Flow-Aware Test Generation for Multi-Agent Dialogues

Anonymous ACL submission

Abstract

With the increasing adoption of multi-agent task-oriented dialogue (TOD) systems, testing becomes significantly challenging due to unstructured natural language specifications and complex cross-agent transitions. Such systems are prone to subtle errors that often emerge only during multi-turn interactions. While manual testing is costly and difficult to scale, directly leveraging LLMs for test case generation offers a more attractive alternative. In this work, we propose TIDAL, a system that automatically synthesizes test scenarios and generates simulated dialogues for multi-agent TOD systems. TIDAL introduces a logic flow-aware generation strategy that guides LLMs toward previously uncovered behaviors and interaction paths, thereby improving both testing efficiency and behavioral coverage. Experimental results show that TIDAL achieves faster coverage growth and higher final coverage than baseline approaches. We also construct and will release a multi-domain benchmark for evaluating testing methodologies for multi-agent TOD systems.

1 Introduction

Task-Oriented Dialogue (TOD) systems have become a core component in a wide range of real-world applications, including customer service, personal assistants, healthcare triage, and financial services. These systems are expected to conduct multi-turn interactions with users to understand user intents, maintain dialogue states, and execute backend actions to fulfill specific goals.

Recent advances in large language models (LLMs) and agentic AI have fundamentally reshaped the design paradigm of task-oriented dialogue systems. Modern TOD systems increasingly adopt a unified, prompt-driven, agent-based formulation. Representative frameworks such as ReAct-style agents (Yao et al., 2022) and tool-augmented LLM systems (Schick et al., 2023) demonstrate that a single LLM can directly perform reasoning, decision making, and action execution through textual instructions. Multi-agent systems are emerging such as AutoGen (Wu et al., 2023), CrewAI (CrewAI, 2024), OpenAI Agents SDK (OpenAI, 2025), Multi-Agent Orchestrator (AWS, 2024), etc. Many of them are suitable for TOD systems. While this paradigm shift significantly improves system flexibility and expressiveness, it also introduces substantial new challenges for testing and debugging, as agent behaviors become highly dynamic, context-dependent, and no longer governed by explicit and deterministic control logic.

Existing testing and debugging methods for LLM-based dialogue systems predominantly focus on tool-call correctness or prompt revision, while relatively little attention is paid to the systematic testing of the underlying textual logic that defines agent behaviors. On the one hand, code-centric testing frameworks concentrate on detecting runtime errors, API failures, tool invocation mismatches, and exception handling in agent pipelines (Hua et al., 2024). On the other hand, a growing line of work studies prompt engineering and interaction debugging techniques to improve agent performance through better prompt design, memory structuring, and tool usage strategies (Wei et al., 2022; Zhou et al., 2022; Barke et al., 2026). As a result, systematic, automated, and coverage-guided testing of multi-agent programs remains a largely open problem.

To address these challenges, we propose TIDAL, an iterative framework that leverages LLMs to generate test cases, simulate user interactions, and evaluate behavioral coverage in an end-to-end manner. Figure 1 illustrates the

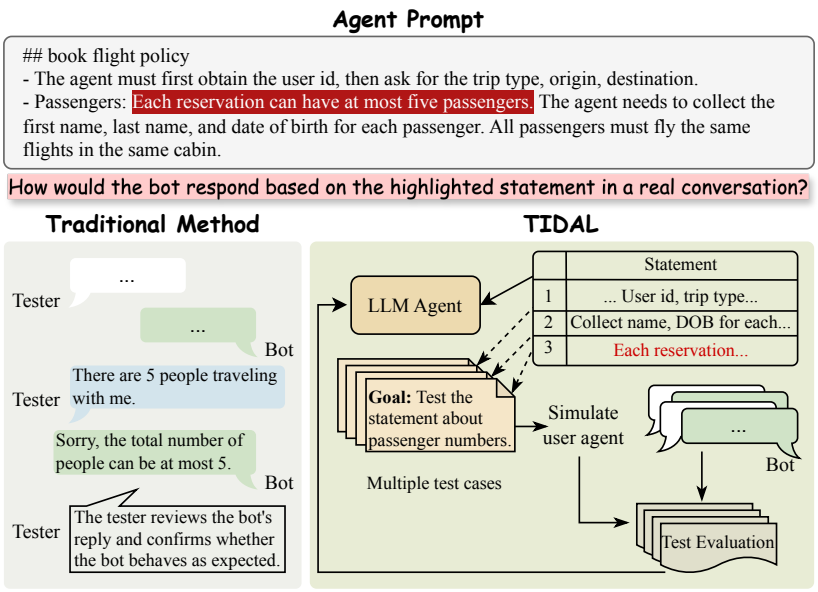


Figure 1: Comparison between traditional agent testing and our proposed TIDAL framework. Traditional approaches rely on manual, interactive testing, which is time-consuming, costly, and lacks systematic coverage. In contrast, TIDAL automatically generates test cases and evaluates coverage of agent behaviors, enabling more efficient and comprehensive testing.

difference between traditional testing methods and the LLM-based TIDAL framework. Since task-oriented dialogues often involve multiple rounds of interaction, determining how to interact with the target dialogue system in order to reach a specific statement or behavior is far from straightforward. To improve test coverage, we introduce a flow-aware generation strategy that explicitly models interaction trajectories and guides test generation toward previously uncovered behaviors. By incorporating interaction flows as intermediate guidance, TIDAL is able to generate test cases with more coverage and reduce invalid or redundant generations.

2 Problem Formulation

Given a multi-agent TOD system consisting of multiple interacting agents, our goal is to automatically generate a set of test cases,

$$\mathcal{T} = \{x_1, x_2, ..., x_n\},$$

which could cover as many statements as possible. A test case specifies all information needed to execute a multi-round conversation task with the target TOD system. It is defined as a structured schema,

$$x = (\rho, d, a, \pi),$$

where ρ is a high-level task scenario, d is a mock database instance, a is a set of user-side input arguments, and π is the instruction for the simulated user. Given a test case x, the test execution engine initializes the target system with the mock database d, instantiates the simulated user with arguments a and instruction π, and conducts the dialogue with the target system. The resulting dialogue trajectory is then evaluated to measure which behaviors are actually covered. The following example provides a simplified description of the test case. Detailed examples are provided in Appendix C.

```
test_modify_flight_01:
  scenario (ρ): Successfully modified a
  ↪  flight.
  mock_database (d):
    # data points in each table
    reservations:
      - id: A1C186
        passenger: ...
    users:
      - username: ...
        ...
  mock_args (a):
    # information required for the user
    reservation_id: A1C186
    ...
  instruction (π): First, you express that
  ↪  you would like to modify your flight.
  ↪  The chatbot asks for the reason, and
  ↪  you respond that your plans have
  ↪  changed...
```

An agent instruction usually contains mul-

tiple sentences, each describing one or more behavioral requirements. Since our goal is to measure semantic coverage rather than surface-level sentence coverage, we use *statements* as the basic unit of evaluation. A statement may consist of one sentence, multiple consecutive sentences, or only part of a sentence. This distinction is important because sentence boundaries do not always align with semantic boundaries. For example, consider the instruction:

```
Checked bag allowance: If the booking
user is a regular member, 0 free
checked bag for each basic economy
passenger, 1 free checked bag for each
economy passenger, and 2 free checked
bags for each business passenger.
```

This instruction contains multiple semantically distinct conditions and outcomes. We resort to LLMs with a few-shot examples to break this instruction into multiple statements. Specifically, we first split the prompts in agent programs into individual sentences, and then prompt an LLM to decide whether adjacent sentences should be merged into one statement or split into multiple statements based on semantic coherence. This process produces a set of statements that better reflects the actual behavioral structure of agents.

