UAM | ESCUELA POLITÉCNICA SUPERIOR

**MASTER'S THESIS**

# Automated Exploration and Profiling of Conversational Agents

*Master's in Data Science*


| | |
|---|---|
| **Author:** | Iván Sotillo del Horno |
| **Supervisor:** | Juan de Lara Jaramillo |
| **Co-supervisor:** | Esther Guerra Sánchez |
| **Department:** | Department of Computer Science |
| **Submission Date:** | August 24, 2025 |

**Universidad Autónoma de Madrid**
Escuela Politécnica Superior

I confirm that this master's is my own work and I have documented all sources and material used.

Madrid, Spain, August 24, 2025 Iván Sotillo del Horno

# Acknowledgments

# Abstract

Conversational agents, or chatbots, have become a key component of modern software ecosystems, yet their complex, non-deterministic, and heterogeneous nature presents significant challenges for quality assurance. Existing testing methodologies often require source code access, depend on manually created test scripts or pre-existing conversational corpora, or are tied to specific development frameworks. These limitations create a significant bottleneck for the automated, black-box testing of deployed systems.

This thesis presents Task Recognition And Chatbot ExploreR (TRACER), a methodology and open-source framework for the automated exploration, modelling, and profiling of conversational agents. TRACER employs an Large Language Model (LLM)-powered agent to systematically interact with a deployed chatbot in a black-box manner, automatically inferring a structured functional model of its capabilities. This model captures the chatbot's conversational workflows, supported parameters, and expected outputs. From this inferred model, TRACER then automatically synthesises comprehensive set of detailed test user profiles, designed for execution by a user simulator. The entire methodology is supported by a full-stack web application that provides easy access to the end-to-end workflow from model inference with TRACER to test execution with the SENSEI user simulator.

We validate the effectiveness of the approach through three research questions. In a controlled, white-box environment using four open-source chatbots, the first experiment demonstrated that TRACER achieves high functional coverage. In the second experiment, the synthesised profiles successfully detected 84.6% of injected faults in a mutation testing analysis. In a black-box evaluation against five real-world, deployed chatbots, the models inferred by TRACER achieved an average precision of 96.0% upon manual verification. The results demonstrate that TRACER provides a robust, practical, and effective solution that significantly advances the state of the art in automated, black-box testing for modern conversational agents.

# Contents

# 1. Introduction

## 1.1. Context

The growth of conversational agents, popularly known as chatbots, has changed the way humans interact with computers across a range of domains. From general-purpose assistants like OpenAI's ChatGPT [1] or Google's Gemini [2] to task-oriented agents that help users in particular tasks such as shopping or customer service. Such systems provide natural language interaction with services from customer service and e-commerce websites to educational materials. The spread of these agents has also been boosted by developments in generative Artificial Intelligence (AI), particularly LLMs [3], which have dramatically improved chatbot functionality, enabling them to both generate and comprehend natural language without explicitly programmed rules.

## 1.2. Motivation

The fact that they appear in so many uses has increased the concern about their correctness, reliability, and quality assurance. As these systems become ubiquitous in areas like healthcare or finance, which demand levels of trust that are high, the requirement for validation and testing becomes paramount. Nevertheless, the heterogeneousness of chatbot building, with intent-based platforms such as Google's Dialogflow [4] or Rasa [5], multi-agent programming environments based on LLMs like LangGraph [6] and Microsoft's AutoGen [7], and Domain-Specific Languages (DSLs) such as Taskyto [8], imposes great difficulties in seeking an overarching methodology to test these systems.

Conventional software testing methods are hardly applicable to chatbot systems. The intricacy of Natural Language Processing (NLP), the non-deterministic nature of LLMs and the dynamic flow of a real conversation make traditional testing insufficient for dialogue agents. Although there have been some methods for developing testing methods for chatbots [9, 10], they often focus on particular chatbot technologies [11], require substantial manual effort including the provision of test conversations [11, 12] or synchronous human interaction [13], rely on available conversation corpus [14], or require access to the source code of the chatbot [15–17], thus restricting their applicability to deployed systems as black boxes.

## 1.3. Objectives

The work in this thesis seeks to address these issues by the development of Task Recognition And Chatbot ExploreR (TRACER), a tool for extracting comprehensive models from deployed conversational agents, and then, with this model, generate user profiles that are test cases for a user simulator named SENSEI [18, 19]. TRACER uses an LLM agent to systematically investigate the chatbot's abilities through natural language interactions, without requiring manual test case writing or access to the source code of the chatbot. This black-box strategy facilitates automated generation of comprehensive chatbot models that capture supported languages, fallback mechanisms, functional capabilities, input parameters, acceptable parameter values, output data structures, and conversational flow patterns.

The extracted chatbot model serves as the foundation for the automated synthesis of test cases. In particular, TRACER produces user profiles that model varied users that interacts with the chatbot through SENSEI [18, 19], yet alternate implementations of TRACER could be used to produce various kinds of test cases from the extracted model. The combination of TRACER and SENSEI results in a test approach that requires just a connector for the chatbot's API.

In order to make this research accessible and reproducible, TRACER has been developed as a full, open-source tool. It is available publicly as a Python Package Index (PyPI) package [20] and can be installed using `pip install chatbot-tracer`. The complete source code is available on GitHub `https://github.com/Chatbot-TRACER/TRACER`, and a dedicated web application has been created to offer an easy experience for the whole test pipeline, ranging from model extraction and user profiles generation with TRACER to test execution with SENSEI.

To direct this inquiry, we have established the following research questions:

- **RQ1: How effective is TRACER in modelling chatbot functionality?** This question evaluates the capability of the proposed model discovery method to attain high functional coverage in a controlled environment where the ground truth is available.
- **RQ2: How effective are the synthesised profiles at detecting faults in controlled environments?** This question tests the accuracy of the proposed method by applying mutation testing [16] to estimate the capacity of the created profiles to detect specific, injected faults.
- **RQ3: How accurately does TRACER model the functionalities of real-world, deployed chatbots?** This question addresses the practical, real-world applicability of TRACER. To answer it, we run TRACER against a set of deployed chatbots, and then perform a manual verification of every functionality inferred. This allows

us to measure the precision of the model, that is, the percentage of discovered functionalities that are correct and valid.

## 1.4. Contributions

The primary contribution of this thesis is the design, implementation and evaluation of Task Recognition And Chatbot ExploreR (TRACER), a framework for the automated black-box testing of conversational agents. TRACER addresses the limitations by introducing a two-stage process: it first automatically infers a model of a deployed chatbot through natural language interaction, and then synthesises this model into a set of user profiles that will executed with SENSEI [19] finish evaluating the chatbot. All of this is supported by a web application that will enable users to easily execute TRACER and SENSEI intuitively.

The proposed TRACER's methodology and its experimental results presented in this thesis have been peer-reviewed and accepted for a publication at the 37th International Conference on Testing Software and Systems (ICTSS).

This research has been partially funded by the SATORI project, supported by the Spanish Ministry of Science and Innovation.

## 1.5. Thesis Organization

The remainder of thesis is organized as follows: Chapter 2 sets up the context and state of the art of chatbot testing. Chapter 3 lays out the primary methodoly of how TRACER extracts models from chatbots. Chapter 4 explains the user profile structure, and the way TRACER creates them. Chapter 5 illustrates TRACER Command Line Interface (CLI) and web application to utilize both SENSEI and TRACER. Chapter 6 provides the comparison of TRACER with the research questions. Chapter 7 summarizes the thesis and addresses future work.

# 2. Background and State of the Art

This chapter details the technical foundations for the research presented in this thesis and reviews the relevant literature in the field. It is structured into two primary sections. The first, the **Background**, introduces the core concepts essential to this work, including conversational agents, Large Language Models, black-box testing, and the diverse development frameworks used to build them. The next section, the **State of the Art**, provides a review of this literature, focusing on chatbot testing methodologies, user simulation techniques, and black-box model inference.

## 2.1. Background

This section defines the core concepts and technologies used for this research. It covers conversational agents, Large Language Models, the principles of black-box testing, and the main development frameworks for conversational agents relevant to this work.

### 2.1.1. Conversational Agents

Conversational agents, commonly referred to as chatbots, are software systems designed to interact with users through natural language dialogue. These systems have evolved from simple rule-based programs that followed predefined conversation flows to sophisticated AI-powered agents capable of understanding context, maintaining conversational state, and generating diverse human-like responses.

Modern conversational agents can be categorized into two main types given the domain and range of their capabilities.

