Pre-print of paper presented at ICSE'2026 (demos track)

# Automated Testing of Conversational Agents with Chatbot Dōjō

Iván Sotillo del Horno
ivan.delhorno@uam.es
Univ. Autónoma de Madrid
Madrid, Spain

Alejandro del Pozzo
alejandro.delpozzo@uam.es
Univ. Autónoma de Madrid
Madrid, Spain

Esther Guerra
esther.guerra@uam.es
Univ. Autónoma de Madrid
Madrid, Spain

Juan de Lara
juan.delara@uam.es
Univ. Autónoma de Madrid
Madrid, Spain

## Abstract

Conversational agents (or chatbots) interact in natural language, and may provide access to software services for specific tasks, like ordering food or obtaining customer support. Given their prominence and ubiquity, they need to be reliable. However, automated chatbot testing techniques, able to cope with the large variety of existing chatbot technologies, are missing.

To fill this gap, we present Chatbot Dōjō: a web platform for end-to-end automated chatbot testing that treats chatbots as black boxes. It comprises: (i) a *user simulator* that can be customised via testing profiles to generate conversations with a chatbot, extract relevant data from conversations, and detect problems like crashes or timeouts; (ii) a *domain-specific testing language* to define testing rules to analyse conversations; and (iii) a *chatbot explorer* that probes a chatbot under test to generate profiles for focused testing sessions. The platform is deployed at http://miso.ii.uam.es:8081/. A demo video is available at https://youtu.be/dFsbN1FKGVg.

## CCS Concepts

• **Software and its engineering → Software testing and debugging**.

## Keywords

Conversational Agents, Testing, User Simulation, LLMs

**ACM Reference Format:**
Iván Sotillo del Horno, Alejandro del Pozzo, Esther Guerra, and Juan de Lara. 2026. Automated Testing of Conversational Agents with Chatbot Dōjō. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3774748.3787604

## 1 Introduction

Conversational agents (or chatbots) are pervasive nowadays. They can be either open-domain, like OpenAI's ChatGPT [13] and Google's Gemini [9], or oriented to specific tasks, like ordering products, obtaining customer support, or completing code. Given their ubiquity, it is vital to assess their quality and proper behaviour.

However, assessing chatbot quality is challenging for two main reasons: the heterogeneity of chatbot technologies, and the manual effort that this assessment entails. Nowadays, there are many approaches to build chatbots, including intent-based frameworks like Google's Dialogflow [8] and Rasa [14], which require the a-priori definition of user intents and training phrases; multi-agent frameworks based on Large Language Models (LLMs) like LangGraph [11] or Microsoft's AutoGen [12]; and domain-specific languages like Taskyto [17]. Thus, most chatbot quality assurance techniques are tailored to a specific technology [14] or need the chatbot source code [2, 7, 20], limiting their applicability and reusability. The second major challenge is automation, as most chatbot testing approaches require substantial manual work, like providing explicit test conversations [1, 14], interacting synchronously with the chatbot [15], or gathering a conversation corpus [21].

To alleviate these challenges, we present a web platform called Chatbot Dōjō, which automates the testing of chatbots built with arbitrary technologies. Its envisioned users are chatbot testers and developers. Its architecture comprises a *chatbot explorer* called Tracer [4], and a *user simulator* called Sensei [3]. Tracer employs an LLM-based agent to autonomously explore the functionality of a deployed chatbot by conversing with it. Then, it analyses the conversations to derive a model of the chatbot functionality, and synthesises *conversation profiles* aimed at systematically testing that functionality. Sensei leverages LLMs to emulate users consistent with these profiles who interact with the chatbot under test, and reports all detected chatbot crashes, timeouts, unachieved conversation goals, and conversations loops. It also offers a *testing language* for defining testing rules to analyse the generated conversations. Overall, our platform advances the state of the art by only requiring the running chatbot (no source code or previous conversation logs are needed) and reducing manual work (model extraction, profile generation, user simulation, and testing rules are automated).

## 2 End-to-End Testing of Conversational Agents

Our approach is able to cope with running, deployed chatbots, since it treats them as black boxes. Fig. 1 shows its working scheme.