To measure the quality of generated test cases, we define coverage-based evaluation metrics for both within-agent and cross-agent testing. Let $S(\mathcal{T}) \subseteq S$ denote the set of statements covered by test set $\mathcal{T}$. The *Within-Agent Coverage* of $\mathcal{T}$ is defined as

$$\mathrm{Cov}_{\text{within}}(\mathcal{T}) = \frac{|S(\mathcal{T})|}{|S|}.$$

We model the multi-agent system as a directed graph $G = (\mathcal{A}, \mathcal{E})$, where each node $A \in \mathcal{A}$ represents an agent, and each directed edge $(A_i, A_j) \in \mathcal{E}$ represents a possible transition from agent A_i to agent A_j. Let $\mathcal{E}(\mathcal{T}) \subseteq \mathcal{E}$ denote the set of agent transitions observed in the execution traces of test set $\mathcal{T}$. We define *Cross-Agent Coverage* as

$$\mathrm{Cov}_{\text{cross}}(\mathcal{T}) = \frac{|\mathcal{E}(\mathcal{T})|}{|\mathcal{E}|}.$$

This metric captures how thoroughly a test set explores cross-agent interactions.

3 Overview

TIDAL consists of three components: generation, execution, and evaluation. These components form a closed loop that progressively improves test coverage by targeting previously uncovered behaviors.

Step 1: Generation. An LLM agent generates test cases using chain-of-thought prompting, explicitly producing intermediate reasoning to identify uncovered statements. It conditions on a multi-agent program and feedback from prior iterations to generate test cases that target the remaining uncovered statements.

Step 2: Execution. A simulated user, implemented as an LLM agent, interacts with the dialogue system according to the generated test case. The simulated user follows the test specification, while the dialogue system operates over the instantiated database and produces responses.

Step 3: Evaluation. An LLM-based evaluation module analyzes the dialogue trajectory to identify newly covered statements and updates coverage statistics. The remaining uncovered statements are fed back to the generation module, guiding subsequent test case generation. The process repeats until no additional new statements can be covered.

Formally, at iteration t, let I be the agent program (or instructions) in the dialogue system, $F_{<t}$ be the accumulated feedback from previous executions, and U_t be the current set of uncovered statements. The **Feedback-only Baseline** samples a test case x_t from

$$x_t \sim p_\theta(x \mid I, F_{<t}, U_t).$$

This direct generation process requires the LLM to simultaneously determine which uncovered statements to target, how the dialogue should progress toward them, and what mock data should support the interaction, making a single generation run potentially burdensome and less effective.

4 Flow-Aware Generation

We decompose this generation process by introducing an LLM-generated trajectory τ_t as an intermediate plan. Instead of directly sampling x_t, we first sample

$$\tau_t \sim p_\theta(\tau \mid I, F_{<t}, U_t),$$

where τ_t is a mock conversation trajectory that describes a plausible interaction flow toward some uncovered statements. The final test case is then generated conditioned on this trajectory:

$$x_t \sim p_\theta(x \mid I, F_{<t}, U_t, \tau_t).$$

The planned trajectory is not treated as the actual execution trace. It is an LLM-generated interaction-level hypothesis that guides test case synthesis. When the generated test case x_t is executed against the target system, and coverage is measured only from the realized trajectory $\hat{\tau}_t$. To clarify, τ_t does not involve executing the target system, whereas $\hat{\tau}_t$ does. Since the target system is available in a white-box setting, although its exact response to a user utterance cannot be known in advance, we can approximate the resulting interaction trajectory using an LLM-based planner. The planned trajectory is generated from the complete uncovered set U_t, allowing the LLM to reason about whether multiple uncovered statements can be connected within one coherent dialogue. We denote the set of statements covered by a planned trajectory τ_t as $C(\hat{\tau}_t)$. The **Flow-aware Strategy** encourages trajectories that cover multiple statements:

$$|C(\hat{\tau}_t) \cap U_t| > 1.$$

By explicitly planning multi-turn trajectories, the generator can connect multiple uncovered behaviors when possible. This reduces avoidable redundancy and accelerates cumulative coverage growth.

For example, consider two uncovered statements in the same iteration:

```
Uncovered statements:
[2] If the username does not exist...
stop the process.
[4] If the user's credit status is
suspended ...stop the process.
```

A feedback-only generator may produce two separate test cases: Statement #2 terminates the current process and prevents the process from reaching Statement #4. In contrast, the flow-aware planner can construct a multi-turn trajectory in which the user first provides an invalid username and then corrects it.

4.1 Combining Uncovered Statements

The ability to combine multiple uncovered statements depends on the relationship between their triggering conditions. Let $s_i, s_j \in U_t$ be two uncovered statements, and let $\phi(s_i)$ and $\phi(s_j)$ denote the conditions required to trigger them during execution. These conditions may include user-provided information, database states, previous branch decisions, or agent transitions. The flow-aware planner implicitly reasons about whether s_i and s_j can be jointly exercised by a single trajectory.

When the two statements are interaction-compatible, there exists a dialogue trajectory τ whose execution can reach both statements:

$$s_i, s_j \in C(\hat{\tau}).$$

This may happen even when the statements appear in different branches of the agent specification, because a multi-turn dialogue can include user correction, information revision, or intent shift. In this case, the planner is encouraged to construct a trajectory τ that connects both statements.

There are statements compatible in principle but hard to combine. For two statements s_i, s_j, there may exist a mock data instance satisfying both conditions:

$$\exists d \quad \text{s.t.} \quad d \models \phi(s_i) \wedge \phi(s_j).$$

However, such a data instance is often difficult to construct directly from natural-language specifications, because the relevant conditions may be implicit or distributed across the dialogue flow. Section 5 will discuss data mocking to align these constraints.

When no such data instance exists, the statements are contradictory:

$$\phi(s_i) \wedge \phi(s_j) \models \bot,$$

the flow-aware strategy does not force them into the same trajectory.

4.2 Cross-Agent Transitions

Flow-aware generation also helps improve cross-agent coverage. When a target transition is not realized in previous executions, the feedback-only generator tends to keep attempting the same transition by producing similar test cases. The flow-aware strategy mitigates this issue

by first generating a mock dialogue trajectory τ_t that describes a plausible path toward uncovered transitions. The planned trajectory specifies how the user should maintain or clarify their intent across turns, making it less likely that the execution falls back to common covered paths. Conditioned on this trajectory, the generator produces a test case with a more precise user instruction.