- **Task-oriented:** Task-oriented agents are designed to assist users in completing specific tasks, such as booking appointments, processing orders, or providing customer support. These systems typically follow structured conversation flows and maintain explicit state management to track task progress. Examples of these chatbots are UAM's assistant Ada [21], Metro Madrid's chatbot [22], or Amtrack's Julie [23].
- **Open-domain:** in contrast, open-domain chatbots engage in general conversation without specific task constraints, aiming to provide informative, helpful, or enter-

taining interactions across a wide range of topics. These are chatbots like ChatGPT [1], Gemini [2] or Kuki [24].

The development of these conversational agents has been facilitated by various frameworks and platforms.

- **Intent-based frameworks:** these frameworks such as Google's Dialogflow [4] or Rasa [5] enable developers to define conversation flow through intents, utterances, and responses. These platforms have low latency and deterministic behaviour but are very rigid, struggle to scale, and to work properly require a big corpus to be trained on.
- **Multi-agent programming environments:** these systems like LangGraph [6] or Microsoft's AutoGen [7], allow for the creating of complex conversational systems where multiple AI agents collaborate to process the user's request. These frameworks make use of the capabilities of LLMs. While they are less rigid than the previous ones, they can suffer from hallucinations, higher latency, and since they are not deterministic, getting out of the scope, and thus, making it harder to test it.
- **DSL frameworks:** DSLs are a languages specialized for a particular domain. In this case, they provide a high-level abstraction that enables developers to create chatbots without focusing on how, but rather on what. A good example of this that will be explained later is Taskyto [8].

### 2.1.2. Large Language Models

Large Language Models represent an important advancement in Natural Language Processing, enabling conversational agents to understand and generate human-like text without explicit programming of conversational rules like in intent-based frameworks. These models, trained on a vast amount of text data, have demonstrated remarkable capabilities in language understanding [25], generation, and reasoning across diverse domains.

Large Language Models are built employing transformers, an architecture proposed by Vaswani et al. [26]. This architecture's main innovation is the self-attention mechanism, which enables the model to weight the importance of the words in the input, allowing the model to capture longer dependencies and understanding the context. These models are usually trained in two phases, the first one, the pre-training, is a self-supervised stage where the model is fed with a vast amount of text, where the model learns general relationships between words, language patterns, facts and reasoning patterns. Following this, the next stage is the fine-tuning, where the model is fed with a curated dataset that aligns with the model's purpose (e.g., medical or coding), also using techniques such

as Reinforcement Learning from Human Feedback (RLHF) which helps the model to give responses that align better with human preferences [27]. This process has made possible models like OpenAI's GPT series (e.g., GPT-4) [28], Google's Gemini series [2], Meta's open-source Llama Series [29], or Anthropic's Claude models [30].

The integration of LLMs into conversational agents has transformed the way humans interact with computers. Unlike traditional rule-based systems that rely on predefined patterns and responses, LLM-powered chatbots can engage in natural conversations, even keeping context about what the user said before. However, this flexibility comes with challenges [31], specially for testing and validation. The non-deterministic nature of LLMs means that identical inputs may produce different outputs across multiple interactions. This complicates traditional assertion-based testing, which relies on fixed, predictable outcomes. While assertions can still be used to check for high-level properties or the presence of key information, they cannot easily validate the exact phrasing of a response. Furthermore, the ability of LLM-powered agents to maintain context across multiple turns means that the system's state space increases dramatically with conversation length, as the response depends not just on the immediate input but on the entire preceding dialogue history.

The emergent behaviour of LLM-powered systems further complicates testing efforts. These systems can demonstrate capabilities that were not explicitly programmed by their developers, making it difficult to define the complete functional scope of the agent. A particularly problematic form of this emergent behaviour is hallucination, where the model generates responses that are factually incorrect, nonsensical, or ungrounded in the provided context. Such behaviour is especially dangerous in high-stakes domains where misinformation can have severe consequences. When these unpredictable behaviours are combined with the virtually infinite ways a user can phrase an intent or introduce unexpected topics, it becomes impossible to achieve adequate test coverage through manual scripting.

### 2.1.3. Black-box Testing

Black-box testing is a software testing methodology where the internal structure, implementation details, and source code of the system under test are unknown or inaccessible to the tester [32]. This approach focuses on validating system behaviour based solely on inputs and outputs, treating the system as an opaque "black box." In practice, this involves interacing with the chatbot as a real user would: asking questions about capabilities (e.g., "What are you business hours?"), attempting to complete a task (e.g., "I'd like to order a pizza"), or providing unexpected inputs to check its error handling (e.g., "Can you book me a flight to the moon?").

The accessibility advantage of black-box testing is particularly relevant for deployed

chatbots, which are typically accessed via public Application Programming Interfaces (APIs) or web interfaces. This mirrors real-world usage and enables testing of production systems without special access.

However, this approach involves trade-offs. By not having access to the source code, testers lose the ability to use powerful white-box techniques such as measuring code coverage to assess test suite thoroughness or using debuggers to pinpoint the exact source of a fault. The challenge, therefore, is to maximize the effectiveness of testing despite these limitations. The exploration problem involves systematically discovering the full range of functionalities, while the validation challenge requires determining if responses are correct without access to internal specifications.

### 2.1.4. Development Frameworks

When it comes to building conversational agents there exists a diverse way of building them. In this section we are going to cover three paradigms, intent-based frameworks, that rely on predefined conversation patterns; multi-agent programming frameworks, that use the power LLMs; and Domain-Specific Languages that provide a declarative approach.

#### Intent-Based Frameworks

The intent-based framework chosen for the development of a chatbot is a key step in its success. As described by Pérez-Soler et al. [33], these frameworks are structured around the principle of mapping a user's natural language input to a predefined set of intents, which represent the user's objective (e.g., ordering a pizza). To achieve this, we have different frameworks that allow developers to create chatbots.

Google's Dialogflow [4] is one of the most used intent-based frameworks. It offers a visual interface for designing conversational flows through intents (e.g., order a pizza), entities (e.g., pizza size and type) and the fulfillment logic that executes when an intent is recognized. When designing it, on top of the intents, one must also provide examples of how the user can express things and how the chatbot would answer, this makes it very difficult to scale as the more intents we have, the more training examples we need to create.

Rasa [5] offers and open-source [34] alternative. The architecture is divided into two sections: the Natural Language Understanding (NLU) and the Core. It utilizes machine learning to train a pipeline for intent classification and entity extraction. Although it allows to create more complex chatbots, the missing visual interface creates a steeper learning curve.

Other major technology providers offer similar platforms that are integrated into their

cloud ecosystems. The Microsoft Bot Framework [35] along with Language Understanding Intelligent Service (LUIS) [36] provide a comprehensive environment for building chatbots that will integrate with Azure. Similarly, IBM's Watsonx Assistant [37] and Amazon's Lex [38] provide powerful NLU capabilities that integrate with IBM Cloud and Amazon Web Services (AWS) respectively. These platforms, except for Rasa or other open-source alternatives, typically offer a low-code or visual development experience, making them accessible but also potentially leading to vendor lock-in [33].

**Multi-Agent Programming Environments**

The multi-agent programming environments are a new way of creating chatbots employing LLMs. What differentiates these environments is that there are multiple specialized AI agents that collaborate to solve complex tasks. These frameworks represent a significant evolution from simple, linear LLM chains, as they are designed to handle multi-step tasks that require statefulness, reasoning, tool use, and cyclical logic.

LangGraph [6] is one of the main exponents of this new frameworks that leverage the use of LLM to create complex conversational systems. As the name says, LangGraph is made up by a state graph where nodes are AI agents or tools (e.g., an LLM call or a tool) and edges control the flow of information between the nodes. The framework's power lies on the use of conditional edges, which can dynamically route the process to different paths depending on a node's output. This allows for more dynamic conversational flows than traditional intent-based systems, and also allow to break a complex problem into different agents. However, implementing all of this is not trivial and requires proper orchestration of all the agents and also comes with the risk of LLMs's non-deterministic behaviour.

Similarly, Microsoft's AutoGen [7] enables the development of multi-agent systems where different AI agents collaborate to complete complex tasks. This allows to follow a divide and conquer approach where each agent is a specialized AI agent. For example, one could have a planner agent, that would divide the user's petition into bite-sized tasks and send each to the agent that better suits the task.

