

Automated End-to-End Testing for Conversational Agents

Juan de Lara^a, Alejandro del Pozzo^a, Esther Guerra^a, Jesús Sánchez Cuadrado^b

^a*Modelling & Software Engineering Research Group
Universidad Autónoma de Madrid, Madrid, Spain*

^b*Models & Languages Lab
Universidad de Murcia, Murcia, Spain*

Abstract

The advances in generative artificial intelligence, especially Large Language Models (LLMs), have prompted the proliferation of conversational agents (or chatbots). These can be general-purpose – like ChatGPT – or tailored to specific tasks – like buying tickets or obtaining customer support. Although chatbots play a significant role in today’s software ecosystem, they are hard to test: defining meaningful, thorough tests is time-consuming, and setting an oracle flexible to conversational variations is challenging. This is aggravated when testing LLM-based chatbots, as their conversation is natural but unpredictable.

To alleviate this problem, we present an end-to-end testing approach for conversational agents, comprising two components. First, a highly customisable user simulator that generates meaningful conversations with a chatbot under test, for the given goals (e.g., setting an appointment) and communication styles (e.g., long/short phrases, spelling mistakes). Second, a domain-specific language to specify and check correctness conditions (assertions and metamorphic relations) on the generated conversations. The conditions can assess functional correctness (e.g., booking more tickets costs more) and interaction styles (e.g., the chatbot responds in English and does not deviate from certain topics). This paper describes the approach, an implementation

Email addresses: juan.delara@uam.es (Juan de Lara),
alejandro.delpozzo@uam.es (Alejandro del Pozzo), esther.guerra@uam.es (Esther Guerra), jescusc@um.es (Jesús Sánchez Cuadrado)

enabling chatbots’ testing independently of their technology, and an evaluation of its effectiveness in finding defects. We tested our tool on chatbots with artificially injected errors, and on third-party, real-world chatbots. Our tool detected between 81.25% and 100% of the injected errors, and identified actual functional issues in the real-world chatbots by applying manually defined correctness rules.

Keywords: Conversational Agents, Chatbots, User Simulation, Testing, Metamorphic Testing

1. Introduction

Recent advances in Natural Language Processing (NLP) and generative artificial intelligence – especially Large Language Models (LLMs) [1] – are popularising conversational user interfaces [2]. Today, many companies offer conversational agents (i.e., chatbots) to access their services, e.g., for ordering products, performing operations or customer support. In addition to such service-oriented chatbots, general-purpose ones – like ChatGPT [3] or Gemini [4] – can converse on any topic.

Both types of chatbots need to be thoroughly tested in terms of functionality (i.e., the data in their responses is correct) and conversation style (e.g., concise answers, do not stray in off-topic conversations). In particular, *end-to-end chatbot testing* is concerned with simulating real user scenarios to validate that a chatbot responds properly to diverse inputs. This involves testing the entire conversation flow, from the initial user query to the final chatbot response, which may comprise several interactions. However, testing chatbots is hard for several reasons [5, 6]. First, manually populating meaningful sequences of user-chatbot test interactions is time-consuming and may result in a limited coverage of input data and communication styles. Second, setting an oracle for the test cases is difficult, as there may not be a specific correct answer to the user (several may be admissible), and the data to be assessed is embedded in the chatbot answers and may appear in different forms at different conversation steps. This makes fully automated end-to-end testing of chatbots problematic.

Some emerging works employ generative artificial intelligence to simulate users [7, 8, 9], e.g., for evaluating conversational assistants. However, an automated framework for end-to-end testing of chatbots must go beyond sentence generation of simulated users. As an example, consider testing a

chatbot that takes pizza orders for a pizzeria. A comprehensive chatbot testing system would require means to set the goals of the simulated users when interacting with the chatbot (e.g., ordering a custom or a predefined pizza), user inputs and their combinations (e.g., ordered pizza types and sizes), and data to extract from the chatbot responses (e.g., order price and ID). It should also be able to verify correctness conditions of the conversations (e.g., the order price meets the expectations, and order IDs are unique). For practical use, the system should support testing any deployed chatbot, and report timing information and detected issues (e.g., crashes, timeouts, conversation loops). Our work focuses on building such a testing system, which is currently missing.

In this paper, we propose a novel end-to-end testing approach for conversational agents. First, we provide a highly configurable *user simulator* that interacts with the chatbot under test to produce a bulk of user-chatbot conversation sequences. The simulator is configured with a profile that declares the conversation goals, the user input data, the interaction style and *persona* [10] of the simulated user, and the number of conversations to be generated. The simulator produces natural conversations that make use of input data as specified, allowing testers to manually inspect them for any incorrect behaviour. Moreover, the simulator also reports about crashes, timeouts, conversations stuck in loops, or unachieved goals.

Second, we provide a *chatbot testing Domain-Specific Language* (DSL) [11] to specify and automatically assess more specific correctness conditions about the conversations. These conditions may be either assertions to be checked in individual conversations (e.g., the chatbot answers in English), across all given conversations (e.g., to check if a certain output is unique), or metamorphic relations [12, 13] to be assessed on subsets of conversations (e.g., the price that a chatbot reports for a medium pizza is equal to or greater than that reported for a small pizza of the same type). The DSL has primitives for extracting data from user-chatbot interactions and for text analysis of chatbot answers (e.g., language, verbosity). Metamorphic relations help alleviating the oracle problem since testing is based on comparing conversations [12].

We have implemented our approach in the tool `SENSEI` (`uSER` simulation and `teSting` for `conVErsatIonal` agents). It permits generating conversations and testing conversational agents built with any technology, as long as they are accessible programmatically (e.g., via a REST API). The tool is freely available at <https://github.com/sensei-chat/sensei>.

We have evaluated our approach in two ways. First, we have applied mutation testing to evaluate the ability of SENSEI to catch errors of different kinds. Second, we have used SENSEI to find defects in chatbots deployed on the Internet, built using heterogeneous technologies (Rasa, Dialogflow and LLM-based frameworks). Our approach was effective at finding relevant errors in both cases. It successfully detected between 81.25% and 100% of the errors artificially injected in chatbots. It also uncovered real functional issues in the publicly deployed chatbots, including crashes, inability to complete some goals, timeouts, deviations from the chatbot topics, repeated or incorrect chatbot answers, and empty buttons, among others. The evaluation results' raw data and a replication package are available at <https://github.com/satori-chatbots/user-simulator-evaluation>.

Overall, the contributions of this work are the following:

- A highly customisable user simulator designed for *testing* conversational systems. It allows the generation of user utterances conformant to a given conversation profile that declares the user context and persona, conversation goals, user inputs (along with ways to generate them and combine them), and data to extract from the chatbot responses. The simulator uses these utterances to interact with the chatbot under test, producing a wide range of conversations. These conversations are enriched with informative meta-data, including interaction errors such as crashes, timeouts, and unachieved goals.
- A DSL for expressing correctness rules and assessing them on the simulated conversations and their meta-data. The defined rules can be assessed on individual conversations, or on sets of them. An evaluation shows that rule execution is fast, with per-conversation times between 0.003 and 3.4 seconds depending on the rule's nature.
- An evaluation on both artificially created faulty chatbots (both based on LLMs and intent-based) and real-world chatbots deployed on the wild, demonstrating that our approach is effective in detecting a wide variety of errors. The used dataset of chatbots is available to the community as open source.










Paper organisation. Section 2 motivates the work through a running example, and Section 3 overviews our approach. Then, Section 4 describes the user simulator, and Section 5 introduces the testing DSL. Section 6 presents tool

support, and Section 7 reports on the evaluation. Finally, Section 8 compares with related research, and Section 9 presents conclusions and future work.

2. Motivation and Running Example

Assume we would like to assess a simple chatbot for a pizzeria, in charge of collecting orders that include a pizza and drinks. Pizzas have a size (small, medium or large), and can be either predefined (like margherita or marinara) or custom, where the user chooses the toppings. As for drinks, the pizzeria sells three types: coke, sprite and water. Once the user orders the pizza and the drinks, the chatbot informs of the total price and provides a unique order identifier, which the user can use to collect the pizza in the shop. The chatbot can also inform about the opening hours, address of the shop, pizza prices and expected elaboration times.

The following is an example conversation with the chatbot, where the customer orders a small margherita pizza (*I5*), and then two sprites (*I7*). The chatbot confirms the receipt of the order (*I6* and *I8*) and informs about the price, estimated preparation time, order ID, and pickup location.

I0		Chatbot: Hello
I1		User: Hi
I2		Chatbot: Welcome to Fast Pizza! I'm a chatbot assistant that can help you order pizzas and drinks. What can I do for you?
I3		User: I'd like to order a pizza
I4		Chatbot: Would you like to order a predefined pizza or create a custom one? We have predefined options like margherita, carbonara, marinara, hawaiian, four cheese, and vegetarian.
I5		User: A small margherita
I6		Chatbot: Thanks for ordering a small margherita pizza! Could you please let me know how many drinks you would like to order and which type of drinks you prefer? The available options are coke, sprite, and water.
I7		User: 2 sprites
I8		Chatbot: Sure, I've received your order for two sprites. The cost for a small margherita pizza and two sprites is \$13.00. Your order will be ready in 15 minutes. Please visit our shop at 23 Main Street, NY. Your order ID is 1e8390.

Testing a chatbot, even if simple as the previous one, is effort-intensive and time-consuming [6]. This is so as it requires creating many conversations

that help assessing:

- *Chatbot robustness.* As the most basic feature, we would like to exercise the chatbot to discover possible breakdowns. This includes the chatbot service crashing and returning error codes instead of answers, or experience timeouts.
- *Chatbot functionality.* We need to test the correctness of chatbot outputs (e.g., ordering custom and predefined pizzas with different sizes, toppings and drinks). This includes not only checking that the input data is understood correctly (e.g., small margherita, 2 sprites), but also analysing the correctness of the chatbot output (e.g., the order price is \$13), and testing how the chatbot handles incorrect input data (e.g., it should refuse to serve pizzas of non-available types).
- *User input space.* We need to test the chatbot using utterances that reflect the range of suitable combinations of user input data (e.g., testing the pizza ordering functionality with test utterances that vary the pizza type, pizza size, number of drinks, and drink types).
- *User interaction style.* To assess the understanding capabilities of the chatbot, it should be confronted with realistic and diverse user interaction styles, like providing all data in one step (e.g., “*I’d like a small margherita pizza and two sprites*”), in several steps (as in the example conversation), or even breaking the conversation flow (e.g., users who change their mind about what they have just ordered).
- *User types.* The chatbot should be capable to converse with diverse user personalities and profiles. This way, we need to test it using different *personas*, e.g., to emulate verbose and elaborate inputs, short inputs, inputs with spelling mistakes, emoticons, slang, etc.
- *Chatbot interaction style.* We would like to assess not only the functional correctness of the chatbot, but also its interaction style (e.g., if its answers are polite and concise, or the language used), or if it deviates from its goals (e.g., a pizza chatbot should probably not be allowed to converse about football with the user).
- *Chatbot performance.* We may be interested in checking response times of certain chatbot interactions, to ensure good user experience and improved perceived quality.

Without automation, a tester would need to manually define test conversation scenarios, manually interact with the chatbot, and manually assess the chatbot responses. Since this is costly, a practical end-to-end chatbot testing framework must address the following challenges: supporting the automatic generation of test user utterances, while allowing a fine-grained control of the user personality, interaction style, and input values; automatically interacting with the chatbot under test; extracting relevant data from the conversations, and checking their correctness against oracles that, for flexibility, could consider either single or multiple conversations (e.g., as in metamorphic testing [12, 13]); being able to test deployed chatbots from arbitrary technologies uniformly; capturing non-functional aspects of the interaction (like response time); and reporting events like timeouts or crashes.

To attack the previous challenges, the next section describes our proposal for automating the chatbot testing process, end-to-end. Testing concerns such as privacy (e.g., of user data and conversations), ethics (e.g., usage of stored conversations between the chatbot and real users) and security (e.g., data protection, rate limiting, access control) is also important, but is out of the current scope of our framework. However, we will consider those aspects in future work, as we discuss in Section 9.

3. End-to-End Chatbot Testing Approach

We propose an automated end-to-end chatbot testing approach for alleviating the burden to manually create and test realistic conversations. Our approach comprises two components (cf. Figure 1): a user simulator, called SENSEI-chat, which emulates users with a given profile to generate meaningful user utterances and conversations with the chatbot; and a testing DSL, called SENSEI-check, for checking the generated conversations for possible errors or defective chatbot answers. Next, we introduce briefly both components, and Sections 4 and 5 will provide further details.

In a first stage (label 1 in Figure 1), the simulator is configured with a *conversation profile*. This describes the user goals (e.g., ordering a pizza), user profile (e.g., name, age, dietary restrictions), interaction style (e.g., language, spelling mistakes, long/short phrases), data the user must provide during the conversation (e.g., pizza toppings and size), data to be extracted from the chatbot outputs (e.g., pizza price), and number of conversations to be generated. User profiles can be defined by user experience (UX) experts, e.g., by means of *persona* designs (archetypal users and their needs) [10].

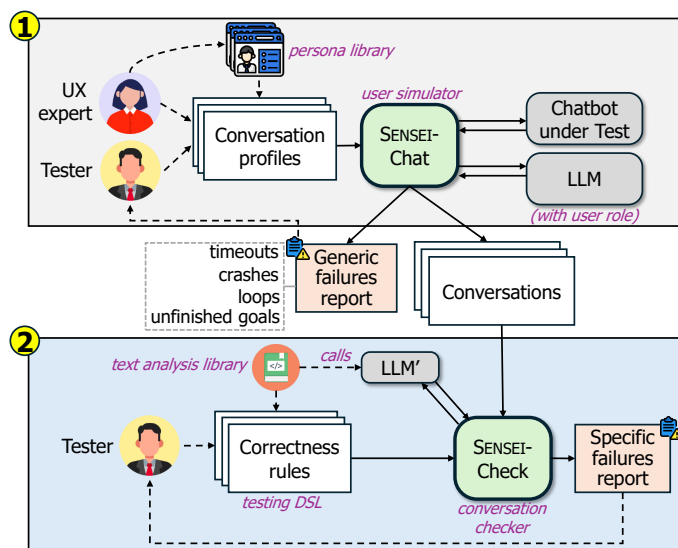


Figure 1: Scheme of our end-to-end chatbot testing approach.

There can be one or more conversation profiles, accounting for different user types and goals.