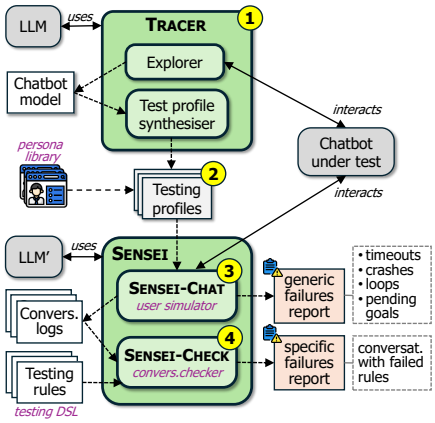


**Figure 1: Our testing workflow.**

First (label 1), Tracer converses with the chatbot under test to infer a model of its functionalities. This model contains general chatbot properties (e.g., spoken languages, fallback response when it does not understand the user); and a node per detected functionality. Such functionality nodes have a description (e.g., order a pizza), expected user inputs (e.g., pizza

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
*ICSE-Companion '26, Rio de Janeiro, Brazil*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3787604

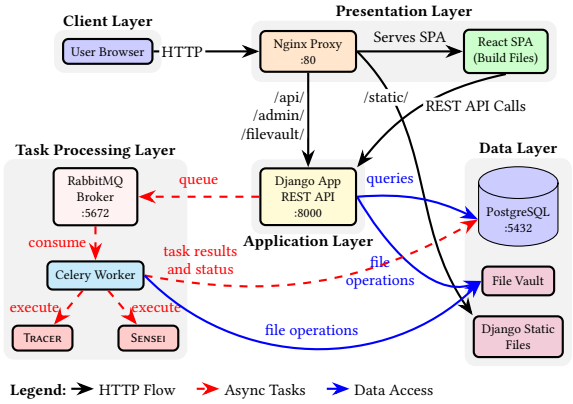


**Figure 2: Chatbot Dōjō's architecture.**

size and type), chatbot outputs (e.g., pizza price), and dependencies with other nodes (e.g., ordering a pizza precedes payment).

Next, Tracer generates a set of testing profiles (label 2), which are used to customise the user simulator Sensei-chat. A testing profile specifies how a simulated user should interact with the chatbot. Thus, Tracer creates profiles tailored to the discovered functionalities to emulate realistic situations. A profile describes the user traits and persona (e.g., non-techy old person), conversation goals (e.g., ordering a pizza), user input parameters and their possible values (e.g., pizza types, and a list of them), how to choose the parameter values in a conversation (e.g., selecting a random pizza type), and chatbot output data (e.g., pizza price). Other customisable aspects include the number of conversations to simulate, the user interaction style (e.g., short phrases, spelling mistakes), and the end-of-conversation condition (e.g., maximum number of interactions, or when achieving all conversation goals). If needed, testers can edit the generated profiles and create new ones by hand.

Sensei-chat (label 3) executes the profiles to hold as many conversations with the chatbot as specified, emulating users coherent with the profiles. The conversations are persisted, enriched with metadata like the used input parameter values, timing information, and extracted output data. It also reports failures found, like chatbot crashes, timeouts, unachieved goals, or conversation loops.

Finally, for detailed analysis, testers can use the language Sensei-check (label 4) to define testing rules to be applied on the persisted conversations. These rules enable checking if the chatbot output aligns with the expectations, given the conversation inputs. Akin to metamorphic testing, the rules can compare inputs and outputs of multiple conversations (e.g., checking if pizzas with more toppings cost more than those with fewer toppings). The rules' execution produces a detailed report of the conversations and rules that failed.

## 3 Architecture of Chatbot Dōjō

Tracer and Sensei are Python programs that can be executed from the console. Additionally, for usability, we have integrated them into a web platform called Chatbot Dōjō. Its code at https://github.com/satori-chatbots/chatbot-dojo is open source. The platform has a multi-tiered architecture designed for scalability, security, and asynchronous processing. Fig. 2 shows its five layers: Client, Presentation, Application, Task Processing, and Data.