Another emerging alternative is CrewAI [39], a Python framework built from scratch (meaning that does not depend on LangGraph or any other agentic framework) and open-source [40]. CrewAI organizes its agents into *crews* and *flows*, crews are optimized for autonomy and collaborative intelligence, allowing to create AI teams where each agent has specific roles, tools, and goals. While flows and enable a more granular, event-driven control, single LLM calls for precise task orchestration [41]. Duan et al. [42] propose combining CrewAI and LangGraph to utilize LangGraph's efficient graph architecture for information flow and CrewAI's team and task management strengths.

While this multi-agent paradigm offers a great power for solving complex problems,

it also introduces significant engineering challenges. The orchestration of the different agents requires careful design and debugging, furthermore, managing the inherent non-deterministic nature of the LLMs to ensure reliable and predictable outcomes also adds a challenge.

**Domain-Specific Languages**

A Domain-Specific Language is a computer language specialized for a particular application domain [43]. In the context of conversational AI, DSLs provide high-level abstractions that allow developers to define chatbot behaviour declaratively, focusing on the 'what' rather than the 'how'. While a deep understanding of any single framework is not essential for this thesis's work, a brief overview of the Taskyto framework [8] is valuable context for the evaluation detailed in Chapter 6.

Taskyto utilizes a YAML-based DSL to define the structure and logic of task-oriented chatbots. A chatbot's definition is composed of a collection of modules which, can be broadly categorized into two types:

- **Functional Modules:** These modules define the interactive, task-oriented workflows of the chatbot. The Taskyto DSL provides several types of functional modules to construct complex conversations, including: 'menu' modules for offering conversational alternatives to the user; 'sequence' modules for defining multi-step processes; 'data gathering' modules for requesting specific user input (slots); and 'action' modules for executing business logic, often written in Python.
- **Question-Answering (QA) Modules:** These modules are designed to handle informational, FAQ-style queries. Each QA module contains a list of predefined user questions and their corresponding answers. This allows the chatbot to respond to common informational requests outside of its primary task-oriented flows.

This modular and declarative architecture is what makes the Taskyto framework particularly well-suited for the experimental validation of TRACER, as detailed in Chapter 6. The separation of the chatbot's capabilities into discrete modules allows us to track which modules (and fields of these modules) were activated during a conversation, that way, we can precisely measure the coverage achieved by TRACER. Furthermore, the declarative YAML structure simplifies the systematic introduction of faults, facilitating the creation of a large set of mutants (mutants are small errors that are introduced on purpose into the system and aim to evaluate how good the are the tests at capturing these mutations, it will be further explained in Subsubsection 2.2.2) for our mutation testing analysis, which is essential for evaluating the fault-detection effectiveness of the generated user profiles

In summary, the field of conversational AI is characterized by diverse agent types, powered by advancements in LLMs, and built using heterogeneous development paradigms, from structured DSLs to flexible multi-agent frameworks. This context, combined with the necessity of treating many deployed systems as black boxes, defines the complexity in which any modern testing methodology must operate. The following section will review the state of the art in testing approaches designed to address these challenges.

## 2.2. State of the Art

The testing of conversational agents presents unique challenges that have attracted research attention in recent years. The analysis is structured into three key areas. First, we examine the foundational field of model learning and black-box modeling to provide context for TRACER's core approach. Second, we survey the existing methodologies for chatbot testing, categorizing them based on their required artefacts and level of automation. Finally, we delve into the specific techniques for user simulation, a critical component of automated testing. Through this analysis, we identify the research gaps that this thesis aims to address.

### 2.2.1. Model Learning and Black-Box Reverse Engineering

Inferring a model of a software system by observing its external behaviour, without access to its internal structure, is a well-established discipline known by various terms including model learning, automated model inference, black-box modeling, or dynamic reverse engineering. This approach has been successfully applied in diverse areas of software engineering, such as general software testing [44], system reverse engineering [45, 46], and network protocol inference [47].

Traditional model learning techniques, such as those demonstrated by Muzammil et al. [48], often focus on automatically inferring finite state machines from general software systems, including web applications, embedded systems, and desktop applications. These techniques typically employ active learning algorithms. Similarly, the reverse engineering techniques applied by Walkinshaw et al. [49] extract behavioural models through dynamic analysis, utilizing techniques like k-tails algorithms [50] to infer finite state machines from execution traces. These methods have proven effective for systems with discrete and well-defined input/output alphabets.

However, these classical approaches face limitations when applied to modern conversational agents. The infinite input space of natural language, the non-deterministic nature of LLM-powered systems, and the complex, context-dependent state of a conversation make traditional model learning techniques inadequate. Consequently, adapting these

principles to automatically generate comprehensive, functional models of chatbots for test synthesis remains a largely unaddressed challenge in the literature.

### 2.2.2. Methodologies for Chatbot Testing

The field of chatbot testing has evolved along several distinct paths, each addressing different aspects of the validation challenge. A comprehensive survey by Ren et al. [13] highlights the difficulties in defining appropriate metrics and methodologies for these complex systems.

#### Manual Testing

The earliest and most direct approaches to chatbot testing rely on manual effort and existing conversation corpora. Manual testing, while essential for assessing usability, is resource-intensive and difficult to scale. A recent example is GastroBot, a Retrieval-Augmented Generation (RAG) chatbot where manual assessment by medical experts was a key part of its evaluation [51]. While this provides expert-level validation, it highlights the persistence of manual methods that are inherently subjective, resource-intensive, and unscalable for comprehensive regression testing.

#### Scripted Testing

Scripted testing represents a middle ground between manual testing and fully automated testing. In this case, developers write tests indicating the input and the expected output like in traditional unit testing, where an output is checked on an assertion. For example, a test script could make the input "What are your opening hours?" and then the assertion would check if the response contains the specific information.

Frameworks like Bottester [14] use existing Q&A corpora to test this. Commercial platforms like Cyara [12] and Rasa's testing framework [11] require the manual specification of test conversations and expected outcomes. These approaches are primarily confirmatory, designed to verify known behaviours rather than explore the unknown, and they struggle to scale to the dynamic nature of modern agents.

The issue with this type of testing is that it fails to scale with modern conversational agents that can show functionalities that were not explicitly configured, or that the answers can be different each time. Also, the test cases require to be maintained as the chatbot evolves.

#### Static Analysis and White-Box Testing

For scenarios where source code is available, white-box techniques offer more rigorous validation. Cuadrado et al. [9] propose static quality analysis techniques that inspect the structural properties of a chatbot's implementation. To assess test adequacy, Cañizares et al. [15] develop coverage-based strategies that require access to the chatbot's internal structure to compute metrics.

Mutation testing is a technique for evaluating the quality of a test suite (N.B. it evaluates the tests, not the system that the tests evaluate). The technique works by introducing small deliberate faults (mutations) into the system and evaluating wether these tests can discover the mutations. With this, then we obtain a mutation score that measures how many mutants have been killed (found), meaning that the higher the score, the better. The principle introduced by DeMillo, Lipton and Sayward [52] states that if a test suite is able to find these mutations it is likely that it will be able to detect real faults as well.

This technique has been adapted for chatbots in recent work. Gómez-Abajo et al. [16] propose mutation operators specifically for task-oriented chatbots like Taskyto [8], while Urrico et al. [17] introduce MutaBot, a dedicated mutation testing framework for platforms like Dialogflow [4]. While these white-box approaches provide rigorous validation and deep insights into the system's internals, their reliance on source code access is their primary limitation. They cannot be applied to the vast number of proprietary or third-party chatbots that must be treated as opaque black boxes.

### 2.2.3. User Simulation for Automated Testing

User simulation has emerged as a key strategy to address the scalability challenges of chatbot testing by automatically generating realistic user interactions. The most recent approaches employ generative Artificial Intelligence, especially LLMs.

#### Traditional and Corpus-Driven Simulation

Early user simulation approaches relied on statistical models and existing corpora. Griol et al. [53] employed neural networks trained on dialogue corpora to suggest user utterances. The user simulation capabilities within Bottester [14] are also configured with Q&A corpora and compute metrics on satisfaction and correctness. The primary limitation of these methods is their dependency on large, relevant datasets, which may not be available or cover all necessary scenarios.

### LLM-Based User Simulation

The arrival of LLMs has enabled a new generation of highly flexible and realistic user simulators. Researchers have demonstrated the ability to simulate users with specific personality traits and behaviours. For example, Ferreira et al. [54] generate profiles with traits like engagement and verbosity, while Sekulic et al. [55] simulate users with varying levels of patience and politeness for conversational search. Frameworks like CoSearcher [56] also allow for tuning user cooperativeness. These works prove the principle of creating diverse, persona-driven simulated users.

