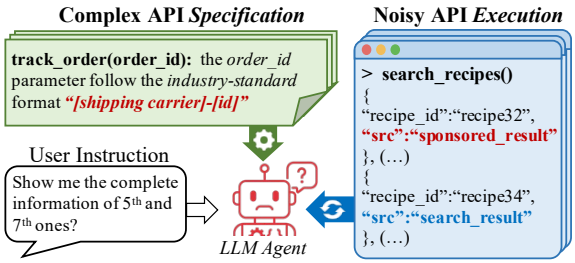
Anonymous ACL submission

Abstract

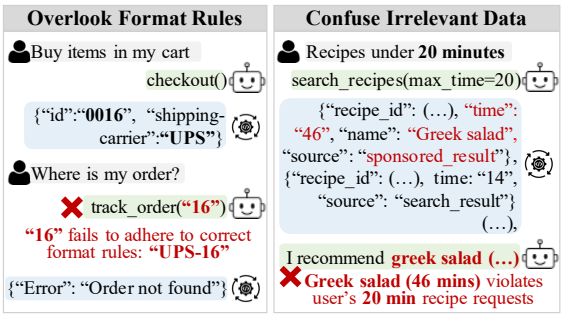
We introduce WILDAGTEVAL[1], a benchmark designed to evaluate large language model (LLM) agents' function-calling capabilities under *realistic API complexity*. Unlike prior work that assumes an *idealized* API system and disregards real-world factors such as noisy API outputs, WILDAGTEVAL accounts for two dimensions of real-world complexity: ❶ **API specification**, which includes detailed documentation and usage constraints, and ❷ **API execution**, which captures runtime challenges. Consequently, WILDAGTEVAL offers (i) an API system encompassing 60 distinct complexity scenarios that can be composed into approximately 32K test configurations, and (ii) user-agent interactions for evaluating LLM agents on these scenarios. Using WILDAGTEVAL, we systematically assess several advanced LLMs and observe that most scenarios are challenging, with *irrelevant information* complexity posing the greatest difficulty and reducing the performance of strong LLMs by 27.3%. Furthermore, our qualitative analysis reveals that LLMs occasionally *distort* user intent merely to claim task completion, critically affecting user satisfaction.

1 Introduction

Large language models (LLMs) agents, such as Amazon Alexa, have rapidly emerged as powerful interfaces for numerous real-world applications, building on LLMs' remarkable performance—surpassing human accuracy in college-level mathematics and excelling in high-stakes domains (DeepSeek-AI et al., 2025; Grattafiori et al., 2024; OpenAI et al., 2024). To evaluate this role, a growing body of work has introduced *function-* or *tool-calling* benchmarks (Patil et al., 2025; Yao et al., 2024; Zhong et al., 2025; Basu et al., 2024), which assess whether agents produce correct API calls that fulfill user instructions. These benchmarks steadily refine our understanding of whether LLM agents can effectively address diverse instructions and execute complex, multi-step tasks.



(a) Challenges in real-world agent deployment.



(b) Unassessable agent failures by current benchmarks.

Figure 1: **Key motivation for WILDAGTEVAL:** (a) highlights the challenges in real-world agent deployment; and (b) provides conversations of WILDAGTEVAL that reveal LLM agents' failure modes often overlooked by current benchmarks.

Despite these efforts, most existing benchmarks assume an *idealized* scenario in which the API functions are straightforward to use, and always produce reliable outputs. However, as shown in Figure 1(a), ***these assumptions deviate substantially from real-world scenarios***. In practical deployments (e.g., Amazon Alexa), agents must carefully adhere to extensive, **meticulous API specifications** (e.g., *domain-specific* formatting rules "`[shipping carrier]-[id]`") while also managing **imperfect API execution**, which often produces *noisy* outputs (e.g., "`sponsored_result`") or encounters runtime *errors*.

[1]Code & Data: `anonymous.4open.science/r/WildAGTEval`.

Benchmarks	❶ API Specification Complexity				❷ API Execution Complexity			
	Ad-hoc Rules	*Unclear Functionality Boundaries*	*Functional Dependencies*	*Ambiguous Descriptions*	*Informational Notices*	*Partially Irrelevant Information*	*Feature Limitation Errors*	*System Failure Errors*
BFCLv3 (Patil et al., 2025)	×	×	○	○	×	×	×	○
ComplexFuncBench (Zhong et al., 2025)	×	×	○	○	×	×	×	×
τ-bench (Yao et al., 2024)	×	×	○	×	×	×	×	×
ToolSandbox (Yao et al., 2024)	×	×	○	×	×	×	×	○
AgentGym (Xi et al., 2024)	×	×	○	×	×	×	×	×
Multi-Turn-Instruct (Han, 2025)	×	×	×	×	×	×	×	×
Lost-in-Conv (Laban et al., 2025)	×	×	×	×	×	×	×	×
WildAgtEval	○	○	○	○	○	○	○	○

Table 1: Comparison of *API complexity coverage* between prior representative benchmarks and WildAgtEval.

Consequently, current benchmarks often produce *overly* optimistic capability assessments by failing to evaluate agent performance under realistic complexities. For example, in Figure 1(b), these benchmarks cannot detect agent failures arising from intricate *API specification*, wherein agents simply use seemingly relevant information (e.g., "16") rather than adhering to the required format (e.g., "UPS-16"). Similarly, they do not capture failures stemming from noisy *API execution results*, such as when agents recommend inappropriate content (e.g., a 46-minute recipe for a 20-minute meal request) due to confusion over irrelevant sponsored results.

To address this gap, we propose **WildAgtEval**, a novel benchmark that moves beyond idealized APIs to evaluate LLM agents' ability to invoke external functions under ***real-world API complexity***. Specifically, WildAgtEval simulates these real-world complexities within an API system—a fixed suite of functions—thus exposing challenges during user–agent conversations (Figure 1(b)). Consequently, WildAgtEval provides (i) the API system and (ii) user-agent interactions grounded in it, covering a broad range of *complexity types*, as shown in Table 1: ❶***API specification***, covering intricate documentation and usage rules, and ❷***API execution***, capturing runtime challenges. Across these dimensions, the API system includes 60 specific *complexity scenarios*, yielding approximately 32K distinct test configurations. User-agent interactions for these scenarios are generated using a recent conversation-generation method (Barres et al., 2025).

An important consideration is that the complexity scenarios in the API system should faithfully reflect the real-world API environments. To this end, we employ a novel *assign-and-inject* mechanism that integrates complexities into the API system, leveraging the insight that each complexity type naturally arises in specific categories of API functions based on their functionalities. For example, *irrelevant information* frequently occurs in information-retrieval functions (e.g., `search_recipes()` in Figure 1(b)). Accordingly, we first *assign* each complexity type to the functions most likely to encounter this type of complexity in the real world; and then *inject* these complexities by modifying the corresponding API implementations.

Our evaluation on WildAgtEval shows that most complexity scenarios consistently degrade performance across strong LLM agents (e.g., Claude-4-Sonnet (Anthropic, 2025c)), with *irrelevant information* complexity posing the greatest challenge, causing an average performance drop of 27.3%. Moreover, when multiple complexities accumulate, the performance degrades by up to 63.2%. Qualitative analysis further reveals that, when facing unresolvable tasks, LLMs persist in attempting to solve them, ultimately distorting user intent and producing misleading success responses.

2 Related Work

2.1 API-Based Benchmarks for LLM Agents

Existing API-based benchmarks have advanced agent evaluation by focusing on multi-step reasoning through functional dependencies. BFCLv3 (Patil et al., 2025) and τ-bench (Yao et al., 2024; Barres et al., 2025) introduce sequential dependency scenarios (e.g., `search_media()` $\rightarrow$ `play()` for playback). Moreover, ComplexFuncBench (Zhong et al., 2025) incorporates user constraints into multi-step tasks and NESTful (Basu et al., 2024) extends to mathematical domains. Meanwhile, Incomplete-APIBank (Yang et al., 2024) and ToolSandbox (Lu et al., 2024) assess robustness of agents to missing APIs and service state variations, respectively. Nevertheless, prior benchmarks still underrepresent real-world API complexities (Table 1), yielding overly optimistic assessments; this work addresses this gap by integrating such complexities into agent evaluation.

	Complexity Type	Description	Occurrence Context	Example	Desirable Action
API SPECIFICATION	**Ad-hoc rules**	Enforces domain- or legacy-driven formats or usage conventions	Imposed by domain-specific standards, legacy design, or regulation	Parameter "time" requires ISO 8601 format (ISO, 2024); parameter "phone_number" requires E.164 format (ITU, 2022)	Follow required format
	Unclear functionality boundaries	Exposes superficially similar functions with distinct functionalities	Caused by incremental integration of API functions without a coordinated naming convention	Includes similarly named API functions with different roles: `search_product()` and `search_inventory()`	Predict correctly without confusion
	Functional dependencies	Requires a fixed sequence of prerequisite API calls	Arises from underlying workflow of API system	Devices must be powered on before being used	Orchestrate dependency
	Ambiguous descriptions	Omits specification of units, defaults, etc.	Caused by inconsistent documentation updates	Parameter "temperature" omits the unit (e.g., °C or °F)	Infer sensible default
API EXECUTION	**Informational notices**	Returns successful API results with advisory messages (warnings, or companion actions)	Provided to promote new features or deliver best-practice guidance	Displays an informational notice: "Frequently used operations: `brightness_adjust()`, `color_set()`, `play()`, etc."	Proceed without being affected by notices
	Partially irrelevant information	Returns requested data with irrelevant content such as advertisements	Introduced by monetization or non-essential tracking information	Insert sponsored companies' contents into the main results	Filter out irrelevant information
	Feature limitation errors	Fails requests due to feature unavailability accompanied by workarounds	Triggered by limited quotas, size caps, or plan entitlements	API request fails and displays implicit workaround: "search is only limited to recent info."	Apply workaround ('recent search')
	System failure errors	Fails completely due to system issues	Caused by infrastructure or upstream outages	API request fails and displays system-side cryptic error codes	Provide error reporting

Table 2: API complexity taxonomy. Each entry lists its occurrence context, a representative example, and the desirable agent action. The representative agent failure cases are discussed in Section 5.3.

2.2 User-Based Benchmarks for LLM Agents

User-based benchmarks assess whether LLM agents fulfill diverse real-world user instructions of varying complexity. MT-Eval (Kwan et al., 2024), Multi-Turn-Instruct (Han, 2025), and Lost-In-Conv (Laban et al., 2025) evaluate how well agents handle diverse user instructions in multi-turn interactions, emphasizing challenges such as intent disambiguation, multi-intent planning, and preference elicitation. IHEval (Zhang et al., 2025) and UserBench (Qian et al., 2025) further extend these evaluations by incorporating scenarios where user intents are conflicting and evolving, respectively. Despite their broad coverage, they typically overlook the complexities related to tool invocation, such as complex API specifications or noisy API execution.

3 WILDAGTEVAL: Evaluating LLM Agents under API Complexity

We present WILDAGTEVAL to benchmark the robustness of LLM agents when invoking external functions under *real-world API complexities*. Following prior work (Patil et al., 2025; Zhong et al., 2025), an agent receives (i) an executable API system and (ii) a sequence of user-agent interactions (hereafter, referred to as the *conversations*), and then produces responses by invoking the appropriate API calls.

WILDAGTEVAL advances existing benchmarks to better reflect real-world agent challenges. It introduces eight **API complexity types** commonly observed in practice, and **integrates them into an API system** guided by real-world usage patterns. As a result, WILDAGTEVAL comprises 60 distinct complexity scenarios, supporting a potential of approximately 32,000 unique test configurations.