Then, the user simulator employs the conversation profiles to generate the specified number of conversations, coherent with the profiles. For this purpose, it relies on an LLM that plays the role of user in the simulated conversation, producing utterances compliant with the specified goals and style. Such utterances are sent to the chatbot under test, which is used as a black-box to obtain responses. The chatbot under test may use an LLM or other technologies (e.g., Dialogflow or Rasa). If it does use an LLM, it may or may not be the same one used by SENSEI-chat, which is configurable. If testers know which LLM the chatbot under test uses, then they can configure the conversation profile to use a different one (typically a chat model, such as GPT 4.1 or 4o-mini). In any case, even if both SENSEI-chat and the chatbot under test used the same LLM, there is no risk of information leakage or interference between them, as each one uses its own prompts, and accesses the model via independent API calls in different processes. The conversations so simulated are stored for further processing, enriched with extracted relevant information as meta-data (e.g., selected pizza toppings and size, order price). Moreover, the simulator produces a report of detected generic failures, like chatbot service crashes, timeouts, conversation loops or unfinished

conversations.

In a second stage (label 2 in Figure 1), a tester can define correctness rules to be checked on the conversations. Each rule specifies an evaluation policy, determining the number of conversations involved in the rule assessment. Specifically, a rule can be evaluated: (i) on each individual conversation, in which case it must define an oracle or assertion; (ii) on several conversations, if the goal is comparing their inputs and outputs; or (iii) on all conversations, e.g., to check if an output value is unique among all conversations, or the chatbot eventually returns certain value. Case (i) corresponds to traditional oracle-based testing, and case (ii) is akin to metamorphic testing. We provide a DSL to define correctness rules, which can use the conversation meta-data, and a library for text analysis. This library has functions to check the used language, tone of answers, or conversation deviations from certain topics, among others. Most of these functions are implemented by prompting an LLM, typically a reasoning model like GPT’s o1, o3 or o4-mini. As previously discussed, if the chatbot under test uses an LLM – regardless of whether this is the same used by our library or not – there is no information leakage between the chatbot and our text functions as they do not share prompts and operate independently. `SENSEI-check` applies the rules on the conversation set, yielding a specific failures report with the unsatisfied conditions and problematic conversations.

Overall, our approach involves two levels of testing. In stage 1, a test case has a conversation profile as input, and the oracle is fixed. Executing `SENSEI-chat` with the profile generates as many conversations as specified in the profile, and the test result consists of a coarse-grained generic failures report identifying chatbot crashes, timeouts, conversation goals and unachieved goals. In stage 2, testers can specify further test cases which have the conversations obtained in stage 1 as input, and correctness rules specified with the DSL as oracle. Executing `SENSEI-check` with such test cases produces a fine-grained specific-failures report. Thus, by separating conversation generation from its validation, we obtain the following advantages:

- The conversation generation process already detects generic failures, while the correctness rules investigate domain-specific failures.
- New correctness rules can be added at any time, which can be checked on existing conversations without requiring conversation regeneration.

- The scheme facilitates the evaluation of metamorphic rules, and rules involving all conversations.

While both SENSEI-chat and SENSEI-check use LLMs, they do not share “memory”, i.e., no prompt with history is passed between them, so there is no risk that SENSEI-check performs better when executed right after SENSEI-chat.

In the next section, we describe the details of the user simulator, and then we explain how its results are used to specify tests of different nature to check that the chatbot behaves as expected.

4. User Simulation

Our user simulator generates conversation sequences based on a conversation profile that establishes the user context, interaction style and conversation goals. The simulator uses this profile to create a prompt, which is passed to an LLM to obtain user utterances aligned with the profile. Then, it sends these utterances to the chatbot under test to obtain its responses, and overall compose a complete conversation.

Next, Section 4.1 describes the structure of conversation profiles, and Section 4.2 shows how they are used to generate conversations.

4.1. Specifying conversation profiles

A conversation profile allows configuring the user goals (including input data and their values) and persona; the expected information in the chatbot outputs; the style and number of conversations to be generated; and the underlying LLM that will play the role of the user. Figure 2 shows a conceptual model of the information structure, which comprises:

- *LLM*. Since the user simulator employs an LLM, it is possible to specify the model name (gpt-4o-mini¹ by default) and temperature. The latter is a real number between 0 and 1 controlling the randomness and diversity of the responses generated by the LLM. Profiles may indicate models from all vendors integrated with LangChain², including those of Anthropic, Mistral, OpenAI, Gemini, and many others. Technically, many of them (like those of DeepSeek) can be used with OpenAI API by just changing

¹<https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>

²<https://python.langchain.com/docs/integrations/chat/>

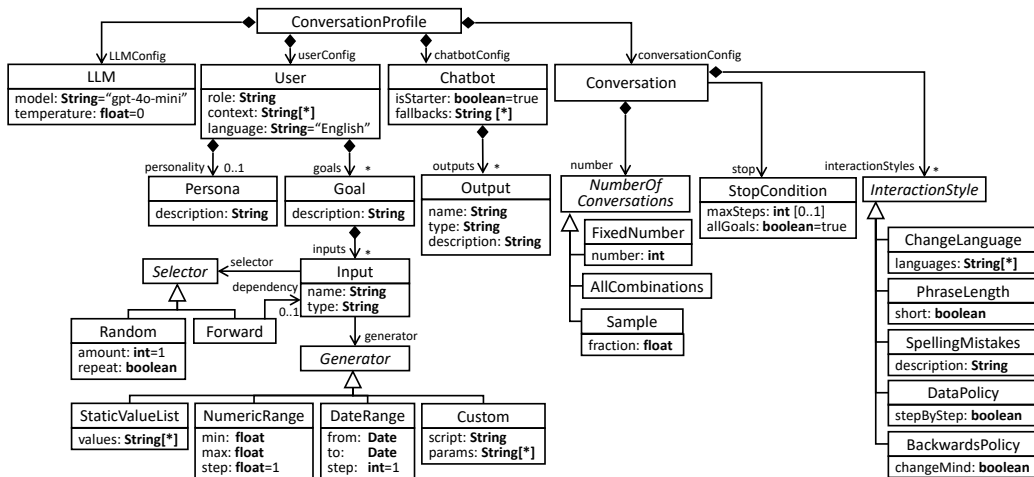


Figure 2: Conceptual model of the conversation profile.

the URL base. The selected model will be used to generate user utterances and guide the interaction with the chatbot.

- *User*. The profile must describe the user role in the conversation (e.g., a customer in a pizzeria), additional user context (e.g., “*you are in a hurry since you have an appointment*”), the interaction language (English by default), the user personality (class *Persona*, to emulate different user behaviours) and the user goals (class *Goal*). Goals describe what the user aims for in the conversation, like ordering a pizza. They can be parameterised with a set of input variables (e.g., pizza size, pizza type). Each input variable (class *Input*) has a name, a type, a generator, and a selector. A *generator* produces values for an input variable, to be used in different conversations with the chatbot. We currently consider generators of: a static list of values, a range (min, max, and step) of admissible numeric or date values, and custom Python code to generate values in arbitrary ways. Then, *selectors* define policies to assign the generated values to the input variables in each conversation. We support two such selectors: random and forward. The *random* selector (class *Random*) randomly selects 1 or more values, and can be instructed to avoid repetitions with previous selections (attribute *repeat*). The *forward* selector (class *Forward*) selects the generated values in sequence. The forward selector for an input variable v_1 can set a dependency to another variable v_2 , in which case, the next value of v_1 is only picked when v_2 exhausts its values. Dependency

sequences of arbitrary length are also supported, which allows emulating nested loops among the values of dependent variables (but cyclic dependencies are forbidden). For each generated conversation, the simulator will save the selected values of the input variables as meta-data.

- *Chatbot.* The profile can specify whether the chatbot should start the conversation (`isStarter`, *true* by default), fallback phrases that the chatbot answers when it fails to understand the user, and data to be extracted from the chatbot outputs (name, type, and description of the data). During conversation generation, the simulator will use the provided fallbacks to detect when a conversation is stuck in a loop [14]. As Section 4.2 will detail, the simulator uses an LLM to extract the output data values from the chatbot responses, and saves them as meta-data.
- *Conversation.* It is also possible to control these further aspects of conversations: number of conversations to be generated, stop condition, and interaction style. For the number of conversations, the profile can select a fixed number (class `FixedNumber`), the number of all combinations of input variables according to their generators and selectors (class `AllCombinations`), or a sample of this latter number (class `Sample`). Conversations can stop either when a maximum number of conversation steps (`maxSteps`) is reached, or when all goals (`allGoals`) are fulfilled (i.e., the simulated user asked about all specified goals, and the chatbot provided all the necessary data). Both conditions (`maxSteps` and `allGoals`) can be set at the same time, in which case, the conversation stops as soon as all goals are fulfilled or the maximum number of steps is reached. Interaction styles influence the shape of the user utterances, and include changing languages mid-conversation (e.g., the user switches from English to Spanish), using either short or long phrases, making spelling mistakes, providing values for all input variables specified in the goals either in one sentence or step-by-step, and changing the user’s mind (e.g., ordering a pizza with some toppings, and modifying the order in later steps).

Table 1 illustrates these interaction choices on conversations generated for the pizzeria running example. The first three rows show the effect of styles *default*, *make spelling mistakes*, and *long phrases* when ordering a small marinara pizza. The two latter styles are especially challenging for intent-oriented chatbots, where intent recognition is based on training phrases. “Obfuscating” the user request on long phrases, or emulating mistakes

(which are common, especially when interacting via mobile devices), is useful to test the recognition capabilities of those chatbots³.

Table 1: Effects of different interaction styles in conversations.

Interaction style	Example conversation
default	User: Hi there! I'd like to order a small marinara pizza for dinner.
make spelling mistakes	User: Hi, I wuld like to order a smal marinara pizzzza for dinneer, please
long phrases	User: I would like to place an order for a small marinara pizza, if you could kindly accommodate my request, as I find myself in the mood for something simple yet delicious after a long day at work, where the hustle and bustle of the office has left me yearning for a comforting meal that doesn't require much thought or preparation on my part. [...]
all questions	User: Can I order a small marinara pizza, and do you have three cans of Coke available?
single question	User: Can I order a small marinara pizza, please? Chatbot: Thanks for ordering a small marinera pizza! Now, could you please let me know how many drinks you would like to order and what type of drinks? The options for drinks are coke, sprite, or water. User: Could I get three cans of coke with that order?
change your mind	User: I'd like to order a small marinara pizza, please. And can I also get three cans of Coke with that? Chatbot: Thanks for ordering a small marinera pizza! Just to clarify, could you please confirm the number of drinks you'd like to order? User: Actually, I think I'd like to change my order. Instead of the small marinara pizza, can I get a large pepperoni pizza? And let's stick with just two cans of Coke this time.
change language	User: I'd like to order a small marinara pizza, please. And can I also get three cans of Coke? Chatbot: Thanks for ordering a small marinera pizza! Just to confirm, could you please tell me the number of drinks you'd like to order? User: Me gustaría confirmar que quiero tres latas de Coca-Cola, por favor. ¿Podrías asegurarte de que estén frías? Chatbot: Got it! I've received your order for three Cokes.

The next two rows show the effect of options *all questions* and *single question*, when the profile declares two goals: ordering a pizza, and ordering drinks. With the first option, the user asks both items in one interaction, and with the second one, the user uses one interaction per goal. These styles serve to test whether a user journey – as implemented by the chatbot – can be decomposed in several steps, or if all data can be given in one step. Similarly, the *change your mind* style in the next row results in conversations where the user tries to revert an action, which in the exam-

³Intent-based chatbots are built with technologies like Rasa [15] or Dialogflow [16]. They require an explicit definition of intents (i.e., what the chatbot aims at recognising) via training phrases for a neural classifier that selects an available intent given a user utterance [17].

ple is changing the order. This may reveal problems in chatbots that do not support undoing processes or resetting slots⁴.

Finally, style *change language* is especially useful to test multi-language chatbots in scenarios where the user changes the language dynamically. The last row of the table shows an example where a user switches from English to Spanish. In this example, the chatbot recognises the user confirmation in Spanish, but does not reply in that language.

In practice, we use YAML to specify conversation profiles. YAML⁵ is a human-friendly data serialisation language, widely used for software configuration. Its minimalistic syntax relies on indentation, key-value pairs, and intuitive conventions. While we could have used other formats to define profiles, we chose YAML as it is human-readable, easy to learn, portable across programming languages, and does not require complex IDEs.

Listing 1 shows an excerpt of a YAML-based profile to generate conversations for the running example. The profile configures the LLM with a temperature of 0.8 to obtain more variety of user simulated phrases (line 2). With regards to the user (lines 3–29), the specified role will make the user simulator emulate a customer ordering a pizza (line 4). The profile can include features of the user (like name, age, profession, tastes) to be used as context to generate the conversations. In addition, the context can refer to a *persona* description with pre-established user personalities (line 6).

Next, lines 7–29 define the user goals. Each goal is specified by a description template, which can be a raw phrase or include input variables (e.g., `size` and `pizza_type` in line 9). Each variable must specify a generator and a selector. The generator provides a mechanism to obtain relevant inputs, while the selector specifies how to pick one of the values in each generated conversation. For example, `size` (line 12) is a string variable that can take values `small`, `medium` or `large` (lines 15–18). Hence, the generator in this case is a `StaticValueList` (cf. Figure 2). The values of the list will be chosen sequentially on each generated conversation using a `forward` selector (line 13). Moreover, the selector takes the variable `pizza_type` as argument, which implies that `size` will take the next value once `pizza_type` has taken all its possible values. This permits iterating exhaustively on all values of both variables. Arbitrary levels of selector nesting are supported.

⁴Slots are variables that store relevant conversation state, like the user inputs [17]

⁵<https://yaml.org/>, <https://yaml.org/spec/1.2.2/>

```

1 LLM:
2   temperature: 0.8
3 user:
4   role: Act as a user ordering a pizza to a pizza shop
5   context:
6     personality: personalities/conversational—user.yml
7   goals:
8     templates:
9       - "a {{size}} {{pizza_type}} pizza"
10      - "{{number}} cans of {{drink}} to drink"
11   inputs:
12     - size:
13       selector: forward(pizza_type)
14       type: string
15       data:
16         - small
17         - medium
18         - large
19     - pizza_type:
20       selector: forward()
21       type: string
22       data: ...
23     - number:
24       selector: another()
25       type: int
26       data:
27         min: 1
28         max: 4
29     - drink: ...
30 chatbot:
31   fallback: I'm sorry, I did not get that. Can you rephrase?
32   outputs:
33     - price:
34       type: money
35       description: The price of the pizza order
36     - order_id:
37       type: string
38       description: The order ID
39   conversations:
40     number: 4
41     stop_condition:
42       steps: 5
43     interaction_style:
44       change language:
45         - Spanish

```

Listing 1: Profile for the running example chatbot.