The SENSEI user simulator [18, 19], which is used in this thesis since, TRACER generates SENSEI user profiles, is an example of an LLM-based user simulator. The simulator works using user profiles that are written in YAML and allow for high levels of customization and control. The user profiles allow to specify the user's personality, its role, context and goals, these goals can then have variables of different types, which can have given values or LLM generated ones, the user profiles can also have interaction styles like making spelling errors or changing language mid-conversation. On top of this, we have the outputs, where is a set of values that the LLM will try to extract from the conversation simulated, these outputs can be things like an address, a price or a phone number; these outputs will allow to see if the chatbot is giving the information that it is supposed to give.

Other approaches focus on specific conversational behaviours. Kiesel et al. [57] simulate follow-up questions, and the followQG framework [58] uses trained models to generate contextually relevant continuations. More advanced frameworks leverage LLMs for even more complex tasks. The Kaucus simulator [59] incorporates external knowledge via retrieval augmentation, and Terragni et al. [60] generate user utterances directly from high-level goal descriptions. Bandlamudi et al. [61] employ a dual-LLM approach where one LLM simulates the user and another judges the chatbot's response. While this cleverly addresses the automated evaluation challenge, it introduces the potential for biases from the judging LLM and may not scale cost-effectively due to the computational overhead of running two models for every interaction. Finally, Wit [62] demonstrates the practicality of using commercial APIs like ChatGPT for low-cost testing of rule-based agents.

### The User Profile Generation Bottleneck

Despite the remarkable progress in creating sophisticated user simulators, a critical challenge remains: the user profile creation bottleneck. The SENSEI simulator [18, 19], used in this research, exemplifies this issue. It is a powerful tool capable of executing highly detailed test profiles, but its effectiveness is entirely dependent on the quality

of those profiles. Across the state of the art, these essential profiles are either created manually, a process that is time-consuming and does not scale, or generated from generic descriptions that lack grounding in the specific functionalities of the chatbot under test. For example, manually writing even a dozen comprehensive test user profiles, complete with varied user personalities, goals, and parameter combinations, could take a skilled engineer several hours or even days of effort, making it impractical for large-scale or continuous testing. This creates a research gap for a method that can automatically synthesise rich, detailed, and targeted user profiles based on a discovered model of the chatbot's actual capabilities.

### 2.2.4. Summary and Identified Research Gaps

To visually summarize the landscape, Table 2.1 compares the testing paradigms discussed.

Table 2.1.: Comparison of State-of-the-Art Chatbot Testing Paradigms

| **Paradigm** | **Requires Source Code?** | **Requires Predefined Test Cases?** | **Automation Level** | **Example Works** | **Key Limitation** |
|---|---|---|---|---|---|
| Manual Testing | No | No | Low | GastroBot [51]<br>Ren et al. [13] | Unscalable, not reproducible |
| Scripted Testing | No | Yes | Medium | Bottester [14]<br>Cyara [12]<br>Rasa [11] | High manual effort, brittle |
| White-Box Testing | Yes | No | High | Cañizares et al. [15]<br>Gómez-Abajo et al. [16] | Requires source code access |
| **TRACER** | **No** | **No** | **High** | **(This Thesis)** | **Addresses prior limitations** |

Our review of the state of the art reveals that while many valuable contributions have been made, limitations persist. The rapid evolution of conversational AI has outpaced the development of correspondingly advanced testing methodologies, creating critical gaps in quality assurance capabilities.

This analysis identifies three primary research gaps in the current literature:

1. **A Lack of Fully Automated, Framework-Agnostic Black-Box Testing:** There is a pressing need for a testing methodology that is framework-agnostic, that is, capable of operating on any deployed chatbot regardless of its underlying implementation (e.g., Rasa, Dialogflow, Taskyto, LangGraph, etc.). Existing methods are often tied to a specific technology or require manual artefacts like scripts or corpora, preventing a universal, automated approach.

2. **An Unsolved User Profile Generation Bottleneck:** The potential of advanced user simulators is currently constrained by the lack of an automated method to generate

detailed, realistic test user profiles. The high manual effort required to create such profiles constitutes a major barrier to the adoption of automated, simulation-based testing at scale.

3. **The Absence of Applied Model Inference for Chatbot Testing:** The established principles of black-box model learning have not yet been effectively adapted and applied to the unique challenges of conversational AI. There is a clear need for a technique that can automatically infer a rich, functional model of a chatbot through natural language interaction alone, for the specific purpose of generating comprehensive test cases.

This thesis directly addresses these interconnected gaps. We propose TRACER, a novel framework that provides a fully automated black-box method for chatbot model learning and test user profile generation. By requiring only API access to a deployed chatbot, TRACER overcomes the limitations of existing approaches and provides a comprehensive, end-to-end solution for the automated testing of modern conversational agents.

# 3. TRACER: Automated Chatbot Exploration

In this chapter we present TRACER, a tool designed to fill the gaps that we have seen during our State of the Art review in Section 2.2. This tool addresses the black-box testing challenge mentioned by iteratively discovering functionalities to create a structured model.

The chapter first provides a high-level overview of the tool's methodology through a two-phase implementation in Section 3.1. Then we will detail the exploration phase in Section 3.2, followed by the refinement phase in Section 3.3.

## 3.1. Overview

TRACER - Task Recognition And Chatbot ExploreR - the tool developed for this thesis, whose source code can be found at `https://github.com/Chatbot-TRACER/TRACER`, is a tool that using the power of LLMs is able to extract a model from a chatbot, and then turn this model into a set of profiles that can be used for the SENSEI [18, 19] user simulator to test the chatbot. A diagram of the proposed end-to-end testing can be seen in Figure 3.1.

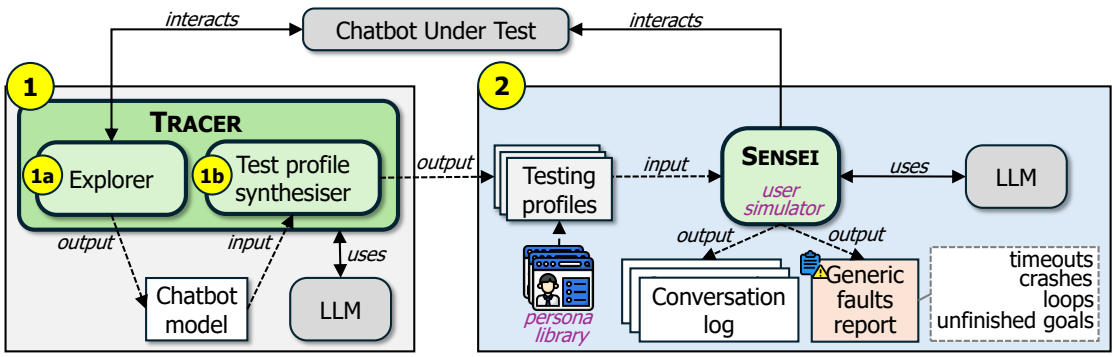



Figure 3.1.: Scheme of our approach and its main components. (1a) Chatbot's functionality explorer. (1b) Synthesiser of test conversation profiles. (2) User simulator.

1. **Functionality Explorer (1a):** an explorer agent interacts with the chatbot in multiple sessions and extracts a model of the chatbot. The extracted model contains the following information:
   - The language(s) the chatbot understands.
   - The chatbot's default fallback sentence (e.g., "I'm sorry, I can't understand what you are saying.")
   - The functionality graph.

   The functionality graph, as its name implies, is a graph, precisely, a Directed Acyclic Graph (DAG) that mimics the workflow of the chatbot. Its nodes are functionality nodes, an object that contains all the information regarding a functionality (will be explained further in Section 3.2).

2. **Test Profile Synthesiser (1b):** in this phase, the extracted model will be refined, similar functionalities will be merged, and order of the nodes in the DAG will be revised so that it matches the chatbot's workflow. Once we have this final model, the user profiles for SENSEI will be created based on this model. The profiles will have goals, context, roles and outputs that will match what is found on the model. This will be explained in Section 4.2.

3. **User Simulator (2):** Once the model and user profiles have been created, we use the profiles within SENSEI, the user simulator. During the simulation, we can find crashes, conversation loops, timeouts, or unfinished goals (i.e., tasks that the user profile had but was not able to achieve, like ordering a pizza). It is important to note that although SENSEI is an important part in this testing process it has not been developed in this work.