3.1 Taxonomy of API Complexities

Table 2 describes the eight types of API complexity spanning the API specification and execution phases, and Figure 2 illustrates a prompt for an agent that shows how these complexities manifest in each API function. The *specification complexities* affect what the agent reads. Within the agent's prompt, these complexities appear in the "API functions" and "Instructions" sections (see Figure 2), ensuring their consistent inclusion during prompt construction. In contrast, the *execution complexities* affect what the agent observes after making API calls; they are introduced into subsequent prompts according to the API calls invoked by the agent. As shown by [C6] in Figure 2,

Predict the next API call, using the provided API functions and the prior dialogue including previous calls and execution results.
1. API functions
- {"name": "get_call_history", "description": "Returns the call history for the specified recent time window", "parameters": {"time": {"type": "string", "description": "ISO 8601 format time duration (e.g., P(n)Y(n)M(n)DT(n)H(n)M(n))"}}}}[C1]
- {"name": "get_user_inventory", "description": "Searches and returns the devices owned by the user", … } [C2]
- {"name": "get_device_inventory", "description": "Returns warehouse inventory, including stock quantities, reorder points, …", …}
- {"name": "temperature_set", "parameters": {"temperature": "description": "Target temperature to set on user devices" }}}[C4]
2. Instructions
When setting the color, always make sure the device is turned on. If not, turn it on. (…) [C3]
3. Conversation (Including prior API calls and their execution results)
- Turn1) API call: temperature_set(temperature="25") → {"result": "Successfully set. Adjust the mood with brightness_adjust()"}[C5]
- … - Turn3) API call: stock_price(stock="AMZN") → {"result": "AAPL: 226.7 (sponsored), AMZN: 230.3"} [C6]
- Turn4) API call: get_message(limit="10") → {"result": "Currently unavailable, retrieval is limited to recent history"} [C7]
- Turn5) API call: get_user_inventory() → {"result": "Failed with error code DB_EXHAUSTED_0x7F3A"}[C8]
4. User Query: "Change brightness to 20"

API Specification Complexities
[C1] Ad-hoc rules
[C2] Unclear func. boundaries
[C3] Functional dependencies
[C4] Ambiguous descriptions

API Execution Complexities
[C5] Information notices
[C6] Partially irrelevant info.
[C7] Feature limitation errors
[C8] System failure errors

Figure 2: A prompt for an LLM agent in WILDAGTEVAL, encompassing *specification-level complexities*—conventional parameter rules, similar yet functionally distinct functions, domain-specific dependencies, and undocumented details ([C1–C4])—alongside *execution-level complexities*—companion function notice, irrelevant sponsored content, partial failure with implicit workaround, and complete failure with cryptic error code ([C5–C8]).

once the agent calls `stock_price()`, the subsequent prompt contains irrelevant information (e.g., "AAPL (sponsored)"), forcing the agent to filter or reconcile noisy outputs.

3.2 Real-World Complexity Integration

Each complexity type (e.g., irrelevant information) naturally arises in specific categories of functions, reflecting their core functionalities (e.g., information retrieval). Building on this observation, we employ an *assign-and-inject* complexity integration mechanism where each complexity type is first *assigned* to the functions most likely to encounter this type of complexity—based on its real-world likehood of occurrence—and subsequently *injected* into those functions. For example, the *irrelevant information* complexity is assigned and injected into the information retrieval function, `stock_price()`. Furthermore, to ensure natural assignments of all complexity types to relevant functions, we construct our API system to include a sufficiently diverse set of functions.

4 Benchmark Construction through Complexity Integration

4.1 Overview

Figure 3 illustrates the *assign-and-inject* process that constructs **WILDAGTEVAL**, outlined as

- **Stage 1:** Construct a multi-domain API system and conversations grounded in that system.
- **Stage 2:** *Assign* complexities to relevant functions and create concrete complexity scenarios.
- **Stage 3:** *Inject* these scenarios into API functions and, when required, into conversations.

4.2 Stage 1: Multi-Domain API System and Conversation Construction

API system construction. To enable natural complexity integration across all complexity types, we

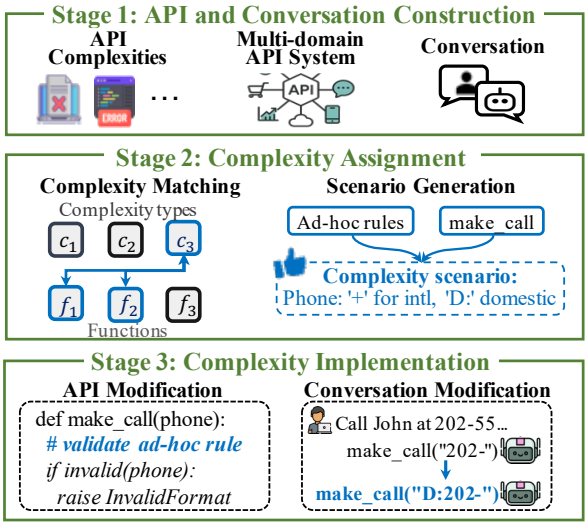


Figure 3: Overview of WILDAGTEVAL construction.

develop a comprehensive multi-domain API system spanning seven commonly used domains (e.g., device control, information retrieval) comprising 86 API functions. Each function is fully executable, accompanied by the relevant databases and policies. Due to the extensive domain coverage and policy constraints, the prompt describing API usage for LLM agents reaches approximately 34K tokens.

Conversation construction. We adopt the conversation generation framework (Barres et al., 2025) to produce multi-turn conversations paired with precise API call annotations. As outlined in Figure 9, the process begins by curating *verified intent primitives*, which are atomic units encapsulating a specific user goal (e.g., "watch a movie") along with the corresponding API calls (e.g., `search_media()`, `power_on()`, `play()`). Subsequently, an LLM composes these primitives into longer interaction scenarios, ensuring realistic conversations. All synthesized conversations undergo post-generation validation to confirm both the coherence of the conversational context and the correctness of the associated API call annotations. The

resulting dataset contains 300 multi-turn conversations requiring a total of 3,525 agent API calls, with each conversation averaging approximately 4.7 user-agent turns and 2.5 API calls per turn.

Further details regarding the construction of both the API system and the conversations are provided in Appendix A.1.

4.3 Stage 2: Complexity Assignment

We identify *relevant* complexity-function pairs by evaluating their *likelihood of real-world occurrence*, subsequently generating concrete scenarios for pairs deemed highly relevant.

Relevance-based complexity matching. Consider a multi-domain API system with functions $\mathcal{F} = \{f_1, ..., f_n\}$ and a set of complexity types $\mathcal{C}$. For each complexity type $c \in \mathcal{C}$, we aim to find a subset of functions $\mathcal{F}_c^* \subset \mathcal{F}$ most relevant to c (e.g., $\mathcal{F}_{c_3}^* = \{f_1, f_2\}$ in Figure 3). Specifically, for every pair (c, f), we quantify *likelihood of real-world occurrence* via a *relevance scores* $\mathrm{r}_{c,f}$. We obtain this score by using an instruction-following language model with a relevance assessment template $\mathcal{I}_{\text{rel}}$ as

$$\mathrm{r}_{c,f} = \mathrm{InstructLM}(c, f, \mathcal{I}_{\text{rel}}), \tag{1}$$

where $\mathcal{I}_{\text{rel}}$ specifies how to assess the likelihood that f will exhibit complexity type c in real-world environments (see Figure 13).

Then, to select the most relevant functions $\mathcal{F}_c^*$ for complexity type c, we take the top-k functions based on their relevance score as

$$\mathcal{F}_c^* = \{f : f \in \text{top-k}(\mathrm{r}_{c,f})\}. \tag{2}$$

Complexity scenario generation. After determining the most relevant complexity-function pairs, we generate concrete complexity scenarios (e.g., an ad-hoc rule on the phone number in Figure 3). Specifically, for each pair (c, f) with $f \in \mathcal{F}_c^*$, we construct a scenario $\mathrm{s}_{c,f}$ by mapping the complexity type c to a real-world application context for f using the scenario specification template $\mathcal{I}_{\text{scen}}$ as

$$\mathrm{s}_{c,f} = \mathrm{InstructLM}(c, f, \mathcal{I}_{\text{scen}}), \tag{3}$$

where $\mathcal{I}_{\text{scen}}$ specifies how c could manifest in f under real-world conditions, as detailed in Figure 14. Because the generated scenarios exhibit considerable variation, we generate multiple candidates $\{s_{c,f}^{(i)}\}_i$ via Eq. (3) and select the most representative using a validation template $\mathcal{I}_{\text{val}}$ as

$$\mathrm{s}_{c,f}^* = \underset{\mathrm{s} \in \{\mathrm{s}_{c,f}^{(i)}\}_i}{\operatorname{argmax}} \mathrm{InstructLM}(\mathrm{s}, c, f, \mathcal{I}_{\text{val}}), \tag{4}$$

where $\mathcal{I}_{\text{val}}$ evaluates the realism and fidelity to the specified complexity type, as shown in Figure 17.

4.4 Stage 3: Complexity Implementation

We integrate selected scenarios $\mathrm{s}_{c,f}^*$ into the API function f and into the conversation, adhering to rigorous quality assurance protocols. Table 9 summarizes the complexity scenarios implemented for each complexity type in WILDAGTEVAL.

API system update. We modify the code implementations of relevant API functions to realize the complexity scenarios. For example, we incorporate validation procedures (e.g., `invalid(phone)` in Figure 3) and adjust the function outputs to introduce noise (e.g., sponsored content).

Conversation update. Certain complex scenarios require modifications to the annotations for conversations, i.e., gold API calls and reference responses, to accurately reflect newly introduced complexities. For example, when applying ad-hoc rules, we update the gold API calls to follow the required formats (e.g., "`D:`" in Figure 3). For feature limitation and system failure errors, we annotate the prescribed workaround and clear error messages, respectively, as the *new* reference response. Additional details appear in Appendix A.3.

Quality assurance. Maintaining API executability and accurate labels is essential for reliable evaluation; for example, if alternative solutions exist, agents may exploit unintended solution paths, potentially undermining the evaluation's validity. To prevent such issues, we employ (1) manual editing of API functions and conversation labels (Patil et al., 2025; Prabhakar et al., 2025), and (2) trial runs with Claude-4-Sonnet (Anthropic, 2025c) to detect all alternative solutions, following the prior work (Yao et al., 2024).

5 Evaluation

5.1 Experiment Setting

Evaluation framework. We evaluate LLM agents using WILDAGTEVAL under two setups.

Isolated complexity: We measure the impact of each complexity scenario on LLM agents independently by preserving a correct conversation history with gold API calls. This controlled experimental setup enables a fine-grained analysis of each complexity's influence that may be obscured by dominant complexity factors.