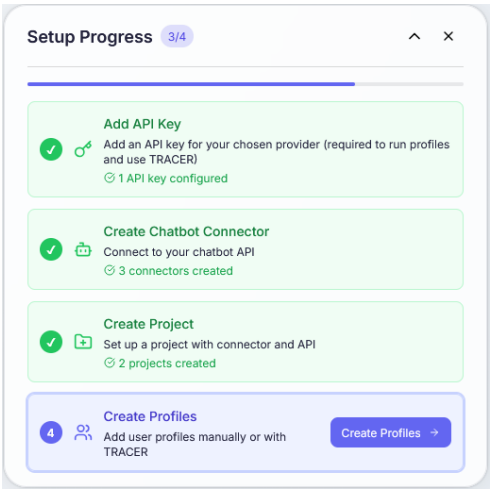


**Figure 3: Setup guide.**

The technology stack uses Django for the backend, chosen for its Python compatibility with Tracer and Sensei and its REST API capabilities through Django REST Framework. The frontend uses React to create a single page application that consumes the Django API. PostgreSQL serves as the database, selected due to its concurrency and performance characteristics.

Asynchronous task handling is critical as Tracer and Sensei executions can take from minutes to hours. We resort to Celery, a distributed task queue, with RabbitMQ as the message broker. This allows users to execute long-running tasks and navigate away from the interface, preventing server overload through controlled job queuing. Request routing is managed by an Nginx reverse proxy that directs traffic based on URL paths. Calls to `/api/` or `/admin/` are redirected to Django, while the rest serve the React build.

Deployment uses Docker containerisation with separate configurations for development and production environments.

Security follows Django's deployment checklist, implementing Django-Rest-Knox for improved token-based authentication with a token per device, encrypted database storage, and configurable expiration times. API keys are encrypted using Fernet Symmetric Encryption before database storage for additional security.

## 4 Chatbot Dōjō Walkthrough

The platform provides a complete workflow to test deployed chatbots. The process is facilitated by a *Setup Guide* (cf. Fig. 3) that walks testers through three steps: adding their LLM API key, creating a chatbot connector, and setting up a testing project. This dynamic process advances as testers complete each step. More in detail, the first step of the setup is configuring the LLM credentials, where testers provide their OpenAI or Gemini API keys. Secondly, testers can define reusable chatbot connectors either by: (i) completing a dynamic form for *built-in* connectors of supported chatbot technologies like Rasa, Taskyto, and 1MillionBot; or (ii) creating a *custom* YAML connector for other chatbot technologies that offer a REST API, which is supported by a dedicated editor with documentation and examples. In the third step, testers can create projects by specifying their name, chatbot connector, API key, LLM used for chatbot exploration, LLM used in testing profiles, and project

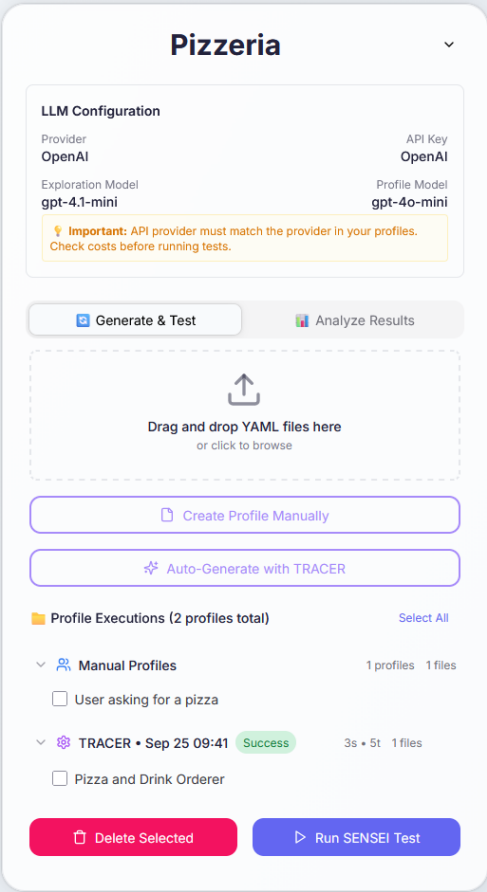


**Figure 4: Test center.**

visibility (public projects make results viewable to others). Each project serves as the central workspace to test a chatbot, storing all the configuration and execution results.

