(authors' preprint of paper accepted at ICTSS 2025)

Automated Exploration of Conversational Agents for the Synthesis of Testing Profiles

Iván Sotillo del Horno, Alejandro del Pozzo, Esther Guerra, and Juan de Lara

Universidad Autónoma de Madrid, Madrid, Spain
{ivan.delhorno, alejandro.delpozzo, esther.guerra, juan.delara}@uam.es

Abstract. Conversational agents – or chatbots – are increasingly being used to access all sorts of services, like citizen services in city halls, customer support, or shopping. Moreover, recent advances in generative artificial intelligence are prompting the integration of conversational assistants into many applications, like programming IDEs, office automation software, or operating systems. Given the prominence of these agents, their correctness is a rising concern. However, automated and robust testing techniques for conversational systems are still needed.
In this paper, we present a technique for extracting a model of a deployed chatbot (i.e., treated as a black-box) through the automated exploration of its functionality via Large Language Models. This model is used for automated testing by generating testing conversation profiles, which a user simulator employs to conduct focused conversations with the chatbot-under-test. We describe our tool support, and report on an evaluation showing that our exploration technique can accurately model the chatbot-under-test, and the subsequent testing can discover existing errors in the chatbot.

Keywords: Conversational Agents · Large Language Models · Testing.

1 Introduction

Conversational assistants (also called chatbots) are pervasive in today's digital ecosystems. Not only general-purpose chatbots like OpenAI's ChatGPT [23] or Google's Gemini [13] are heavily used, but task-oriented chatbots assisting users in specific tasks like shopping, contracting services, or customer support, are deployed in all sorts of channels such as websites, social networks and smart speakers. Given their prominence nowadays, it is vital to assess their quality.

Chatbots are built using a plethora of approaches [24], from intent-based technologies like Google's Dialogflow [14] or Rasa [26], to multi-agent programming frameworks based on Large Language Models (LLMs) like LangGraph [18] or Microsoft's AutoGen [21], and also Domain-Specific Languages (DSLs) like Taskyto [31]. This heterogeneity of approaches, supported chatbot capabilities, and deployment channels makes chatbot testing difficult. While there are some emerging techniques for chatbot quality assurance [6,30], they either target a particular technology [27], demand substantial manual work (e.g., the provision of test conversations [5,27], the synchronous interaction with humans [28]), require an existing corpus of conversations [35], or need the chatbot source code (limiting their applicability) [7,12,34].

To address these challenges, we propose an automated technique for extracting models from deployed chatbots. The technique uses an LLM-based agent that systematically explores the functionality of the deployed chatbot by conversing with it in natural language. The extracted model can be used for various purposes; in this work, we focus on automated testing. Specifically, from the extracted model, we synthesise input *conversation profiles* for a user simulator called Sensei [19]. Sensei emulates users consistent with the profiles who interact with the chatbot-under-test, and reports any detected issues such as crashes, timeouts, unachieved conversation goals, and conversations loops. Overall, our chatbot testing approach advances the state of the art by only requiring the running chatbot (no source code or previous conversation logs are needed) and eliminating the need for manual work (model extraction, profile generation and user simulation are fully automated). We have implemented our model exploration technique in an open-source tool called Tracer, available at `https://github.com/Chatbot-TRACER/TRACER`. We have successfully evaluated the effectiveness of Tracer in discovering the functionality of deployed chatbots by coverage analysis [7], and the effectiveness of the synthesised testing conversation profiles in finding chatbot errors by mutation testing [9].

Paper organisation. Sec. 2 overviews our approach to automate chatbot exploration and testing. Sec. 3 details the techniques for chatbot exploration, and Sec. 4 the synthesis of testing profiles. Sec. 5 describes tool support. Sec. 6 reports on the evaluation. Sec. 7 compares with related works, and Sec. 8 concludes.

2 Overview of the Approach

Fig. 1 shows a scheme of our approach. In a first phase (label 1), an *explorer* agent interacts with the chatbot-under-test to create a model of the chatbot (label 1a). The explorer uses an LLM to produce phrases that are sent to the chatbot with the goal of discovering its functionality. The chatbot model that is learnt makes explicit the languages the chatbot understands, its fallback phrases, the functionality it offers (e.g., ordering a pizza), the input parameters users should provide to achieve the functionality (e.g., pizza size and type), the admitted values for those parameters (e.g., small, medium and large for pizza size), the output data the chatbot should produce (e.g., the pizza order price), and possible conversation flows. Then, a *synthesiser* agent analyses the extracted model to generate scoped testing conversation profiles for a user simulator called Sensei (label 1b). The profiles characterise meaningful conversations with the chatbot by declaring conversation goals, parameters, parameter values, number of conversations to be generated, and *persona* [8] of the simulated user (e.g., talkative, elderly).