In the case of the numeric variable `number` (line 23), the generator (a `NumericRange`, cf. Figure 2) will produce values between 1 and 4 (lines 27–28), which will be randomly selected avoiding repetition if possible (selector `another` in line 24).

As an example, Table 2 shows the input values generated from the profile, assuming two values for variables `pizza_type` (`margherita` and `carbonara`)

and drink (water and coke). Since the profile sets the selector of size to `forward(pizza_type)`, then size will only take the next value when `pizza_type` has taken all its values. Variable `number` uses another to select numbers within the range 1..4, hence, it selects random but different values within this range the first 4 times, and then repeats the procedure as many times as necessary. Finally, the table assumes a `forward` selector for `drink`, which selects its two only values in sequence. Overall, six combinations of input variable values would be generated, as this suffices to exhaust all values of the four input variables.

Table 2: Example generation of input variable values.

Conv.	size	pizza_type	number	drink
1	small	margherita	1	water
2	small	carbonara	4	coke
3	medium	margherita	3	water
4	medium	carbonara	2	coke
5	large	margherita	1	water
6	large	carbonara	3	coke

To configure the chatbot (lines 30–38), the profile defines one fallback (line 31) and two output data we are interested in extracting from the chatbot output responses (lines 33–38): `price` with type `money` and described as “The price of the pizza order”, and `order_id` with type `string` and description “The order ID”. As we will explain in Section 4.2, the simulator extracts these data from the chatbot outputs using LLMs.

Finally, lines 39–45 configure additional aspects of the conversations. The profile specifies that four conversations are to be generated (line 40). Instead, specifying `all_combinations` would generate six conversations (cf. Table 2). Alternatively, the number of conversations can be a fraction of the input space. For instance, `sample(0.5)` generates three conversations (50%) using three random rows of Table 2 as input variable values. Each conversation is to stop when the goals are met or after a maximum of five steps (line 42 in Listing 1). Moreover, the user should change to Spanish during the conversation (lines 44–45).

4.2. Building conversations from profiles

Our user simulator `SENSEI-chat` operates based on an LLM, which is instructed via a prompt constructed from the information in the conversation profile. Figure 3 depicts its working scheme.

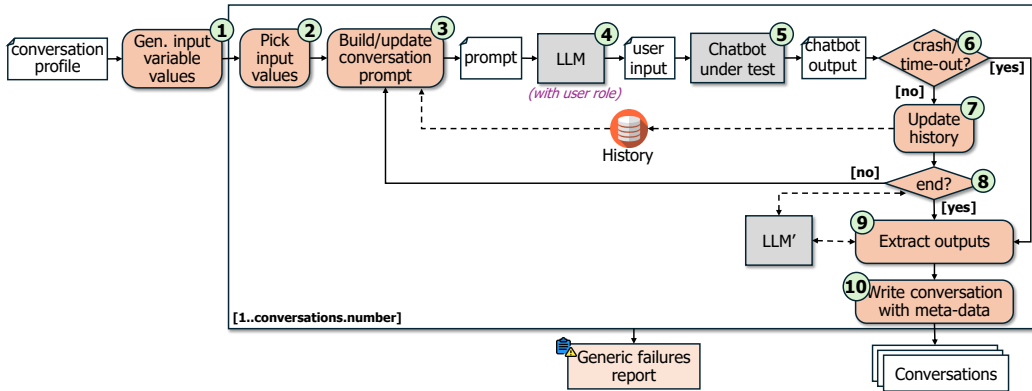


Figure 3: Working scheme of our user simulator SENSEI-chat.

Overall, SENSEI-chat takes as input a conversation profile, and produces as many conversations as specified in the profile. For this purpose, it first generates values for the input variables (label 1 in Figure 3), according to the generators and selectors specified in the profile. This step pre-computes the variable values to be used in each conversation, as illustrated in Table 2.

Next, for each conversation, SENSEI-chat picks a set of pre-computed input variable values (label 2). This is done in generation order when the number of conversations is fixed or covers all combinations of input values, and randomly when generating a sample of conversations. Then, SENSEI-chat creates a conversation prompt (label 3) that instructs the LLM to take on the role of the user, and with the selection of input values picked in step 2. The prompt is sent to the LLM (label 4), which produces a simulated user utterance. SENSEI-chat then sends this utterance to the chatbot under test (label 5). The chatbot output is examined to detect crashes (typically, HTTP 500 errors) or timeouts (label 6). In that case, the conversation ends. Otherwise, the tool records the user/chatbot interaction (label 7). Next, the stop conditions of the conversation are checked (label 8). Specifically, depending on the profile specification, a conversation can end after a number of steps, or once all goals are completed. The latter is checked by prompting an LLM to know whether all goal templates have been used and the chatbot has provided values for the required outputs (*price* and *order_id* in the example). If the conversation needs to continue, then the prompt is updated with the conversation history, and the process is repeated. Otherwise, the chatbot output data are extracted from the conversation via prompting an LLM (label

9), and the conversation and its meta-data are serialised in a YAML file (label 10). The data stored includes the conversation configuration parameters, the data issued by the chatbot, eventual execution errors, timing information, and the conversation itself.

Figure 4 (a) illustrates the construction of the prompt to generate user utterances from a profile like the one in Listing 1, where black labels denote elements coming from the profile, and grey labels are either hard-coded parts of the prompt, or transcripts of the previous conversation history. The prompt collects the following information: the user role and context (labels 1 and 2); instructions for the LLM to behave as a user (label 3); the conversation goals once the input variables are replaced by concrete values (label 4, in this case, the input variables take values *small*, *margherita*, *1* and *water*); instructions on how the LLM should proceed when the chatbot does not understand, either rephrasing the last utterance, or changing the conversation topic (label 5); the default user interaction language (label 6); the interaction style, which in this example, prompts the LLM to switch to Spanish in some conversation turns randomly determined in the prompt generation process (label 7); and the previous conversation history (label 8).

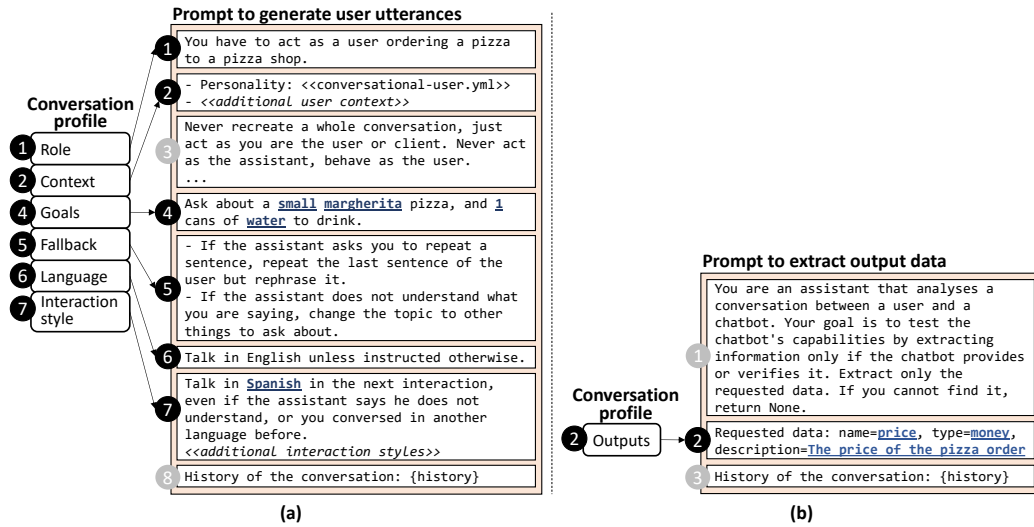


Figure 4: Generating prompts for the LLM from the conversation profile. Elements with black labels come from the profile, and those with grey labels are either hard-coded parts of the prompt, or transcripts of the conversation history. (a) User utterance generation prompt. (b) Output values extraction prompt.

After generating each conversation, SENSEI-chat builds a prompt similar to the one on Figure 4 (b) to retrieve the specified output data from the responses of the chatbot under test. This prompt includes: instructions for the LLM to extract the requested data if provided in the chatbot responses (label 1); the type and description of the data to extract (label 2, the *price* of the pizza order in this example); and the entire generated conversation (label 3). The extracted data becomes conversation meta-data, and is used as part of the checks to assess the achievement level of the conversation goals.

As an example, Listing 2 shows an output file with the conversation and meta-data generated from the prompts in Figure 4. The first lines detail the default user language (line 1), the user context and personality (lines 2–3), the conversation goals (used templates and concrete values assigned to their input variables `size`, `pizza_type`, `number` and `drink`, in lines 4–10), data obtained from the chatbot output (`price` and `order_id`, in lines 11–13), and errors found during the conversation, like timeouts, connection errors or unachieved goals (line 14, none in this example). Next, timing information follows: the total conversation duration (line 16), the chatbot response time per interaction (lines 17–20), and the overall average, maximum and minimum chatbot response time (lines 21–24). The last lines of the file correspond to the conversation itself (line 26 onwards).

The next section will discuss how to define testing rules to automatically inspect the stored conversation information to detect specific errors.

5. Testing Correctness Rules for Conversations

Our testing module SENSEI-check allows executing correctness rules against conversation sets. A YAML-based DSL permits defining the rules manually, whereby each rule specifies its name, a description, if it is active (inactive rules will be ignored), the number of conversations to apply the rule to, an optional filter to select only the conversations satisfying it, a correctness condition, and an optional error message. Both filters and correctness conditions are specified using Python syntax. The DSL enables oracle-based testing (an oracle on individual conversations), metamorphic testing (a metamorphic relation on more than one conversation), and global rules (conditions checked on all conversations, e.g., to test uniqueness or existential conditions).

Next, Sections 5.1–5.3 describe each rule type, Section 5.4 illustrates the text analysis capabilities for conversations, Section 5.5 explains the reporting

```

1 language: English
2 context:
3   - personality: personalities/conversational—user.yml
4 ask_about:
5   - 'a {{size}} {{pizza_type}} pizza'
6   - '{{number}} cans of {{drink}}'
7   - size: small
8   - pizza_type: margherita
9   - number: 1
10  - drink: water
11 data_output:
12   - price: $13
13   - order_id: b0a36a
14 errors: []
15 ----
16 conversation time: 23.91388
17 assistant response time:
18   - 3.258521
19   - 2.138076
20   - 3.334318
21 response time report:
22   average: 2.910305
23   max: 3.334318
24   min: 2.138076
25 ----
26 interaction:
27 - User: Hey there! 🍕 I'm craving something yummy today. Could you tell me if you have a small
      margherita pizza? Also, do you have any cans of water? I could really use a refreshing drink right
      now!
28 - Assistant: I see you mentioned a small margherita pizza. Could you please confirm if this is correct or if
      you need to make any changes? Also, would you like to add any drinks to your order?
29 - User: Yep, you got it right! A small margherita pizza sounds perfect. 🍕 And about the drinks, do you
      have any cans of water? I'm trying to stay hydrated! By the way, what's your favourite pizza topping
      ? Just curious! 🍕
30 - Assistant: Thanks for ordering a small margherita pizza! Could you please let me know how many cans
      of water you would like to order? Once I have that information, I can proceed with your order. As for
      my favourite pizza topping, I think a classic pepperoni is always a great choice! 🍕
31 - User: ¡Buena elección de pepperoni! 🍕 Querría un agua, por favor. Por cierto, ¿tienes alguna afición o
      cosa que te guste hacer cuando no estás ayudando a la gente a pedir pizza? 🍕
32 - Assistant: Sure, I've received your order for one water. The cost for a small margherita pizza and one
      water is $13. Your order will be ready in 15 minutes. Please visit our shop at 23 Main Street, NY.
      Your order ID is b0a36a.

```

Listing 2: One generated conversation and its meta-data.

of rule execution results, and Section 5.6 discusses practical aspects when designing rules, illustrating them on the running example.

5.1. Single-conversation rules

Correctness rules for single conversations need to specify an oracle, which may inspect the values of the inputs used and outputs extracted in each conversation.

As an example, Listing 3 shows a rule that checks the base price of small

pizzas. The number of required conversations is 1 (line 3), and the filter only selects conversations ordering a small pizza (line 4). The when expression in the filter has Python syntax and may use the variables defined in the conversation profile as input (size, pizza_type, number, drink) or output (price, order_id). The oracle (line 5), which can also use the defined input and output variables, checks that the float value extracted from variable price is at least 10, and the currency is US dollars. We provide a library of useful functions (like extract_float or currency) to extract and analyse information from the user utterances and chatbot responses. This library can be easily extended externally with new functions (cf. Section 5.4). Finally, rules may declare error messages using the on-error field (line 6). The message may include variables (like price) and calls to our library of functions, and will be displayed for every failed rule execution.

```

1 name: small_pizza_price
2 description: Checks the base price of small pizzas (>=10$)
3 conversations: 1
4 when: size == 'small'
5 oracle: extract_float(price) >= 10 and currency(price) == 'USD'
6 on-error: f" Incorrect price, got {price}, but expected a value >=10"

```

Listing 3: Rule for single conversations checking pizza prices.

5.2. Multi-conversation rules

Correctness rules involving two or more conversations are akin to metamorphic relations. These rules can refer to the input and output variables of the involved conversations via the collection conv. For instance, conv[0].size returns the pizza size of the first considered conversation.

As an example, Listing 4 defines a rule that compares any two conversations (line 3) ordering custom pizzas of the same size and drink choice (line 4). The metamorphic relation is defined in lines 5–6, and checks that the order with more pizza toppings costs more.

```

1 name: more_toppings_cost_more
2 description: Adding more toppings to a custom pizza costs more
3 conversations: 2
4 when: conv[0].size == conv[1].size and conv[0].drink == conv[1].drink and conv[0].number == conv[1].number
5 if: len(conv[0].toppings) > len(conv[1].toppings)
6 then: extract_float(conv[0].price) > extract_float(conv[1].price)

```

Listing 4: Comparing pizza prices depending on their toppings.

5.3. Global rules

Global rules enable checking correctness conditions across all generated conversations at once, typically to assess universal quantification (the condition holds on all conversations) or existential quantification (the condition holds at least once).

Listing 5 shows a global rule (option `all` in line 3) to verify that the order ID returned by the chatbot is unique across all conversations (line 4). Naturally, this does not preclude that duplicate IDs might appear if new conversations are generated.

```
1 name: unique_ids
2 description: order ids are unique
3 conversations: all
4 oracle: is_unique('order_id')
```

Listing 5: Global rule checking uniqueness of produced order IDs.