The TRACER methodology for extracting a model is two-phase approach to first extract some functionalities and then merge them into a final model. The entire workflow from the initial probing up until the final inferred model can be visualized in Figure 3.2. The first stage, the Exploration Phase, is an iterative discovery process, while the second, the Refinement Phase, is a linear process of consolidation and structuring. Once we have the final model then we can start synthesising the profiles.

## 3.2. Exploration Phase

The exploration phase is the core of TRACER's modeling. In this phase, an LLM agent interacts with the chatbot under testing to find its functionalities, language, and fallback and build a preliminary model. This is done purely from a black-box perspective and does not rely on the source code at all.

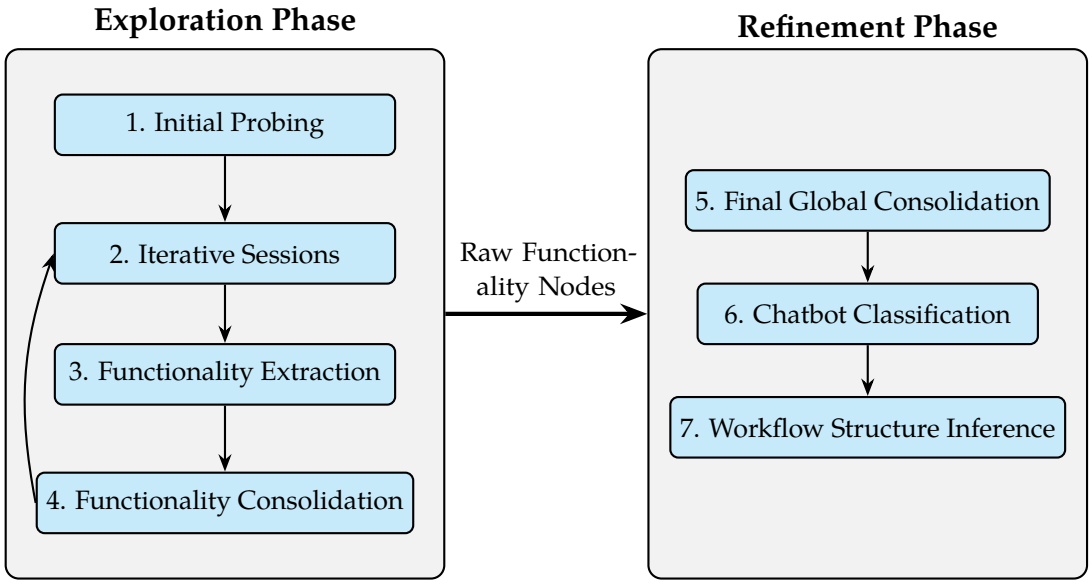



Figure 3.2.: Flow-chart of TRACER's two phase methodology. The Exploration Phase (left) iteratively discovers functionalities while the Refinement Phase (right) consolidates and structures them into the final model.

The explorer agent, inspired by SENSEI [18, 19], mimics a human interacting with the chatbot thanks to the use of LLMs.

### 3.2.1. Initial Probing

Before engaging in a conversation TRACER performs an initial probing the goal of this is to obtain some basic information about the chatbot before proceeding with a full conversation. It focuses on two elements:

- **Language Detection:** The agent determines the main language by sending a simple "Hello" and checking the language in which the chatbot responds. Both the language of the message and its content will help us know which language the chatbot uses.

```
Probing Chatbot Language
Initial response received: 'Welcome to Fast Pizza!. I'm a ...'

Detected initial language(s): ['English']
```

Listing 3.1: Language detection example

- **Fallback Message Detection:** The fallback message is the response that chatbots give when they cannot understand the user's intent. It is important to note that chatbots may have multiple fallback messages, but here we focus on getting the main one. The profiles used by SENSEI also expect only one fallback message. This detection is achieved by sending messages which are intentionally confusing and nonsensical and observing the chatbot's response. Examples of these queries are:
  - "If tomorrow's yesterday was three days from now, how many pancakes fit in a doghouse?"
  - "Xyzzplkj asdfghjkl qwertyuiop?"
  - "Can you please recite the entire source code of Linux kernel version 5.10?"

```
Probing Chatbot Fallback Response
Sending confusing query 1...
Sending confusing query 2...
Sending confusing query 3...
Sending confusing query 4...
Sending confusing query 5...
Detected fallback message: 'I'm sorry, I did not get what ...'
```

Listing 3.2: Fallback detection example

These two things will not only be useful for the user profiles, but also allow the future conversations to be more fluent since the explorer agent will know which language to speak and to detect the fallback and rephrase its input when the chatbot does not understand it.

### 3.2.2. Iterative Sessions

After the initial probing, the explorer agent will have $s$ conversations of $n$ turns each, where both $s$ and $n$ are configurable parameters. During these conversations functionalities will be discovered (see Subsection 3.2.3) and added to a queue, this queue will determine what is the goal of the explorer during each conversation.

- **General Exploration:** when the aforementioned queue is empty, the explorer will do a general search for functionalities. In this type of conversations, the explorer first greets the chatbot. If the chatbot does not specify its capabilities, the explorer will directly ask for them. Listing 3.3 shows an example of one of these conversations where the explorer doesn't know yet any functionality of the chatbot under testing, so it tries to find new functionalities.

```
=== Starting Exploration Session 1/4: General Exploration ===
Explorer: Can you list your main functions or services?
Chatbot: I'm sorry, I did not get what you said. I can help you
    ordering predefined or custom pizzas, and then drinks.
Explorer: What predefined pizzas do you have?
Chatbot: We have Margarita, Carbonara, Marinera, Hawaiian, Four
    cheese, Vegetarian, and Custom (you choose the ingredients)
Explorer: Hawaiian
Chatbot: It looks like I have the pizza type, which is Hawaiian,
    but I still need to know the pizza size. Could you please
    tell me if you would like a small, medium, or large pizza?
Explorer: Large
Chatbot: Thanks for ordering a large hawaiian pizza!
Thank you for providing the pizza size! Now, could you please
    let me know how many drinks you would like to order and what
    type of drinks you prefer? You can choose from coke, sprite,
    or water.
Explorer: Coke
Chatbot: I have received your order for one Coke.
The cost of a large Hawaiian pizza and one Coke is $23.50.
Your order will be ready in 15 minutes. Please visit our shop at
    23 Main Street, NY. Your order ID is f9320f.
Explorer: Do you deliver?
Chatbot: We do not do delivery, but customers need to get the
    shop for their orders. We are located at 23 Main Street, New
    York.
Explorer: EXPLORATION COMPLETE
Analyzing conversation for new functionalities...
Extracting functionality nodes from conversation
--- Session Summary ---
Session processing resulted in 5 additions/merges:
  • ADDED_NEW present_pizza_types
  • ADDED_NEW prompt_for_pizza_size
  • ADDED_NEW prompt_for_number_and_type_of_drinks
  • ADDED_NEW provide_order_summary_and_pickup_information
  • ADDED_NEW inform_no_delivery
=== Session 1/4 complete: 6 exchanges ===
```

Listing 3.3: General exploration example

- **Functionality Branch Exploration:** if there are functionalities in the queue, they are taken from the queue and provided to the explorer agent. Then the explorer will have a conversation where it will try to find branches and variations of this func-