In a second phase (label 2), Sensei uses those profiles to interact with the chatbot-under-test via an LLM that plays the role of the user. The simulator

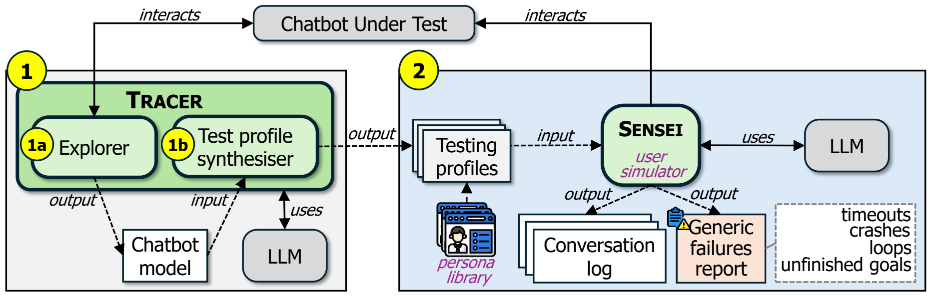

Fig. 1. Scheme of our approach and its main components. (1a) Chatbot's functionality explorer. (1b) Synthesiser of test conversation profiles. (2) User simulator.

persists the resulting conversations, along with a fault report that may signal crashes, timeouts, conversation loops, or unfinished goals (i.e., tasks that the simulator was set to achieve, like ordering a pizza, but did not complete). The conversations are annotated with meta-data (e.g., their duration time, used parameter values, chatbot output data) and can be assessed against correctness rules specified in a Python-based DSL.

Next, Sec. 3 describes the explorer responsible for generating chatbot models, and Sec. 4 details the testing profile generation. A full description of the SENSEI user simulator and the definition of correctness rules is beyond the scope of this paper, but is available on its website [19].

3 Extracting Chatbot Models

To automatically generate test conversation profiles for a chatbot, we first need a model that captures its capabilities. We treat chatbots as black-boxes, accessible only via their conversational interface. Hence, to infer their model, we implemented an automated exploration process driven by LLMs. The process involves an initial *Exploration Phase*, where an explorer agent interacts with the chatbot in multiple sessions (Sec. 3.1), followed by a *Refinement Phase* that consolidates the sessions' data into a final chatbot model (Sec. 3.2).

3.1 Exploration Phase

The core of this phase is an agent, called the *Explorer Agent*. It is designed to simulate a human user interacting with the chatbot-under-test, with the primary goal of systematically uncovering the functionalities the chatbot offers. The result is a model conformant to the schema shown in Fig. 2. This model consists of functionality nodes that are organised in functional categories and may point to following nodes, as discovered in the conversations. Nodes may also have parameters that the user should provide, and output data that the chatbot produces. Overall, the exploration phase comprises the following four steps.

1. Initial Probing. The process starts with the Explorer Agent performing initial probing interactions aimed to obtain the chatbot's language and its *fallback* message. The latter is a default response that task-oriented chatbots use when they do not understand the user intent. These two elements allow the next steps to work properly. The explorer will talk in the detected language, it will rephrase its questions when it receives the fallback response, and it will completely change the topic if it receives several fallback messages, preventing the conversation from getting stuck.

2. Iterative Sessions. After the initial probing, the agent will engage in S conversations of N turns each (both S and N configurable). During each conversation, the agent will have a focus: either to explore possible branches of a previously discovered functionality, or if the functionality queue is empty, to run a general exploration to find new functionalities. During these conversations, the agent will act as a user of the chatbot, selecting options when offered, answering questions, and changing the topic if the conversation stalls due to repeated fallbacks.

3. Functionality Extraction. When each conversation ends, the agent analyses the conversation transcript using an LLM to extract functionalities. Each functionality is represented as a Functionality Node that describes a capability, action, or piece of information provided by the chatbot. As depicted in Fig. 2, a Functionality Node has a name (e.g., *prompt_for_pizza_size*), a description (e.g., *asks the user for the size of the pizza*), parameters, which are fields that the user should input along with the possible options, if any (e.g., *size*: small, medium, large), and outputs, which are pieces of data that the chatbot will provide, if any (e.g., *order_id*, *total_cost*).

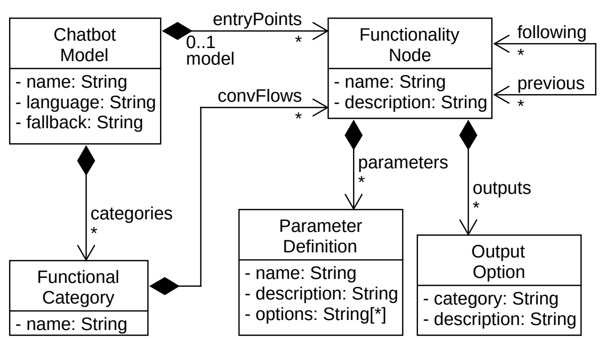

Fig. 2. Chabot model schema.

4. Functionality Consolidation. As every session results in the creation of one or more Functionality Nodes, the agent performs a post-processing to unify them. This helps keeping the Functionality Nodes non-redundant by merging similar ones through two actions:

- **Session-Local Merge:** First, given the Functionality Nodes extracted in the current session, the agent merges those nodes that are semantically similar into a more complete node.
- **Global Merge:** The agent then compares each newly obtained node against the collection of unique functionalities discovered in all previous sessions. Specifically, it uses an LLM to determine whether a node is a semantic duplicate of an existing one, in which case, both are merged into a refined node that captures the functionality more accurately. Nodes confirmed as distinct are added to the collection of functionalities.

3.2 Refinement Phase

Once the interaction with the chatbot finishes, the *Refinement Phase* is focused on refining the obtained Functionality Nodes and building the final chatbot model. The phase comprises the following three steps.

1. Global Consolidation. While the previous Functionality Consolidation step removes duplicates and redundancies, variations of the same functionality might still appear across different sessions. To unify these, once all the conversation sessions are complete, our process merges all the Functionality Nodes that capture variants of the same functionality into a single, more comprehensive node.

2. Chatbot Classification. Next, based on the discovered functionalities and the conversation history, an LLM is used to classify the chatbot as *Transactional* (it guides users through specific tasks or workflows, like placing orders or making bookings) or *Informational* (it primarily answers questions or provides information) [22]. As explained next, this classification influences how the relations between functionalities are inferred.