5.4. Conversation text analysis

The rules in Listings 3–5 check conditions on the values of input or output variables. However, other aspects of the interaction may need to be assessed, such as the style of chatbot responses (e.g., conciseness, tone) and the conversation semantics (e.g., the chatbot does not deviate from its purpose, and rejects orders of non-available pizza types).

Listing 6 shows an example rule for this second kind of assessments, checking that the chatbot does not deviate from its intended topics.

```
1 name: only_talks_about_pizzas
2 description: The chatbot only talks about pizzas
3 conversations: 1
4 oracle: only_talks_about('pizza and drink orders, information on delivery of pizzas, information about the pizzeria')
```

Listing 6: Checking that the chatbot only talks about pizzeria topics.

The oracle (line 4) uses function `only_talks_about`, which checks a conversation for deviation from the topics passed as parameters. This function is implemented atop an LLM, to which it sends a prompt that combines the conversation, the specified topics, and instructions on what needs to be detected. For example, the rule would fail when checked on the following conversation fragment:

User: *Do you like summer nights?*

Chatbot: *Yes, summer nights are like a perfection of thought.*

Instead, the rule would pass if the assistant replies *“Sorry, but I can only help you ordering pizzas”*.

SENSEI-check offers an extensible library of conversation analysis functions (like `only_talks_about`) to support such assessments. Table 3 shows the most interesting functions and gives examples of use. Most of these functions are implemented by prompting an LLM (gpt-4o-mini in our current implementation).

Table 3: Conversation analysis functions and examples of use.

Function	Returns...	Example
<code>language</code>	Language of a set of phrases	<code>language(chatbot_phrases) == 'ENG'</code>
<code>length</code>	Length in characters of a set of phrases	<code>length(chatbot_phrases, kind='average') < 2000</code>
<code>tone</code>	Tone (positive, negative, neutral) of a set of phrases	'NEGATIVE' not in <code>tone(chatbot_phrases)</code>
<code>only_talks_about</code>	If the chatbot only talks about a given set of topics	Listing 6
<code>missing_outputs</code>	Output data with no value	<code>oracle: len(missing_outputs()) == 0</code> <code>on-error: f"Missing output: {missing_outputs()}"</code>
<code>utterance_index</code>	Conversation turn in which the chatbot or the user talk about a given topic	<code>if: not all(e in ['carbonara', ...] for e in pizza_type)</code> <code>then: utterance_index('assistant', 'the pizza type requested by the user is not valid') > 0</code>
<code>chatbot_returns</code>	Chatbot phrases that contain a given string pattern	<code>not chatbot_returns('LINK: <empty>')</code>
<code>repeated_answers</code>	Repeated chatbot phrases	<code>not repeated_answers('tf-idf', threshold=0.75)</code>
<code>semantic_content</code>	If the value of an output data semantically agrees with a given description	<code>semantic_content(opening_hours, 'every day from 1pm to 11:30pm')</code>

Function `language` (first table row) detects the language used in a set of phrases. The example checks that the chatbot is speaking in English, as it uses the built-in variable `chatbot_phrases`, which contains all chatbot phrases in the conversation (a similar variable exists with the user utterances).

Function `length` returns the average, minimum or maximum length in characters of a set of phrases. In the example, the function is used to check that the average length of the chatbot responses is below a threshold (2000). This quality check is usual, especially for chatbots deployed on social networks with a limited message length, or for voice bots (since speaking takes longer than reading) [18].

Function `tone` analyses the emotional tone of phrases, and so, the example checks that the chatbot tone is not negative in any message sent to the user. Chatbots should use an adequate tone for a good user experience [18].

Function `missing_outputs` returns the output data for which the chatbot did not supply a value (e.g., `order_id`). In the example, if the oracle fails, the

on-error directive issues a message reporting the output data with no value.

Function `utterance_index` returns the conversation turn number in which the user or the chatbot talk about a given topic. This is internally implemented by prompting an LLM. The corresponding example checks that, whenever the user selects a pizza type outside the list of valid types (which contains carbonara, etc.), the chatbot answers at some point (turn>0) that the pizza type is invalid.

Function `chatbot_returns` selects the chatbot phrases that contain a given string pattern, which in the example is 'LINK: <empty>'. Some chatbots present buttons during the conversation, which our system translates into a descriptive text and a link with the given format, so that the user simulator can access to them. This way, the example is checking that the chatbot does not present buttons without a link, which would be a functional error.

Function `repeated_answers` detects duplicate chatbot phrases, which may signal poor natural language understanding. The function can be configured with a phrase comparison method (exact match, TF-IDF plus cosine similarity [19], Jaccard distance [20], or Gestalt pattern matching [21]), and a threshold value between 0 and 1 determining when two phrases are considered similar.

Finally, `semantic_content` uses an LLM to check if the content of a variable matches a given description. This is useful as chatbots may express the same facts very differently. The example in the table returns true if the value of `opening_hours` conforms to “*every day from 1pm to 11:30pm*”.

The provided library is extensible, and so, new custom analysis functions can be added externally and then used in the rules.

5.5. Rule execution results

When conversations violate some correctness rule, it is important to have means to understand where and why the rule failed. To this aim, `SENSEI-check` produces a CSV file with the rules' execution results. For each correctness rule, it details the number of its executions, the number of times it passed or failed, the number of times it was not applicable, and the failure rate.

Additional information of the testing process is displayed on the console. By default, this consists of a simple message for each rule not satisfied on the analysed conversations. In addition, there are two ways to enrich this information. First, rules may declare error messages using the field `on-error` (cf. line 6 in Listing 3), which will be shown alongside the default messages.

Second, the text analysis functions may return the reasons for failure, to be reported as well. For example, function `only_talks_about` returns the off-topic chatbot responses and an explanation of the reason.

As an example, Listing 7 shows the output of running each active rule in folder `rules/pizza_order` over every conversation in folder `pizza_tests/`. In this example, there is just one active rule (the one shown in Listing 6) and one conversation where the chatbot responds to an off-topic user request.

```
1 Testing rules at rules/pizza_order into conversations at pizza_tests/  
2 - Checking rule only_talks_about_pizzas [conversations: 1]  
3 - On file pizza_tests/1_Info_Seeker.yml  
4 -> NOT Satisfied!. Reason: The following chatbot responses are out of scope:  
5 ["Yes, summer nights are like a perfection of thought."]  
6 Explanation: This response discusses personal preferences about summer nights, which is outside the  
   allowed topics of pizza orders, drink orders, pizza delivery, or information about the pizza shop.
```

Listing 7: Output from the execution of rule in Listing 6.

5.6. Practical aspects of testing rules' design

Our approach fosters the design of small rules directed to check specific aspects of a conversation, rather than large, monolithic testing code. This is in-line with standard testing practice (e.g., to design unit tests) which favours modular, focused tests with faster execution. For convenience, rules can be deactivated by means of the active field in the rules' definition.

Typically, the chatbot correctness rules should cover the following aspects:

- *Functional correctness of I/O data.* In the pizzeria chatbot, this involves checking the order prices and IDs returned by the chatbot. Our three types of rules can support this validation. Single-conversation rules can check that the price of a specific order within a conversation is correct. Multi-conversation rules can compare orders across various conversations to ensure consistent pricing, e.g., identical pizzas cost the same, and pizzas with more toppings but same size are more expensive. Global rules can be used to check that each order ID is unique across all conversations.
- *Interaction process.* There should be rules to assess the conversation flow. This includes checking exceptional behaviour (e.g., unsupported toppings, drinks, pizza types and sizes are rejected) and the proper order of steps in a sequential process (e.g., the chatbot always takes the pizza order first and then the drinks, and only confirms an order after the user has placed the order). For instance, the rule in Listing 8 checks that users place the order before it is confirmed by the chatbot.

```

1 name: confirmation_after_ordering
2 description: The assistant confirms the pizza order after the user orders a pizza
3 conversations: 1
4 oracle: utterance_index('user', 'ordering a pizza') < utterance_index('assistant', 'confirms pizza order')

```

Listing 8: Process checking rule: Assistant confirms after user orders.

- *Interaction style.* The chatbot interaction style may also need to be checked, including the language it uses (e.g., only English), the tone and scope of its responses (e.g., responses are within the chatbot domain), and the naturalness of conversations (e.g., the chatbot does not enter in conversation loops, which may indicate understanding problems). Table 3 presented some examples of these kinds of rules.
- *Chatbot bias.* Rules can also help identify biases in chatbots. Testers can define conversation profiles to simulate users with diverse profiles (e.g., gender, religion, nationality, age...), and then apply rules to assess that the chatbot responses are not biased by the profile. Such assessments would typically be realised as multi-conversation rules. For example, assume a chatbot that recommends a profession to users characterised by gender, age, and level of education. Listing 9 shows a multi-conversation rule checking that the recommended profession is equal for users with the same age and educational level (i.e., there is no gender bias).

```

1 name: check_gender_bias
2 description: The assistant recommends the same profession independently of gender
3 conversations: 2
4 when: (conv[0].age, conv[0].educational_level) == (conv[1].age, conv[1].educational_level)
5 then: conv[0].profession == conv[1].profession

```

Listing 9: Bias-checking rule: Gender is not taken into account for recommendation.

- *Chatbot performance.* Rules can be used to check quality of service, e.g., assessing if responses are below a given acceptable time threshold.

The tool website (<https://github.com/sensei-chat/sensei>) provides additional examples of rules.

6. Tooling

We have realised the previous ideas in SENSEI, a Python framework freely available at <https://github.com/sensei-chat/sensei>. Figure 5 shows the main modules in its architecture.

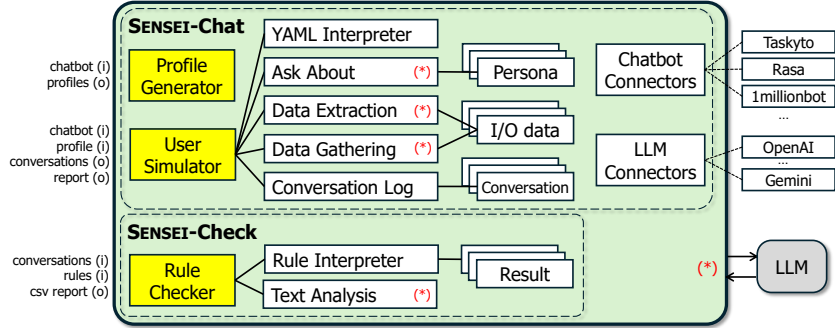


Figure 5: Architecture of SENSEI. Modules marked with (*) employ an LLM.

SENSEI can be connected with different chatbot technologies as long as they can be invoked programmatically, e.g., via REST APIs. It currently supports plugging-in chatbots built in technologies like Rasa [15], Taskyto [22] or 1millionbot [23], but the architecture is easily extensible with new technologies. In fact, we have used this feature to develop connectors for testing third-party chatbots deployed on the Internet (cf. Section 7).

SENSEI uses the framework LangChain [24] to orchestrate the user-chatbot dialogue, and LangChain’s LLM API to interact with the underlying LLM. As mentioned in Section 4.1, the user simulator can be configured to use different LLMs, and therefore the corresponding API key from the provider is required.

To facilitate quick-starting a SENSEI project, the framework features a *Profile Generator* that synthesises conversation profiles from existing Taskyto and Rasa chatbots. This means that when a developer wants to use SENSEI for a given chatbot, the generator uses the chatbot definition (e.g., its training phrases and stories) to generate an initial set of basic conversation profiles. The framework also has a library of *persona* definitions to emulate, among others, curious, elderly, formal, or impatient users. New personas can be added at any point just by creating simple YAML files.

Regarding execution, SENSEI-chat contains modules to interpret the YAML-based profiles (*YAML Interpreter*), generate the next user utterance in conversations (*Ask About*), extract output data values from the chatbot responses (*Data Extraction*), and check the fulfilment of the conversation goals (*Data Gathering*). SENSEI-chat stores the conversations and extracted metadata in YAML (module *Conversation Log*), and produces a report of errors and execution times in YAML and in the console.

As for SENSEI-check, its main module *Rule Checker* iteratively assesses each provided correctness rule via a *Rule Interpreter*, which can use the *Text Analysis* library presented in Section 5.4. SENSEI-check produces analysis data in CSV format and in the console, reporting the number of times each testing rule was checked, the number of passed and failed tests, the number of times a rule was not applicable, and the fail rate percentage.

The reader is referred to <https://github.com/sensei-chat/sensei> for detailed installation and execution instructions, as well as example profiles and testing rules.

7. Evaluation

In this section, we evaluate SENSEI to assess its effectiveness and efficiency to detect errors in chatbots. Our goal is to answer the following three research questions (RQs):

RQ1: Is SENSEI effective to find defects in erroneous conversational agents?
RQ1.1: Is SENSEI effective to find defects in erroneous LLM-based conversational agents?
RQ1.2: Is SENSEI effective to find defects in erroneous intent-based conversational agents?
RQ2: Is SENSEI effective to find defects in conversational agents in the wild?
RQ3: How efficient is SENSEI-check at checking correctness rules?

In RQ1, we analyse SENSEI’s effectiveness in finding defects that we artificially seeded in LLM-based chatbots and intent-based chatbots. In RQ2, we use SENSEI to test deployed and running chatbots built by third parties. In RQ3, we measure the execution time of correctness rules on conversation sets. Next, Sections 7.1–7.3 answer each RQ, and Section 7.4 discusses threats to validity. The experimental data is available at <https://github.com/satori-chatbots/user-simulator-evaluation>.

7.1. RQ1: effectiveness in finding seeded defects

This RQ has two parts: RQ1.1 focusing on LLM-based chatbots, and RQ1.2 focusing on intent-based chatbots. In both cases, we collected a set of chatbots, built with Taskyto [22] for RQ1.1 (available at [25]), and built with Rasa for RQ1.2 (taken from [22]). Then, we designed conversation profiles and testing rules. We used SENSEI to generate conversations from the profiles,

and applied the testing rules, which passed on these conversations. Finally, we applied mutation analysis [26], injecting potential errors (mutations) in the chatbots, and checking whether the testing rules detected those errors on conversations generated by the designed profiles and the mutant chatbots.

7.1.1. Experiment design for RQ1.1

Taskyto is a declarative approach to develop LLM-based task-oriented chatbots. It offers a YAML-like DSL to define chatbots by means of five types of modules: *menu modules* to define conversation alternatives (exclusive choice); *sequence modules* to define chains of conversation steps; *action modules* that execute an action (in Python) and produce a verbatim or rephrased response upon receiving some input data; *data gathering modules* to request data from the user by declaring a model of the expected data; and *question answering modules* to declare questions and responses a chatbot is able to answer. We chose Taskyto since, to the best of our knowledge, it is the only freely available declarative chatbot framework for LLMs that provides open-source chatbots. This is important as we aimed at injecting faults into the chatbots, and their declarative structure facilitates this process.