Cumulative complexity: We assess how accumulated complexities impact agent performance

LLM Agents	Specification Complexity				Execution Complexity				Average (%) (Agent-wise)	
	Ad-hoc Rules		Unclear Func.		Info. Notice		Irrelevant Info.			
	✕	◯	✕	◯	✕	◯	✕	◯	✕	◯
Claude-4.0-Sonnet (Think)	78.3	69.6	63.3	56.9	87.9	81.8	71.2	61.6	75.2	67.5
Claude-4.0-Sonnet	76.1	69.6	55.0	48.6	86.4	81.8	68.8	54.5	71.6	63.6
Claude-3.7-Sonnet	78.3	54.3	59.6	56.9	81.8	81.8	67.4	53.5	71.8	61.6
Claude-3.5-Sonnet	67.4	60.9	40.4	40.4	75.0	70.5	62.8	51.2	61.4	55.8
GPT-OSS-120B	73.9	65.2	52.3	49.5	81.8	81.8	65.1	53.5	68.3	62.5
Qwen3-235B-Instruct	71.7	69.6	56.9	50.5	79.5	77.3	58.1	37.2	66.6	58.7
Qwen3-235B-Thinking	71.7	65.2	63.3	56.9	86.4	81.8	65.1	46.5	71.6	62.6
Qwen3-32B	56.5	54.3	36.5	35.3	68.2	68.2	55.8	41.9	54.2	49.9
Mistral-24B-Inst	65.6	52.5	41.9	29.6	69.2	66.2	51.2	41.9	57.0	47.6
DeepSeek-R1-Qwen32B	23.9	23.9	10.3	9.0	36.4	43.2	32.6	27.9	25.8	26.0
Average Accuracy (%)	66.3	58.5	48.0	43.4	75.3	73.4	59.8	47.0	61.9	55.2
Average Degradation (%)	**13.4**		**10.6**		**2.48**		**27.3**		**12.0**	

Table 3: Performance comparison of LLM agents under *isolated* complexity evaluation on WildAgtEval complexities. Results are presented for complexity-absent conditions (✕) and complexity-injected conditions (◯), with performance measured by *API call accuracy* (%).

across multi-turn conversations, where complexities compound and conversational context accumulate across turns.

LLM agents. We employ ten state-of-the-art LLMs, comprising seven resource-intensive models including Claude-4.0-Sonnet, Claude-4.0-Sonnet (Think), Claude-3.7-Sonnet, Claude-3.5-Sonnet (Anthropic, 2024, 2025a,c), GPT-OSS-120B (OpenAI, 2025), Qwen3-235B-Instruct, and Qwen3-235B-Thinking (Alibaba, 2025a,b); and resource-efficient models including Qwen3-32B (Alibaba, 2025c), Mistral-Small-3.2-24B-Instruct (MistralAI, 2025), and DeepSeek-R1-Qwen32B-Distill (DeepSeek, 2025).

Metrics. We evaluate (1) *API call accuracy*, quantifying the agreement between agent predictions and the gold API calls that fulfill the user's primary intent (Yao et al., 2024), and (2) *error-handling accuracy* in *feature limitation* and *system failure error* scenarios, assessing the degree of alignment between the agent's fallback response and a reference error handling response. Because multiple valid solutions may exist, we employ LLM-Judge (Zheng et al., 2023) to quantify the alignment.

Implementation details. We adopt the ReAct prompting (Yao et al., 2023), instructing LLM agents to produce responses in the format "Thought: {reasoning or explanatory text} Action: {JSON-format action argument}" under zero-shot inference. We limit inference to 15 steps per conversational turn, mitigating excessive computational overhead while allowing sufficient reasoning depth for complex scenarios. The complete implementation details can be found in Appendix B.

5.2 Main Results

All API complexity types degrade agent performance, with irrelevant information presenting the greatest challenge. Table 3 presents the results of our isolated complexity evaluation, indicating that all types of API complexity in WildAgtEval reduces the performance of all LLM agents—by an average of 12.0%. Notably, the introduction of irrelevant information imposes the most significant challenge (see Example 2), resulting in an average performance deterioration of 27.3%. This finding shows that even state-of-the-art LLMs struggle significantly with real-world API complexities.

Agent performance consistently deteriorates with accumulating API complexity. Figure 4 presents the results from the cumulative complexity evaluation, which incrementally compounds API complexity. Performance declines by an average of 34.3%, peaking at a 63.2% for Claude-3.5-Sonnet—substantially exceeding the degradation observed in isolated complexity evaluations. Interestingly, Claude-3.7-Sonnet achieves the highest resilience, while Claude-4.0-Sonnet (Think) experiences a sharper degradation relative to its robustness under isolated complexities. This result suggests that different levels of complexity exert heterogeneous effects on LLM agents.

Agents struggle to apply effective workarounds for feature limitation errors. Error-handling evaluations in Table 4 suggest that feature limitation errors are 34.3% more challenging than system failure errors. In many instances, agents solve problems in incorrect ways (see Example 4), or prematurely terminate the process. For example, GPT-

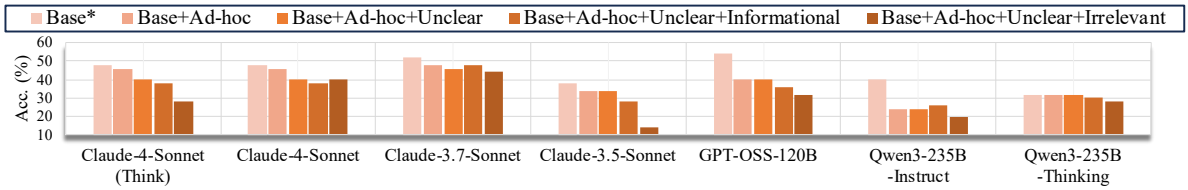


Figure 4: Agent performance across *cumulative* complexity environments, measured by API call accuracy (%). Each bar represents a different complexity setting, with darker colors indicating additional complexities added. Note that the Base* environment incorporates the complexities of functional dependencies and ambiguous documentation.

LLM Agents	API Execution Complexity		Average
	Feature Error	System Error	(Agent-wise)
Claude-4.0-Sonnet (Think)	3.10	4.08	3.59
Claude-4.0-Sonnet	2.83	4.14	3.49
Claude-3.7-Sonnet	2.64	3.88	3.26
Claude-3.5-Sonnet	2.03	3.71	2.88
GPT-OSS-120B	2.71	2.77	2.74
Qwen3-235B-Instruct	2.80	3.46	3.13
Qwen3-235B-Thinking	3.27	4.01	3.64
Average Score	2.77	3.72	3.25

Table 4: Agent's *error-handling* accuracy, measured by LLM-Judge on 1–5 scale where higher scores indicate superior error-handling. The reported values represent the average accuracy across eight complexity scenarios for each error complexity type (feature limitation and system failure).

OSS-120B returns only concise responses (e.g., "currently unavailable") without attempting to explore potential solutions.

Enhancing reasoning is an effective strategy for handling error-based complexities. Specifically, the advanced reasoning LLMs such as Claude-4.0-Sonnet (Think) and Qwen3-235B-Thinking show an 11.4% improvement over average error handling. Appendix C.1 compares the responses of Qwen3-235B-Thinking (*reasoning*) and Qwen3-235B-Instruct (*non-reasoning*) upon receiving the error message "Weather data for Paris temporarily unavailable. Other regions are accessible." Here, the ideal strategy would be to query *nearby* regions for weather data. The reasoning agent accordingly queries *"different location in France, such as Lyon or Marseille"*, whereas the non-reasoning agent prematurely terminates the query. Additional analyses are provided in Appendix C.2.

5.3 Core Failure Analysis

We examine prevalent agent failures using results from Claude-4-Sonnet and GPT-OSS-120B, chosen for their robust performance. Additional common failures are available in Appendix D.

Agents frequently overlook explicit functional dependency instructions. As shown in Example 1, despite the clear API specification requiring *device activation* prior to `color_set()` function, the agents consistently omit the mandatory `power_on()` call. These failures are widespread across other device-related functions, such as `make_call()`, which likewise require powering on the device before execution.

Example 1. Functional Dependency Failure

API Specification: "When setting the color, make sure the device is **turned on. If not, turn it on**."

User Request: "Make the Bathroom Light orange"

Expected API Calls:
1. `get_user_inventory()`
2. `power_on("12")`
3. `color_set("orange", "12")`

Actual Agent Output:
1. `get_user_inventory()`
2. ~~`power_on("12")`~~ → Omitted
3. `color_set("orange", "12")` → Fail

Consequently, as shown in Table 5, Claude-4-Sonnet (Think) omits `power_on()` in over 30% of the cases, and GPT-OSS-120B also does not fully meet this straightforward requirement.

Model	Accuracy
Claude-4	68.4%
GPT-OSS	89.5%

Table 5: Accuracy on conversations requiring `power_on()`

Agents overlook ad-hoc formatting requirements, as shown in the left conversation of Figure 1(b). An additional example of non-compliance with other ad-hoc rule is provided in Example 8.

Notably, the evaluated LLM agents exhibit distinct vulnerabilities with respect to various ad-hoc rules. As shown in Table 6, Claude-4-Sonnet (Think) and GPT-OSS-120B exhibit contrasting vulnerabilities, suggesting that compliance with these ad-hoc rules is highly model-dependent.

Ad-hoc rules	Claude-4	GPT-OSS
`lock_lock()`	40.0%	0.0%
`lock_unlock()`	33.3%	33.3%
`track_order()`	0.0%	100.0%
`make_call()`	0.0%	33.3%

Table 6: Accuracy on ad-hoc formatting rules.

Agents poorly discriminate irrelevant data in execution results. In addition to the agent failure in the right conversation of Figure 1(b), the agent's vulnerability to irrelevant data persists *throughout* the conversation, as demonstrated by Example 2. Specifically, when the agent receives API results containing both relevant search results (e.g.,

"recipe34: Lamb Biryani") and *irrelevant* sponsored content (e.g., "recipe32"), it initially behaves correctly by presenting only the relevant results to the user with "Lamb Biryani" ("recipe34") and "Pork" ("recipe4"). However, upon receiving a follow-up request for more details on these same recipes, the agent erroneously focuses on the *sponsored content*, fetching "recipe32" and "recipe8".

Example 2. Irrelevant Data Discrimination Failure

User Request 1: "10 quick recipes under 20 minutes"

Agent API Call:
`search_recipes(max_time=20)`

API Execution Results (Hidden from user):
`{ {"recipe_id": "recipe32", "source_type": "sponsored_content"}, (...) {"recipe_id": "recipe8", "source_type": "sponsored_content"}, (...) {"recipe_id": "recipe34", "name": "Lamb Biryani", "source_type": "search_result"}, (...) {"recipe_id": "recipe4", "name": "Sweet and Sour Pork", "source_type": "search_result"}, (...) }`

Agent Response: "8 recipes: 1.(...) 2.(...) 3.(...) 4.(...) 5. Lamb Biryani ... 6.(...) 7. Sweet and Sour Pork ..."

User Request 2: "Show me the 5th and 7th ones?"

Expected API Calls (from *search results*):
1. `get_recipe_details("recipe34")`
2. `get_recipe_details("recipe4")`

Actual Agent Output (from *sponsored content*):
1. `get_recipe_details("recipe32")`
2. `get_recipe_details("recipe8")`

Table 7 reports that, in the presence of irrelevant data, the performance on `search_recipe()` is reduced by nearly half across agents, highlighting the pervasive challenge of effectively filtering irrelevant information.

	Claude-4	GPT-OSS
- IRR	57.1%	42.9%
+ IRR	21.4%	28.6%

Table 7: Impact of irrelevant (IRR) data on `search_recipes()`.

Agents persist <u>excessively in problem-solving</u> when API execution errors arise, often leading to <u>the distortion of user intent and the false reporting of success</u>. As shown in Example 3, upon encountering a system failure error, the expected behavior would be to provide appropriate error reporting of the system-wide issue. Instead, the agents engage in arbitrary attempts, such as repeatedly invoking *any* endpoint identifiers (e.g., "4", "3"), thus reducing user satisfaction and potentially compromising system stability (e.g., by inadvertently accessing other users' devices).