Once created, the main interface of a project is the *Test Center* (cf. Fig. 4), where testers can execute Tracer and Sensei. Initially, testers can specify testing profiles for the chatbot under test manually through a YAML editor equipped with validation, autocompletion and documentation. Alternatively, testers can automatically generate the profiles using Tracer, which runs for as many conversations as the tester specifies in a modal window. Once the process is complete, the generated profiles and a model of the chatbot functionalities are available from the *Tracer Dashboard*. Testers can modify the generated profiles if so desired.

Profiles are executed with Sensei-chat, which simulates being the user described in the profiles. Upon completion, the *Sensei Dashboard* (cf. Fig 5(a)) presents a summary of the simulation results, like total execution time, testing errors and cost. The *Profiles* tab displays the results per profile, reporting execution times, errors and profile details. Moreover, an expandable *Conversation* section displays the specific information that Sensei-chat asked about in the conversation, the data extracted from the chatbot responses, and the complete interaction between Sensei-chat and the chatbot.

For deeper analysis, testers can define chatbot-specific testing rules in an integrated YAML editor, or upload them from their local system. For expressiveness, the correctness conditions in rules use Python syntax. These rules allow checking specific errors in conversations, validating the chatbot outputs (e.g., correct pizza prices) and the interaction (e.g., the chatbot answers in the expected language). Testers can select which rules to enable, and run them using Sensei-check (cf. Fig. 5(b)). The results are shown in a modal window, as well as in the *Sensei-check Dashboard* alongside the results of other executions.

## 5 Evaluation

In previous work [3], we evaluated the effectiveness of Sensei in finding faults in both in-house created chatbots (using mutation analysis) and deployed third-party chatbots, and compared with the chatbot testing framework Botium. Overall, Sensei found most injected errors, detected errors in chatbots in production, and surpassed the default tests generated by Botium. The results are at https://github.com/satori-chatbots/user-simulator-evaluation.

In [4], we showed that Tracer achieves good precision in discovering the functionality of in-house created chatbots, and generates profiles able to find errors. Here, we report a new evaluation to address the RQ: *How accurately does Tracer model the functionality of real-world deployed chatbots?* The aim is to assess Tracer's precision for chatbots deployed in the wild, regardless of their technology. To this end, we ran Tracer on 5 third-party chatbots in production: Ada-UAM[1] for university student support, Gestri[2] for tax payment, Madrid te Cuida[3] for healthcare assistance, and 2 citizen-service chatbots for the town halls of Gallo de Morón[4] and Arucas[5]. We selected these chatbots as they are in production, cover various public-sector domains, and provide API access. We ran Tracer with 20 sessions of 12 turns, using Gemini 2.0 Flash as the LLM.

Next, we measured the precision of the inferred chatbot models, i.e., the percentage of found functionalities that are correct. We validated each functionality by interacting with the chatbots, classifying it as a *true positive* (TP) if it was present in the chatbot, or as a *false positive* (FP) if it was absent or duplicated in the model.

**Table 1: Precision of Tracer.**

| Chatbot | #Func. | TP | FP | Precision |
|---|---|---|---|---|
| Ada-UAM | 22 | 22 | 0 | 100.0% |
| Gallo de Morón | 29 | 27 | 2 | 93.1% |
| Madrid te Cuida | 91 | 88 | 3 | 96.7% |
| Gestri | 27 | 27 | 0 | 100.0% |
| Arucas | 41 | 37 | 4 | 90.2% |
| **Total** | 210 | 201 | 9 | **96.0%** |

Table 1 shows the results. Precision ranges from 90.2% (Arucas) to 100% (Ada-UAM, Gestri), with an average of 96%. Madrid te Cuida achieved 96.7%, being the most complex chatbot with 91 discovered functionalities, 88 of them correct and non-duplicated. All FPs but one are duplicated functionalities (e.g., the model of Arucas has separate nodes for "Present electronic office options" and "Provide electronic office link", which could be merged). The only FP due to hallucinated functionality was a non-existent lottery prize service attributed to Gallo de Morón. Excluding this FP, the chatbots really exhibited 99.52% of the 210 discovered functionalities. More details at: https://github.com/Chatbot-TRACER/TRACER-evaluation.