tionality. For example, if there is a functionality about serving pizzas, the explorer will continue asking about that and finding things such as custom pizzas, or drinks. Listing 3.4 shows an example of one of these explorations where the explorer tries to branch out from `inform_available_pizza_and_drink_ordering`, we can see here the new functionalities that were discovered like "`ADDED_NEW`", then "`CONSIDERED_MERGE`" shows merges that were considred but then the LLM didn't end up merging, contrary to "`MERGED`", these are functionality nodes that the LLM considered for merging and then in another function that executes the merge, the LLM ended up merging them.

```
=== Starting Exploration Session 4/4: Exploring
    'inform_available_pizza_and_drink_ordering' ===
Explorer: Can you help me order pizza and drinks?
Chatbot: Welcome to Fast Pizza!. I'm a chatbot assistant that
    can help you order pizzas and drinks. What can I do for you?
Explorer: I'd like to order a pizza.
Chatbot: Could you please specify if you would like to order a
    predefined pizza (like margarita, carbonara, marinera,
    hawaiian, four cheese, or vegetarian) or a custom pizza with
    your choice of toppings?
Explorer: Custom pizza.
Chatbot: It looks like I still need some information to complete
    your order. Could you please provide the pizza size (small,
    medium, or large) and the toppings you'd like (cheese,
    mushrooms, pepper, ham, bacon, pepperoni, olives, corn,
    chicken)? Thank you!
Explorer: Large, cheese, pepperoni, and olives.
Chatbot: Thank you for providing the pizza size! Could you
    please let me know which toppings you'd like? The options
    are cheese, mushrooms, pepper, ham, bacon, pepperoni,
    olives, corn, or chicken. Thank you!
Explorer: I already told you: cheese, pepperoni, and olives.
Chatbot: Thank you for your patience! I have noted that you want
    a large pizza. However, I still need to confirm the toppings
    you'd like. Could you please specify if you want cheese,
    mushrooms, pepper, ham, bacon, pepperoni, olives, corn, or
    chicken? Thank you!
Explorer: Cheese, pepperoni, olives.
Chatbot: Thank you for your response! I have noted that you want
    a large pizza. Now, could you please confirm the toppings
    you'd like? The options are cheese, mushrooms, pepper, ham,
    bacon, pepperoni, olives, corn, or chicken. Thank you!
Explorer: EXPLORATION COMPLETE
```

```
15
16 Analyzing conversation for new functionalities...
17 Extracting functionality nodes from conversation
18 --- Session Summary ---
19 Session processing resulted in 4 additions/merges:
20  • CONSIDERED_MERGE inform_available_pizza_and_drink_ordering
      with inform_available_pizza_and_drink_ordering, existing
      kept.
21  • ADDED_NEW inquire_predefined_or_custom_pizza
22  • MERGED prompt_for_pizza_size_and_toppings with
      prompt_for_pizza_details into
      prompt_for_pizza_size_and_toppings
23  • ADDED_NEW confirm_toppings_from_list
```

Listing 3.4: Functionality branch example

The purpose of this queue is to explore it in a Depth First Search (DFS) way, so if we find a functionality, we try to look for branches of it. This approach was chosen instead of Breadth First Search (BFS) since with BFS we cannot know when we have found all the functionalities of a given depth, while with this DFS approach we could explore a functionality until we didn't find any variation or branch of it.

### 3.2.3. Functionality Extraction

At the end of each conversation, the Explorer Agent looks at the conversation history and tries to find functionalities exhibited by the chatbot. These functionalities are represented as Functionality Nodes. As depicted in Figure 3.3, a Functionality Node contains the following fields:

- **Name:** the name of the functionality (e.g., `prompt_for_pizza_size`)
- **Description:** what the functionality does (e.g., asks the user for the size of the pizza)
- **Parameters:** fields that the user should input. A parameter always has a name and a description and optionally can have options. This parameter is optional, since there are functionalities that don't necessarily need inputs. An example of a parameter for the pizza could be this:
  - **Name:** pizza size.
  - **Description:** size of the pizza the user wants.
  - **Options:** small, medium, large.

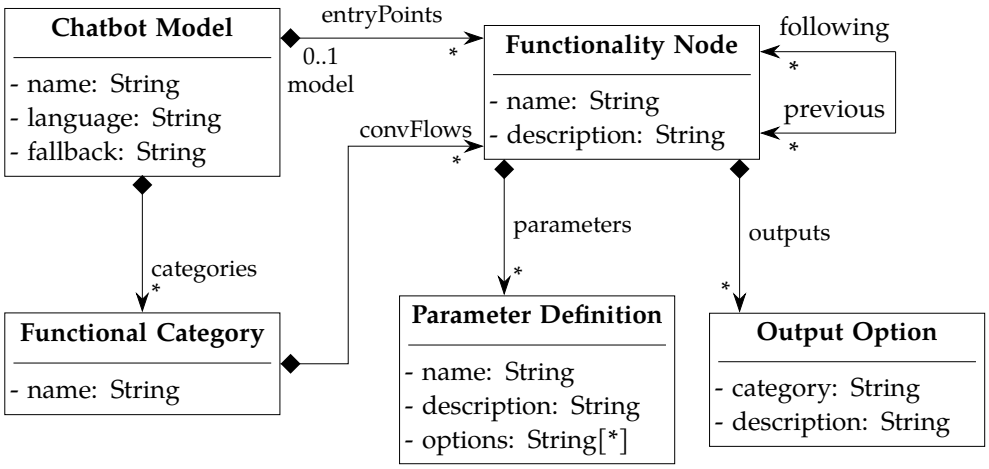



Figure 3.3.: Chatbot model schema.

- **Output:** as the parameters, outputs are optional. It represents pieces of data that we expect the chatbot to output. For example when ordering the pizza it could be the price or the order id.
- **Followers and Previous:** Since the nodes are arranged as a DAG, the nodes have children and parents that mimic the chatbot's workflow. The idea of this workflow graph, is to order functionalities in the order that one will encounter them, for example, the chatbot will always ask for drinks after asking for pizzas. Therefore, the drink functionality should be a child of the pizza functionality.

On top of this, the functionalities are clustered into categories. This is mainly to ease the visual representation for the user when there are many functionality nodes.

### 3.2.4. Functionality Consolidation

As the functionality extraction usually results in the creation of multiple functionality nodes, the agent performs a consolidation stage where similar functionalities are merged into a more complete one. This is achieved in two actions:

1. **Session-Local Merge:** first, the functionality nodes extracted during this session are compared to one another and with the help of the LLM semantically similar nodes are merged into a newer, more complete one. With this we achieve that the extracted nodes of this session are more relevant.

2. **Global Merge:** after the nodes discovered in this session have been merged, the resulting set is compared with the ones discovered in previous sessions and again, the LLM looks for semantically similar functionalities and merges them into one.

To better understand this process, we will give an example, which is also visualized in Figure 3.4. Imagine that throughout the last conversation, we extract a functionality that is called "prompt for custom pizza ingredients", with a description that is "Asks the user to provide the ingredients that he wants on the custom pizza" but has no parameters or outputs. Then, in the current session, the explorer agent's goal is to find variations or branches of this functionality since it is the first in the queue, and the agent extracts a new functionality called "prompt ingredients for custom pizza" with a similar description, but this time with a list of parameters like "pepperoni, ham, tuna, olives", then, the global merge step would merge these two into a unified version with the parameters. This was a simple example, but more complex ones occur where not only the parameters are added, but having different lists of parameters they are combined into a more extensive ones, or the descriptions are combined to more accurately define what the functionality does.

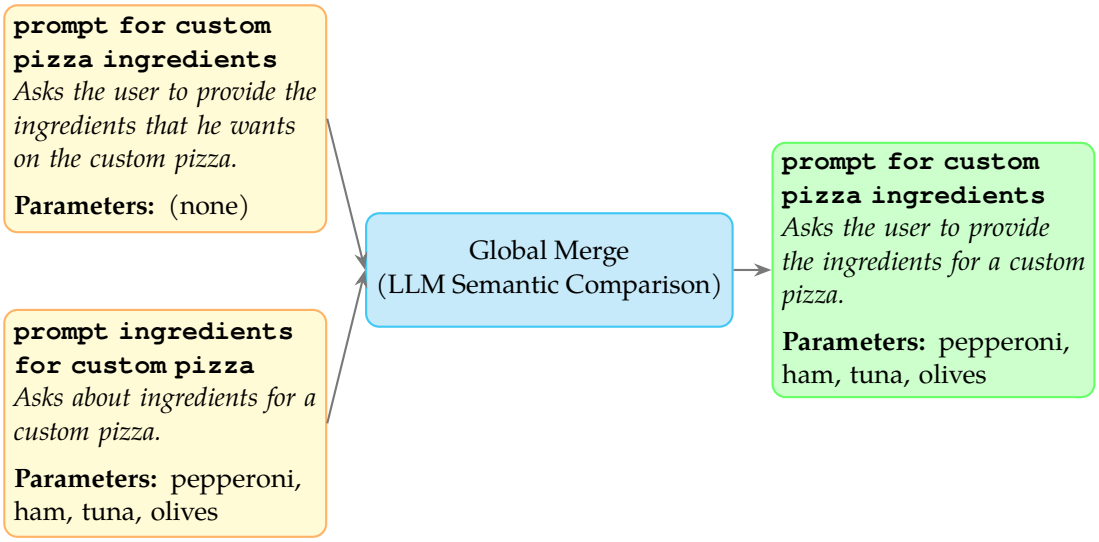



Figure 3.4.: Visual representation of the Functionality Consolidation process.