3. Workflow Structure Inference. The final step is to define *previous/following* relations between the Functionality Nodes (cf. Fig. 2) to form a directed acyclic graph that models the chatbot workflow. This is done by asking an LLM to identify likely sequences, branches, and joins based on conversational evidence and the dependencies between functionalities. The prompt for the LLM depends on the chatbot type:

- **For Transactional chatbots:** The prompt emphasizes the identification of task sequences. It instructs the LLM to look for functionalities that logically follow others (e.g., selecting drinks after selecting pizzas) and to represent optional choices offered by the chatbot as branches (e.g., ordering either a custom pizza or a predefined pizza).
- **For Informational chatbots:** The prompt is conservative on the identification of relations, assuming that functionalities are independent conversation entry points by default. *Previous/following* relations are only established if there is conversational evidence of the chatbot explicitly forcing a sequence (e.g., it says that the user cannot access a certain page, before offering help to solve the issue) or if an undeniable functional dependency exists.

Example. Fig. 3 shows the model that results from applying the Exploration and Refinement phases to a chatbot for a pizzeria built with Taskyto, which we took from [31]. The model captures the relations between the identified Functionality Nodes, and the conversation entry points are connected to a black dot. The Explorer agent has identified four main functionalities, classified in two categories. The *Chatbot Meta* category includes a node for asking the capabilities of the chatbot (label *State Available Inform...*), and a welcome greeting node (label *Provide Welcome Messag...*). The *Order Placement* category includes two node sequences, one to ask for available pizza types, and another to order a pizza. The latter sequence comprises four nodes, where the chatbot successively asks the user for the pizza size and type (parameters of the node *Prompt For Pizza Details*), a confirmation of the pizza type, the number and type of drinks (parameters of the node *Prompt For Drinks*), and a final confirmation that includes the total cost of the order (output of the chatbot in node *Provide Order Total*).

The extracted chatbot models can be used for many purposes, like reverse engineering, reengineering, migration, or maintenance. The next section shows how we use them for testing the original chatbot.

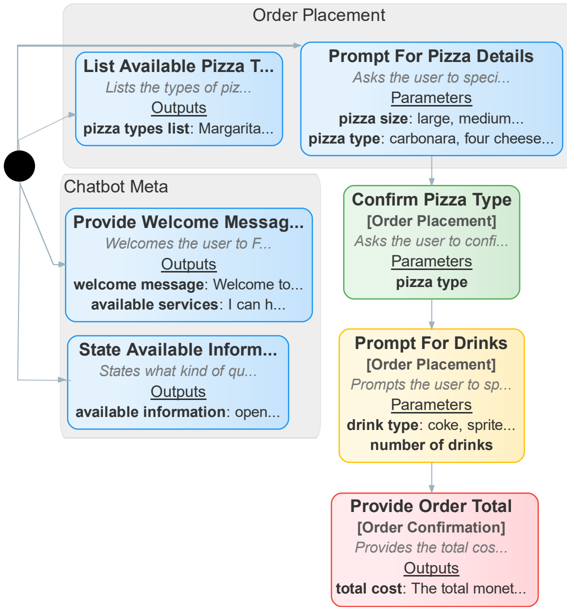

Fig. 3. Model of a chatbot for ordering pizza.

4 Profile Generation and Testing

After extracting the chatbot model, we use it to automatically generate conversation profiles. These serve as test cases to verify the discovered functionalities of the chatbot, its handling of inputs, and the correctness of its outputs using the user simulator Sensei. This section details the structure of these conversation profiles (Sec. 4.1) and their generation from chatbot models (Sec. 4.2).

4.1 Conversation Profile Structure

A conversation profile characterises a user and the intended interaction with the chatbot-under-test. It is defined as a YAML file that describes the user context, the conversation goals, and the interaction style of the simulated user. Listing 1 shows an example of a conversation profile automatically generated by our tool Tracer (cf. Sec. 5) for a pizzeria chatbot. The profile has the following sections:

- `test_name`: A unique, descriptive name for the profile.
- `llm`: Configuration of the LLM used in the user simulator.
  - `model`: The identifier of the LLM to be used (e.g., *gpt-4o-mini*).
  - `temperature`: A real number between 0 and 1 controlling the creativity and randomness of the simulated user's responses.
- `user`: Description of the persona [8], context, and goals of the simulated user.
  - `language`: The language in which the simulated user will interact.
  - `role`: The role of the simulated user (e.g., a customer ordering a pizza).

```yaml
test_name: Pizza Order Placement
llm:
  model: gpt-4o-mini
  temperature: 0.8
user:
  language: English
  role: A customer ordering a pizza and drinks for delivery or pickup.
  context:
  - personality: personalities/conversational-persona
  - You are planning a casual get-together with friends tonight and want to order pizza and drinks.
  goals:
  - "Ask what pizza options are available."
  - "Select the pizza type {{pizza_type}}."
  - "Specify the pizza size as {{pizza_size}}."
  - "Specify that I want {{number_of_drinks}} drinks."
  - "Confirm the order details and total cost."
  - number_of_drinks:  # parameter of the conversation
      function: random() # in each conversation, selects a random int value in the range defined below
      type: int
      data:
        min: 1
        max: 6
        step: 1
  - pizza_size:  # parameter of the conversation
      function: forward() # in each conversation, iterates forward in the following list of values
      type: string
      data:
      - large
      - medium
      - small
  - pizza_type:  # parameter of the conversation
      function: forward() # in each conversation, iterates forward (the forward function supports nesting)
      type: string
      data:
      - Carbonara
      - Four cheese
      - ...
chatbot:
  fallback: "I'm sorry, I did not get what you said. Can you rephrase?"
  output:
  - pizza_types:
      type: string
      description: "A comma-separated list of the available pizza types offered by the chatbot."
  - number_of_drinks_confirmation:
      type: int
      description: "The number of drinks confirmed by the chatbot."
  - final_order_summary:
      type: string
      description: "A summary of the order, including pizza type, size, number of drinks, and total cost."
  - total_cost:
      type: money
      description: "The total cost of the pizza order, including the pizza and drinks."
conversation:
  number: 8
  goal_style:
    all_answered:
      limit: 18
  interaction_style:
  - single question
```

Listing 1. Example of a conversation profile for a pizzeria chatbot.