Overall, we can answer the RQ positively, as Tracer accurately extracted the functionalities of real-world chatbots. The few FPs

[1] https://tinyurl.com/yvdd7np3
[2] https://tinyurl.com/mym5ecm9
[3] https://tinyurl.com/4jjtnvsv
[4] https://tinyurl.com/39pssnt7
[5] https://tinyurl.com/yyatbp8c

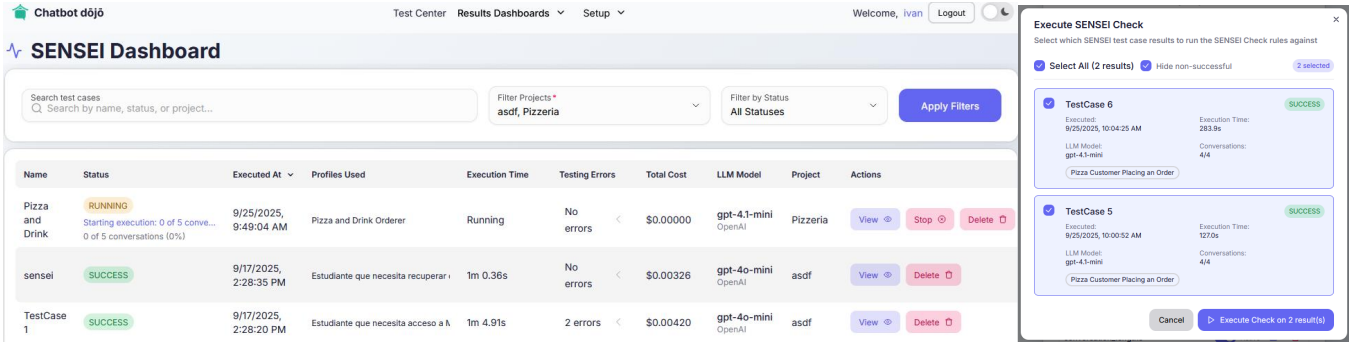


**Figure 5: Sensei-chat dashboard (a). Sensei-check dashboard (b).**

were not hallucinations, but stemmed from an over-granularity in distinguishing between semantically similar features. This confirms Tracer as a precise tool for black-box modelling of chatbots, as a previous step to their testing.

Concerning threats to validity, a construct threat is the inability to measure recall, as we lacked access to the chatbots' code to identify missing functionalities in the inferred models. This is common in black-box settings where ground-truth models are infeasible. With regards to external validity, our evaluation uses real-world chatbots, but only considers 5. Even if they are from different domains, we aim at expanding our evaluation in future work.

## 6 Related Work

Inferring a model of a system by observing its external behaviour has been applied to domains like software testing or reverse engineering. However, to our knowledge, Tracer is the first black-box tool for chatbot model learning. It requires neither the chatbot code nor dialogue logs, and it generates meaningful chatbot-specific test profiles by interacting with the chatbot.

Regarding chatbot testing, most automated proposals target a concrete technology [14] or need the chatbot source code [2, 7, 20]. Instead, our approach is technology-independent.

Real users can also assess chatbots, but since this is costly, researchers have proposed user simulators with diverse customisations options including Q&A corpora [21], conversational traits like engagement and verbosity [6], or personality traits like patience, cooperativeness and politeness [16, 18]. To simulate users, some proposals use neural networks trained on a dialogue corpus [10], perform retraining via retrieval augmentation and summary control [5], or prompt LLMs with a description of the user goals [19]. As for the output, all simulators save the simulated conversation, and only a few offer further data such as user satisfaction (answers' length, repeated answers, response time) and correctness (expected vs. actual answers) [21]. Compared to these user simulators, Sensei has greater customisability, covering not only user traits and persona, but also conversation goals, user inputs and expected chatbot outputs. It is technology-agnostic, and can check for crashes, timeouts, loops, unfinished goals, and chatbot-specific testing rules.