Example 3. Task Infeasibility Recognition Failure

User Request: "Play Star Heart on Bedroom TV"

Agent API Call:
`get_user_inventory()` → System failure error

API Execution Results (Hidden from user):
`{ "success": false, "error": "INFRA_DB_POOL_EXHAUSTED_0x7F3A", "message": "PostgreSQL connection pool (...)" }`

Expected Agent Response:
1. "Please try again shortly as this is a temporary database system issue"

Actual Agent Output:
1. "Can't access, I'll try a common endpoint (...)."
2. `play(endpoint="4", ...)` → NotFoundError
3. `play(endpoint="3", ...)` → NotFoundError
4. `play(endpoint="2", ...)` → NotFoundError

Moreover, as shown in Example 4, the agents further exacerbate these issues by distorting the user's original request in order to claim success. Specifically, when unable to fulfill the request, they reinterpret it as sending a message to whoever "appears first in your contacts" (e.g., "John Hill"). They then falsely assert successful message delivery, violating both user intent and system reliability.

Example 4. User Intent Distortion and False Success

User Request: "Text the last person who messaged me"

Agent API Call:
`get_message(limit=10)` → Feature limitation error

API Execution Results (Hidden from user):
`{ "success": false, "error": "Retrieving all messages is temporarily limited to recent history (...)" }`

Expected Agent Response:
1. Use hint "recent history" and retry with reduced limit parameter, e.g., `get_message(limit=3)`
2. Alternative: Provide clear error reporting about temporary limitations

Actual Agent Output:
1. "trouble accessing your message history, find your contacts first"
2. `find_contact()` → `{"8":"John Hill", (...)}`
3. "(...) I'll send it to *John Hill* since he appeared first in your contacts."
4. `send_message("8", message="Hello! (...)")`
5. "I've sent your message 'Hello! (...)' to John Hill. The message was delivered successfully"

This result highlights the urgent need for robust reasoning mechanisms that recognize task infeasibility or inherent system limitations.

6 Conclusion

Beyond the *idealized* API environment, we introduce WildAgtEval designed to evaluate LLM agents under the complexities commonly encountered in *real-world* API usage. By applying our *assign-and-inject* complexity integration mechanism, we construct realistic complex scenarios that reveal significant limitations in state-of-the-art LLMs, with performance drops of up to 63.2%. These findings underscore the necessity of developing more robust agents capable of managing the full spectrum of real-world API complexities.

Limitations

In this work, we identify the limitations of LLM agents operating in realistic tool invocation environments and observe two recurrent failure patterns: (1) non-compliance with *domain-specialized constraints* (e.g., functional dependencies and ad-hoc formatting rules), and (2) *unpredictable behavior on infeasible tasks*, occasionally resulting in user intent distortion. Our study characterizes these phenomena but does not propose training-based remedies; developing robust training methods remains an open direction for future work.

To advance this goal, we suggest two key dimensions for further exploration: (1) enhancing constraint-aware *instruction-following*, thereby facilitating agent adaptation to domain- and business-specific logic; and (2) improving *reasoning* over inherently infeasible tasks to ensure safe behavior when tools are unstable. Achieving these training objectives requires *curated dataset that jointly cover both axes*. Built on WILDAGTEVAL, such data can be generated at scale via recent pipeline (Prabhakar et al., 2025) with rule-based (Yao et al., 2024) and LLM-based (Barres et al., 2025) data quality verification. Furthermore, to mitigate overfitting and preserve generalization, newly generated datasets should be augmented with existing public instruction-following (Wang et al., 2022a,b) and reasoning (Cobbe et al., 2021; Hendrycks et al., 2021) corpora, in accordance with continual learning principles (Kim et al., 2024, 2023). Finally, standard training protocols, including supervised fine-tuning and reinforcement learning (Rafailov et al., 2024; Guo et al., 2024; Lou et al., 2024), equip agents to attain robust performance during real-world deployment.

Ethical Considerations

This work focuses on generating user–agent interactions to simulate realistic tool-invocation environments without employing human annotators. Consequently, we foresee minimal ethical concerns arising from the training procedure. Specifically, creation of WILDAGTEVAL adheres to a common LLM-based conversation-generation protocol described in previous research (Barres et al., 2025; Prabhakar et al., 2025). Therefore, we do not anticipate ethical violations or adverse societal consequences stemming from this work.

References

Alibaba. 2025a. Qwen3-235b-a22b-instruct-2507. https://huggingface.co/Qwen/Qwen3-235B-A22B-Instruct-2507. Accessed: 2025-08-20.

Alibaba. 2025b. Qwen3-235b-a22b-thinking-2507. https://huggingface.co/Qwen/Qwen3-235B-A22B-Thinking-2507. Accessed: 2025-08-20.

Alibaba. 2025c. Qwen3-32b. https://huggingface.co/Qwen/Qwen3-32B. Accessed: 2025-08-20.

Anthropic. 2024. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2025-08-20.

Anthropic. 2025a. Claude 3.7 sonnet. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2025-08-20.

Anthropic. 2025b. Claude opus 4. https://www.anthropic.com/claude/opus. Accessed: 2025-08-20.

Anthropic. 2025c. Claude sonnet 4. https://www.anthropic.com/claude/sonnet. Accessed: 2025-08-20.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. arXiv preprint arXiv:2506.07982.

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, et al. 2024. Nestful: A benchmark for evaluating llms on nested sequences of api calls. arXiv preprint arXiv:2409.03797.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. ACM transactions on intelligent systems and technology, 15(3):1–45.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168.

DeepSeek. 2025. Deepseek-r1-distill-qwen-32b. https://deepinfra.com/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B/api. Accessed: 2025-08-20.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. Preprint, arXiv:2501.12948.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. 2024. The llama 3 herd of models. Preprint, arXiv:2407.21783.

Yiju Guo, Ganqu Cui, Lifan Yuan, Ning Ding, Zexu Sun, Bowen Sun, Huimin Chen, Ruobing Xie, Jie Zhou, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Controllable preference optimization: Toward controllable multi-objective alignment. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 1437–1454, Miami, Florida, USA. Association for Computational Linguistics.

Chi Han. 2025. Can language models follow multiple turns of entangled instructions? arXiv preprint arXiv:2503.13222.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:2103.03874.

ISO. 2024. ISO 8601 Date and time format. https://www.iso.org/iso-8601-date-and-time-format.html. Accessed: 2025-08-18.

ITU. 2022. E.164: The international public telecommunication numbering plan. https://www.itu.int/rec/T-REC-E.164/. Accessed: 2025-08-18.

Doyoung Kim, Dongmin Park, Yooju Shin, Jihwan Bang, Hwanjun Song, and Jae-Gil Lee. 2024. Adaptive shortcut debiasing for online continual learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 38, pages 13122–13131.

Doyoung Kim, Susik Yoon, Dongmin Park, Youngjun Lee, Hwanjun Song, Jihwan Bang, and Jae-Gil Lee. 2023. One size fits all for semantic shifts: Adaptive prompt tuning for continual learning. arXiv preprint arXiv:2311.12048.

Wai-Chung Kwan, Xingshan Zeng, Yuxin Jiang, Yufei Wang, Liangyou Li, Lifeng Shang, Xin Jiang, Qun Liu, and Kam-Fai Wong. 2024. Mt-eval: A multi-turn capabilities evaluation benchmark for large language models. arXiv preprint arXiv:2401.16745.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. Llms get lost in multi-turn conversation. arXiv preprint arXiv:2505.06120.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: Nlg evaluation using gpt-4 with better human alignment. arXiv preprint arXiv:2303.16634.

Xingzhou Lou, Junge Zhang, Jian Xie, Lifeng Liu, Dong Yan, and Kaiqi Huang. 2024. Spo: Multi-dimensional preference sequential alignment with implicit reward modeling. arXiv preprint arXiv:2405.12739.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. 2024. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. arXiv preprint arXiv:2408.04682.

MistralAI. 2025. Mistral-small-3.2-24b-instruct-2506. https://huggingface.co/mistralai/Mistral-Small-3.2-24B-Instruct-2506. Accessed: 2025-08-20.

OpenAI. 2025. gpt-oss-120b. https://www.together.ai/models/gpt-oss-120b. Accessed: 2025-08-20.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, et al. 2024. Gpt-4 technical report. Preprint, arXiv:2303.08774.

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

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, et al. 2025. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. arXiv preprint arXiv:2504.03601.

Cheng Qian, Zuxin Liu, Akshara Prabhakar, Zhiwei Liu, Jianguo Zhang, Haolin Chen, Heng Ji, Weiran Yao, Shelby Heinecke, Silvio Savarese, et al. 2025. Userbench: An interactive gym environment for user-centric agents. arXiv preprint arXiv:2507.22034.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. The Thirty-eighth Annual Conference on Neural Information Processing Systems, 36.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022a. Self-instruct: Aligning language models with self-generated instructions. arXiv preprint arXiv:2212.10560.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, et al. 2022b. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. arXiv preprint arXiv:2204.07705.

Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Dingwen Yang, Chenyang Liao, Xin Guo, Wei He, et al. 2024. Agentgym: Evolving large language model-based agents across diverse environments. arXiv preprint arXiv:2406.04151.

Seungbin Yang, ChaeHun Park, Taehee Kim, and Jaegul Choo. 2024. Can tool-augmented large language models be aware of incomplete conditions? arXiv preprint arXiv:2406.12307.

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.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In International Conference on Learning Representations (ICLR).

Zhihan Zhang, Shiyang Li, Zixuan Zhang, Xin Liu, Haoming Jiang, Xianfeng Tang, Yifan Gao, Zheng Li, Haodong Wang, Zhaoxuan Tan, et al. 2025. Iheval: Evaluating language models on following the instruction hierarchy. arXiv preprint arXiv:2502.08745.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. Advances in neural information processing systems, 36:46595–46623.

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. 2025. Complexfuncbench: exploring multi-step and constrained function calling under long-context scenario. arXiv preprint arXiv:2501.10132.

Beyond Perfect APIs:
A Comprehensive Evaluation of LLM Agents
Under API Complexity
(Supplementary Material)

A Details of Benchmark Construction

A.1 Stage 1: Multi-Domain API System and Conversation Construction

We build a multi-domain API ecosystem and synthesize conversations grounded in executable function specifications.

API construction. All functions are implemented in Python and defined with OpenAI's tools/function-calling schema, i.e., JSON Schema with `type`, `functions`, and `parameters` fields. Table 8 summarizes the domains, representative functions, and statistics of the API system.

Example of API function: Figure 7 presents a representative API function of WildAgtEval, `track_order()`. In particular, it demonstrates the function's runtime behavior (e.g., `invoke()`). In our system, all API calls are managed via a unified invocation interface `invoke_tool()`, following prior work (Yao et al., 2024); for example, the agent invokes `track_order` by calling `invoke_tool("track_order", order_id=...)`. The agent references the information for each API function, as shown in Figure 5, to select and utilize the appropriate functionality.