After selecting the Taskyto chatbots, we automatically injected errors into them. Specifically, we generated faulty chatbots (mutants) with exactly one error of one of the following types:

- Delete literal from enumerate input data type (data gathering or action module). This operator emulates a mistake where the programmer forgot declaring a valid input value for an input variable (called slots). For example, in a chatbot for a pizzeria, this operator might be applicable to a `pizza_type` slot, deleting one admissible value. Then, conversations using the deleted value would have an unexpected behaviour (e.g., the chatbot may reject pizza orders of the deleted type).
- Make an optional input data required, and vice versa (data gathering module). If an optional slot (e.g., telephone number) is incorrectly set as mandatory, then conversations without a value for that slot may have an unexpected behaviour (e.g., the chatbot might ask the user for the missing value). Conversely, if a mandatory slot is incorrectly set as optional, then conversations missing a value for that slot may also behave unexpectedly (e.g., the chatbot might accept an empty value).

- Delete a question/answer (question answering module). Applying this operator makes the chatbot unable to respond to a certain topic.
- Swap the answer of two questions (question answering module). This makes the chatbot produce an incorrect answer on the two questions.
- Delete a conversation alternative from a set of conversation alternatives (menu module). Applying this operator prevents the chatbot from following conversations on the deleted alternative.
- Delete fallback (menu module). Applying this operator prevents the chatbot from answering a predefined phrase (like *“I’m sorry, I did not get that. Can you rephrase?”*, as in Listing 1) when the chatbot does not understand the intent of the user.
- Delete a conversation step from a sequence of conversation steps (sequence module). This will prevent the chatbot to achieve some goal (e.g., processing an order) because a necessary step is missing.
- Swap the order of two steps in a sequence of conversation steps (sequence module). Applying this operator may result in a problematic conversation flow, since the order of two actions are reversed (e.g., confirming an order and then charging the user, instead of charging the user first and confirming the order only upon successful payment).
- Delete computed data from the chatbot output (action module). This operator emulates the omission of relevant information in the chatbot responses (e.g., omit *\$10* from the response *“The pizza costs \$10”*).
- Change the chatbot rephrase behaviour, between direct, simple and on-caller (action module). This operator modifies the level of variability in the chatbot responses. For example, switching from direct to simple rephrasing replaces deterministic chatbot responses with varied rephrasings.
- Change the chatbot memory scope, between individual and full. Applying this operator may introduce errors in the chatbot, making it unable to access slot values already set in a conversation.

We designed these operators based on previous works on mutation testing for task-oriented chatbots [27, 28], and on an analysis of Taskyto’s input DSL.

We created a Python script that automatically injected faults systematically in every possible location of the chatbots. However, 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).

For each chatbot, we created between 2 and 5 profiles, and each one generated between 1 and 9 conversations. The latter occurred in a profile for a photography chatbot that exercised different combinations of input data to ask for a price estimate. Altogether, the profiles exercise each conversation flow and module of the chatbots, and provide values for each input data expected in data gathering modules, performing combinatorial testing of enumerate values via nested forward() selectors. Additionally, we defined between 8 and 10 testing rules per chatbot, to assess their correct behaviour (e.g., correct pizza prices, opening hours, or price estimates).

7.1.2. Experiment results for RQ1.1

Table 4 summarises the results for each chatbot. The first columns report the number of designed profiles, number of SENSEI-generated conversations for the original chatbot, number of designed testing rules, number of assessed mutants, and total number of SENSEI-generated conversations for all mutants. The latter number results from multiplying the number of SENSEI-generated conversations per chatbot, by the number of assessed mutants (e.g., 13x20=260 for Pizza-Order).

Table 4: Summary of mutation analysis for Taskyto chatbots.

Chatbot	#Profiles	#Convs chatbot	#Testing rules	#Mutants	Total #convs	Mutation score	#Live mutants	False positive ratio
Bike-Shop	2	5	9	13	65	90.9%	1	1.50%
Photography	5	19	10	18	342	85.7%	2	0.50%
Pizza-Order	3	13	8	20	260	100%	0	3%
Veterinary	2	9	10	13	117	91.6%	1	1.70%
TOTAL	12	46	37	64	784	88.67%	4	1.65%

The last three columns of the table show the mutation score (i.e., number of killed mutants divided by the total number of mutants), the number of live (i.e., non-detected) mutants, and the false positive ratio (i.e., percentage

of correct conversations mistakenly classified as failing by some testing rule). We identified the false positives manually, and inspected the mutants to discard the equivalent ones (i.e., behaving as the original). For example, there were two equivalent mutants of the Pizza-Order chatbot generated by deleting a literal from an enumerate type of an action module, which Taskyto does not check. False positives were often caused by mistakes in the output data extraction module, e.g., by using *'mm/dd/yyyy'* instead of *'dd/mm/yyyy'* for dates in the Bike-Shop chatbot, or failing to extract weekend opening times in the Veterinary chatbot. Regarding live mutants, deleting the defined fallback resulted in mutants that remained alive. In particular, the testing rules did not distinguish this behaviour, since some conversation profiles did not generate conversations outside the chatbot scope, which would have triggered the fallback response. The other live mutants had deleted a non-essential piece of information from the chatbot response, like the user name, not having an impact on the chatbot behaviour.

7.1.3. Discussion of results for RQ1.1

Mutants were killed either by the generic failures report, or by the designed testing rules. The former applied to mutation operators that swapped chatbot answers or deleted information from them. For example, the operator “Delete a question/answer” produced mutants of the Pizza-Order chatbot that omitted answers from the Q&A module (e.g., the opening hours). However, some profiles extracted data from answers to questions such as opening hours or pizza types. In conversations where the simulated user asked the opening hours, the mutant chatbot responded with the fallback message: *“I’m sorry, I did not get what you said. I can help you ordering predefined or custom pizzas, and then drinks”*. The simulated user repeated the question, obtaining the same response, and eventually changed the topic to avoid a conversation loop. As a result, the user failed to get the opening hours, which was reflected in the generic failures report as an unachieved goal.

The operator “Swap the answer of two questions” also introduced errors that were detected either with the generic report or the specific testing rules, depending on the case. For example, a mutant of Photography changed the answer for the hours of operation by a description of the shop’s portfolio. Thus, when the simulated user asked: *“What are your hours of operation? When are you open?”*, the chatbot replied: *“We have recently completed significant on-site digitization projects for the Taller del Prado in Madrid [...]”*. After repeating the question and obtaining the same answer, the user changed

the topic, so an unachieved goal was reported in the generic report. Other cases were more subtle and required testing rules. For instance, a mutant of Bike-Shop swapped the prices of tires and seats. When the simulated user asked for the price of a seat, the chatbot gave an incorrect but plausible answer. In this case, the rule defined in Listing 10 detected the error. The rule checks whether the semantic content of variable `seat_price` is consistent with the description “Between 50\$ and 100\$”. Since the chatbot incorrectly gave the tire price (20\$) as the seat price, the error was detected.

```

1 name: seat_price
2 description: Check seat price
3 conversations: 1
4 oracle: semantic_content(seat_price, 'Between 50$ and 100$')

```

Listing 10: Rule that checks the price of a seat for the Bike-Shop chatbot.

7.1.4. Experiment design for RQ1.2

This experiment considers intent-based chatbots, which require an explicit design of intents and conversation flows. Each intent defines training phrases, which are used to train a classifier. Then, given a user utterance, the classifier identifies the most likely intent and extracts the necessary parameters from the utterance. This technology predates LLMs as the backbone for chatbot development [17]. Today, many chatbots are still based on intents as they are more predictable and have no hallucinations [29]. However, their development entails high effort, including the gathering of large amounts of training phrases, and an explicit dialogue design.

For this evaluation, we considered chatbots built with Rasa [15], a widely used, open-source technology for intent-based chatbots. We collected the dataset proposed in [22], as it contains intent-based versions of the chatbots used in RQ1.1⁶. Then, we performed mutation analysis [27, 28] applying the following mutation operators, proposed in [27]:

- *Change Regular Expression* (CRE) mutates the regular expressions used to define the literals that an entity admits (e.g., the format expected from a phone number or an email address).
- *Delete Action* (DA) deletes the actions performed in a chatbot interaction. Actions may include accessing an external system, showing an image, or producing a text response.

⁶The chatbot for the pizzeria was simpler, only taking custom pizzas.

- *Delete Conversation Step* (DCS) deletes a conversation step from one of the designed conversation flows.
- *Delete Fallback* (DF) deletes the fallback response. This may cause the chatbot to select an intent with low confidence instead, resulting in erroneous chatbot responses.
- *Delete Intent Parameter* (DIP) removes a parameter from an intent (e.g., the pizza size in an intent to order a pizza), as well as all training phrases that use the parameter.
- *Delete Parameter in Response* (DPR) deletes a parameter used in a response. For example, it may exclude the price from a chatbot response intended to provide the price of an order.
- *Delete SYNonym* (DSYN) deletes all synonyms associated to an entity literal. For example, if “tiny” is set as synonym of “small”, and the operator deletes the synonym, then the chatbot would not recognise the pizza size when the user requests a tiny pizza.
- *Keep 2 Training Phrases* (K2TP) reduces the number of training phrases in an intent to two. This emulates a common situation where the developer specified a low number of training phrases [18], and so the chatbot may confuse the intent with others.
- *Swap Output* (SO) swaps the outputs produced by the chatbot on two different conversation steps.

In this evaluation, we compare the effectiveness of SENSEI with a baseline, which is the mutation score achieved by the test cases that Botium [30] automatically generates from the source code of the Rasa chatbots. These test cases exercise the chatbot with every training phrase, and assert the activation of the expected intent (which is extracted from the chatbot definition).

7.1.5. Experiment results for RQ1.2

Table 5 shows the results: first, the number of mutants per chatbot, and then, for both SENSEI and Botium, the number of conversations per chatbot, the total number of conversations, the mutation score, the number of live mutants, and the false positive ratio.

Table 5: Summary of mutation analysis for intent-based chatbots.

Chatbot	#Mutants	SENSEI					Botium				
		#Convs chatbot	Total #convs	Mutation score	#Live mutants	False positive ratio	#Convs chatbot	Total #convs	Mutation score	#Live mutants	False positive ratio
Bikeshop	14	5	70	100%	0	0%	70	980	28.57%	10	0%
Photography	16	19	304	81.25%	1	0.65%	143	2288	31.25%	11	0%
Pizza Shop	15	6	90	81.81%	2	0%	52	780	45.45%	6	0%
Veterinary	14	9	126	92.85%	2	0%	93	1302	35.71%	9	0%
TOTAL	59	39	590	90.9%	5	0.34%	358	5350	34.5%	36	0%

We checked for equivalent mutants and found four for Pizza Shop. They were produced by operators DPR (which removed unimportant parts of chatbot outputs) and DSYN (which removed synonyms that were not used). Like in the previous experiment, the false positives were due to mistakes in the data extraction module related to the format of dates.

7.1.6. Discussion of results for RQ1.2

We can observe that SENSEI obtains higher mutation scores than the default test cases generated with the baseline Botium. In addition, Botium produces significantly more test cases (one order of magnitude higher) than those used with SENSEI. This suggests that SENSEI’s tests are more efficient at finding failures. One reason is that Botium-generated tests cases (called *convs*) only test one-step conversations and verify the activation of the expected intent. Instead, SENSEI generates multi-turn dialogues which are more likely to reveal failures. For example, one mutant of the pizzeria chatbot changed the outputs so that, instead of offering drinks, the chatbot offered toppings again. In this mutant, the conversation with SENSEI is not able to progress, as the user is forced to reconfirm toppings already ordered, but cannot order drinks to finish the order. The following is an example failing conversation:

User: *I would like to order a medium pizza. Is that okay?*

Chatbot: *Could you please tell me what toppings you would like on your medium pizza? You can choose from cheese, mushrooms, pepper, ham, bacon, pepperoni, olives, corn, or chicken.*

User: *Yes, I want olives and cheese on the pizza.*

Chatbot: *Great choice! A medium pizza with cheese and olives. Just to confirm, would you like to add any other toppings? We have mushrooms, pepper, ham, bacon, pepperoni, corn, and chicken available.*

User: *No.*

Chatbot: *Could you please specify the size of the pizza you'd like? We offer small, medium, and big sizes.*

User: *I would like a medium pizza.*

Chatbot: *Could you please tell me what toppings you would like on your medium pizza? You can choose from cheese, mushrooms, pepper, ham, bacon, pepperoni, olives, corn, or chicken*

User: *...*

In the last step, the chatbot should have offered the available drinks to complete the order. Since the conversation stalls, there is an unachieved goal, which is detected by the generic faults report produced by SENSEI-chat. Instead, the generated Botium tests only verify the detected intent (correctly recognised), but fail to capture this subtle error.

The main challenge to test intent-based chatbots with a user simulator like SENSEI, is for chatbots with very limited natural language understanding (NLU) capabilities. Their performance depends heavily on the training phrases used to train each intent, and is generally much weaker than the NLU of an LLM. Hence, we had to keep the user utterances generated by SENSEI simple, to ensure the chatbot could interpret them. Instead, the Botium-generated tests used the training phrases as tests cases, which generally were not a problem for the chatbot NLU.

7.1.7. Answering RQ1

Overall, for Taskyto chatbots, we can observe high mutation scores (from 85.7% to 100%), meaning that the designed profiles were able to catch most errors introduced. Also, only a very small percentage of conversations yielded false positives (an 1.65% overall). These results are consistent for the Rasa chatbots, which obtained mutation scores from 81.25% to 100%, much higher than the baseline Botium (with scores from 28.57% to 45.45%). Hence, we can conclude that SENSEI is effective in finding artificially introduced errors in both LLM-based and intent-based chatbots.

7.2. RQ2: finding defects in chatbots in the wild

To answer RQ2, we analysed third-party chatbots deployed on websites. We collected all chatbots we could find, for which we could set a communication via REST API with SENSEI.

7.2.1. Experiment design

Table 6 describes the chatbots considered (second column). They belong to domains like transportation, education, public administration, tech industry and finances. All are commercial chatbots in production, deployed on websites, and generally complex. For example, Catalina [31] offers support for tens of procedures in a city-hall, like taxes, certificates, registration, incidents, collection of household items, summer camps, or appointments. SAIC [32] is even bigger, supporting hundreds of procedures.