- `context`: Relevant contextual details or personality traits.
- `goals`: The specific user objectives. Each objective can have parameters, with each parameter defined by its name (e.g., `pizza_type`), `type` (e.g.,

string), the data values it can take (e.g., Carbonara, Four cheese...), and a function that specifies how to pick those values (e.g., random).
- chatbot: Information related to the chatbot being tested.
  - fallback: The chatbot response when it does not understand a request.
  - output: Output data that the chatbot should deliver in the conversation. Each data is defined by a name (e.g., total_cost), type (e.g., money), and a description to guide their extraction from the chatbot responses.
- conversation: Further aspects of the conversation.
  - number: The number of conversations to be simulated. Values bigger than one permit testing different combinations of parameter values.
  - goal_style: The conversation termination criteria. In Listing 1, the option all_answered indicates that the conversation ends when the chatbot delivers all defined outputs, up to the limit of 18 turns.
  - interaction_style: A list of conversation styles the simulated user should adopt, such as *make spelling mistakes*, *use short phrases*, or address each conversation goal in a different *single question*.

4.2 Conversation Profile Generation

The generation of conversation profiles from the extracted chatbot model is an automated process driven by an LLM. The goal is to generate realistic profiles that encapsulate what a real user would do when interacting with the chatbot, ensuring that all the functionalities are covered while minimising redundancy across the profiles. The process comprises seven steps.

1. Grouping of Functionalities into Profiles. The discovered functionalities are analysed and grouped into a minimal set of logical conversation profiles by an LLM call, which takes as input conversation fragments and the chatbot model with the functionality nodes and their relations. Each group gathers functionalities that are semantically related, frequently used together, or performed sequentially within a user journey, particularly for transactional chatbots. A conversation profile is created for each group. The profile has a name and a user role aligned with the functionalities to be tested in the profile. The aim is to ensure that at least one profile covers each discovered functionality, while avoiding redundancy.

2. Goal Generation. For each created conversation profile, a set of goals is generated that trigger all the functionalities grouped in the profile. If the functionalities have parameters, then the goals will have placeholder variables to enable testing all the possible parameter values. For instance, lines 13–15 of Listing 1 show three goals with placeholders. The goal generation is done by an LLM call which receives the profile name, the user role, the functionalities grouped in the profile, the chatbot model, and conversation fragments.

3. Variable Definition. For each placeholder variable, a variable definition is added to the profile. To obtain this definition, our process queries an LLM to determine whether any data source (parameter or output) of the profile's functionality nodes matches the variable. If so, the data source and its possible values are used as the variable definition (cf. lines 31–37 in Listing 1 for the definition of the variable *pizza_type*). Moreover, to test the chatbot with unexpected inputs, the definition is added an extra value within the variable domain, that is not present in the data source (e.g., *Marinara*, if absent from *pizza_type*). Instead, if no data source matches the variable, the LLM is prompted to generate a definition based on the previously gathered information.

4. Context Generation. A simple context of two or three sentences is included in the profile to make the simulated user more realistic (cf. line 10 in Listing 1). The context is generated by the LLM, which receives the user role, conversation goals and functionalities being tested as input. Additionally, a personality from a set of predefined ones is randomly assigned (cf. line 9 in Listing 1).

5. Output Definition. The LLM is used to generate the set of expected chatbot outputs based on the conversation goals and the functionality nodes' outputs. The outputs are pieces of information that the chatbot should deliver, or confirmations to see if the chatbot understood the user (cf. lines 41–52 in Listing 1).

6. Conversation Style Definition. Setting the conversation style options does not require LLM assistance. The number of simulated conversations is set to the largest number of values for any of the parameters. The limit of conversation turns is a linear combination of the number of goals and outputs in the profile. Lastly, one or two interaction styles are randomly chosen from a predefined set. For example, *single_question* in line 59 of Listing 1 makes the simulated user address each goal in separate interactions.

7. Profile Assembly and Validation. The profile is saved as a YAML file, and validated for correctness and completeness using a validation script. In case of errors, the script returns a warning indicating where and what is failing, and the profile plus the warning is sent to the LLM to fix the profile.

Our approach for generating conversation profiles allows transforming the model inferred from the deployed chatbot-under-test, into a set of realistic and comprehensive test cases. As described in Sec. 2, these conversation profiles serve as input to the Sensei user simulator, which uses an LLM to emulate user utterances that align with the specified goals and style. These utterances, and the corresponding responses of the chatbot-under-test, are stored and analysed to extract relevant meta-data (e.g., selected pizza type and size, order price). Additionally, the simulator reports any chatbot crashes, timeouts, conversation loops, and unfinished conversation goals detected during the simulation.