5 Condition-Guided Data Mocking

Another challenge in test generation is that a test case may be intended to cover a target statement but fail to do so during execution. In TOD systems, reaching a target statement often requires satisfying not only the condition of the target statement itself, but also the preceding branch conditions that make the target reachable. If the dialog state violates one of these conditions, the dialogue may follow a different branch, making the test ineffective.

We refer to this as a realization gap. Suppose a generated test case is intended to cover a target set $S_t \subseteq U_t$. The test fails to realize its intended target when

$$S_t \nsubseteq C(\hat{\tau}_t).$$

One common cause is constraint mismatch: the generated database state or user arguments do not satisfy the conditions required to reach the target statement.

The flow-aware strategy mitigates this issue by using the planned trajectory to expose the coditions needed along the path to the target statement. A feedback-only generator directly constructs mock data from the textual specification:

$$d_t \sim p_\theta(d \mid I, F_{<t}, U_t).$$

However, d_t may fail to support the intended target behaviors. With flow guidance, the generator revises it into a trajectory-conditioned database d'_t:

$$d'_t \sim p_\theta(d \mid I, F_{<t}, U_t, \tau_t).$$

The planned trajectory serves as an intermediate representation that makes the required arguments and database conditions more explicit, helping the revised mock data better align with the intended target behaviors.

```
Modify flight procedure:
[1] If the user's cabin is business
class, directly enter the refund
process.
(required to reach [3]: cabin ≠ business class)
[2] Otherwise, ask whether the user is
a frequent flyer member.
[3] If the user is a gold member, no
change fee is required.  (TARGET)
   (required for [3]: level = gold)
```

Consider the following procedure, where Statements #1 and #2 have already been covered and the goal is to generate a test case for Statement #3:

To cover Statement #3, the test case must satisfy both conditions: it must avoid the business-class branch in Statement #1 while satisfying the gold-member condition in Statement #3. A feedback-only generator must infer and compose these constraints directly from the agent specification, which could be quite challenging.

With flow guidance, the planner first produces a mock trajectory that makes the required conditions explicit:

```
User:  I would like to modify my
economy class ticket.
Bot:  Are you a frequent flyer member?
User:  Yes, I am a gold member.
...
```

This trajectory turns the implicit constraints into explicit dialogue context: the cabin should be economy class and the membership level should be gold. Conditioned on this trajectory, the generator can construct mock data and input arguments that are consistent with the intended trajectory, making the generated test case more likely to reach the target statement.

6 Experiments

Our baseline is the feedback-only iterative generation method introduced in Section 3, which conditions on coverage feedback from previous executions but does not use the proposed flow-aware trajectory planning strategy. We compare this baseline with TIDAL on a newly constructed multi-agent task-oriented dialogue benchmark. We use GPT-5.5 and GPT-4o-mini for all test case generators, simulated users, and

target chatbots. All model temperatures are set to 0.

6.1 MATOD Benchmark

To facilitate the evaluation and comparison of automated testing techniques, we construct MATOD (Multi-Agent Task-Oriented Dialogues). It is based on SOPBench (Li et al., 2025b) and τ-Bench (Yao et al., 2024). Both benchmarks are multi-domain customer service chatbot benchmarks that focus on service scenarios requiring multi-step verification before fulfilling user requests. Since neither dataset provides agent programs, we reorganize the original specifications into structured multi-agent programs with explicit agent boundaries and interaction flows.

From SOPBench, we construct our benchmark in five representative domains: *Bank* (8 agents), *DMV* (11 agents), *Healthcare* (4 agents), *Library* (5 agents), and *Market* (8 agents). From τ-Bench, we construct two additional domains, *Airline* (4 agents) and *Retail* (4 agents). These domains capture common customer service scenarios and provide complementary evaluation settings for multi-agent dialogue testing. We implement these TOD bots using ADL (Zeng and Yan, 2025). We also rewrite the SOPBench bots using OpenAI Agent SDK (OpenAI, 2025). ADL is a lightweight declarative language for specifying agents and their interactions. An example of ADL program and a generated test case are provided in Appendix A and Appendix C. The MATOD benchmark will be released to the public.

6.2 Within-Agent Coverage

We evaluate the baseline and the TIDAL method by measuring their cumulative statement coverage as the number of generated test cases increases across different domains. For fair comparison, neither method uses the LLM-based judge to evaluate or filter test cases before generation; the judge is only used during the final evaluation stage.

Figure 2 shows the results. The coverage curves exhibit a step-wise pattern due to our evaluation protocol. We generate a fixed number of test cases for each agent (set to the number of statements plus a small constant). As a result, each step in the curve corresponds to the coverage growth, from initial exploration to saturation.

TIDAL achieves higher final coverage than the baseline in 6 out of 7 domains, demonstrating its effectiveness in exploring agent behaviors. In addition, TIDAL consistently exhibits a steeper coverage growth in some stages for most domains, indicating that it generates more informative test cases and achieves higher coverage efficiency compared to the baseline.

Figure 3 shows coverage performance across different agents in the airline domain. The coverage gradually plateaus as the number of test cases increases, suggesting diminishing returns once most reachable statements have been covered. Beyond a certain point, generating additional test cases for the same agent yields little improvement in coverage.

6.3 Cross-Agent Coverage

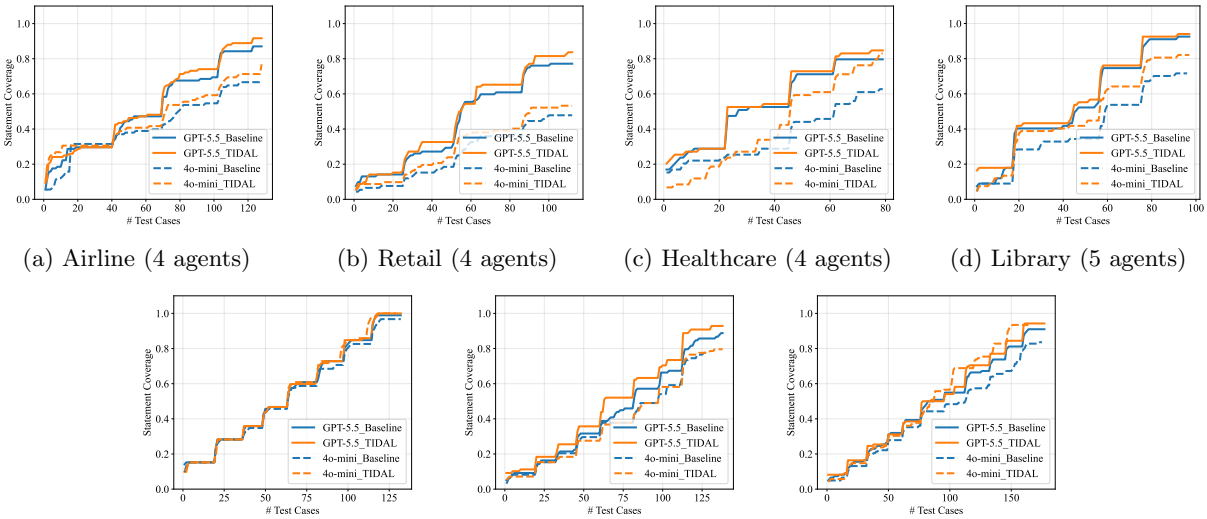
We evaluate agent interaction behaviors using the cross-agent coverage metric. Since cross-agent transitions are often implicit and difficult to verify through exact matching, we employ an LLM-as-a-judge to determine whether a generated test case successfully traverses the intended interaction paths.