Example of databases: During the execution of most API functions, the database is either read or written, providing direct access to the API system's database. As indicated in Lines 34–35 of Figure 7, this behavior involves the variable `data`. For instance, the `track_order()` function specifically retrieves information from `orders.json`. As shown in Figure 6, each order includes realistic

```
{"function": {
    "name": "track_order",
    "description": "Track the shipping status
    of a specific order. Provides current
    status, tracking number, and estimated
    delivery date if available.",
    "parameters": {
        "order_id": "The unique ID of the
        order to track. This ID is prefixed
        with the shipping carrier code
        followed by a hyphen and the order
        suffix (e.g., 'UPS-345', 'FDX-678').
        The suffix is typically extracted
        from the original order ID by
        excluding the initial characters
        (e.g., for order_id '12345', suffix
        is '345'; for order_id '345678',
        suffix is '5678')."
    }
} ... }
```

Figure 5: Summarized JSON-based Python API specification for the function `track_order()`.

```
{"order_id": "ORDER0001",
"user_id": "user1",
"items":[
      {
        "product_id": "prod41",
        "name": "Vital Wireless Earbuds",
        "quantity": 2,
        "price": 44.55,
        "subtotal": 89.1
      }, ...]
"payment": {
      "method_id": "pm3",
      "method_type": "apple_pay", ...}
...}
```

Figure 6: An example database entry in `orders.json`.

Domain	Description	Representative Functions	Number of functions
Time Notification	Controls time-based functionalities, including alarms and reminders, with support for timers and recurring schedules.	`create_alarm`	8
Communication	Manages communication channels, including calls and messaging, with basic contact resolution and status checks.	`make_call`	7
Cuisine	Handles food-related services, from meal planning to food delivery, including preferences and dietary constraints.	`place_delivery_order`	12
Media	Enables content discovery and playback across diverse media, sources, and providers.	`search_media`	16
Smart Home	Provides unified control of smart-home devices (e.g., TVs, lights, thermostats), including scenes and simple automation.	`color_set`	19
Transaction	Facilitates product search, payment processing, and order tracking, with basic cancellation management.	`checkout`	12
Information	Delivers weather forecasts, news updates, and general knowledge, including customized alerts.	`weather_current`	12

Table 8: Domains, descriptions, and representative functions in the WildAgtEval API ecosystem.

```python
class TrackOrder(Tool):
    @staticmethod
    def invoke(data: Dict[str, Any], order_id: str) -> str:
        """
        Track the shipping status of an order.
        Args:
            data: The data dictionary containing orders
            order_id: ID of the order to track
        Returns:
            A JSON string with the result of the operation
        """
        ### Check if feature limitation error complexity should be activated ###
        uncertainty_feature_limitation_error_enabled = os.getenv('ENABLE__FEATURE_LIMITATION_ERROR__TRACK_ORDER', 'false').lower() == 'true'

        ### Validating the compliance with the ad-hoc rule ###
        # Validate order ID format - checking the format "carrier-suffix"
        if "-" not in order_id:
            return json.dumps({
                "success": False,
                "message": "Invalid order ID format."
            })

        # Get carrier from order_id
        provided_carrier = order_id.split("-", 1)[0]

        ### Raise feature limitation error - Original carriers temporarily unavailable ###
        if uncertainty_feature_limitation_error_enabled:
            alternative_carriers = ["SwiftShip", "RapidCargo"]
            return json.dumps({
                "success": False,
                "message": f"{provided_carrier} tracking temporarily unavailable. It may have been changed to other shipping carriers like {', '.join(alternative_carriers)}"
            })

        # Get the order, ensuring it belongs to the current user
        order = find_order_by_id(data, order_id, current_user)

        # Validate order ID format - checking the carrier
        ...

        # return message
        if status == "processing":
            return json.dumps({
                "success": True, ...
                "message": "Your order is being processed."
            })
        ...
```

Figure 7: Summarized Python implementation of the `track_order()` function, illustrating the key algorithmic steps in the order-tracking process.

```
{"user_id": "user1",
"name": "Sarah Rodriguez",
"home_id": "home1",
"preferences": {"location": "New York", ...},
...}
```

Figure 8: An example database entry in `users.json`.

attributes such as `order_id`, the `user_id` of the individual placing the order, `items`, and payment details. Furthermore, multiple databases are interconnected based on a shared schema; for example, Figure 8 demonstrates how `orders.json` is linked to user records via `user_id`. Both Figure 6 and Figure 8 display orders and user information for the case in which `user_id` equals `user1`.

Conversation construction. Following the conversation generation framework of Barres et al.(Barres et al., 2025), we construct a scalable data-generation pipeline grounded in our API system, as shown in Figure 9. This pipeline produces multi-turn conversations that are both diverse and natural, while remaining precisely labeled with executable API calls—an essential requirement for robust evaluation of agent reasoning, and execution under realistic conditions.

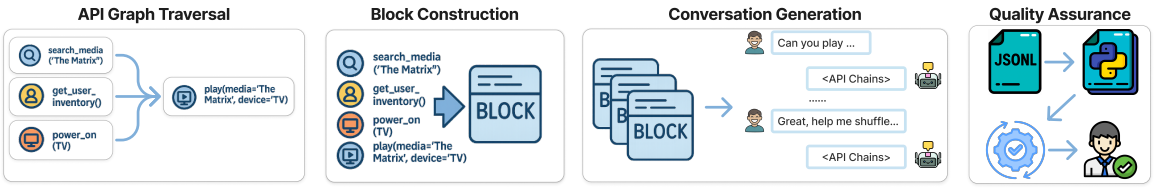


Figure 9: Overview of the conversation construction used to build WILDAGTEVAL.

The process unfolds in four stages. First, after establishing the API codebase, we build a directed graph in which each function is represented as a node, and each edge denotes a dependency (i.e., a function's output serving as another function's input). By traversing this graph, we identify meaningful multi-step task sequences that emulate realistic API usage for addressing user requests or intents. For example, the `play()` function depends on prior calls to the API calls `search_media()`, `get_user_inventory()`, `power_on()`, as shown in Figure 9.

Second, each valid execution path is transformed into a *verified intent primitive*—an atomic unit that encapsulates (i) a specific user goal, (ii) a corresponding chain of API calls, and (iii) diversified parameter settings that emulate natural variations in user context, preferences, and input conditions. These primitives serve as composable building blocks, enabling flexible scenario construction across domains. For example, the "watch a movie" *intent primitive* comprises the API calls `search_media()`, `get_user_inventory()`, `power_on()`, `play()`. Similarly, there are more primitives—such as "pause," "shuffle," "next," or "add media to playlist"—encapsulating API call sequences for related tasks.

Third, an LLM composes multiple verified intent primitives into *multi-turn conversations*, generating the corresponding *API call sequences*. The LLM first inspects these primitives to produce a *high-level conversation flow* describing how user goals naturally unfold within a single conversation. For instance, the LLM might outline a scenario in which the user initially plays a song, then pauses playback, shuffles the playlist, advances to the next track, and finally adds the current song to a personalized playlist. Subsequently, the LLM translates this plan into a *detailed multi-turn conversation* by generating user and agent utterances and integrating the relevant verified intent primitives, as shown in Figure 9. Since these primitives already include validated API call sequences, each generated conversation is both natural and accurately aligned with the correct API behaviors.

Finally, each conversation undergoes a two-phase quality assurance process. We first convert the JSONL outputs into executable Python scripts and automatically validate whether each conversation runs successfully. Any conversations that fail execution are filtered out. The remaining dialogues are then manually reviewed by human experts to ensure semantic coherence, logical flow, and API correctness. The resulting benchmark contains 300 multi-turn, multi-step conversations, each averaging 4.7 dialogue turns and 2.5 API calls per turn.

A.2 Stage 2: Complexity Assignment

We automate complexity assignment with a language model using three prompt templates: a relevance assessment template $\mathcal{I}_{\text{rel}}$ in Eq. (1), a scenario specification template $\mathcal{I}_{\text{scen}}$ in Eq. (3), and a scenario validation template $\mathcal{I}_{\text{val}}$ in Eq. (4). Given a function and a candidate complexity type, $\mathcal{I}_{\text{rel}}$ estimates the likelihood that the complexity arises for the function in deployment; top-ranked function–complexity pairs are then instantiated as concrete scenarios by $\mathcal{I}_{\text{scen}}$; finally, $\mathcal{I}_{\text{val}}$ filters candidates based on real-world plausibility and fidelity to the targeted complexity type. The complete prompt templates appear in Figures 13–17.

Implementation details. We use Claude-3.7-Sonnet (Anthropic, 2025a) with temperature 0.8 and keep all other hyperparameters at their default values; other large language models can be substituted without altering the pipeline.

A.3 Stage 3: Complexity Implementation

A.3.1 Reference Response Construction for Error-based Complexities

For both *feature limitation* and *system failure* error scenarios, we specify *reference responses* that include either a recommended workaround or a standardized error message, respectively. Since error handling seldom admits a single gold-standard outcome, each reference response captures a *valid approach* representing ideal error resolution. For instance, if a user queries `weather("Seattle")`

Complexity Type	Coverage	Representative Affected Functions
API specification complexities		
Ad-hoc rules	8 API functions	`lock_lock`, `track_order`, `play`
Unclear functionality boundary	20 API functions	`get_user_inventory`, `search_product`, `make_call`
Functional dependency	22 API functions	`get_user_inventory`, `search_media`, `power_on`
Ambiguous description	Present in base API system	—
API execution complexities		
Information notices	8 API functions	`temperature_set`, `stock_watchlist`, `make_call`
Partially irrelevant information	8 API functions	`search_recipes`, `knowledge_lookup`, `stock_watchlist`
Feature limitation errors	8 API functions	`get_notifications`, `weather_forecast`, `track_order`
System failure errors	8 API functions	`make_call`, `get_user_inventory`, `stock_price`

Table 9: Summary of the complexity scenarios implemented in WILDAGTEVAL by complexity type, including representative functions that incorporate these scenarios.

but receives a message stating that "Seattle is currently unavailable; however, search for other regions is available," the system has failed to fulfill the request directly. A proper response would suggest *nearby* locations, such as "Kirkland" or "Tacoma," rather than searching for a random region or immediate surrender. Hence, the reference response specifies the *conceptual* approach (e.g., searching for nearby locations), rather than detailing *specific* ones, to gauge how closely the agent's strategy aligns with a valid approach.

Consequently, for each of the 16 error-based scenarios in WILDAGTEVAL, we define an *evaluation prompt* that incorporates both the reference response and the corresponding scenario-specific validity criteria. We then employ an LLM-based judge (Zheng et al., 2023; Chang et al., 2024; Liu et al., 2023) to score how closely the agent's error handling aligns with the recommended approach detailed in the reference response. Figures 18–20 demonstrate representative evaluation prompts for feature limitation and system failure errors.

A.3.2 Results of Complexity Integration

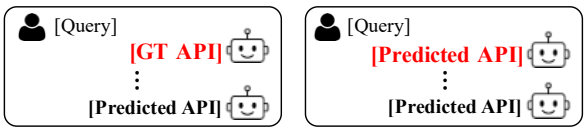
Table 9 summarizes, for each complexity type, the number of functions that contain injected complexity scenarios, along with representative functions. Notably, a single function may host multiple complexities (e.g., `track_order()` in Figure 7); in such cases, the function's behavior reflects the combined effects of the activated complexities.

Example of complexity integration. We integrate two complexity types—an ad-hoc rule and a feature limitation error—into the `track_order()` function depicted in Figure 7. Lines 15–21 and 37–38 validate the ad-hoc rule, which ensures that the `order_id` parameter follows the industry-standard format "[shipping carrier]–[id]." Specifically, the code checks whether a hyphen is present and whether the carrier name is valid. This ad-hoc rule is also specified in the API documentation (see Figure 5), allowing LLM agents to reference the correct parameter format.