5 Tool Support

We have implemented our techniques for automated chatbot exploration, model extraction, and test profile generation as an open-source Python package called Tracer (Task Recognition And Chatbot ExploreR), available at `https://github.com/Chatbot-TRACER/TRACER`.

Tracer uses the LangGraph framework [18] to manage the interactions with the LLMs, and to orchestrate the conversations performed for chatbot exploration and profile generation. It currently employs OpenAI and Gemini models, but adding support for other LLMs that are within LangGraph is easy.

Tracer can extract models from deployed chatbots, as long as they are accessible via an interface, typically a REST API. It provides connectors to communicate with chatbots of different technologies, such as Taskyto [31], Rasa [26] and 1MillionBot [1]. In addition, it can be easily extended with connectors for other chatbot technologies.

Tracer includes three modules for the phases described in Sections 3–4:

- *Explorer Module*: This module contains the Explorer Agent responsible for the initial probing and iterative conversational sessions with the target chatbot. It manages the interaction logic, the conversation history, and the initial extraction of Functionality Nodes.
- *Refinement Module*: It implements the logic for the functionality refinement, the classification of chatbots as transactional or informational, and the inference of the workflow structure (i.e., the relationships between Functionality Nodes). It takes the raw Functionality Nodes and conversation transcripts from the Explorer Module, and produces the final chatbot model.
- *Profile Generation Module*: It generates conversation profiles based on the chatbot model. The profiles can be used with the Sensei user simulator.

Tracer can be executed from the command line. It produces a set of profiles, and a report that contains information such as the token usage, the estimated cost, the Functionality Nodes in JSON format, and a graphical representation of the chatbot model (cf. Fig. 3).

6 Evaluation

In this section, we aim to answer the following research questions (RQs):

RQ1: Is Tracer effective in discovering chatbot functionality?
RQ2: How effective are the generated profiles at detecting errors?

RQ1 evaluates the coverage of our approach in terms of activating the chatbot functionalities. We distinguish between the coverage achieved by Tracer during the exploration phase, and that achieved by Sensei during the execution of the generated profiles. We anticipate higher coverage during profile execution, as profiles are designed to systematically test all discovered parameters and their valid combinations. For instance, Tracer may find that a pizzeria chatbot offers six pizza types, but does not try them all during exploration; instead, the generated profiles are designed to test all options. Evaluating RQ1 is crucial for understanding how thoroughly Tracer exercises the chatbot's functionality. High coverage does not guarantee high quality, but it implies that a significant part of a system has been tested, increasing the confidence in its reliability [3].

RQ2 assesses the practical effectiveness of our approach in detecting chatbot errors. To this aim, we employ mutation testing [9] by introducing defects into correct chatbots to create mutants, and testing whether the generated profiles detect these defects. A profile successfully detects a mutation (i.e., a fault) when the simulated user fails to complete a goal (e.g., ordering a pizza type that was removed in the mutant) or when an expected output is missing or incorrect. This way, assessing RQ2 is important to provide evidence that our approach produces profiles that are sufficiently comprehensive to reveal real-world chatbot errors.

The experiment data are available at: `https://github.com/Chatbot-TRACER/TRACER-evaluation`.

6.1 RQ1: Coverage of Chatbot Functionality

Experiment Setup. To assess the effectiveness of our approach in discovering chatbot functionality, we applied Tracer to four deployed Taskyto chatbots, also used in [31]. We chose chatbots of this technology because its code is open-source, so we could instrument it to trace the modules activated during conversations, as required to assess RQ1. Moreover, the declarative structure of the chatbots facilitates injecting faults into them, as required to assess RQ2.

Taskyto is a declarative framework to develop LLM-based task-oriented chatbots using a YAML-like DSL. Chatbot definitions consist of any number of modules of five possible types: *menu* to define conversation alternatives; *sequence* to define chains of conversation steps; *action* to execute Python code and produce a verbatim or rephrased response upon receiving some input data; *data gathering* to request user input data; and *question answering* to declare a FAQ.

Table 1. Chatbots used in the evaluation.

Chatbot	Mods.	YAML	Inps.	Vals.	Q&A
Bike-shop	3	65	3	2	4
Photography	5	140	9	12	5
Pizza-order	10	282	22	78	6
Veterinary	3	71	3	4	5

Table 1 displays size metrics of the chatbots used (though Tracer uses the deployed chatbots and not their code). We show the number of: modules in their definition, lines of code (in YAML), input parameters, possible values (in case of enums), and questions in the FAQ. *Bike-shop* is for scheduling bike repair appointments and answers bike maintenance questions. *Photography* is a chatbot for a photography shop, which answers questions about the shop, gathers contact details of clients, and gives price estimates. *Pizza-order* handles pizza orders, including their size, toppings and drinks. *Veterinary* sets appointments and answers about a vet clinic.

We instrumented Taskyto's engine to obtain each activated module, expected input data from the user (e.g., pizza type), values provided for the input data with a limited set of options (e.g., Carbonara), and performed questions from the chatbot FAQ. To account for the non-determinism of LLMs, we repeated the experiment three times. We executed Tracer with 20 conversations and 12 turns per conversation, which enabled a diverse set of interactions within a reasonable time frame. The LLM used by Tracer was Google's Gemini 2.0 Flash with the default model's parameters, i.e., temperature 1. This allowed creativity in the conversations, while still following Tracer's instructions. The LLM used by Sensei and Taskyto was OpenAI's GPT-4o-mini. For Taskyto, we employed a

Table 2. Coverage of TRACER (chatbot exploration) and SENSEI (profile execution).