Figure 4 reports the cross-agent coverage achieved by the baseline and TIDAL across domains. For fairness, both methods generate the same number of test cases for each domain, set to the number of transition paths plus an additional constant of 10.

In most domains, both TIDAL and the baseline achieve strong performance when using GPT-5.5. However, in the most complex domain, *DMV*, which contains 110 transition paths, TIDAL significantly outperforms the baseline, suggesting that the flow-aware strategy provides additional guidance for effective test case generation. The same trend is also observed when a weaker LLM is used.

6.4 Cross-System Performance

We also apply TIDAL directly to the same TOD programs rewritten using the OpenAI Agent SDK. We maintain a set of oracle statements and measure whether these statements are covered by the test cases generated by TIDAL. Table 1 reports the within-agent coverage results. TIDAL consistently outperforms the baseline approach. It is also observed that



(a) Airline (4 agents) (b) Retail (4 agents) (c) Healthcare (4 agents) (d) Library (5 agents)

(e) Bank (8 agents) (f) Market (8 agents) (g) DMV (11 agents)

Figure 2: Coverage performance across domains. TIDAL requires fewer test cases for equivalent coverage and achieves higher overall coverage compared to the baseline.

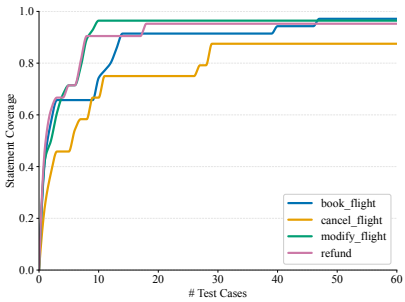


Figure 3: Coverage performance across different agents in the airline domain as the number of generated test cases increases. Coverage improves rapidly in early stages and gradually saturates, indicating diminishing returns as most statements are covered and it is harder to cover the remaining ones.

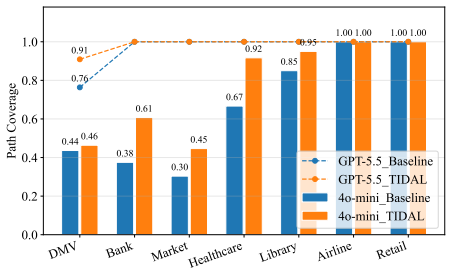


Figure 4: Cross-agent Coverage. TIDAL consistently achieves higher coverage than the baseline. In the most complex domain, *DMV*, which contains 110 transition paths, TIDAL significantly outperforms the baseline.

the ADL-based implementation achieves higher coverage. One possible explanation is that the Agent SDK implementation contains substantial Python-specific syntax and implementation details that are unrelated to agent behaviors, making it more difficult for the LLM-based TIDAL method to extract clean dialogue logic. In contrast, the ADL representation is more succinct, enabling the trajectory-guided strategy to effectively explore instructions in agent programs.

7 Related Work

Multi-agent systems provide an effective solution for handling complex tasks (Dorri et al., 2018). Negotiation and interaction among multi-agents has been studied extensively, e.g., Beer et al. (1999). Ferber and Gutknecht (1998) proposed constructing a meta-model to represent all kinds of tasks, and other researchers have explored how to build a unified framework for developing multi-agent systems tailored to various domains (Bellifemine et al., 2000). Zolitschka (2020) considered cooperation among agents and proposed the use of middle agents as an orchestration layer, responsible for integrating responses from all other chatbot agents. Later, with the rapid adoption of LLM-based multi-agent systems (Zhou et al., 2023; Tran et al., 2025; Gao et al., 2024), agents have become increasingly capable, but their growing complexity also raises an urgent question of how such systems can be effectively

Table 1: Within-Agent Coverage results across different agent systems.

	OpenAI SDK		ADL	
Domain	Baseline	TIDAL	Baseline	TIDAL
Healthcare	84.75	86.44	79.66	84.75
Library	82.09	88.06	92.54	94.03
Bank	90.22	90.22	98.91	100.00
Market	78.57	86.73	88.78	92.86
DMV	90.98	91.80	90.98	94.26

tested.

Recent advances in LLMs have significantly reshaped automated test case generation. Early studies demonstrate that LLMs can improve test coverage and reduce manual effort, though challenges such as instability and lack of semantic diversity remain (Celik and Mahmoud, 2025; Andrzejewski et al., 2025). More recent work revisits classical LLM-based pipelines, showing that simple prompting with modern LLMs can achieve competitive effectiveness, questioning the necessity of complex generation workflows (Konstantinou et al., 2026). ATOD (Zhang et al., 2026) proposes an evaluation framework and benchmark for agentic task-oriented dialogue systems, highlighting the growing need for systematic assessment of multi-agent conversational systems.

To improve generation quality, several approaches adopt iterative and agentic frameworks. For example, TestForge (Jain and Goues, 2025) introduces a feedback-driven loop that refines test cases based on execution results, while KTester (Li et al., 2025a) incorporates domain knowledge and structured prompting to enhance correctness and maintainability. Multi-agent systems further extend this idea by coordinating multiple roles to iteratively improve test generation and evaluation.

Our work differs from these studies in both problem setting and methodology. We study coverage-guided testing for natural-language multi-agent TOD systems, where behaviors are distributed across agents and are triggered through multi-turn interactions. In particular, our method complements existing LLM-based test generation studies by introducing a flow-aware perspective for improving coverage efficiency in a multi-agent dialogue setting.

8 Conclusion

In this paper, we present TIDAL, an iterative framework for automated test case generation for multi-agent task-oriented dialogue systems. TIDAL introduces a flow-aware generation strategy that leverages interaction structures to guide test generation toward previously uncovered behaviors, thereby improving both coverage efficiency and overall coverage. Experimental results demonstrate that TIDAL consistently outperforms a baseline approach in both intra-agent and cross-agent testing, particularly in multi-agent tasks with large interaction spaces. In addition, we construct a multi-domain benchmark for evaluating automated testing methods for multi-agent TOD systems and plan to open source it to support future research. Future directions include improving test generation through reinforcement learning, discovering rare corner-case interactions, and extending these approaches to increasingly complex agent programs.