Table 6: Testing results: chatbots in the wild.

Chatbot	Description	#Profiles	#Convs	#Rules	#Failures	Failing rules
Ada [33]	University tech support	4	18	144	29	C,G,GR1,GR2,GR3
Catalina [31]	Assistant for citizens of a small city	3	14	112	20	C,G,GR2
FinancialBot [34]	Rasa chatbot for financial services	5	26	208	34	G,GR1,GR2
Julie [35]	Rail travelling assistant for Amtrak	4	32	256	53	G,GR2,GR3,GR4
Kuki [36]	General-purpose chatbot by Kuki AI	6	15	150	10	G,GR2,Mul,Geo
Lola [37]	Student registration support	4	14	112	32	C,G,GR1,GR2,GR3
Millie [38]	Info about company 1MillionBot	3	12	96	7	C,G,GR1,GR3
SAIC [32]	Assistant for citizens of a large city	4	14	112	22	C,G,T,GR2
ServiceForm [39]	Support for company Serviceform	4	12	96	9	G,GR1,GR2,GR3

We developed conversation profiles for each chatbot, and used SENSEI-chat to generate conversations using two personalities: standard and non-verbose. For each chatbot, we set to test typical tasks a user could perform, like obtaining a train schedule between two cities, or performing a money transfer. Then, we used SENSEI-check to assess the conversations.

Using SENSEI-chat, we checked for crashes (C), unsatisfied goals (G), timeouts (T) and conversation loops (L). We used SENSEI-check and the DSL to set some general correctness rules to check: if the chatbot deviates from its topics (GR1); if the chatbot repeats itself, as this could imply understanding problems (GR2); if the chatbot is supposedly multi-language but responds to the user in the same language (GR3); and if the chatbot provides malformed

links or empty buttons (GR4). These rules were run on all chatbots. Additionally, we created domain-specific rules for some chatbots using the DSL, e.g., to check the correctness of the computations they performed (e.g., Mul for multiplication operations, and Geo for geography questions). A failure is reported upon detecting crashes, unfinished goals, timeouts, conversation loops, or some of the created correctness rules fails.

7.2.2. Experiment results

Columns 3–7 of Table 6 report on the testing results. They show the number of profiles designed, the number of conversations generated, the number of rule executions (i.e., number of conversations multiplied by number of rules), the number of failures found (one conversation may have multiple defects), and the concrete rules that failed. Additionally, Table 7 details the specific number of generic failures (C, G, T, L) and rule violations (GR1, GR2, GR3, GR4, Mul, Geo) per chatbot.

Table 7: Detailed results: number of violations of each rule (“–” means not applicable).

Chatbot	C	G	T	L	GR1	GR2	GR3	GR4	Mul	Geo	TOTAL
Ada [33]	11	8	0	0	1	5	4	0	–	–	29
Catalina [31]	3	7	0	0	0	10	–	0	–	–	20
FinancialBot [34]	0	15	0	0	2	17	0	0	–	–	34
Julie [35]	0	8	0	0	0	11	15	19	–	–	53
Kuki [36]	0	7	0	0	–	1	–	0	1	1	10
Lola [37]	7	13	0	0	2	6	4	0	–	–	32
Millie [38]	1	3	0	0	2	0	1	0	–	–	7
SAIC [32]	2	8	1	0	0	11	–	0	–	–	22
ServiceForm [39]	0	1	0	0	1	5	2	0	–	–	9
TOTAL	24	70	1	0	8	66	26	19	1	1	216

7.2.3. Discussion of results

Ada and Lola are chatbots deployed on University websites. For Ada, we designed 4 profiles to obtain technical assistance on email credentials and the contact details of the service center. The native language of Ada is only Spanish, and profiles dealt with English communication (to account for Erasmus students) or off-topic conversations. Interestingly, the chatbot crashed most of the times when approached in English. Moreover, we found repetitions in 28% of conversations (GR2). For Lola, we designed 4 profiles with the same approach to Ada, finding similar problems, but the crashes found were not related to the user language.

Catalina and SAIC are chatbots for assisting citizens. We designed 3 and 4 profiles for them, asking questions like how to pay the housing tax, how to

register in the census and off-topic questions. The main difference between both chatbots is that Catalina seems to be intent-based, while SAIC uses an LLM. In them, we found crashes, unsatisfied goals and timeouts, but the fact that SAIC is based on LLMs makes it more robust to language understanding issues.

FinancialBot is a Rasa-based chatbot to serve as a framework for building banking solutions. We designed 5 profiles for tasks like obtaining the account balance, checking the money spent in Amazon in one month, and performing bank transfers. This chatbot is quite robust with respect to crashes, but limited in language understanding since Rasa is an intent-based technology. This is shown by the fact that 65% of the conversations fail GR2 (repeated phrases). Moreover, when the user insists with phrases that the chatbot does not understand, the chatbot ends up saying *“I’m a virtual assistant made with Rasa”*, which our rules correctly catch as out of scope. In general, most problems that our user simulator helped detecting in intent-based chatbots were related to poor understanding of common phrases. For instance, when FinancialBot asks *“Would you like to continue scheduling the money transfer?”*, it is not able to understand the user response *“Go ahead, please”*. Instead, if we instruct the simulator to emulate a *“direct user profile”* who answers just *yes* or *no*, then the chatbot understands the user better. This was also the strategy followed in the experiment of RQ1.2. In this respect, developers can use the generated failing conversations to improve the chatbot NLU.

For Julie, we created 4 profiles to obtain schedules of trains between two US cities (among a set). The profiles considered changing the conversation language to Spanish, talking only in Spanish, or giving wrong cities (i.e., outside the US). Julie always responds in English, but if approached in Spanish, she redirects to the Amtrack website in Spanish, and is not able to accomplish the task. In addition, Julie provided repeated answers in 34% of conversations, and sometimes presented buttons with no links.

Since Kuki is general-purpose and open-domain, we tested it with tasks about maths and geography. After building domain-specific rules for these tasks, we found that the chatbot failed in multiplication tasks, providing always wrong (but close) values for multiplications of 4-digit numbers, as shown in the following conversation:

User: *Hey Kuki! I’m working on some math exercises and I need a little help. Can you help me figure out the result of multiplying 4950 and 35437?*

Kuki: *Hi. How many hours do you work? What’s wrong? If you ask for*

help in the form of a question, I might be able to solve your problem. Is it part of a car? Let me think... Sure thing! The result of multiplying 4950 and 35437 is 175,471,500. 35437 is a number.

The chatbot response 175,471,500 was correctly extracted from the conversation, and the rule Mul caught the error (the correct result is 175,413,150). However, it succeeded in other arithmetical operations like division and square roots. Also, Kuki incorrectly answered that the capital of Canada was Toronto instead of Ottawa, which was caught by a rule (Geo) in charge of checking capitals.

Specific to companies, Millie is a multi-language assistant from a chatbot development company. Its testing involved 3 profiles designed to obtain the possible deployment channels of chatbots and the company phone number, changing the language and doing small talk with off-topics. We found crashes, but good handling of multi-language except in one case, in which Millie responded in the wrong language. For the ServiceForm chatbot, we set 4 profiles, some of them to gather the best plan for a service and its price. The recommended plan is based on the number of visits of the user to the website. We encountered unfinished goals (G) when the number of visits was expressed as “dozens per week”, which the chatbot did not understand. In this case, SENSEI could not obtain the plan or the price, and led to the chatbot repeating very similar phrases in some conversations (GR2).

As a final note, the generation time of each conversation depends on the efficiency of the tested chatbot and the performance of our system. We record both times as part of the conversation generation process. In our experiments, SENSEI-chat needed around 5 seconds in average per generated interaction. Generating a typical conversation with 5 steps takes between 30 to 60 seconds, depending on the response time of the tested chatbot.

7.2.4. Answering RQ2

We can answer RQ2 positively, since we detected problems in all analysed chatbots. By defining just a few (3–6) profiles, it is possible to generate tens or even hundreds of conversations using input generators and different personas. Errors can already be caught without testing rules. In addition, generic rules – reusable across chatbots – built with the DSL can catch errors related to malformed links or language problems. Finally, adding domain-specific rules with the DSL permits detecting functionality problems related to the information provided by the chatbots.

7.3. RQ3: performance of checking correctness rules

We assess the scalability of our DSL by measuring the execution time of various types of correctness rules over conversation sets of increasing size.

7.3.1. Experiment design

Firstly, we used SENSEI to simulate 10 varied conversations of different user persona (e.g., talkative, direct) interacting with the Pizza Order chatbot. Conversations had between 4 and 18 turns (median 8), between 106 and 347 words (median 192), and between 603 and 2086 characters (median 1015). Then, we created conversation sets of increasing sizes, from 10 up to 1000 conversations, by replicating the simulated ones.

Secondly, we used our DSL to define 7 varied rules, whose definition is available in the Appendix. The rules are as follows:

- `oracle_1_conv`: single-conversation rule, similar to Listing 3.
- `oracle_2_conv`: multi-conversation rule similar to Listing 4, checked on conversation pairs.
- `global_rule`: global rule in Listing 5, checked on the whole conversation set.
- `repeated_answers`: single-conversation rule checking for duplicate answers. It calls function `repeated_answers` in Table 3, which uses NLP comparison techniques.
- `semantic_content`: single-conversation rule checking the semantic content of an output variable. It calls function `semantic_content` in Table 3, which uses an LLM (GPT-4o-mini).
- `conversation_order`: single-conversation rule checking the conversation flow. It calls function `utterance_index` in Table 3 twice. This function uses an LLM (GPT-4o-mini).
- `only_talks_about`: single-conversation rule checking the conversation scope. It calls function `only_talks_about` in Table 3, which uses a reasoning LLM (o1-mini).

Finally, we executed the rules on the conversation sets using SENSEI-check. To rule out spurious effects, we repeated the execution five times, taking the median execution time. We performed the experiment on an HP Z4 G4 Workstation Windows machine, with 32Gb RAM, and an Intel i9-10920X CPU @ 3.50GHz.

7.3.2. Experiment results

Table 8 shows the experiment results. For each rule, it displays the LLM used, the median time to process each conversation set (in seconds), and the per-conversation-time (PCT) for the sets with 1000 conversations. Figure 6 represents the same data in graphic form.

Table 8: Rule checking times (in seconds, median over 5 executions, with estimated standard deviation).

Rule	LLM	Number of Conversations					PCT
		10	100	300	500	1000	
oracle_1_conv	n/a	0.04 ± 0.0009	0.38 ± 0.001	1.15 ± 0.004	1.90 ± 0.007	3.86 ± 0.012	0.0038
oracle_2_conv	n/a	0.04 ± 0.0003	1.15 ± 0.006	8.16 ± 0.003	21.59 ± 0.24	84.02 ± 1.1	0.0840
global_rule	n/a	0.03 ± 0.0005	0.37 ± 0.001	1.11 ± 0.007	1.84 ± 0.01	3.69 ± 0.05	0.0037
repeated_answers	n/a	0.11 ± 0.02	1.14 ± 0.05	3.45 ± 0.2	5.69 ± 0.3	11.39 ± 0.6	0.0114
semantic_content	GPT-4o-mini	2.73 ± 0.4	30.78 ± 1.7	86.54 ± 1.7	153.00 ± 11.4	291.42 ± 9.4	0.2914
conversation_order	GPT-4o-mini	11.24 ± 1.1	116.56 ± 5.1	348.13 ± 4.7	586.40 ± 3.2	1192.29 ± 20.8	1.1923
only_talks_about	o1-mini	32.40 ± 2.8	343.37 ± 22.8	1079.83 ± 58.3	1792.95 ± 83.6	3476.96 ± 59.7	3.4769

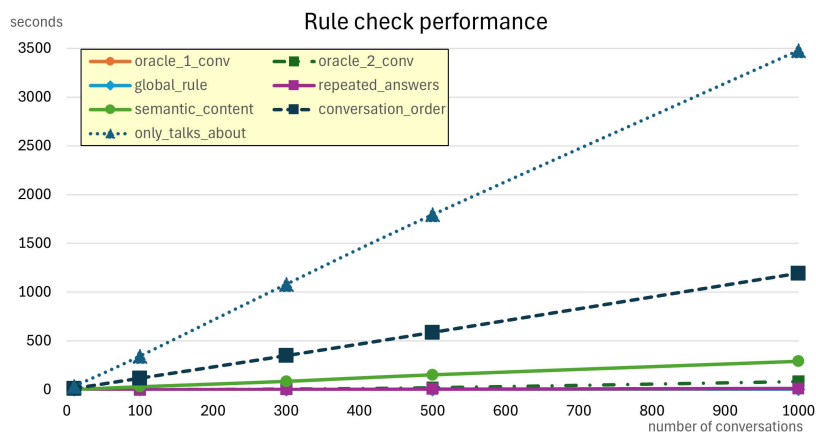


Figure 6: Median rule execution times in conversation sets of increasing size.

7.3.3. Discussion of results

We observe that fastest rules are the single-conversation ones that do not use LLMs: `oracle_1_conv` and `global_rule`, each taking less than 0.004 seconds per conversation. Rule `repeated_answers` is also fast but slightly slower (0.0114 seconds per conversation), due to its use of TF-IDF for text comparison. The multi-conversation rule `oracle_2_conv` has quadratic time growth, as it checks each conversation pair. The slowest rules are those using text functions based on LLMs, where the time cost depends on the specific LLM used and the

number of calls made. The chat model GPT-4o-mini is faster than o1-mini, which is optimised for reasoning. Moreover, rule `conversation_order` calls the LLM twice to identify the index of both a user request and a chatbot response.

7.3.4. Answering RQ3

We generally obtained fast processing times, with per-conversation times between 0.003 and 3.4 seconds, depending on the rule nature (single- or multi-conversation, using LLM-based text processing functions or not). As expected, we observed linear growth time in all rules, except for the multi-conversation rule, which had a quadratic execution time (cf. Figure 6).

7.4. Threats to validity

This subsection discusses the threats to the validity of our experiments.

7.4.1. External validity

Our evaluation involves a limited number of chatbots (4 in RQ1.1, 4 in RQ1.2, 9 in RQ2, 1 in RQ3), and so, stronger results could be achieved with more chatbots. The problem is that there is no standard dataset of chatbots for testing, and those available in public repositories (e.g., GitHub) are many times too simple, contain errors, or cannot be deployed [18]. To alleviate this threat, our experiment considers chatbots made with heterogeneous technologies: Taskyto in RQ1.1 and RQ3; Rasa in RQ1.2, Pandorabots [40] and mashups of Rasa, Dialogflow and LLMs for RQ2. Moreover, RQ1.1, and RQ1.2 employed chatbots published in the literature, and RQ2 uses chatbots in production deployed on the web.