## 7 Conclusions and Future Work

Given the ubiquity of chatbots and conversational user interfaces, there is an urgent need for automated mechanisms that ensure their quality. To fill this gap, we have presented Chatbot Dōjō, a web platform that automates chatbot testing independently of the chatbot technology. The platform automates chatbot exploration, chatbot interaction via focused profiles and a user simulator, and rule testing to examine conversations and chatbot outputs.

In the future, we will consider specific security testing strategies (e.g., prompt injection), and use the chatbot model extracted by Tracer for other purposes, like reengineering and modernisation.

## Acknowledgments

Work funded by the Spanish MICINN with projects PID2021-122270OB-I00, PID2024-155231OB-I00, and RED2022-134647-T.

## References

[1] Botium. last access in 2025. https://cyara.com/products/botium/.
[2] P. C. Cañizares et al. 2024. Coverage-based strategies for the automated synthesis of test scenarios for conversational agents. In *AST*. ACM, 23–33.
[3] J. de Lara, A. del Pozzo, E. Guerra, and J. Sánchez Cuadrado. 2026. Automated end-to-end testing for conversational agents. *J. Syst. Softw.* 233 (2026), 112685.
[4] I. Sotillo del Horno et al. 2025. Automated exploration of conversational agents for the synthesis of testing profiles. In *ICTSS (LNCS, Vol. 16017)*. Springer, 1–18.
[5] K. D. Dhole. 2024. KAUCUS: Knowledge augmented user simulators for training language model assistants. In *SCI-CHAT*. ACL, 53–65.
[6] R. Ferreira et al. 2024. Multi-trait user simulation with adaptive decoding for conversational task assistants. In *EMNLP*. ACL, 16105–16130.
[7] P. Gómez-Abajo, S. Pérez-Soler, P. C. Cañizares, E. Guerra, and J. de Lara. 2024. Mutation testing for task-oriented chatbots. In *EASE*. ACM, 232–241.
[8] Google's Dialogflow. last access in 2025. https://cloud.google.com/dialogflow.
[9] Google's Gemini. last access in 2025. https://gemini.google.com/.
[10] D. Griol et al. 2013. An automatic dialog simulation technique to develop and evaluate interactive conversational agents. *App. Art. Intell.* 27, 9 (2013), 759–780.
[11] LangGraph. last access in 2025. https://www.langchain.com/langgraph.
[12] Microsoft's AutoGen. last access in 2025. https://microsoft.github.io/autogen/.
[13] OpenAI. last access in 2025. https://openai.com/.
[14] Rasa, Rasa Test. [n. d.]. https://rasa.com/, https://tinyurl.com/r7mcwetu.
[15] R. Ren et al. 2019. Evaluation techniques for chatbot usability: A systematic mapping study. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1673–1702.
[16] A. Salle et al. 2022. CoSearcher: Studying the effectiveness of conversational search refinement and clarification through user simulation. *Inf. Retr. J.* 25, 2 (2022), 209–238.
[17] J. Sánchez Cuadrado, S. Pérez-Soler, E. Guerra, and J. de Lara. 2024. Automating the development of task-oriented LLM-based chatbots. In *CUI*. ACM, 1–17.
[18] I. Sekulic, L. Lu, N. S. Bedi, and F. Crestani. 2024. Simulating conversational search users with parameterized behavior. In *SIGIR-AP*. ACM, 72–81.
[19] S. Terragni et al. 2023. In-context learning user simulators for task-oriented dialog systems. arXiv:2306.00774 [cs.CL]
[20] M. F. Urrico, D. Clerissi, and L. Mariani. 2024. MutaBot: A mutation testing approach for chatbots. In *ICSE Companion*. ACM, 79–83.
[21] M. Vasconcelos et al. 2017. Bottester: Testing conversational systems with simulated users. In *IHC*. ACM, Article 73, 4 pages.