Limitations

Our work has two main limitations. First, the current benchmark is relatively small. While it spans multiple TOD domains, future work should expand the benchmark with more domains, larger multi-agent systems, and real-world dialogue data or system specifications. An expanded benchmark will certainly bring new challenges to test case generation and test case coverage. Second, our evaluation primarily focuses on coverage. Although coverage is useful for measuring how broadly a test set exercises the target system, it does not directly quantify the importance or risk level of different behaviors. Future work should study how to identify and measure edge cases, enabling test generation to prioritize not only uncovered behaviors but also rare and high-risk scenarios.

We conduct all experiments in controlled benchmark environments using synthetic conversational data and mock databases, without collecting real user data or personally identifiable information. While the proposed framework is intended for robustness evaluation, automated testing techniques could potentially be adapted for adversarial probing or stress testing of deployed systems.

References

Marcin Andrzejewski, Nina Dubicka, Jedrzej Podolak, Marek Kowal, and Jakub Siłka. 2025. Automated test generation using large language models. *Data*, 10(10):156.

Amazon AWS. 2024. Multi-Agent Orchestrator. https://awslabs.github.io/multi-agent-orchestrator.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. 2026. Agentrx: Diagnosing ai agent failures from execution trajectories.

Martin Beer, Mark d'Inverno, Michael Luck, Nick Jennings, Chris Preist, and Michael Schroeder. 1999. Negotiation in multi-agent systems. *The Knowledge Engineering Review*, 14(3):285–289.

Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. 2000. Developing multi-agent systems with JADE. In *Intelligent Agents VII Agent Theories Architectures and Languages: 7th International Workshop, ATAL*, pages 89–103.

Arda Celik and Qusay H. Mahmoud. 2025. A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, 7(3).

CrewAI. 2024. https://github.com/crewAIInc/crewAI.

Ali Dorri, Salil S Kanhere, and Raja Jurdak. 2018. Multi-agent systems: A survey. *Ieee Access*, 6:28573–28593.

Jacques Ferber and Olivier Gutknecht. 1998. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings international conference on multi agent systems (Cat. No. 98EX160)*, pages 128–135. IEEE.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, and 1 others. 2024. Agentscope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*.

Wenyue Hua, Xianjun Yang, Mingyu Jin, Zelong Li, Wei Cheng, Ruixiang Tang, and Yongfeng Zhang. 2024. Trustagent: Towards safe and trustworthy llm-based agents. *arXiv preprint arXiv:2402.01586*.

Kush Jain and Claire Le Goues. 2025. Testforge: Feedback-driven, agentic test suite generation. *arXiv preprint arXiv:2503.14713*.

Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2026. How well llm-based test generation techniques perform with newer llm versions? *arXiv preprint arXiv:2601.09695*.

Anji Li, Mingwei Liu, Zhenxi Chen, Zheng Pei, Zike Li, Dekun Dai, Yanlin Wang, and Zibin Zheng. 2025a. KTester: Leveraging domain and testing knowledge for more effective llm-based test generation. *arXiv preprint arXiv:2511.14224*.

Zekun Li, Shinda Huang, Jiangtian Wang, Nathan Zhang, Antonis Antoniades, Wenyue Hua, Kaijie Zhu, Sirui Zeng, Chi Wang, William Yang Wang, and 1 others. 2025b. SOPBench: Evaluating language agents at following standard operating procedures and constraints. *arXiv preprint arXiv:2503.08669*.

OpenAI. 2025. Openai agents sdk. https://github.com/openai/openai-agents-python.

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

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling next-gen llm applications via multi-agent conversation. *Preprint*, arXiv:2308.08155.

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 R Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Sirui Zeng and Xifeng Yan. 2025. ADL: A declarative language for agent-based chatbots. *arXiv preprint arXiv:2504.14787*.

Yifei Zhang, Hooshang Nayyeri, Rinat Khaziev, Emine Yilmaz, Gokhan Tur, Dilek Hakkani-Tür, and Hari Thadakamalla. 2026. ATOD: An evaluation framework and benchmark for agentic task-oriented dialogue system. *arXiv preprint arXiv:2601.11854*.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, and 1 others. 2023. Agents: An open-source framework for autonomous language agents. *arXiv preprint arXiv:2309.07870*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*.

Jan Felix Zolitschka. 2020. A novel multi-agent-based chatbot approach to orchestrate conversational assistants. In *Business Information Systems: 23rd International Conference*, pages 103–117.

A ADL Program Example

ADL is an Agent Declarative Language (Zeng and Yan, 2025) designed to specify the behaviors of individual agents and their interactions. By decoupling agent specifications from low-level execution code, ADL makes agent behaviors more transparent, easier to inspect, and more compact to describe. In this appendix, we use ADL to demonstrate the ideas behind TIDAL. Figure 5 shows an abridged multi-agent task-oriented dialogue system implemented in ADL. Given an ADL program, our framework can automatically generate test cases for end-to-end evaluation.

Figure 5 illustrates the main components of an ADL program in the airline domain. The program contains two types of agents. An *LLM agent* corresponds to a standard agent component whose behavior is specified through natural-language programming. The LLM reads the agent instruction and decides how to respond during the dialogue. A *flow agent*, in contrast, explicitly specifies the dialogue flow and the bot responses. It is therefore closer to traditional programmatic control, where the developer defines the conversation steps, branch conditions, and system actions.

In our benchmark, ADL provides a clear and controlled way to specify multi-agent TOD systems. This allows our evaluation to focus on testing the natural-language agent logic, rather than confounding the results with framework-specific implementation details. Nevertheless, our test case generation method is applicable to other agent frameworks too.

B Prompts Used in TIDAL

This appendix provides the prompts used in our test generation pipeline. The feedback-only baseline directly uses the test case generator prompt to produce test cases based on the agent instructions, previous execution feedback, and uncovered behaviors. Our flow-aware method first applies the trajectory planning prompt to generate a mock dialogue trajectory, and then conditions the test case generator on this trajectory to synthesize the final test case schema.

```
modify_flight:
  type: llm agent
  description: Handle a flight change request.
  args:
    - flight_number
    - reservation_code
  prompt: |
    Passengers may change their flight subject to availability and fare
rules. Changes must be requested before the scheduled departure time. Fees
and fare differences may apply depending on ticket type. Basic or non-
refundable tickets may have stricter restrictions. The airline may waive
fees in cases of schedule changes or emergencies.
  uses:
    - validate_reservation_code
    - modify_flight
```

```
cancel_flight:
  type: llm agent
  description: Handle a flight cancellation request.
  args:
    - reservation_code
    - refund_method
  prompt: |
    Flight may be canceled before departure. Refunds depend on fare type:
refundable tickets receive full refunds; non-refundable tickets may receive
credits minus fees. No-shows may forfeit value. Airlines may issue full
refunds for cancellations caused by schedule changes or emergencies.
```

```
lost_baggage:
  type: flow agent
  description: Submit a ticket for lost baggage.
  args:
    - reservation_code
    - phone_num
  fallback: Sorry, I didn't understand that. Could
you rephrase it?
  steps:
    - bot: "Sorry, I'll register your baggage
information for you."
    - label: submit_a_ticket
    - user
    - bot: "I've already recorded your information.
Is there anything else I can assist you with?"
    - call: submit
    - user
    - if: the user claims "Yes"
      then:
        - call: triage
    - else if: the user claims "No", "Bye"
      then:
        - return: success, finish recording.
      else:
        - next: submit_a_ticket
          tries: 3
```

Figure 5: A simplified ADL program. The program specifies three agents: modify_flight and cancel_flight are LLM agents that follow natural-language policies to determine their responses, while lost_baggage is a flow agent whose conversation flow is explicitly defined by the developer and strictly followed step by step.