Our evaluation is focused on “service-oriented chatbots” (except Kuki, which is general-purpose). This is so as our proposal is particularly useful for chatbots having a specific objective, from which SENSEI can extract output data. It would be interesting to use SENSEI to test chatbots like OpenAI’s GPTs [41], which combine LLMs and a specific purpose (e.g., converse about travelling). However, they are currently not available through an API.

7.4.2. Internal validity

For RQ1.1 and RQ1.2, we manually assessed each conversation to detect false positives and discard equivalent mutants. The assessment was performed by one of the authors, and confirmed by the other authors.

For RQ2, we could have generated more conversations. However, we were limited by targeting chatbots already deployed, and flooding them with

thousands of fake conversations would have impacted their statistics and may have had an economic impact on the owning companies. Still, tens of conversations served to reveal faults in the considered chatbots.

For RQ3, to discard spurious effects in the computation of performance, we executed the rules five times and took the median. However, for RQ1.1 and RQ2, the lack of repetitions that account for potential answer variations due to the randomness of LLMs poses a risk. In this respect, although each profile was executed once, each execution resulted in several conversations targeting the same conversation goals (on average, 3.8 conversations per profile in RQ1.1, and 4.2 in RQ2). This mitigates this threat by trying several conversations per profile, akin to replications. For the case of RQ1.2, Rasa chatbots are more deterministic, but the same strategy was followed.

7.4.3. Construct validity

SENSEI uses LLMs to generate user utterances, and some chatbots under test use LLMs for user intent recognition or response generation. In the evaluation, SENSEI-chat used gpt-4o-mini, Taskyto chatbots used gpt-4o, and we do not know the underlying model of the LLM-based chatbots of Table 6. Thus, there may be a potential risk of using LLMs to test other LLMs. However, as discussed in Section 3, there is no information leakage between the chatbot under test and SENSEI, as they access the LLMs via their APIs without sharing prompts. Likewise, there is no leakage between SENSEI-chat and SENSEI-check, as their conversation history is not shared. This avoids any bias that could make SENSEI-check perform better after SENSEI-chat.

For RQ1, we do not claim that the set of mutation operators is complete, models all possible faults, or mimics realistic errors found in practice. Still, the operators cover all elements of Taskyto and Rasa specifications, and SENSEI was able to find most injected faults, as well as real errors in third-party chatbots.

Finally, we defined all parameters, profiles and testing rules manually. This may introduce bias in RQ2, favouring configurations that expose errors while omitting others that do not. However, this would only imply that some errors might require a careful design of testing profiles and rules, without undermining the usefulness of our method for error detection. Likewise, for RQ3, we may have inadvertently favoured faster rules. To mitigate this risk, we defined a diverse rule set (including single-conversation, multi-conversation, and global rules) both with and without relying on different types of LLMs.

8. Related Work

Next, we analyse related work on user simulation (Section 8.1), chatbot testing (Section 8.2) and metamorphic testing (Section 8.3).

8.1. User simulation for conversational systems

Testing chatbots with real users is valuable, but also costly and time-consuming. To address this issue, researchers have developed user simulation methods. These vary on their input, output, and simulation techniques.

First, the customisation options for user simulators are diverse. For example, Vasconcelos *et al.* [42] stress the difficulty to evaluate chatbots and the need for automated testing, and propose the user simulator Bottester that is configured with a Q&A corpus, a number of executions, and a user delay. UserSimCRS [43] is an agenda-based user simulator [44] for conversational recommendation systems, and requires providing: a set of slot names; preference data (item ratings); sample dialogues annotated with matching intent and slot-value pairs; an interaction model tailored to conversational recommendations; and a model of the user preferences, persona and context. Ferreira *et al.* [45] produce user profiles with conversational traits, such as engagement or verbosity. Specific for conversational search, Sekulic *et al.* [46] apply in-context learning and LLMs to simulate users with varying levels of patience, cooperativeness and politeness. CoSearcher [47] is a stochastic user simulator for conversational search refinement that supports tuning the users' cooperativeness and patience. Kiesel *et al.* [48] simulate follow-up questions in conversational searches using LLMs and simple prompts, but no user traits. In comparison, SENSEI does not need pre-existing questions and answers [42] or a definition of the space of intents and responses [43], but generates user inputs automatically. Moreover, compared to other simulators' inputs, SENSEI provides an expressive DSL to configure the user profile and conversation strategy, covering not only user traits and persona, but also conversation goals, user inputs and expected chatbot outputs.

With regards to techniques for simulating users, the most recent approaches employ generative artificial intelligence, especially LLMs. For instance, Griol *et al.* [49] 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 [7] user simulator can include external knowledge via retrieval augmentation and summary control,

converting free-form text into training data without user intervention. Terragni *et al.* [8] employ LLMs to generate user utterances based on the user goals (described by a sentence) and a set of dialogue examples. Bandlamudi *et al.* [50] employ an LLM-based simulated user to test conversational assistants generated from application program interfaces (APIs) and robotic process automation (RPA), and another LLM to judge the assistants’ handling of the interactions. Wit [9] 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 [51] uses language models trained on an in-domain corpus of question, response and follow-up triplets to generate follow-up questions. Similar to these approaches, SENSEI uses LLMs to simulate users coherent with a given description. However, compared to them, it does not require the availability of a dialogue corpus [7, 8, 49, 50, 51], and offers a finer-grained characterisation of users and conversation goals [9].

Different from the previous user simulation techniques, ConvLab-3 [52] is a toolkit to build and evaluate dialogue systems. It uses a unified format for storing dialogues, and dialogue datasets can be used for both construction and evaluation. For evaluation, ConvLab-3 can be integrated with user simulators to explore the generalisation capabilities of the dialogue systems with new user behaviours. The authors explore candidate user simulators like TUS [53], GenTUS [54], EmoUS [55], and ABUS [56]. TUS and ABUS work at the semantic level, and apply template-based natural language (NL) generation to convert the generated user actions into utterances, which can lead to less diverse and dynamic conversations. In contrast, GenTUS and EmoUS produce both user actions and NL utterances, but require a dataset of human-chatbot dialogues to train the models. This might not be realistic in our context, chatbot testing, which sees chatbots as black-boxes. Neither ConvLab-3 or these simulators support testing capabilities or interacting with deployed chatbots.

As for the user simulators’ output, all save the simulated conversations for their inspection. For instance, Terragni *et al.* [8] analyse the interaction between the simulated user and the dialogue system manually, which permits uncovering some mistakes such as misclassifications, hallucinations, or cases where the user simulator fails to mention all user goals. Some simulators also report further data. For example, Bottester computes metrics on user satisfaction (answers’ length, repeated answers, response time) and correctness (expected vs. actual answers); ConvLab-3 [52] provides scripts for calculating metrics like accuracy, BLEU [57], and slot error rate; Zhan *et*

al. [58] simulate users that give feedback on recommended items, and measure conversation quality solely on recommendation metrics such as recall; and Bernard *et al.* [14] simulate users of conversational recommendation systems, reporting patterns causing breakdowns (bugs, communication breakdowns, or deviations from the defined interaction model). In comparison to these simulators, SENSEI reports timeouts, loops, and unachieved conversation goals. More importantly, it offers a DSL that permits defining richer domain-specific checks and metamorphic rules, as well as their automated assessment on the simulated conversations.

Finally, also for LLMs, Prompt Fuzzer [59] simulates LLM-based attacks to evaluate the security of generative AI systems. It extracts the required context from the provided system prompt, adapting the fuzzing process accordingly. The fuzzer focuses on security vulnerabilities, and lacks an equivalent to our DSLs to configure the simulated users, goals, and testing rules.

8.2. Testing of conversational systems

Assessing chatbots is often based on user evaluations [60]. However, while humans are good oracles to assess chatbot quality, a manual approach incurs in high effort and cost. Among automated approaches to test conversational systems, we can distinguish white-box approaches, which need the chatbot source code, and black-box approaches, which do not need it.

Within white-box approaches, mutation testing has been proposed for intent-based chatbots [27, 28]. Those approaches define mutation operators, e.g., to modify the training phrases of the intents, the conversation flows or the chatbot outputs. Our end-to-end approach treats chatbots as black-boxes, and hence does not need their source code. The evaluation in Section 7.1 used fault seeding, but with the purpose of evaluating the strength of SENSEI in discovering chatbot defects.

Also for white-box testing, Asymbob [61] analyses the conversation design of intent-based chatbots and generates test scenarios fulfilling coverage criteria of interest. The generated scenarios exercise conversation flows based on the utterances used to train the chatbot. The approach is applicable to Rasa and Dialogflow agents, and runs the tests atop Botium (now part of Cyara [62]).

Other white-box approaches target specific parts of chatbots. For example, CHARM [63] tests the NLU component of intent-based chatbots by generating variants of the intents’ training phrases to use them as testing utterances. Hence, it cannot produce full conversations nor be used with

LLM-based chatbots. CTG [64] exploits the chatbot definition to generate test scenarios that cover the chatbot conversational paths and entity values, being specific for Dialogflow chatbots. Finally, many intent-based chatbot construction frameworks, like Rasa [15] or Dialogflow [16], feature facilities to test the NLU component, to assess whether a chatbot recognises the expected intent and extracts the relevant parameters from the user utterance.

Some black-box approaches are integral, end-to-end. Rasa test e2e [65] permits defining test scenarios as YAML files, which are executed against deployed Rasa chatbots. Test conversation scenarios can have different pre-configured contexts, execute custom actions, verify response texts or names, and assert when slots (i.e., data required by the chatbot) are filled or not. In comparison, SENSEI works for any API-enabled chatbot technology, supports not only assertions over data values but also checking non-functional properties and metamorphic relations, and integrates a user simulator that automates the generation of test conversation scenarios.

Finally, Botium [30] (used to address RQ1.2) is a testing framework for conversational agents. Similar to Rasa test, it permits defining scenarios (*convos*) comprising user utterances and expected chatbot reactions. The correctness of the latter can be checked by assertions on the activated intent or chatbot output. Thus, Botium benefits from having available the chatbot source code, being able to inspect its internal state (e.g., for assertions about the activated intent). However, Botium was designed to test intent-based chatbots, and hence, it presents limitations for its application to LLM-based chatbots. First, it requires the explicit definition of user utterances and conversation scenarios, which is difficult to predict for LLM-based chatbots. Second, it lacks mechanisms to extract relevant data from the chatbot responses, and to define testing rules like we do with the DSL.

Overall, our work is novel by providing mechanisms to automate end-to-end chatbot testing. These include support for generating conversations via simulated users, and for testing rules that can be checked on the generated conversations.

8.3. Metamorphic testing

Metamorphic testing has been applied to many domains [66], but scarcely to chatbots. Some metamorphic relations (MRs) are proposed in [67, 68], maintaining or changing the meaning of user utterances, e.g., replacing keywords by synonyms, or changing numerical values. The approach generates test cases from the MRs, and checks for equality or difference of chatbot

outputs in conversation pairs. Morales *et al.* follow a somewhat related approach to test bias in LLMs [69]. They use sets of prompt templates expressing ethical requirements and having “holes” (e.g., gender), which are filled by variable values (e.g., male, female). Then, the answer of the LLM in each case is checked and compared. Compared to these works, SENSEI-check supports the definition of arbitrary MRs (i.e., it is not limited to a predefined set or scope) that can be applied to any number of conversations (rather than just pairs). However, it lacks means to derive follow-up test cases from the MRs. Instead, our testing framework is integral, including a user simulator that emulates meaningful conversations coherent with a given profile.

9. Conclusions and Future Work

Given the prominence and momentum of conversational agents, techniques to ensure their quality are needed. In this paper, we presented SENSEI, an approach to reduce the effort to test chatbots based on a user simulator that produces conversations with the chatbot under test and detects errors like crashes or timeouts. Then, a DSL permits defining testing rules for further assessing the chatbot. We have introduced the approach, described an implementation, and reported an evaluation showing that the approach is able to detect faults in real chatbots in reasonable time.

Recently, researchers have proposed guidelines to design chatbot conversations, like those consolidated by Ramos and Dias [70, 71] or by Moore and Arar [72]. SENSEI may also help testing some of these principles automatically to enable effective chatbot conversational design. For instance, one recommendation is to avoid repetitive messages [70], and our evaluation showed that SENSEI-check can detect conversational repetitions. Other guidelines that our system can help validate include: avoiding errors (typos) in the chatbot responses to make it look *natural*, hiding the chatbot’s non-human identity, addressing users by their name, incorporating graphical media and exclamatory feedback, or informing users about the chatbot’s capabilities [70].

Regarding usability, to facilitate the use of SENSEI, we are currently automating the generation of conversation profiles from black-box chatbots. We do so by conducting automated exploratory sessions of the chatbots’ functionality via LLMs. These sessions simulate interactions with the chatbot, from which conversation profiles are derived. Moreover, as part of our ongoing work, we plan to conduct a user study to check with practitioners the

effort of defining the conversation profiles, the testing rules using the DSL, and the whole usability of the approach.

As mentioned in Section 2, we will consider extending our framework to support assessing specific aspects of chatbots concerning ethics (e.g., with more advanced mechanisms to check response bias, transparency, inclusivity, safety and harm prevention), security (e.g., by customisable libraries of malicious prompts, and mechanisms for injection testing in inputs) and privacy (e.g., to assess data collection and storage, or privacy leakage).

In the future, we would like to extend our approach to test voice-driven chatbots, offer a more extensive text analysis library, and develop modularisation techniques for conversation profiles to enable better reuse. Moreover, it would be interesting to provide automated assistance, potentially using LLMs, to mitigate the cost of defining and validating the metamorphic relations, which are currently specified manually. Since the functionality of SENSEI is based on LLMs, we would also like to use budget cost as stopping criteria for conversation generation. On the technical side, we would like to deploy SENSEI as a testing service, on a server. Regarding the supported kinds of interaction, when a chatbot displays buttons, we translate them as text for the user simulator; still, we would need to extend the simulator to emulate clicking on them. Likewise, we would like to extend our simulator to handle chatbot outputs with multimedia content (images, short videos) by relying on multi-modal LLMs. Finally, we plan to develop a full-fledged mutation testing framework for LLM-based chatbots.

Acknowledgements

Work funded by the Spanish MICINN (projects TED2021-129381B-C21, TED2021-129381B-C22, PID2024-155231OB-I00, PID2021-122270OB-I00, and RED2022-134647-T). We would like to thank the reviewers for their useful comments, which served to improve previous versions of the paper.