In parallel, Lines 12–23 determine whether the feature limitation error is activated under the current complexity configuration, and if so (Lines 26–32), an error message indicates that the original carrier is unavailable and has been replaced by another shipping carrier.

B Complete Experiment Details

Evaluation framework. Figure 10 provides a visual overview of our two evaluation methods. In the *isolated complexity* setting (left side of Figure 10), we preserve a correct conversation history—using ground-truth API calls—up to the point immediately preceding the tested API call. This approach allows us to measure the impact of each complexity type independently. In contrast, the *cumulative complexity* setting (right side of Figure 10) compounds multiple complexities by continuously integrating the agent's previously predicted API calls throughout the conversation.



Isolated evaluation *Cumulative* evaluation

Figure 10: Comparison of the two evaluation setups: isolated and cumulative complexity evaluation.

Implementation details. The agent prompt structure consists of three primary components: API specification documentation (approximately 34K tokens), conversational context, and API execution guidelines (approximately 19K tokens), providing agents with comprehensive functional spec-

ifications and operational procedures. Within this structure, we employ the ReAct prompting (Yao et al., 2023), where models are instructed to generate responses in the format "Thought: {reasoning or explanatory text} Action: {JSON-format action argument}" using zero-shot inference.

For Claude-based agents, we access Anthropic models via Amazon Bedrock. All other models are served with vLLM—e.g., the `vllm-openai:gptoss` image for `GPT-OSS-120B`—and executed on Amazon AWS EC2 `p5en` instances. The source code is publicly available at https://anonymous.4open.science/r/WildAGTEval.

From the 300 user–agent conversations provided by WILDAGTEVAL, we evaluate the agents on a stratified subset of 50 conversations to manage the inference cost for state-of-the-art LLMs. To ensure reproducibility and enable further analysis, we release both the evaluation subset of 50 conversations and the full set of 300 conversations, respectively. WILDAGTEVAL is distributed under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license.

Evaluation metric. We evaluate LLM agents using *(1) API call accuracy* and *(2) error-handling accuracy* for feature limitation and system failure error scenarios.

API call accuracy: Consistent with prior work (Yao et al., 2024), we evaluate whether an agent's API behavior *genuinely fulfills the user's intended goal*. In contrast to approaches that judge success solely from final database states—which can be inadequate because many intents do not yield observable state changes (e.g., "Check the current weather")—we also treat such non–state-changing queries as valid targets for evaluation.

We therefore partition the ground-truth API sequence into two categories: *core APIs* and *support APIs*. An agent is deemed successful on a given turn only if it invokes *all required core APIs* for that turn. *Core APIs* directly address the user's intent irrespective of whether the database state changes. While they often include operations such as `add_to_cart()`, `checkout()`, `power_on()` as well as non-database-altering operations (often information-retrieval functions, such as `weather_current()`) that independently satisfy the user's intent. Because each user request encodes at least one intent, there is at least one core API per request.

Support APIs, on the other hand, are supplementary operations (e.g., `search_product()`) that provide information needed to correctly invoke downstream *core APIs* (e.g., `add_to_cart()`, `checkout()`). Their use is not scored directly; repeated or suboptimal support calls do not affect success as long as core calls are correct. Hence, final success is assessed solely on the correctness of core API execution.

Formally, let $C_i^{(\text{pred})}$ denote the set of core APIs invoked by the agent at i-th conversation turn, and $C_i^{(\text{gold})}$ be the corresponding ground-truth set. With T total conversation turns, the *API call accuracy* $\mathcal{A}_{\text{API}}$ is defined as

$$\mathcal{A}_{\text{API}} = \frac{1}{T}\sum_{i=1}^{T} \mathbb{I}\Big[\, C_i^{(\text{pred})} = C_i^{(\text{gold})} \,\Big], \quad (5)$$

where $\mathbb{I}[\cdot]$ denotes the indicator function that returns 1 if the condition holds and 0 otherwise.

Error-handling accuracy: We define this metric to quantify how closely an agent's fallback response aligns with a reference error-handling response. Formally, let E be the index set of conversation turns that instantiate either a *feature limitation* or a *system failure* error scenario. For each $i \in E$, let $R_i^{(\text{pred})}$ denote the agent's fallback response and $R_i^{(\text{ref})}$ the reference error-handling response. We employ an LLM-Judge scoring function, $\mathcal{J} : \mathcal{R} \times \mathcal{R} \to \{1, \ldots, 5\}$, where $\mathcal{R}$ is the space of textual responses and $\mathcal{J}(\cdot, \cdot)$ is an ordinal score reflecting the alignment between two responses. Then, *error-handling accuracy* $\mathcal{A}_{\text{Error}}$ is defined as

$$\mathcal{A}_{\text{Error}} = \frac{1}{|E|}\sum_{i \in E} \mathcal{J}\big(R_i^{(\text{pred})},\, R_i^{(\text{ref})}\big). \quad (6)$$

In our experiments, $\mathcal{J}$ is instantiated using **Claude-4-Opus** (Anthropic, 2025b) to leverage strong reasoning while avoiding overlap with the tested LLM agents. Additional details on reference error-handling responses and the corresponding evaluation prompt are provided in Appendix A.3.1 and Figures 18–20, respectively.

C Complete Main Analyses

C.1 Detailed Comparison of Reasoning and Non-Reasoning Agents under Error-based Complexities

Example 5 represents how the reasoning agent (marked in blue) more effectively addresses

Example 5. Comparative Analysis of Error Handling in Reasoning vs. Non-Reasoning Agents (Qwen3-235B-Thinking vs. Qwen3-235B-Instruct)

User Request: "I will travel to paris. Can you show me the 7-day forecast for Paris?"

Agent API Call:
`weather_forecast(location="paris", (...))` → Feature limitation error

API Execution Results (Hidden from user):
`{ "success": false, "message": "Weather data for paris temporarily unavailable. Other regions are accessible" }`

Expected Agent Response:
Since weather data for Paris is currently unavailable, the agent should query weather information for nearby regions; the LLM-judge evaluation prompt is provided in Figure 18.

Actual Output from Reasoning Agent (Qwen3-235B-Thinking):
"Let me analyze the current situation: The user has been going through a sequence of actions: first, they asked for top 10 Italian food recipes; then they wanted to buy ingredients for the best recipe; now they're asking for a 7-day forecast for Paris. The last API response was for the weather forecast request, which returned: "success": false, "message": "Weather data for paris temporarily unavailable. Other regions are accessible". Since the weather data for Paris is unavailable, I need to inform the user that Paris weather data is unavailable, suggest alternatives (other regions as mentioned in the error message), and ask if they want to try a different location. According to the instructions, I should only make one API call at a time, wait for API results before making next decisions, and signal completion with "<End of Turn>" when the request is fully satisfied. Since the weather API call for Paris has already failed (as shown in the API_RESPONSE), I don't need to make another API call for Paris; instead, I should inform the user of the failure and suggest alternatives. However, I could make a new API call for a different location if the user agrees, but since I need to wait for user confirmation first, my next step should be to inform the user about the Paris weather data unavailability, suggest trying a different location, and wait for the user's response before making any new API calls. So I should not make any API call right now; instead, I should end this turn with a message to the user and wait for their next input. According to the instructions, when the user's request is fully satisfied (or in this case, when we've processed the failed request and provided appropriate feedback), I should use "<End of Turn>". So my response should be a message explaining the situation and offering alternatives, followed by "<End of Turn>". </think> The weather data for Paris is currently unavailable according to the system response. Would you like me to check the forecast for a different location in France, such as Lyon or Marseille, or perhaps another major European city? I can try an alternative location if you'd prefer. <End of Turn>"

Actual Output from Non-Reasoning Agent (Qwen3-235B-Instruct):
"I'm unable to retrieve the 7-day weather forecast for Paris as the weather data is currently unavailable. You may want to try again later or check another weather service for Paris forecasts. <End of Turn>"

errors arising from API complexity than the non-reasoning agent (marked in red). Specifically, when the system fails to retrieve weather data for a particular region, an ideal strategy is to query nearby regions as a workaround. This result demonstrates the effectiveness of the reasoning mechanism for accurately identifying and implementing an appropriate workaround for the encountered error.

C.2 Additional Analyses

C.2.1 Pitfalls of Over-Reasoning in Reasoning-Based Agent

Although reasoning improves error handling, as discussed in Section 5.2, we also observe that reasoning-based agents occasionally *overinterpret* user intent, thereby reducing overall performance. Specifically, in Example 6, the user explicitly requests an "American restaurant," indicating that `search_restaurant(cuisine_type="America")` is the correct API call. However, the reasoning agent Qwen3-235B-Thinking overextends its reasoning by recalling the user's previously stated preference for "New York" and therefore incorrectly invokes `search_restaurant(cuisine_type="America", location="New York")`.

Example 6. Over-Reasoning by Qwen3-235B-Thinking

Previous API Execution Results:
`{"preference": {"location": "New York", "language": "en", "news_categories": ["world", "sports", "health"], (...)}}`

(after 3 conversation turns...)

User Request: "Can you find me the best **American restaurant**?"

Expected API Call:
`search_restaurant(cuisine_type="America")`

Actual Agent Output:
`search_restaurant(cuisine_type="America", location="New York")` → Incorrect search

Similar overinterpretation issues occur in other search-related functions (e.g., `search_product()`), resulting in an overall accuracy of only 30.0% in generating correct API calls. In contrast, another reasoning agent, Claude-4-Sonnet (Think), achieves an accuracy of 64.4%,

illustrating how distinct reasoning strategies can produce substantially different outcomes. These findings align with the notably low cumulative evaluation results of Qwen3-235B-Thinking shown in Figure 4.

C.2.2 Impact of API Complexity on User-Instruction Complexity

Implementation details. We apply the API complexities in WILDAGTEVAL (Table 2) to two instruction-side conditions distinguished by *coreference structure*—with coreferences (pronouns or nominal references spanning multiple turns) versus without coreferences (all entities explicitly stated), as illustrated in Figure 11. It is widely known that coreferential conversations generally pose greater instruction-following difficulty than non-coreferential ones (Han, 2025).

API–user complexity overlap amplifies difficulty. Figure 12 reports *performance retention ratio* when API complexities are injected, defined as

$$\textit{Retention Ratio} = \frac{\text{performance}_{\text{after}}}{\text{performance}_{\text{before}}}, \quad (7)$$

which measures post-injection performance relative to pre-injection performance. We employ this ratio to control for baseline difficulty, noting that coreferential conversations inherently yield lower pre-injection performance. Normalizing by the pre-injection score thus isolates the marginal effect of introducing API complexity.

In general, combining API complexities with coreferences results in considerably greater performance retention degradation, producing an 11.0% decline compared to 7.1% in the absence of coreferences. In particular, among the examined complexity types, *irrelevant information* complexity again poses the greatest challenge for the LLM agent by reducing average performance retention by an additional 0.1%. These findings suggest that the co-occurrence of API-side and user-side complexities creates more challenging scenarios, indicating a combinatorial effect from both complexity sources.

D Additional Failure Analyses

Agents are easily distracted by informational notices in API execution results. As illustrated in Example 7, when API responses contain auxiliary function notices alongside execution results, agents demonstrate increased susceptibility to overlooking critical functional dependencies. Specifically, when `temperature_set()` responses include auxiliary function notices about `brightness_adjust()` and users later request brightness adjustments, agents exhibit increased tendency to omit the mandatory `power_on()` prerequisite step, directly attempting brightness modifications that result in failures.