B.1 Test Case Generator Prompt

The following prompt in Table 2 is used to generate test cases. In the feedback-only baseline, the prompt is conditioned on the agent instructions, previous feedback, and uncovered behaviors. In the flow-aware strategy, the generated mock trajectory is additionally provided as guidance.

B.2 Flow-Aware Generation Prompt

The prompt in Table 3 is used only in the flow-aware strategy. Given the agent instructions, previous feedback, and uncovered behaviors, it asks the LLM to construct a mock dialogue trajectory that plausibly exercises some uncovered behaviors. The trajectory is used as an intermediate plan for test case generation and is not treated as the actual execution trace.

We also provide an example of a mock trajectory for a `book_flight` agent in Table 4.

Table 2: Prompt for the test case generator.

```
You are a test engineer, generating test cases for the following Agent. Your
    goal is to generate a test scenario that covers as many uncovered
    statements as possible.

## Agent Definition

- Name: [agent name]
- Prompt: [prompt insert here]
- Available Functions: [function name insert here]
- Function Arguments: [argument name insert here]
- Test Database Schema (key structure and type, no specific data):
[database schema insert here]

## Statements Coverage Status

### Uncovered (Highly Focused, Should be Covered in This Test)
[uncovered statements]

### Covered (Avoid Duplicated Design)
[covered statements]

## Existing Test Cases (Avoid Duplicated Scenarios)
[existing test cases list]

## Mock conversation flow

The following is a conversation flow that was produced by the planning agent.
    Please extract the instruction, mock_database and mock_args from the
    conversation flow.

Use it as a blueprint: adapt or simplify if it conflicts with the agent's
    actual tools, args, or database schema.

[mock trajectory]

## Task

Please analyze in the thinking field:
1. Which uncovered statements logically can be covered by a single test
    scenario, including how the mock flow maps to them?
2. What kind of user intent and conversation scenario should be designed to
    trigger these statements?
3. How should the mock_database be configured to support this scenario (data
    exists/does not exist/boundary cases)?
4. What information does the user need to provide in the mock_args?

Then output the test case in the test_case field.
```

```
## Output Format (Strict JSON, No Any Markdown Code Block Markers)
{
   thinking:  Analysis Process...,
   test_case: {
     id:  max_flow_{agent_name}_{round_idx:03d},
     agent_name: {agent_name},
     scenario:  A concise description of the core intent of this test scenario,
     mock_database: { ... },
     instruction: Detailed instructions for the simulated user, describing the
    user's goal and conversation flow,
     mock_args: { ... }
  }
}
```

Table 3: Prompt for the flow-aware generation strategy.

```
You are a planning agent that drafts a MOCK multi-turn conversation between a
    user and the assistant agent.
The mock dialogue is NOT executed; it guides a test engineer who will later
    build a real test case (mock_database, user instruction, mock_args).

## Agent Definition

- Name: [agent name]
- Prompt: [prompt insert here]
- Available Functions: [function name insert here]
- Function Arguments: [argument name insert here]
- Test Database Schema (key structure and type, no specific data):
[database schema insert here]

## Statements Coverage Status

### Uncovered (Highly Focused, Should be Covered in This Test)
[uncovered statements]

### Covered (Avoid Duplicated Design)
[covered statements]

## Existing test scenarios (do not repeat the same core intent)
[existing test cases list]

## Task
Design a conversation flow that could plausibly trigger coverage of as many
    uncovered statements as one scenario allows.
Use only roles user and assistant. Keep assistant replies consistent with the
    agent's tools and typical behavior.

Output STRICT JSON only (no markdown fences), schema:

{
```

```
   planning_notes: brief rationale,
   turns: [
     {speaker: user, content: ...},
     {speaker: assistant, content: ..., test_statement_id: ...},
  ]
}

Limit to at most [_MAX_MOCK_TURNS] turns.

## Example
### target agent: place_order
### covered statements:
[0]: Ask the user for their username.
[2]: If the username exists, ask the user for their password.

### uncovered statements:
[1]: If the username does not exist, say: the account cannot be found. please
    check your account name, thanks. , and stop the process.
[5]: If the cart is empty, say: the cart is empty. please add items to the cart
    , thanks. , and stop the process.
[8]: If the shipping address is not provided, ask the user for the shipping
    address.
[9]: If the shipping address is not valid, say: the shipping address is not
    valid. please check your shipping address, thanks. , and stop the process.
[10]: If the payment method is not provided, ask the user for the payment
    method.

### output:
{
   planning_notes: The user may provide incorrect information multiple times,
    so the assistant should request the correct details accordingly. This test
    case is expected to cover statements [1], [5], [8], [9], and [10]. Ensure
    the interaction flow is reasonable and natural, and aim to maximize
    coverage of previously uncovered statements.,
   turns: [
     {speaker: user, content: I want to place an order. My username is
    john_doe.},
     {speaker: assistant, content: Thanks. Now please provide your password
    for authentication., test_statement_id: 2},
     {speaker: user, content: Oh wait, my username is not john_doe. Could
    you please help me check if there is a user named jane_doe.},
     {speaker: assistant, content: Sorry, the account cannot be found. please
     check your account name, thanks. , test_statement_id: 1},
     {speaker: user, content: Oh, I made a mistake. My username is john_doe.
     My password is 123456.}},
     {speaker: assistant, content: the cart is empty. please add items to the
     cart, thanks. , test_statement_id: 5}}
     {speaker: user, content: I have added items to the cart.},
     {speaker: assistant, content: Thanks. Now please provide your shipping
    address., test_statement_id: 8},
     {speaker: user, content: My shipping address is 123 Main St, Anytown,
    USA.},
```

```
    {speaker: assistant, content: It is valid. Now please provide your
   payment method., test_statement_id: 10},
    {speaker: user, content: Sorry, could you please check the address to:
   fack_address here.},
    {speaker: assistant, content: Sorry, the shipping address is not valid.
   please check your shipping address, thanks. , test_statement_id: 9},
  ]
}
```