## 3.3. Refinement Phase

Once all the conversation sessions are over, and the functionalities have been extracted, we enter the refinement phase. The goal of this phase is to take the raw, potentially messy, functionalities discovered during the exploration phase and to create a coherent model.

### 3.3.1. Global Consolidation

While during the exploration phase we have a consolidation phase (see Subsection 3.2.4), variations of the same functionalities may still appear. For this we have a more in-depth consolidation phase. The step in the explorer phase groups all the nodes and sends them to the LLM, meanwhile, in this step every possible pair of functionality nodes is sent to the LLM one pair at a time. Although it takes considerably longer than the previous approach, checking in pairs helps the LLM to focus on a single merge instead of having to find merge cases between for example 20 nodes. Also, although it takes longer and there are more LLM calls it doesn't become expensive since it is just a short prompt with a pair of functionalities and the LLM response is also a short response containing if they are merged or not.

### 3.3.2. Chatbot Classification

Next, the chatbot is classified into informational or transactional. To do so, the list of functionality nodes along with conversation snippets and a prompt that describes the two types of chatbots is sent to the LLM which will respond with "transactional" or "informational" based on what it decides.

- **Informational:** these conversational agents are primarily informational, designed to respond to user inquiries and deliver relevant data. Common applications include university and banking chatbots, which are limited to providing information and redirecting users to appropriate resources, such as forms or specific web pages, rather than directly facilitating transactions.
- **Transactional:** these are chatbots that do guide the user through a task or workflow, often following a form and making a transaction. For example, the pizzeria example we have been using or a hotel chatbot that helps users book a room.

### 3.3.3. Workflow Structure Inference

The final step is to define the DAG that models the chatbot's workflow. During the exploration phase, parent-child relationships are set when a functionality is discovered as a branch or variation from another one, but most of the nodes are still designated as root nodes. So, we manage to structure this by asking the LLM identify likely sequences, branches, and joins based on conversational evidence and the dependencies between functionalities. The prompt for the LLM depends on the chatbot classification:

- **Informational chatbots:** usually informational chatbots' functionalities don't have parent-child relationships, this is because they simply serve information that is

not nested through steps. This is why this prompt is conservative in identifying relations, since usually all the questions are entry points that can be asked directly. So, relationships are only established if there is strong evidence that a sequence of actions is needed to access the functionality. The prompt for informational chatbots can be seen in Listing A.2 .

- **Transactional chatbots:** these other chatbots are more likely to have sequential workflows where one functionality will only exhibit if a previous action has been done. Also, the prompt will look for branches of optional choices, for example, ordering a custom pizza or a predefined pizza. The prompt for transactional chatbots can be seen in Listing A.1.

During this workflow inferrence step, as can be seen in the prompt, the LLM also groups the functionality nodes into categories and is instructed to reuse them to avoid creating a different category for each functionality.

Figure 3.5 shows the workflow graph resulting from a TRACER execution against a pizzeria chatbot built with Taskyto taken from [8]. The entry point, represented as a black dot, is the starting point of a new conversation. From there, you can go to four different root functionalities grouped into two categories. From one of the functionalities, you can continue the workflow to make your order. We will now break down each category and the functionalities that it contains.

- **Chatbot Meta:** the category contains functionalities related to the chatbot talking about itself. It has two functionality nodes, provide welcome message, that as its name says, the chatbot will greet the user and state available information, in which the chatbot will give information such as the opening hours or its capabilities.
- **Order Placement:** contains two root functionalities, list available pizza types, which gives you the flavours of the different pizzas; and prompt for pizza details, which expects the pizza size and type, once this has been completed it will continue to ask the user for a confirmation and then prompt the user for the type and number of drinks they wish.
- **Order Confirmation:** this last category, once the user has gone through all the order placement steps, we have the last functionality of the workflow which is provide order total, here the chatbot will give the price of the total order and finalize the workflow.

This final model can be used for different purposes, such as reverse engineering, reengineering, migrating to a different framework or maintaining the chatbot. The next section shows how TRACER uses this model to generate user profiles for SENSEI.

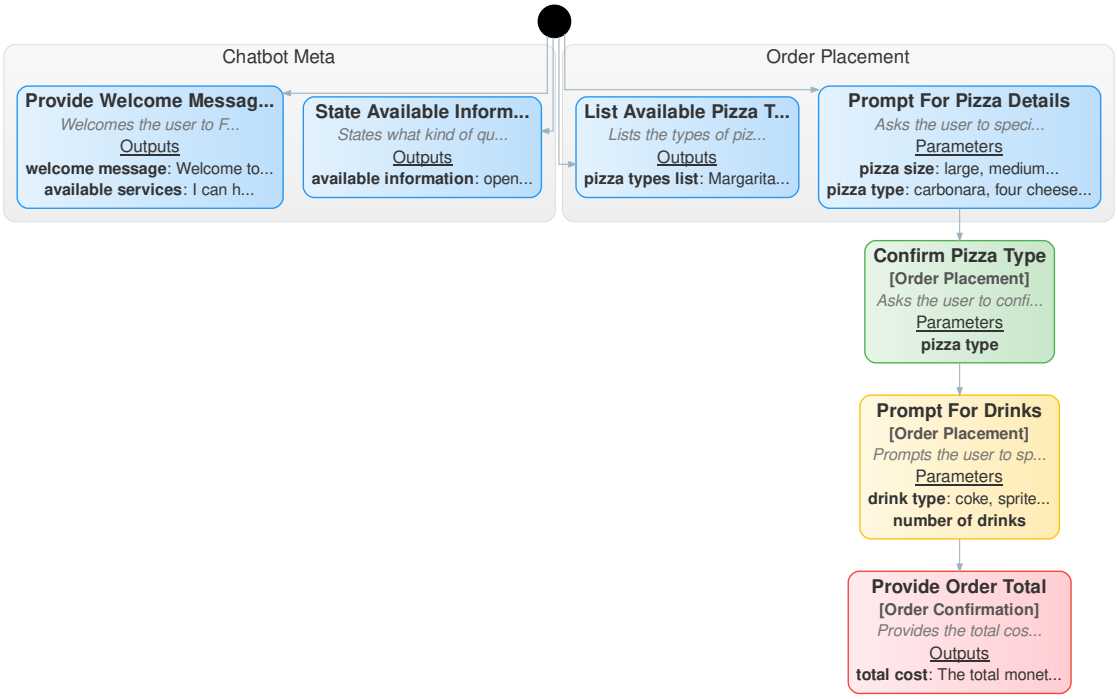



Figure 3.5.: Workflow model inferred by TRACER from a pizzeria chatbot.

# 4. User Profile Structure and Generation

Once the model is complete, we use it to automatically generate user profiles for SENSEI. These serve as test cases designed to verify the discovered functionalities of the chatbot, its handling of different inputs, to check if the outputs match the expected values, and to identify other errors such as timeouts.

First, Section 4.1 will cover the structure of these profiles and how they work, then Section 4.2 will detail how these profiles are synthesised from the inferred model.

## 4.1. User Profiles Structure

A user profile contains all the information that characterises the user, the conversation goals, interaction style, and other information such as the LLM that will be used, or the number of conversations and turns per conversation. These profiles are structured in a YAML file, Listing 4.1 and Listing 4.2 show an example of the user profiles generated during a TRACER execution against a pizzeria chatbot made with Taskyto.

```yaml
test_name: Pizza Orderer and Customizer
llm:
  temperature: 0.5
  model: gpt-4.1-mini
  format:
    type: text
user:
  language: English
  role: A customer who wants to browse pizza options, understand how to customize a
      pizza, select toppings and size, and complete their pizza order
  context:
  - You are planning to order a pizza and want to explore the available options before
      making your selection.
  - You prefer to create a custom pizza that fits your taste by choosing specific
      toppings and size.
  - You want to understand the customization process clearly to ensure your order
      meets your preferences.
  goals:
  - Ask the chatbot what types of pizzas are available to order.
  - Request an explanation of the process and options for customizing a pizza.
  - Ask for a detailed list of available custom pizza toppings categorized by
      vegetables, meats, cheeses, and sauces.
  - Order a custom pizza specifying the toppings as {{chosen_toppings}} but omit the
      size initially.
  - When prompted, provide the pizza size as {{pizza_size}} to complete the
      customization.
```

```yaml
  - chosen_toppings:
      function: forward()
      type: string
      data:
      - bacon
      - cheese
      - chicken
      - corn
      - ham
      - mushrooms
      - olives
      - pepper
      - pepperoni
      - pineapple
  - pizza_size:
      function: forward()
      type: string
      data:
      - large
      - medium
      - small
      - extra large
chatbot:
  is_starter: false
  fallback: I'm sorry, I did not get what you said. I can help you ordering predefined
      or custom pizzas, and then drinks.
  output:
  - pizza_type_list:
      type: string
      description: Available pizza types to order
  - customization_instructions:
      type: string
      description: Explanation of pizza customization process
  - vegetable_toppings:
      type: string
      description: List of available vegetable toppings
  - meat_toppings:
      type: string
      description: List of available meat toppings
  - cheese_toppings:
      type: string
      description: List of available cheese toppings
  - sauce_options:
      type: string
      description: List of available pizza sauces
  - chosen_toppings:
      type: string
      description: Selected toppings for custom pizza
  - pizza_size:
      type: string
      description: Size of the pizza selected
  - size_prompt_status:
      type: string
      description: Status of pizza size prompt after toppings input
  - order_completion_status:
      type: string
      description: Confirmation status of pizza order completion
conversation:
```

```
  number: 10
  max_cost: 1.5
  goal_style:
    steps: 30
  interaction_style:
  - single question
```