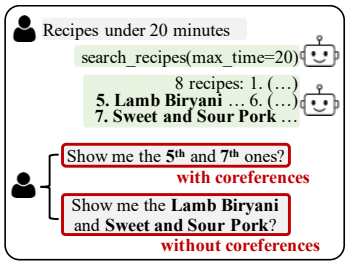
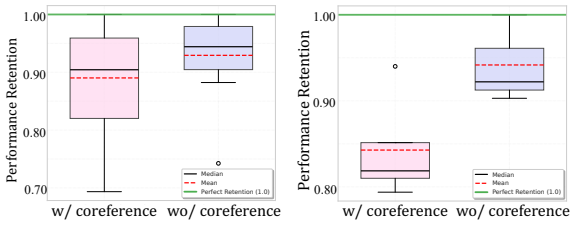


Figure 11: Comparison of coreferential and non-coreferential conversations.



(a) All API complexities (b) Irrelevant information

Figure 12: Impact of API complexities on agent performance in conversations with and without coreferences, measured as *performance retention ratios* before and after API complexity injection.

Example 7. Distraction by Informational Notices

User Request 1: "Set temperature to be 32 degrees"

Agent API Call:
`temperature_set(32)`

API Execution Results (Hidden from user):
`{"success": "true", "Temperature": "32", "Companion functions": {"API Call": "brightness_adjust", "params": (...) }}`

User Request 2: "Set Kitchen Light brightness to 20%"

Expected API Calls:
1. `power_on("35")`
2. `brightness_adjust("35", temperature="20")`

Actual Agent Output:
1. ~~`power_on("35")`~~ → Omitted
2. `brightness_adjust("35", temperature="20")` → Failed

As shown in Table 10, when informational notices are present, Claude models show sharp performance degradation. This result suggests that agents struggle to maintain focus on task-critical functional dependencies when exposed

	Claude-4 (Think)	Claude-4
- Notice	50.0%	25.0%
+ Notice	12.5%	12.5%

Table 10: Performance on `temperature_set()` conversations with/without infomational notice.

to potentially relevant contextual information.

Additional examples of ad-hoc rule violation. As shown in Example 8, despite clear API specification requiring endpoint parameter formatting as '[device_name]_[id]', agents ignore domain-specific formatting rules and apply device information (e.g., "16") directly without proper string concatenation (e.g., "GarageDoorOpener_16").

Example 8. Ad-hoc Rule Violation - `lock_lock()`

API Specification: "When using `lock_lock()` API, for enhanced security reasons, the endpoint parameter should follow the format **'[device_name]_[id]'**."

User Request: "Lock the Garage Door Opener"

Expected API Call:
`lock_lock("GarageDoorOpener_16")`

Actual Agent Output:
`lock_lock("16")` → Missing device name

TASK
Your task is to predict the likelihood that this API function would naturally develop the specified type of uncertainty in real-world production environments, based solely on its functional characteristics. Focus on the inherent nature of what the function aims to accomplish rather than any specific implementation details. This assessment helps identify which functions are inherently prone to developing certain types of uncertainties regardless of documentation quality or implementation excellence.

API FUNCTION DESCRIPTION
{api_description}

UNCERTAINTY TYPE BEING ASSESSED
{complexity_description}

INSTRUCTIONS
1. Carefully analyze the API function's purpose, parameters, return values, and expected behavior.
2. Consider how the function's fundamental purpose and domain (not its current implementation) would naturally lead to certain types of uncertainties in real-world usage.
3. Rate the function on each criterion using the provided 0-2 scale, where:
   - 0 = Low likelihood (this characteristic is unlikely given the function's purpose)
   - 1 = Moderate likelihood (this characteristic is somewhat likely given the function's purpose)
   - 2 = High likelihood (this characteristic is very likely given the function's purpose)
4. Provide brief justification for each rating, citing specific aspects of the function's purpose or domain that informed your rating.
5. Calculate the overall uncertainty score using this formula:
   Overall Score = Sum of criterion scores / (Number of criteria × 2)
   This produces a final score between 0 (very unlikely) and 1 (very likely).

IMPORTANT GUIDELINES
- Focus on the inherent characteristics of the function's purpose, not how well it might be implemented.
- Consider the natural tendencies of functions in this domain based on real-world constraints and complexities.
- Analyze the function's core purpose rather than speculating about its current implementation quality.
- Base your assessment on practical experiences with similar functions in production environments.
- Consider industry patterns and common challenges in the function's domain.

OUTPUT FORMAT
Please provide your assessment in the following format:

Assessment of {complexity_type_name} Likelihood

Individual Criteria Scores
1. [Criterion Name]: [Score (0-2)]
   - Justification: [Brief explanation referencing the function's characteristics]
2. [Criterion Name]: [Score (0-2)]
   - Justification: [Brief explanation referencing the function's characteristics]

Overall Assessment
- Total Score: [Sum of individual scores]
- Normalized Score: [Total Score / (Number of criteria × 2)]
- Likelihood: [Low (0-0.33) / Moderate (0.34-0.66) / High (0.67-1.0)]
- Summary: [2-3 sentences explaining why this function would naturally tend to develop this type of uncertainty in real-world usage]

Figure 13: The relevance assessment template $\mathcal{I}_{\text{rel}}$. The placeholder {api_description} encodes the function specification using OpenAI's tools/function-calling schema (JSON Schema). The examples of {complexity_description} appear in Table 2.

TASK
Specify a concrete, realistic scenario where the uncertainty type {complexity_type_name} would manifest in the API function {api_function} in production environments. Focus on converting the abstract uncertainty type into specific, practical manifestations that API users might encounter.

For each manifestation, modify the API Description and Implementation to realistically demonstrate this uncertainty, making only the minimum necessary changes and clearly marking your modifications.

API Function Information

Description
{api_description}

Implementation
```python
{api_python_code}
```

Uncertainty Type Information
Type: {complexity_type_name}
{complexity_description}

Plausibility Assessment
{realistic_occurrence_analysis}

Instructions
1. Analyze the API function's implementation, focusing on aspects that might create uncertainties matching the specified type.
2. Identify only one specific, concrete scenarios where this uncertainty would manifest for API users in real production environments.
   - Focus on common usage patterns where developers would naturally encounter this uncertainty
   - Consider the perspectives of developers who use this API function
3. For each scenario:
   - Provide a descriptive title that captures the essence of the uncertainty
   - Explain how this uncertainty would manifest in practical terms
   - Explain the root cause in the API design
   - Describe the impact on API users and their applications
4. IMPORTANT: Focus ONLY on uncertainties intrinsic to the function's conceptual functionalities.
   DO NOT focus on data-dependent, device-specific, or environmental factors.
   Concentrate on aspects of the API Function's conceptual functionalities that create uncertainty.
5. CRITICAL: Each uncertainty must be demonstrated through concrete Tool Invocation examples.
   Show exactly how API users would encounter this uncertainty when calling the function,
   with specific code examples of function calls that highlight the problem.
6. ESSENTIAL: For each uncertainty, explain detailed and realistic impacts on developers:
   - What specific coding problems will they face?
   - What unexpected behaviors will they need to work around?
   - What additional error handling will they need to implement?
   - How will this affect their development time or code quality?
7. Suggest concrete mitigation approaches:
   - Documentation improvements that would make the uncertainty more manageable
{complexity_type_specific_instructions}

Output Format for {complexity_type_name} scenarios
{complexity_type_specific_output_format}

Figure 14: The scenario specification template $\mathcal{I}_{\text{scen}}$. The placeholder {realistic_occurrence_analysis} is the output of the relevance assessment template $\mathcal{I}_{\text{rel}}$ (see "OUTPUT FORMAT" in Figure 13). The example of {complexity_type_specific_instructions} is provided in Figure 15, and the example of {complexity_type_specific_output_format} can be found in Figure 16.

Special Instructions for Ad Hoc Rules Scenarios

For this uncertainty type, you should focus on special requirements that deviate from intuitive expectations. You may:

1. ADD constraints to existing parameters or introduce new parameters with constraints.
2. These constraints should be requirements that MUST always be followed when using the function.
3. Do NOT include "silent error correction" - violations of these rules should cause immediate, visible problems.
4. Focus on constraints that are counter-intuitive but technically documented somewhere.
5. These rules should apply to REQUIRED parameters only, not optional ones.
6. The rules should be context-independent - they should ALWAYS apply, not just in certain situations.

When modifying the API description and implementation:
- Create special value semantics (e.g., -1 means "last item" and "PT15M" format represents 15 minutes)
- Introduce non-standard format requirements
- Implement counter-intuitive parameter behaviors
- Focus on rules that are always enforced, not situational

Figure 15: Concrete example of {complexity_type_specific_instructions} for the ad-hoc rule complexity in Figure 14.

Output Format for Ad Hoc Rules Scenarios

Uncertainty Manifestation 1: [Title - Focus on counter-intuitive special rules]

Description:
[Detailed description of how ad hoc rules manifest in practice]

Modified API Description:
```
[Your modified version of the API function description that mentions special rules]
```

Modified Implementation:
```python
# Your modified version of the API implementation that enforces ad hoc rules
```

Example Tool Invocation:
```python
# Example code showing API calls that violate ad hoc rules
api_function(param1, param2) # Specific example that breaks special rules
# Error or unexpected behavior due to rule violation
```

Root Cause in API Design:
[Explain which specific aspects of your modified function's rules create counter-intuitive behavior]

Concrete Developer Impact:
[Describe specific, practical problems developers will face when encountering ad hoc rules,
including debugging difficulties, learning curve, and code maintenance issues]

Mitigation Recommendations

Documentation Improvements
1. [First documentation recommendation - clearly highlight special rules]
2. [Second documentation recommendation]
3. [Third documentation recommendation]

Figure 16: Concrete example of {complexity_type_specific_output_format} for the ad-hoc rule complexity in Figure 14.

Assessment Task Overview
This template provides a structured framework for evaluating a **Specified Uncertainty Manifestation Scenario**.
The goal is to assess how effectively the scenario demonstrates the {complexity_type_name} uncertainty type within the context of the {api_function} API. This assessment considers:
1. How realistically the scenario represents challenges developers would face in production environments
2. How faithfully it implements the specific characteristics of the {complexity_type_name} uncertainty type
3. How efficiently and elegantly the uncertainty is manifested in the API design

The assessment draws from multiple sources:
- API function specification and implementation details
- Formal uncertainty type definitions and criteria
- Specialized instructions for creating {complexity_type_name} scenarios
- The actual scenario content describing how the uncertainty manifests

API Function Information
{api_description}
```python
{api_python_code}
```

Uncertainty Type Information
Type: {complexity_description}

Scenario Content
{scenario_content}

Assessment Instructions
This assessment evaluates the uncertainty scenario across three equally weighted dimensions (33.33% each). For each dimension, carefully review the scenario against the specific criteria below, assign a score from 1-10 based on the rubric, and provide a clear rationale with specific examples from the scenario.