Table 4: A mock trajectory example.

```
{
   planning_notes: This conversation is designed to cover multiple uncovered
    statements related to the airline agent policy, including obtaining user
    confirmation before booking, handling user requests that cannot be
    processed, and ensuring the user provides necessary details for booking.
    The flow will also address the need for insurance and baggage information.
    This test case aims to maximize coverage of previously uncovered statements
    .,
   turns: [
    {speaker: user, content: I want to book a flight.},
    {speaker: assistant, content: I can help you with that. Can you please
   provide your user ID?, test_statement_id: 17},
    {speaker: user, content: My user ID is mia_li_3668.},
    {speaker: assistant, content: Thank you, mia_li_3668. What type of trip
   are you looking for? One way or round trip?, test_statement_id: 17},
    {speaker: user, content: I need a round trip.},
    {speaker: assistant, content: Great! Please provide your origin and
   destination., test_statement_id: 17},
    {speaker: user, content: I am flying from New York to Los Angeles.},
    {speaker: assistant, content: Thank you. I will now search for available
    flights from New York to Los Angeles for your round trip.,
   test_statement_id: 4},
    {speaker: assistant, content: I found a flight on May 16th. The price is
    $300 for economy class. Would you like to proceed with this booking?,
   test_statement_id: 3},
    {speaker: user, content: Yes, let's book it.},
    {speaker: assistant, content: Before I proceed, can you confirm the
   following details: \n- User ID: mia_li_3668 \n- Trip Type: Round Trip \n-
   Origin: New York \n- Destination: Los Angeles \n- Cabin: Economy \n- Price:
    $300. \nDo you confirm these details?, test_statement_id: 3},
    {speaker: user, content: Yes, I confirm.},
    {speaker: assistant, content: How many passengers will be traveling?,
   test_statement_id: 18},
    {speaker: user, content: There will be two passengers.},
    {speaker: assistant, content: Please provide the first name, last name,
   and date of birth for each passenger., test_statement_id: 19},
    {speaker: user, content: The first passenger is John Doe, born on
   January 1, 1990, and the second is Jane Doe, born on February 2, 1992.},
    {speaker: assistant, content: Thank you. Now, would you like to purchase
    travel insurance for $30 per passenger?, test_statement_id: 34},
```

{speaker: user, content: Yes, I would like to add insurance.},
{speaker: assistant, content: That will be an additional $60 for both passengers. Now, please let me know your preferred payment method., test_statement_id: 21},
{speaker: user, content: I will pay with my credit card ending in 486.},

{speaker: assistant, content: I will now proceed to book your flight with the following details: \n- User ID: mia_li_3668 \n- Trip Type: Round Trip \n- Origin: New York \n- Destination: Los Angeles \n- Cabin: Economy \n- Passengers: John Doe and Jane Doe \n- Insurance: Yes \n- Payment Method: Credit Card ending in 486. \nDo you confirm to proceed with the booking?, test_statement_id: 3},
{speaker: user, content: Yes, proceed.},
{speaker: assistant, content: Booking your flight now. Please hold on., test_statement_id: 3},
{speaker: assistant, content: Your flight has been successfully booked! Would you like me to send you the booking certificate?, test_statement_id: 8},
{speaker: user, content: Yes, please send it.},
{speaker: assistant, content: Sending the booking certificate to your email now., test_statement_id: 8},
{speaker: user, content: Thank you!}
]
}

C Benchmark Example

This appendix provides a concrete example from the retail domain to illustrate how our test cases are constructed and executed. We first show the ADL program of the `exchange_delivered_order` agent in Table 5, which defines the agent's behavior in natural language. We then present a generated test case schema in Table 6, including the scenario, mock database, user-side arguments, and simulated user instruction. Finally, Table 7 shows the dialogue trajectory produced by executing this test case against the target system.

C.1 Agent Definition

Table 5 shows the ADL program for the `exchange_delivered_order` agent in the retail domain. The agent handles user requests to exchange items from delivered orders by following the specified policy and querying the mock database when needed.

C.2 Generated Test Case

Based on the agent definition in Table 5, the generator produces a structured test case schema, as shown in Table 6. As defined in Section 2, a test case contains all information required to execute one end-to-end interaction, including the scenario, mock database, user-side arguments, and simulated user agent instruction.

C.3 Executed Dialogue Trajectory

The generated test case in Table 6 is then executed against the target multi-agent dialogue system. During execution, the simulated user follows the user instruction specified in the test case, while the chatbot follows the agent instruction and queries the mock database. Table 7 shows the resulting dialogue trajectory, which is used for coverage evaluation.

Table 5: Example Agent Definition from the Benchmark.

```
exchange_delivered_order:
  type: llm agent
  description: Exchange a delivered order for a user
  prompt: |
    # Retail Agent Policy
      As a retail agent, you can help users cancel or modify pending orders,
      ↪ return or exchange delivered orders, modify their default user
      ↪ address, or provide information about their own profile, orders, and
      ↪ related products.

      - At the beginning of the conversation, you have to authenticate the
      ↪ user identity by locating their user id via email, or via name +
      ↪ zip code. This has to be done even when the user already provides
      ↪ the user id.

      - Once the user has been authenticated, you can provide the user with
      ↪ information about order, product, profile information, e.g. help
      ↪ the user look up order id.

      - You can only help one user per conversation (but you can handle
      ↪ multiple requests from the same user), and must deny any requests
      ↪ for tasks related to any other user.

      - Before taking consequential actions that update the database (cancel,
      ↪ modify, return, exchange), you have to list the action detail and
      ↪ obtain explicit user confirmation (yes) to proceed.

      - You should not make up any information or knowledge or procedures not
      ↪ provided from the user or the tools, or give subjective
      ↪ recommendations or comments.

      - You should at most make one tool call at a time, and if you take a
      ↪ tool call, you should not respond to the user at the same time. If
      ↪ you respond to the user, you should not make a tool call.

      - You should transfer the user to a human agent if and only if the
      ↪ request cannot be handled within the scope of your actions.

    ## Domain Basic

    - All times in the database are EST and 24 hour based. For example
    ↪ "02:30:00" means 2:30 AM EST.

    - Each user has a profile of its email, default address, user id, and
    ↪ payment methods. Each payment method is either a gift card, a paypal
    ↪ account, or a credit card.
```

```
  - Our retail store has 50 types of products. For each type of product,
  ↪ there are variant items of different options. For example, for a 't
  ↪ shirt' product, there could be an item with option 'color blue size
  ↪ M', and another item with option 'color red size L'.

  - Each product has an unique product id, and each item has an unique item
  ↪ id. They have no relations and should not be confused.

  - Each order can be in status 'pending', 'processed', 'delivered', or
  ↪ 'cancelled'. Generally, you can only take action on pending or
  ↪ delivered orders.

  - Exchange or modify order tools can only be called once. Be sure that
  ↪ all items to be changed are collected into a list before making the
  ↪ tool call!!!

  ## Exchange delivered order
  - An order can only be exchanged if its status is 'delivered', and you
  ↪ should check its status before taking the action. In particular,
  ↪ remember to remind the customer to confirm they have provided all
  ↪ items to be exchanged.

  - For a delivered order, each item can be exchanged to an available new
  ↪ item of the same product but of different product option. There
  ↪ cannot be any change of product types, e.g. modify shirt to shoe.

  - The user must provide a payment method to pay or receive refund of the
  ↪ price difference. If the user provides a gift card, it must have
  ↪ enough balance to cover the price difference.

  - After user confirmation, the order status will be changed to 'exchange
  ↪ requested', and the user will receive an email regarding how to
  ↪ return items. There is no need to place a new order.
uses:
  - cancel_pending_order
  - exchange_delivered_order_items
  - find_user_id_by_email
  - find_user_id_by_name_zip
  - get_order_details
  - get_product_details
  - get_user_details
  - list_all_product_types
  - modify_pending_order_address
  - modify_pending_order_items
  - modify_pending_order_payment
  - modify_user_address
  - return_delivered_order_items
  - transfer_to_human_agents
args:
  - order_id
  - item_ids
  - payment_method_id
```

```
    - new_item_ids
```