Chatbot	Stat.	Tool	Module	Input	Value	Question
Bike-shop	Median	TRACER	100%	85.71%	83.33%	75.00%
		SENSEI	100%	71.43%	50.00%	50.00%
	Aggregate	TRACER	100%	85.71%	83.33%	75.00%
		SENSEI	100%	85.71%	83.33%	50.00%
Photography	Median	TRACER	100%	73.33%	64.71%	20.00%
		SENSEI	100%	80.00%	58.82%	40.00%
	Aggregate	TRACER	100%	73.33%	76.47%	20.00%
		SENSEI	100%	93.33%	94.12%	80.00%
Pizza-order	Median	TRACER	83.33%	67.86%	27.38%	100%
		SENSEI	100%	96.43%	69.05%	100%
	Aggregate	TRACER	100%	100%	48.81%	100%
		SENSEI	100%	100%	90.48%	100%
Veterinary	Median	TRACER	100%	50.00%	44.44%	20.00%
		SENSEI	100%	62.50%	44.44%	40.00%
	Aggregate	TRACER	100%	62.50%	55.56%	40.00%
		SENSEI	100%	87.50%	77.78%	80.00%

temperature 0 to ensure the behaviour of the chatbot was deterministic. SENSEI used the temperatures defined in the generated profiles, which were set randomly.

Experiment results. Table 2 summarises the results. For each chatbot, it shows the median coverage, and the aggregate coverage of the three executions, obtained by TRACER when exploring the chatbot, and by SENSEI when running the generated profiles. The green cells highlight the highest average and aggregate coverage per chatbot. We distinguish coverage of modules, input parameters, input values, and FAQ questions.

We observe that TRACER achieved 100% average module coverage in all but one chatbot (though 100% aggregate coverage), and SENSEI (i.e., the profile execution) reached full module coverage for all chatbots. Input coverage was also high, with minimum aggregate values 62.50% for TRACER and 85.71% for SENSEI, reaching 100% in the *Pizza-order* chatbot. Input value coverage was good, with aggregate values between 77.78% and 94.12% for SENSEI. Aggregate question coverage was slightly lower, as SENSEI reached between 50% and 100%.

Answer to RQ1. We can state that TRACER effectively discovered most functionality of the considered chatbots. As expected, executing the generated profiles yielded generally higher coverage than chatbot exploration alone. This shows the added value of profile generation for testing, while remaining fully automatic.

6.2 RQ2: Ability to Detect Errors

Experiment setup. To answer RQ2, we applied mutation analysis [9] to the four chatbots used in the previous RQ, injecting potential errors (mutations) in the chatbots. Then, we checked whether SENSEI detected those errors when using the conversation profiles generated from the correct chatbots (i.e., those obtained when assessing RQ1) and the mutant chatbots.

Table 3. Summary of mutation analysis.

Chatbot	#Mutants	#Profiles	#Convs. per chatbot	#Convs. all mutants	MS	#Live muts.
Bike-shop	13	3	9	117	91.0%	1
Photography	18	3	12	216	76.9%	3
Pizza-order	20	4	22	440	75.0%	3
Veterinary	13	5	21	273	91.7%	1
TOTAL	**64**	**15**	**64**	**1046**	**84.6%**	**8**

To create the mutants, we implemented a Python script that generates as many mutant chatbots as possible from a given original, introducing a single error in each mutant. The errors can be one of eleven types: deleting a possible value from an input parameter (e.g., deleting value *small* from parameter *pizza_size*); making an optional input parameter required, or vice versa; deleting a question from a question answering module, so that the chatbot becomes unable to respond it; swapping the answer of two questions; deleting a conversation alternative from a menu module; deleting the fallback response; deleting a conversation step from a sequence module; swapping the order of two conversation steps in a sequence module; deleting some output data (e.g., omitting the price in the chatbot response *"The pizza costs $10"*); changing the chatbot rephrasing behaviour between the three possible modes; changing the chatbot memory scope, so that it becomes able/unable to access data values previously set in the conversation. We designed these mutation operators based on previous works on mutation testing for task-oriented chatbots [12,34], and trying to maximise the coverage of Taskyto components. The script systematically injects faults in every possible location of the chatbots, but due to cost and time budget (both Taskyto and SENSEI run atop OpenAI models), we randomly selected at most 2 mutants of each type (less if there were not so many). Such mutants were evaluated with the profiles generated in one execution of TRACER over the original chatbot.

Experiment results. Table 3 shows the results for each chatbot. The columns show the number of assessed mutants, the number of profiles and conversations, the mutation score (MS, number of killed mutants divided by total number of non-equivalent mutants), and the number of live (i.e., non-detected) mutants.

We observe a good MS, over 75% in all cases, peaking at 91.7% in *Veterinary*, and with an overall average of 84.6%. The most complex chatbots had the lowest scores (75% and 76.9%). Most errors (58%) were detected fully automatically, as 27.5% caused chatbot crashes, 25.3% unachieved goals, 2.6% timeouts, and 2.6% loops. However, some cases required inspecting the conversation, e.g., when the chatbot omitted a specific piece of data (like the photography shop location) but included other requested information. For instance, the *Photography* chatbot outputs a summary of the shop information, and some but not all information is set in this output, so there is no error of unachieved goal, even if the response is incomplete. To detect such errors automatically, we should define testing rules with oracles that dig into the expected chatbot outputs, as explained in [19].

Answer to RQ2. The generated profiles were effective in detecting most errors injected in the considered chatbots (84.6% overall). However, full automation would require complementing the profiles with (manually created) testing rules that search for specific data in the profile outputs.