References

- [1] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, J.-R. Wen, A survey of large language models (2023). [arXiv:2303.18223](https://arxiv.org/abs/2303.18223).

- [2] B. A. Myers, *Pick, click, click!: The story of interaction techniques*, Vol. 57 of ACM Books, ACM, 2024.
- [3] Chat GPT, <https://chatgpt.com/> (last access in 2025).
- [4] Google, Gemini, <https://gemini.google.com/> (last access in 2025).
- [5] X. Li, C. Tao, J. Gao, H. Guo, A review of quality assurance research of dialogue systems, in: *IEEE International Conference On Artificial Intelligence Testing (AITest)*, IEEE, 2022, pp. 87–94.
- [6] J. Cabot, L. Burgueño, R. Clarisó, G. Daniel, J. Perianez-Pascual, R. Rodríguez-Echeverría, Testing challenges for NLP-intensive bots, in: *International Workshop on Bots in Software Engineering (BotSE@ICSE)*, IEEE, 2021, pp. 31–34.
- [7] K. D. Dhole, KAUCUS: Knowledge augmented user simulators for training language model assistants, in: *Workshop on Simulating Conversational Intelligence in Chat (SCI-CHAT)*, ACL, 2024, pp. 53–65.
- [8] S. Terragni, M. Filipavicius, N. Khau, B. Guedes, A. Manso, R. Mathis, In-context learning user simulators for task-oriented dialog systems (2023). [arXiv:2306.00774](https://arxiv.org/abs/2306.00774).
- [9] J. de Wit, Leveraging large language models as simulated users for initial, low-cost evaluations of designed conversations, in: *International Workshop on Chatbot Research and Design (CONVERSATIONS)*, Vol. 14524 of LNCS, Springer, 2023, pp. 77–93.
- [10] A. Cooper, R. Reimann, D. Cronin, C. Noessel, *About Face: The essentials of interaction design* (4th edition), Wiley, 2014.
- [11] A. Wasowski, T. Berger, *Domain-specific languages - Effective modeling, automation, and reuse*, Springer, 2023.
- [12] T. Y. Chen, S.-C. Cheung, S.-M. Yiu, *Metamorphic testing: A new approach for generating next test cases*, Tech. rep., HKUST-CS98-01 (1998).
- [13] S. Segura, D. Towey, Z. Q. Zhou, T. Y. Chen, *Metamorphic testing: Testing the untestable*, *IEEE Softw.* 37 (3) (2020) 46–53.

- [14] N. Bernard, K. Balog, Identifying breakdowns in conversational recommender systems using user simulation, in: ACM Conference on Conversational User Interfaces (CUI), ACM, 2024.
- [15] Rasa, <https://rasa.com/> (last access in 2025).
- [16] Dialogflow, <https://cloud.google.com/dialogflow> (last access in 2025).
- [17] S. Pérez-Soler, S. Juárez-Puerta, E. Guerra, J. de Lara, Choosing a chatbot development tool, *IEEE Softw.* 38 (4) (2021) 94–103.
- [18] P. C. Cañizares, J. M. López-Morales, S. Pérez-Soler, E. Guerra, J. de Lara, Measuring and clustering heterogeneous chatbot designs, *ACM Trans. Softw. Eng. Methodol.* 33 (4) (2024) 90:1–90:43.
- [19] G. Salton, M. J. McGill, Introduction to modern information retrieval, McGraw-Hill, Inc., New York, NY, USA, 1983.
- [20] C. D. Manning, P. Raghavan, H. Schütze, Introduction to information retrieval, Cambridge University Press, Cambridge, UK, 2008.
- [21] J. W. Ratcliff, D. Metzener, Pattern matching: The gestalt approach, *Dr. Dobb’s Journal* 13 (1988) 46–51.
- [22] J. Sánchez Cuadrado, S. Pérez-Soler, E. Guerra, J. de Lara, Automating the development of task-oriented LLM-based chatbots, in: ACM Conf. on Conversational User Interfaces (CUI), ACM, 2024, pp. 1–17.
- [23] 1millionbot, <https://1millionbot.com/en/> (last access in 2025).
- [24] LangChain, <https://www.langchain.com/> (last access in 2025).
- [25] Taskyto chatbots, <https://github.com/satori-chatbots/taskyto> (last access in 2025).
- [26] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, *Computer* 11 (4) (1978) 34–41.
- [27] P. Gómez-Abajo, S. Pérez-Soler, P. C. Cañizares, E. Guerra, J. de Lara, Mutation testing for task-oriented chatbots, in: International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, 2024, pp. 232–241.

- [28] M. F. Urrico, D. Clerissi, L. Mariani, MutaBot: A mutation testing approach for chatbots, in: IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE Companion), ACM, 2024, pp. 79–83.
- [29] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, Q. Yang, Why Johnny can't prompt: How non-AI experts try (and fail) to design LLM prompts, in: 2023 CHI Conference on Human Factors in Computing Systems, CHI, ACM, 2023.
- [30] Botium, <https://www.botium.ai/> (last access in 2023).
- [31] Catalina chatbot, <https://www.rivasciudad.es/> (last access in 2025).
- [32] SAIC chatbot, <https://saic.malaga.eu/> (last access in 2025).
- [33] Ada chatbot, <https://www.uam.es/uam/tecnologias-informacion> (last access in 2025).
- [34] Financial chatbot, <https://github.com/dDeuce222/Financial-chatbot> (last access in 2025).
- [35] Amtrak's Julie chatbot, <https://www.amtrak.com/home.html> (last access in 2025).
- [36] Pandorabot's Kuki chatbot, <https://chat.kuki.ai/chat>, see also https://en.wikipedia.org/wiki/Kuki_AI (last access in 2025).
- [37] Lola chatbot, <https://www.um.es/web/estudios/acceso/> (last access in 2025).
- [38] Millie chatbot, <https://1millionbot.com/> (last access in 2025).
- [39] Serviceform chatbot, <https://www.serviceform.com/> (last access in 2025).
- [40] Pandorabots, <https://home.pandorabots.com/> (last access in 2025).
- [41] OpenAI GPTs, <https://chatgpt.com/gpts> (last access in 2025).
- [42] M. Vasconcelos, H. Candello, C. Pinhanez, T. dos Santos, Bottester: Testing conversational systems with simulated users, in: Brazilian Symposium on Human Factors in Computing Systems (IHC), ACM, 2017.

- [43] J. Afzali, A. M. Drzewiecki, K. Balog, S. Zhang, UserSimCRS: A user simulation toolkit for evaluating conversational recommender systems, in: ACM International Conference on Web Search and Data Mining (WSDM), ACM, 2023, pp. 1160–1163.
- [44] J. Schatzmann, S. J. Young, The hidden agenda user simulation model, *IEEE Trans. Speech Audio Process.* 17 (4) (2009) 733–747.
- [45] R. Ferreira, D. Semedo, J. Magalhães, Multi-trait user simulation with adaptive decoding for conversational task assistants, in: Findings of the Association for Computational Linguistics (EMNLP), ACL, 2024, pp. 16105–16130.
- [46] I. Sekulic, L. Lu, N. S. Bedi, F. Crestani, Simulating conversational search users with parameterized behavior, in: Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region (SIGIR-AP), ACM, 2024, pp. 72–81.
- [47] A. Salle, S. Malmasi, O. Rokhlenko, E. Agichtein, CoSearcher: Studying the effectiveness of conversational search refinement and clarification through user simulation, *Inf. Retr. J.* 25 (2) (2022) 209–238.
- [48] J. Kiesel, M. Gohsen, N. Mirzakhmedova, M. Hagen, B. Stein, Simulating follow-up questions in conversational search, in: European Conference on Information Retrieval (ECIR), Vol. 14609 of LNCS, Springer, 2024, pp. 382–398.
- [49] D. Griol, J. Carbó, J. M. Molina, An automatic dialog simulation technique to develop and evaluate interactive conversational agents, *Appl. Artif. Intell.* 27 (9) (2013) 759–780.
- [50] J. Bandlamudi, K. Mukherjee, P. Agarwal, R. Chaudhuri, R. Pimplikar, S. Dechu, A. Straley, A. Ponniah, R. Sindhgatta, Framework to enable and test conversational assistant for APIs and RPAs, *AI Mag.* 45 (4) (2024) 443–456.
- [51] P. Rao, M. Agnihotri, D. B. Jayagopi, Improving asynchronous interview interaction with follow-up question generation, *Int. J. Interact. Multim. Artif. Intell.* 6 (5) (2021) 79–89.

- [52] Q. Zhu, C. Geishauser, H. Lin, C. van Niekerk, B. Peng, Z. Zhang, S. Feng, M. Heck, N. Lubis, D. Wan, X. Zhu, J. Gao, M. Gasic, M. Huang, ConvLab-3: A flexible dialogue system toolkit based on a unified data format, in: Conference on Empirical Methods in Natural Language Processing (EMNLP) – System Demonstrations, ACL, 2023, pp. 106–123.
- [53] H.-C. Lin, N. Lubis, S. Hu, C. van Niekerk, C. Geishauser, M. Heck, S. Feng, M. Gasic, Domain-independent user simulation with transformers for task-oriented dialogue systems, in: Annual Meeting of the Special Interest Group on Discourse and Dialogue, ACL, 2021, pp. 445–456.
- [54] H. Lin, C. Geishauser, S. Feng, N. Lubis, C. van Niekerk, M. Heck, M. Gasic, GenTUS: Simulating user behaviour and language in task-oriented dialogues with generative transformers, in: Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL), ACL, 2022, pp. 270–282.
- [55] H. Lin, S. Feng, C. Geishauser, N. Lubis, C. van Niekerk, M. Heck, B. M. Ruppik, R. Vukovic, M. Gasic, EmoUS: Simulating user emotions in task-oriented dialogues, in: International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), ACM, 2023, pp. 2526–2531.
- [56] J. Schatzmann, B. Thomson, K. Weilhammer, H. Ye, S. J. Young, Agenda-based user simulation for bootstrapping a POMDP dialogue system, in: Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, ACL, 2007, pp. 149–152.
- [57] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, BLEU: a method for automatic evaluation of machine translation, in: Annual Meeting on Association for Computational Linguistics, ACL, 2002, pp. 311–318.
- [58] Q. Zhan, X. Guo, H. Ji, L. Wu, User simulator assisted open-ended conversational recommendation system, in: Workshop on NLP for Conversational AI (NLP4ConvAI), ACL, 2023, pp. 89–101.
- [59] Prompt Fuzzer, <https://github.com/prompt-security/ps-fuzz> (last access in 2025).

- [60] R. Ren, J. W. Castro, S. T. Acuña, J. de Lara, Evaluation techniques for chatbot usability: A systematic mapping study, *Int. J. Softw. Eng. Knowl. Eng.* 29 (11&12) (2019) 1673–1702.
- [61] P. C. Cañizares, R. D. A. Ortiz, S. Pérez-Soler, E. Guerra, J. de Lara, Coverage-based strategies for the automated synthesis of test scenarios for conversational agents, in: *ACM/IEEE International Conference on Automation of Software Test (AST)*, ACM, 2024, pp. 23–33.
- [62] Cyara, <https://cyara.com/> (last access in 2025).
- [63] S. Bravo-Santos, E. Guerra, J. de Lara, Testing chatbots with Charm, in: *International Conference on Quality of Information and Communications Technology (QUATIC)*, Vol. 1266 of CCIS, Springer, 2020, pp. 426–438.
- [64] R. G. Rapisarda, D. Ginelli, D. Clerissi, L. Mariani, Test case generation for Dialogflow task-based chatbots, in: *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICST)*, IEEE, 2025, pp. 93–102.
- [65] Rasa e2e test, <https://rasa.com/docs/rasa-pro/production/testing-your-assistant/#end-to-end-testing> (last access in 2025).
- [66] S. Segura, G. Fraser, A. B. Sánchez, A. R. Cortés, A survey on metamorphic testing, *IEEE Trans. Software Eng.* 42 (9) (2016) 805–824.
- [67] J. Bozic, F. Wotawa, Testing chatbots using metamorphic relations, in: *IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS)*, Vol. 11812 of LNCS, Springer, 2019, pp. 41–55.
- [68] J. Bozic, Ontology-based metamorphic testing for chatbots, *Softw. Qual. J.* 30 (1) (2022) 227–251.
- [69] S. Morales, R. Clarisó, J. Cabot, A DSL for testing LLMs for fairness and bias, in: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM, 2024, pp. 203–213.
- [70] G. Ramos, E. Dias, Towards user-centric guidelines for chatbot conversational design, *Int. J. Hum. Comput. Interact.* 40 (2) (2024) 98–120.

- [71] G. Ramos Sousa Silva, E. Dias Canedo, Designing from experience: on the adoption of chatbot conversational design practices by practitioners, *Pers. Ubiquitous Comput.* 29 (2) (2025) 153–168.
- [72] R. J. Moore, R. Arar, Conversational UX design: A practitioner’s guide to the natural conversation framework, ACM, 2019.

Appendix

This appendix includes the testing rules used in the evaluation of RQ3: How efficient is SENSEI-check at checking correctness rules? (cf. Section 7.3).

```

1 name: oracle_1_conv
2 description: Checks the price of small pizzas (>=10$)
3 conversations: 1
4 when: size == 'small'
5 oracle: extract_float(price) >= 10
6 on-error: f"Wrong price for small pizzas, it should be >=10$"
7
8 name: oracle_2_conv
9 description: More drinks cost more
10 conversations: 2
11 when: conv[0].number > conv[1].number
12 then: extract_float(conv[0].price) > extract_float(conv[1].price)
13
14 name: global_rule
15 description: The order IDs are unique
16 conversations: all
17 oracle: is_unique('order_id')
18
19 name: repeated_answers
20 description: The chatbot does not repeat answers
21 conversations: 1
22 oracle: not_repeated_answers(method='tf-idf', threshold=0.75)
23 on-error: f"The chatbots repeats {repeated_answers(method='tf-idf', threshold=0.75)}"
24
25 name: semantic_content
26 description: Checks the price of small pizzas (>=10$)
27 conversations: 1
28 when: size == 'small'
29 oracle: semantic_content(price, '10$ or more')
30
31 name: conversation_order
32 description: The assistant confirms the pizza order after the user orders a pizza
33 conversations: 1
34 oracle: utterance_index('user', 'ordering a pizza') < utterance_index('assistant', 'confirms pizza order')
35
36 name: only_talks_about
37 description: The chatbot only talks about pizzas
38 conversations: 1
39 oracle: only_talks_about('pizza orders, drink orders, information on delivery of pizzas, information about the pizza shop')

```

Listing 11: Testing rules used in the evaluation of performance (RQ3).