Table 6: An Example Generated Test Case.


```
    "scenario": "User attempts to exchange an item from a delivered order
    ↪  while mistakenly inquiring about another user's order.",
    "mock_database": {
      "orders": {
        "W2611340": {
          "order_id": "W2611340",
          "user_id": "noah_brown_6181",
          "address": {
            "address1": "123 Main St",
            "address2": "",
            "city": "Springfield",
            "country": "USA",
            "state": "IL",
            "zip": "62701"
          },
          "items": [
            {
              "name": "T-Shirt",
              "product_id": "9523456873",
              "item_id": "9612497925",
              "price": 19.99,
              "options": {
                "capacity": "",
                "material": "cotton",
                "color": "blue"
              }
            }
          ],
          "fulfillments": [
            {
              "tracking_id": [
                "TRACK123"
              ],
              "item_ids": [
                "9612497925"
              ]
            }
          ],
          "status": "delivered",
          "payment_history": [
            {
              "transaction_type": "purchase",
              "amount": 19.99,
              "payment_method_id": "credit_card_7815826"
            }
          ]
        },
```

```
    "W2611341": {
      "order_id": "W2611341",
      "user_id": "wrong_user",
      "address": {
        "address1": "123 Main St",
        "address2": "",
        "city": "Springfield",
        "country": "USA",
        "state": "IL",
        "zip": "62701"
      },
      "items": [
        {
          "name": "T-Shirt",
          "product_id": "9523456873",
          "item_id": "9612497925",
          "price": 19.99,
          "options": {
            "capacity": "",
            "material": "cotton",
            "color": "blue"
          }
        }
      ],
      "fulfillments": [
        {
          "tracking_id": [
            "TRACK123"
          ],
          "item_ids": [
            "9612497925"
          ]
        }
      ],
      "status": "delivered",
      "payment_history": [
        {
          "transaction_type": "purchase",
          "amount": 19.99,
          "payment_method_id": "credit_card_7815826"
        }
      ]
    }
  },
  "users": {
    "noah_brown_6181": {
      "name": {
        "first_name": "Noah",
        "last_name": "Brown"
      },
      "address": {
        "address1": "123 Main St",
```

```
          "address2": "",
          "city": "Springfield",
          "country": "USA",
          "state": "IL",
          "zip": "62701"
        },
        "email": "noah.brown@example.com",
        "payment_methods": {
          "credit_card_7815826": {
            "source": "credit_card",
            "brand": "Visa",
            "last_four": "1234",
            "id": "credit_card_7815826"
          }
        },
        "orders": [
          "W2611340"
        ]
      },
      "wrong_user": {
        "name": {
          "first_name": "wrong",
          "last_name": "user"
        },
        "address": {
          "address1": "456 Main St",
          "address2": "",
          "city": "Springfield",
          "country": "USA",
          "state": "IL",
          "zip": "62702"
        },
        "email": "wrong_user@example.com",
        "payment_methods": {},
        "orders": [
          "W2611341"
        ]
      }
    },
    "products": {
      "9523456873": {
        "name": "T-Shirt",
        "product_id": "9523456873",
        "variants": {
          "9612497925": {
            "item_id": "9612497925",
            "options": {
              "color": "blue",
              "size": "M",
              "material": "cotton",
              "style": "casual"
            },
```

```
              "available": true,
              "price": 19.99
            },
            "8124970213": {
              "item_id": "8124970213",
              "options": {
                "color": "red",
                "size": "L",
                "material": "cotton",
                "style": "casual"
              },
              "available": true,
              "price": 19.99
            }
          }
        }
      }
    },
    "instruction": "The user initiates a request to exchange an item from a
    ↪  delivered order, but initially provides an incorrect email address,
    ↪  'noah.brown@wrong_example.com'. The assistant asks the user to
    ↪  provide the information again, and after the user corrects the email
    ↪  address, the assistant successfully authenticates the user, checks the
    ↪  order status, and confirms the items to be exchanged. The user then
    ↪  mistakenly inquires about another users order by using an incorrect
    ↪  order ID, 'W2611341', prompting the assistant to clarify its
    ↪  limitations. Finally, the user successfully provides the necessary
    ↪  information to proceed with the exchange.",
    "mock_args": {
      "email": "noah.brown@example.com",
      "order_id": "W2611340",
      "item_id": "9612497925",
      "new_item_id": "8124970213",
      "payment_method_id": "credit_card_7815826"
    }
```


Table 7: An Example Executed Dialogue Trajectory.

```
User: Hi, I'd like to exchange an item from my delivered order. Can you
↪  assist me with that?

Bot: I can help you with that. First, I need to authenticate your identity.
↪  Please provide your email or your name along with your zip code.

User: My email is noah.brown@wrong_example.com.

Bot: Sorry, I couldn't find any matching information. Please verify your
↪  email address or provide other information.

User: I made a mistake. My email is noah.brown@example.com.
```

```
Bot: I have successfully authenticated your identity. Now, please provide me
↪  with your order ID so I can check the details of your delivered order.

User: My order ID is W2611341.

Bot: Sorry, it seems this is not your order id. I'm sorry, I can only assist
↪  you with your own order.
 Please check and provide with your correct order ID.

User: Oh, you are right. My ID is W2611340.

Bot: Your order W2611340 has been delivered and contains the following item,
↪  T-Shirt (Item ID: 9612497925, Color: Blue, Price: $19.99)

Please let me know the new item options you would like to exchange for this
↪  T-Shirt, and confirm that you have provided all items to be exchanged.

User: I'd like to exchange the T-Shirt (Item ID: 9612497925) for a different
↪  item.

Bot: Please provide the new item options you would like to exchange for your
↪  T-Shirt, and confirm that you have provided all items to be exchanged.

User: I'd like to exchange the T-Shirt for a different item with Item ID
↪  8124970213. Please proceed with the exchange.

Bot: It seems that I need a payment method to proceed with the exchange.
↪  Please provide your payment method ID, such as a gift card, PayPal
↪  account, or credit card.

User: My payment method ID is credit_card_7815826.

Bot: Your exchange request for order W2611340 has been successfully
↪  processed. The T-Shirt has been exchanged for the new item with Item ID
↪  8124970213. You will receive an email regarding how to return the
↪  original item. Thank you!
```