6.3 Threats to Validity

External validity. Our evaluation only involves four chatbots built with the same technology, which limits the generalisability of our findings. Thus, stronger evidence could be obtained by applying Tracer to a broader variety of chatbots. We used Taskyto because it provides an open-source framework and open-source chatbots, which we needed for instrumenting the code to collect coverage metrics, and perform mutation testing. In the future, we plan to extend the evaluation to other technologies that meet these requirements, such as Rasa.

Internal validity. For RQ2, we manually reviewed each conversation to detect false negatives and discard equivalent mutants. While we could have generated more conversations, the thousand we used was sufficient to reveal injected faults.

Construct validity. For RQ2, we do not claim that our mutation operators exhaustively model all possible faults. Still, they cover all elements of Taskyto specifications, and our approach detected a significant number of injected faults.

7 Related Work

Inferring a model of a software system by observing its external behaviour, without access to its internal structure, is known as model learning, automated model inference, black-box modelling, black-box reverse engineering, or dynamic reverse engineering. It has been used in software testing [2], reverse engineering [16,20], and protocol inference. However, to our knowledge, it has not been applied to the generation of chatbot models and test conversation profiles.

We exploit chatbot models to test chatbots. Although testing chatbots with real users is valuable, it is also costly and time-consuming. To address this issue, researchers have developed user simulation methods, which we review next.

First, the customisation options of user simulators are diverse. For example, the user simulator Bottester [35] is configured with a Q&A corpus, a number of executions, and a user delay. Ferreira *et al.* [11] produce user profiles with conversational traits, such as engagement or verbosity. Specific for conversational search, Sekulic *et al.* [32] apply in-context learning and LLMs to simulate users with varying levels of patience, cooperativeness and politeness. CoSearcher [29] is a stochastic user simulator for conversational search refinement that supports tuning the users' cooperativeness and patience. Kiesel *et al.* [17] simulate follow-up questions in conversational searches using language models and simple prompts, but no user traits. In this paper, we generate test conversation profiles for Sensei. These are highly customisable, covering not only user traits and persona, but also conversation goals, user inputs and expected chatbot outputs.

For simulating users, the most recent approaches employ generative artificial intelligence, especially LLMs. For instance, Griol *et al.* [15] use a neural network trained on a dialogue corpus to suggest user utterances based on the dialogue history, so the approach depends on the corpus availability. The Kaucus [10] user simulator can include external knowledge via retrieval augmentation and summary control, converting free-form text into training data without user intervention. Sensei uses LLMs to simulate users coherent with a conversation profile. Terragni *et al.* [33] employ LLMs to generate user utterances from dialogue examples and a sentence describing the user goals. Bandlamudi *et al.* [4] employ an LLM-based simulated user to test conversational assistants generated from APIs and RPAs, and another LLM to judge the assistants' handling of the interactions. Wit [36] relies on ChatGPT API as simulated user for the low-cost testing of rule-based conversational agents (i.e., not open-ended ones). Finally, followQG [25] uses language models trained on an in-domain corpus of question, response and follow-up triplets to generate follow-up questions.

As for the output, all user simulators save the simulated conversation, and some offer further data. For example, Bottester computes metrics on user satisfaction (answers' length, repeated answers, response time) and correctness (expected vs. actual answers), and Sensei reports crashes, timeouts, loops, unfinished goals, and can check correctness rules on the conversations.

Overall, while prior work has used user simulation and user profile generation for chatbot testing, our approach is, to our knowledge, the first black-box method for chatbot model learning. It requires neither the chatbot code nor dialogue logs, but it generates chatbot-specific test profiles by an automated interaction with the chatbot. This reduces the effort to create meaningful profiles. While Tracer generates Sensei profiles, it can be adapted to other user simulators. Still, compared to them, Sensei offers greater customisability, and is not limited to specific chatbot types (e.g., for conversational query, or rule-based).

8 Conclusions and Future Work

Conversational agents need to be tested to ensure their quality. To automate this task, we have proposed an approach that extracts a model out of a deployed chatbot, and synthesises profiles for their automated testing via a user simulator. We have presented tool support and reported on an evaluation showing that both Tracer and the execution of the generated testing profiles achieve good coverage of the tested chatbot, and are effective in discovering errors automatically.

In the future, we plan to introduce specific chatbot testing strategies into the synthesised profiles (e.g., the simulated user trying to impersonate the chatbot to confuse it). We also plan to use the extracted chatbot model to enable automatic chatbot migration, e.g., for modernisation. Finally, we aim to conduct ablation studies to analyse the individual contribution of each step of our approach to the overall system performance.

Acknowledgments. Work funded by the Spanish MICINN with projects TED2021-129381B-C21, PID2021-122270OB-I00, and RED2022-134647-T.