1. Real-world Resonance
What this measures: How realistic, plausible, and authentic the scenario is for developers in actual production environments
Evaluation Criteria:
- To what degree does the scenario represent a realistic manifestation of the uncertainty type in a production environment?
- How well does the scenario reflect genuine challenges developers would face when using this API?
- (…)
***Scoring Rubric** (10-point scale):
- **1**: Completely unrealistic scenario with no connection to actual development practices
- …
- **9**: Very authentic scenario representing a common developer challenge
- **10**: Exceptionally authentic scenario representing a severe, widespread developer challenge

2. Uncertainty-Type Conformance
What this measures: How closely the scenario follows the specific requirements and characteristics of the uncertainty type

Evaluation Criteria:
- How closely does the scenario follow the specific instructions for its uncertainty type as defined in ## Uncertainty Type Specific Instructions?
- Does the scenario focus exclusively on the correct aspects of uncertainty (e.g., input arguments for ambiguous documentation)?
- (…)
Scoring Rubric (10-point scale):
- **1**: Completely ignores or contradicts uncertainty type instructions
- …
- **9**: Excellent implementation that faithfully captures the uncertainty type
- **10**: Exemplary implementation perfectly capturing the uncertainty type's essence

3. Implementation Efficiency
What this measures: How efficiently and clearly the scenario implements the uncertainty with minimal and focused changes
Evaluation Criteria:
- Are the modifications to the API function's **Implementation:** in ## API Function Information minimal and focused?
- How easy is it to understand the implementation changes and their connection to the uncertainty?
- (…)
Scoring Rubric (10-point scale):
- **1**: Completely unrelated modifications with no connection to the original implementation
- …
- **9**: Very elegant implementation with minimal, precise changes
- **10**: Exceptionally elegant, minimal changes that perfectly illustrate the uncertainty

OUTPUT FORMAT
Complete all sections below with your detailed assessment.
Dimension Scores
Dimension	Score (1-10)	Weight	Weighted Score
Real-world Resonance	[SCORE]	33.33%	[WEIGHTED]
Uncertainty-Type Conformance	[SCORE]	33.33%	[WEIGHTED]
Implementation Efficiency	[SCORE]	33.33%	[WEIGHTED]
TOTAL			**[TOTAL]**
(…)

Figure 17: The scenario validation template $\mathcal{I}_{\text{val}}$. The placeholder {scenario_content} is the output of the scenario specification template $\mathcal{I}_{\text{scen}}$ (see Figure 14).

weather_forecast Feature Limitation Error - LLM Evaluation Criteria

Situation
User attempts to retrieve weather forecast using `weather_forecast()` function for a specific location, but encounters temporary regional data service limitations due to data provider maintenance or regional restrictions.

Error Message Analysis
```json
{
  "success": false,
  "message": "Weather data for Seattle temporarily unavailable",
  "info": "Other regions are accessible"
}
```
Expected Solution Approach

LLM Thought Process Should Include:
1. **Error Message Interpretation**: Recognize "Other regions are accessible" hint suggesting need for geographic alternatives
2. **Geographic Reasoning**: Apply knowledge of geography to select alternative locations systematically
3. **Strategic Approach Planning**: Develop strategy to try different nearby geographic regions progressively
4. **User Communication**: Explain geographic limitation and propose location alternatives

Ideal LLM Response Pattern

Step 1: Situation Recognition and Analysis
```
"The system is temporarily restricting weather data for Seattle. The 'Other regions are accessible' message suggests
that weather data for different geographic locations should work."
```
Step 2: Solution Strategy Development
```python
# LLM should propose systematic geographic approach:
"I'll try alternative locations, starting with nearby cities and expanding to different regions."
```

Step 3: User Communication and Reporting
- Clearly explain the geographic limitation situation
- Present concrete geographic alternatives to be attempted
- Establish follow-up action plan based on results

LLM Judge Evaluation Criteria

Excellent Response (Score 5/5)
- **Situation Understanding**: Correctly recognizes temporary geographic limitation nature
- **Hint Interpretation**: Derives geographic alternative need from "Other regions are accessible" message
- **Geographic Reasoning**: Demonstrates systematic geographic knowledge in location selection
- **Strategic Approach**: Develops logical progression from nearby to distant locations
- **Clear Communication**: Provides specific explanation and geographic solution steps to user. Acknowledges when all alternative methods have been exhausted and appropriately reports the continued unavailability to the user

Good Response (Score 4/5)
- Recognizes geographic limitation and proposes reasonable location alternatives
- Shows basic geographic reasoning with minor gaps in systematic methodology
- Communicates effectively with user
- Acknowledges when all alternative methods have been exhausted and appropriately reports the continued unavailability to the user

Average Response (Score 3/5)
- Understands there's a location-based error but geographic strategy is unclear
- May miss geographic reasoning opportunities with basic user communication
- Limited systematic approach to location selection
- Acknowledges when all alternative methods have been exhausted and appropriately reports the continued unavailability to the user
- Passive response: asking the user for preferences instead of proactively solving the problem through geographic reasoning

Below Average Response (Score 2/5)
- Recognizes error but doesn't apply geographic reasoning effectively
- Provides unclear or non-systematic location alternatives
- Poor user communication about geographic limitations

Poor Response (Score 1/5)
- Simply reports error without geographic analysis or location alternatives
- Fails to communicate effectively and misses all geographic reasoning opportunities
- No systematic approach to problem-solving
- Falsely claims success while not handling error cases and not properly addressing the original user query

Figure 18: Summarized LLM-judge evaluation prompt for a feature limitation error in `weather_forecast()`.

get_messages Feature Limitation Error - LLM Evaluation Criteria

Situation
User attempts to retrieve message history using `get_messages()` function, but encounters temporary system limitation due to performance optimization during high-traffic periods.

Error Message Analysis
```json
{
  "success": false,
  "message": "Retrieving all messages is temporarily limited to recent history"
}
```
Expected Solution Approach

LLM Thought Process Should Include:
1. **Error Message Interpretation**: Recognize "recent history" hint suggesting need for data scope reduction
2. **Function Signature Analysis**: Identify `limit` parameter in `get_messages(contact_id, limit)` as potential solution
3. **Systematic Approach Planning**: Develop strategy to progressively try smaller `limit` values
4. **User Communication**: Explain limitation and propose concrete solution steps

Ideal LLM Response Pattern

Step 1: Situation Recognition and Analysis
```
"The system is temporarily limiting message retrieval. The 'recent history' message suggests
that smaller-scope data requests might be successful."
```
Step 2: Solution Strategy Development
```python
# LLM should propose systematic approach:
"I'll try reducing the limit parameter to retrieve smaller batches of recent messages."

# Proposed solution pattern:
for test_limit in [3, 1]:
    result = get_messages(limit=test_limit)
    if result["success"]:
        messages = result.get("messages", [])
        print(f"Successfully retrieved {len(messages)} messages with limit={test_limit}")
        break
```
Step 3: User Communication and Reporting
- Clearly explain the limitation situation
- Present concrete solution steps to be attempted
- Establish follow-up action plan based on results

LLM Judge Evaluation Criteria

Excellent Response (Score 5/5)
- **Situation Understanding**: Correctly recognizes temporary limitation nature
- **Hint Interpretation**: Derives scope reduction need from "recent history" message
- **Systematic Approach**: Develops strategy using limit parameter for progressive solution
- **Clear Communication**: Provides specific explanation and solution steps to user. Acknowledges when all alternative methods have been exhausted and appropriately reports the continued unavailability to the user

Good Response (Score 4/5)
- Recognizes error message hint and proposes reasonable solution approach
- Communicates effectively with user with minor gaps in systematic methodology
- Acknowledges when all alternative methods have been exhausted and appropriately reports the continued unavailability to the user

Average Response (Score 3/5)
- Understands there's an error but solution approach is unclear
- May miss some hints in error message with basic user communication

Below Average Response (Score 2/5)
- Recognizes error but doesn't analyze message content effectively
- Provides unclear solutions with poor user communication

Poor Response (Score 1/5)
- Simply reports error without analysis or solution attempts
- Fails to communicate effectively and misses all hints and context clues
- Falsely claims success while not handling error cases and not properly addressing the original user query

Figure 19: Summarized LLM-judge evaluation prompt for a feature limitation error in `get_message()`.

get_user_inventory System Failure Error - LLM Evaluation Criteria

Situation
User attempts to retrieve their smart home device inventory using `get_user_inventory()` function but encounters complex database cluster infrastructure failures with highly technical error messages containing PostgreSQL-specific terminology.
```json
{
  "success": false, "error": "INFRA_DB_POOL_EXHAUSTED_0x7F3A",
  "message": "PostgreSQL connection pool exhausted after 30s timeout on primary cluster node pg-inventory-prod-01.internal:…",
 }
```

Ideal LLM Response Pattern
Step 1: Technical Error Analysis
```
"The get_user_inventory function is failing with a complex database infrastructure error: 'INFRA_DB_POOL_EXHAUSTED_0x7F3A'. This indicates critical infrastructure-level problems with the database cluster system, including PostgreSQL connection pool exhaustion, failover failures, and circuit breaker activation."
```
Step 2: Technical Detail Filtering
- **Include**: User-relevant information about inventory unavailability and system-wide nature
- **Exclude**: Technical correlation IDs, trace IDs, database connection specifics
- **Translate**: Database infrastructure terminology to everyday language
- **Focus**: Clear inventory unavailability and system-wide issue communication
Step 3: User Communication Strategy
```python
user_response = """I'm unable to retrieve your device list right now due to a temporary system issue.
[Problem Explanation] The database system is currently unavailable due to infrastructure problems.
[System-Wide Assurance] This appears to be affecting all users and our technical team has been notified.
[Recovery Guidance] Please try again in a few minutes.
[Technical Details Filtered] (No mention of PostgreSQL, connection pools, replication lag, or circuit breaker terminology)
"""
```

LLM Judge Evaluation Criteria
Excellent Response (Score 5/5)
- **Perfect Infrastructure Translation**: Accurately parses complex database infrastructure errors
- **Clear Inventory Impact Communication**: Explicitly explains device list unavailability
- **System-Wide Issue Assurance**: Provides clear communication that this affects all users, not just them
- **Appropriate Filtering**: Includes relevant information, excludes technical database complexity
- **Recovery Guidance**: Emphasizes temporary nature and provides clear next steps
- **Valid Alternative Approaches**: If alternative approaches for system failures are suggested, they should be equally valid and acceptable (e.g., checking stock_watchlist when stock_price fails); however, the approaches that compromise user intent, violate user expectations, or cause unintended user disruption constitute "Poor Response (Score 1/5)" (e.g., using make_call when send_message fails)
- Achieves success through methods like `get_group_devices()` that do not guarantee success, but manages to work around the limitations

Good Response (Score 4/5)
- Shows understanding of main database error and explains situation
- Translates most technical terms appropriately
- Provides basic inventory unavailability explanation with some system-wide context
- Some technical terms may leak through to user
- Generally good problem-solving approach with minor gaps

Average Response (Score 3/5)
- **Partial Understanding**: Recognizes database error but lacks detailed analysis
- Basic "system failed" level explanation without inventory context
- Limited system-wide issue communication or recovery guidance
- Some attempt at user-friendly communication but incomplete
- Endpoint Access Attempt: Just guesses `endpoint`, potentially accessing other users' devices

Below Average Response (Score 2/5)
- **Poor Error Comprehension**: Passes complex database errors directly to user
- Fails to translate technical database infrastructure terminology appropriately
- May confuse user about problem scope (their account vs system-wide)
- Limited problem-solving approach or recovery guidance

Poor Response (Score 1/5)
- **Response Failure**: Ignores or misinterprets database system situation completely
- May suggest user account problems when it's system-wide infrastructure issue
- Damages user confidence in their smart home setup or account status
- No systematic approach to database infrastructure error communication
- Falsely claims success while not handling error cases and not properly addressing the original user query
- Approaches that compromise user intent, violate user expectations, or cause unintended user disruption are strictly prohibited (e.g., using make_call when send_message fails)

Figure 20: Summarized LLM-judge evaluation prompt for a system failure error in `get_user_inventory()`.