References

1. 1 Million Bot: https://1millionbot.com/ (last access in 2025)
2. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Revised Papers. LNCS, vol. 11026, pp. 74–100. Springer (2018)
3. Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2013)
4. Bandlamudi, J., Mukherjee, K., Agarwal, P., Chaudhuri, R., Pimplikar, R., Dechu, S., Straley, A., Ponniah, A., Sindhgatta, R.: Framework to enable and test conversational assistant for APIs and RPAs. AI Mag. **45**(4), 443–456 (2024)
5. Botium: https://cyara.com/products/botium/ (last access in 2025)
6. Cañizares, P.C., López-Morales, J.M., Pérez-Soler, S., Guerra, E., de Lara, J.: Measuring and clustering heterogeneous chatbot designs. ACM Trans. Softw. Eng. Methodol. **33**(4), 90:1–90:43 (2024)
7. Cañizares, P.C., Ortiz, R.D.A., Pérez-Soler, S., Guerra, E., de Lara, J.: Coverage-based strategies for the automated synthesis of test scenarios for conversational agents. In: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test, AST. pp. 23–33. ACM (2024)
8. Cooper, A., Reimann, R., Cronin, D., Noessel, C.: About Face: The essentials of interaction design (4^{th} edition). Wiley (2014)
9. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer **11**(4), 34–41 (1978)
10. Dhole, K.D.: KAUCUS: Knowledge augmented user simulators for training language model assistants. In: Proceedings of the 1st Workshop on Simulating Conversational Intelligence in Chat, SCI-CHAT. pp. 53–65. ACL (2024)
11. Ferreira, R., Semedo, D., Magalhães, J.: Multi-trait user simulation with adaptive decoding for conversational task assistants. In: Findings of the Association for Computational Linguistics, EMNLP. pp. 16105–16130. ACL (2024)
12. Gómez-Abajo, P., Pérez-Soler, S., Cañizares, P.C., Guerra, E., de Lara, J.: Mutation testing for task-oriented chatbots. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE. pp. 232–241. ACM (2024)
13. Google: Gemini. https://gemini.google.com/ (last access in 2025)
14. Google's Dialogflow: https://cloud.google.com/dialogflow (last access in 2025)
15. Griol, D., Carbó, J., Molina, J.M.: An automatic dialog simulation technique to develop and evaluate interactive conversational agents. Appl. Artif. Intell. **27**(9), 759–780 (2013)
16. Hajipour, H., Malinowski, M., Fritz, M.: IReEn: Reverse-engineering of black-box functions via iterative neural program synthesis. In: Proceedings of the International Workshop on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, ECML PKDD, Part II. Communications in Computer and Information Science, vol. 1525, pp. 143–157. Springer (2021)
17. Kiesel, J., Gohsen, M., Mirzakhmedova, N., Hagen, M., Stein, B.: Simulating follow-up questions in conversational search. In: Proceedings of 46th European Conference on Information Retrieval, ECIR. LNCS, vol. 14609, pp. 382–398. Springer (2024)
18. LangGraph: https://www.langchain.com/langgraph (last access in 2025)

19. de Lara, J., del Pozzo, A., Guerra, E., Sánchez Cuadrado, J.: `https://github.com/sensei-chat/sensei` (last access in 2025)
20. Menguy, G.: Black-box code analysis for reverse engineering through constraint acquisition and program synthesis. Ph.D. thesis, University of Paris-Saclay, France (2023)
21. Microsoft's AutoGen: `https://microsoft.github.io/autogen/stable/` (last access in 2025)
22. Moore, R.J., Arar, R.: Conversational UX design: A practitioner's guide to the natural conversation framework, ACM Books, vol. 27. ACM (2019)
23. OpenAI: `https://openai.com/` (last access in 2025)
24. Pérez-Soler, S., Juárez-Puerta, S., Guerra, E., de Lara, J.: Choosing a chatbot development tool. IEEE Softw. **38**(4), 94–103 (2021)
25. Rao, P., Agnihotri, M., Jayagopi, D.B.: Improving asynchronous interview interaction with follow-up question generation. Int. J. Interact. Multim. Artif. Intell. **6**(5), 79–89 (2021)
26. Rasa: `https://rasa.com/` (last access in 2025)
27. Rasa test: `https://rasa.com/docs/pro/testing/evaluating-assistant/` (last access in 2025)
28. Ren, R., Castro, J.W., Acuña, S.T., de Lara, J.: Evaluation techniques for chatbot usability: A systematic mapping study. Int. J. Softw. Eng. Knowl. Eng. **29**(11&12), 1673–1702 (2019)
29. Salle, A., Malmasi, S., Rokhlenko, O., Agichtein, E.: CoSearcher: Studying the effectiveness of conversational search refinement and clarification through user simulation. Inf. Retr. J. **25**(2), 209–238 (2022)
30. Sánchez Cuadrado, J., Ortiz, R.D.A., Pérez-Soler, S., Cañizares, P.C., Guerra, E., de Lara, J.: Integrating static quality assurance in CI chatbot development workflows. IEEE Softw. **41**(5), 60–69 (2024)
31. Sánchez Cuadrado, J., Pérez-Soler, S., Guerra, E., de Lara, J.: Automating the development of task-oriented LLM-based chatbots. In: Proceedings of the 2024 ACM Conference on Conversational User Interfaces, CUI. pp. 1–17. ACM (2024)
32. Sekulic, I., Lu, L., Bedi, N.S., Crestani, F.: Simulating conversational search users with parameterized behavior. In: Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region, SIGIR-AP. pp. 72–81. ACM (2024)
33. Terragni, S., Filipavicius, M., Khau, N., Guedes, B., Manso, A., Mathis, R.: In-context learning user simulators for task-oriented dialog systems (2023)
34. Urrico, M.F., Clerissi, D., Mariani, L.: MutaBot: A mutation testing approach for chatbots. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion. pp. 79–83. ACM (2024)
35. Vasconcelos, M., Candello, H., Pinhanez, C., dos Santos, T.: Bottester: Testing conversational systems with simulated users. In: Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems, IHC. ACM (2017)
36. de Wit, J.: Leveraging large language models as simulated users for initial, low-cost evaluations of designed conversations. In: Proceedings of 7th International Workshop on Chatbot Research and Design, CONVERSATIONS. LNCS, vol. 14524, pp. 77–93. Springer (2023)