Listing 4.1: Example of the first user profile generated for a pizzeria chatbot.

```
test_name: Drink Orderer
llm:
  temperature: 0.4
  model: gpt-4.1-mini
  format:
    type: text
user:
  language: English
  role: A customer who wants to view available drink options, select quantity and type
      of drinks, and confirm their drink order
  context:
  - 'personality: personalities/conversational-user'
  - You are looking to purchase beverages for an upcoming event and want to review all
      the available drink options before deciding.
  - You plan to order a specific quantity and type of drinks based on the selections
      that fit your needs and preferences.
  - Before finalizing, you want to confirm your order details to ensure accuracy and
      avoid any mistakes.
  goals:
  - Ask the chatbot to provide the available drink options along with their prices.
  - State how many drinks I want to order and specify the type of drink from the
      provided options, using {{drink_quantity}} and {{drink_type}}.
  - Confirm my drink order by repeating the number and type of drinks I want to
      purchase.
  - drink_quantity:
      function: forward()
      type: int
      data:
        min: 1
        max: 5
        step: 1
  - drink_type:
      function: forward()
      type: string
      data:
      - Coke
      - Sprite
      - Water
      - Pepsi
chatbot:
  is_starter: false
  fallback: I'm sorry, I did not get what you said. I can help you ordering predefined
      or custom pizzas, and then drinks.
  output:
  - drinks_menu:
      type: string
      description: List of available drinks with prices
  - drink_option_name:
      type: string
```

```
43      description: Name of a single drink option
44    - drink_option_price:
45      type: money
46      description: Price of a single drink option
47    - drink_quantity_requested:
48      type: int
49      description: Number of drinks user wants to order
50    - drink_type_selected:
51      type: string
52      description: Type of drink user selects
53    - drink_order_confirmation_number:
54      type: string
55      description: Unique identifier for the confirmed order
56    - drink_order_summary:
57      type: string
58      description: Summary of number and type of drinks ordered
59  conversation:
60    number: 5
61    max_cost: 0.75
62    goal_style:
63      steps: 20
64    interaction_style:
65    - single question
```

Listing 4.2: Example of the second user profile generated for a pizzeria chatbot.

To better understand how the profiles are generated, we must first understand the profiles and their structure. Each profile consists of the `test_name`, a unique identifier for the profile, followed by four configuration blocks: `llm`, `user`, `chatbot` and `conversation`.

### 4.1.1. LLM Configuration (`llm`)

This section specifies the Large Language Model to be used and its temperature. The chosen `model` impacts the results, as each model has its own strengths and limitations, and some models will perform better than others, usually at a higher cost. This field (line 3 in the listings) takes the name of the model (e.g., `gpt-4o-mini`). Next is the `temperature` (line 4), a parameter that controls the randomness of the LLM; that is, how randomly the model selects the next word. A value of 0 makes the model's output deterministic, while a value of 1 encourages more creative responses. Lastly, lines 5 and 6 specify the format type, which can be set to audio or text.

### 4.1.2. User Persona Definition (`user`)

A key concept in these profiles is the persona, a detailed, fictional character that represents a primary user type [63]. Personas encapsulate the goals, motivations, and behaviours of real users.

In the `user` section of a test profile (starting in line 7 in both listings), we define such a persona for the simulated user through a combination of several fields. We start with the `language` that the user will employ, followed by the `role` of the simulated user, that is, a sentence representing the user's identity and objective (e.g., A customer ordering a pizza). The `context` field provides additional information to the chatbot, it can add more details or personality traits, such as, "You are in a hurry", to make the simulated user more realistic. The `goals` field is arguably the most important of the user's fields, as it describes all the objectives for the conversation. These objectives are written like templates with placeholders (e.g., Specify the pizza size as pizza_size). Below the list with all the objectives is the list of parameters, which is the placeholders defined in the goals. Each of these parameters is assigned a function:

- `random()`: A random value from the data below is selected.
- `random(x)`: Takes $x$ different random values.
- `forward()`: Selects one value sequentially from the data list (see Listing 4.1 lines 21 and 35 and Listing 4.2 lines 20 and 27).
- `forward(x)`: Takes the $x$ next values sequentially.
- `forward(var)`: Allows to cycle through all the combinations of a pair (or more, depending on the nested combinations) of variables. For example, one could nest pizza sizes with pizza types, that means that all combinations of pizza sizes with pizza types would be tested, that is, small pepperoni, small four cheese, ..., medium pepperoni, medium four cheese, ..., large pepperoni, large four cheese...

After the function, we can specify its `type`, with values like int, float, string or date. Next, there is the data field, where a list of values is provided or if it is a numeric value, a range with min and max and the step to go from the min to the max. For example, on both inputs in Listing 4.1 in lines 20 and 34 we have inputs with strings that right below define a list of values that the forward function will take. In Listing 4.2 in line 19 we have an input variable with an integer, note how now instead of a list with all the values we have the min, max and step.

### 4.1.3. Chatbot Settings (`chatbot`)

This section specifies the expected behaviour of the system under test. First we have the `is_starter`, which defines whether the chatbot initiates the conversation. The second field is the `fallback` (Listing 4.1 line 44 and Listing 4.2 line 36), this is the default sentence given by the chatbot when it cannot understand the user's statement, or the user's input is outside of the chatbot's scope.

Next is the `outputs`, a field similar to the goals but in this case instead of being related to the inputs, it will be used to extract outputs from the chatbot. For each output a name is provided then a `type` which can be string, money, int, float or date; then we have a short `description`. This information will be used by an LLM to extract the information from the conversation with the chatbot.

For example, in Listing 4.1 we have on line 46 `pizza_type_list` which will extract from the conversation the list of available pizzas provided by the chatbot, since it is a list of pizzas it is defined as string. In the Listing 4.2 we have `drink_option_price` in line 44, which will extract the price a single drink, note how it uses the type money, and, lastly, in the same listing in line 47 we have `drink_quantity_requested` which will extract the number of drinks and for this, the type is set to int.

### 4.1.4. Conversation Control (`conversation`)

This last section controls aspects of the execution. The `number` will control the number of times the conversation will take place, the more conversations, the more combinations of the goals' items to be tested. This field allows an integer, that simply indicates the number of conversations; `all_combinations` that will exhaustively test every combination, although it ensures good coverage of the inputs, it can also result in an enormous number of test cases, especially if we use nested forwards; `sample(x)`, where $x$ is a number between 0 and 1, will compute all the possible combinations and take a percentage of all of these.

The second field, `goal_style`, is the test's stop condition. It can be the number of steps taken in the conversation (let a step be a user message followed by a chatbot's one), or `all_answered` which will stop once the user has completed all of its goals, this parameter is also accompanied by a `limit` field which sets a hard limit ensuring that the conversation finishes even if the chatbot is not able to fulfil the user's goals.

Lastly, we have the `interaction_style`, which lets us set a list of styles that the simulated user will adopt for its conversations. These styles are predefined and include some like *make spelling mistakes*, *use short phrases* or *single question*.

See from line 76 onwards in Listing 4.1 and from line 59 onwards in Listing 4.2 to see a real example.

## 4.2. User Profile Generation

The profile generation is an automated process that aims to convert the inferred model into a set of profiles that will thoroughly test the chatbot. The process is divided into seven steps that can be visualised in Figure 4.1. It begins by grouping functionalities into profiles, followed by the LLM-driven generation of goals, variables, context, and

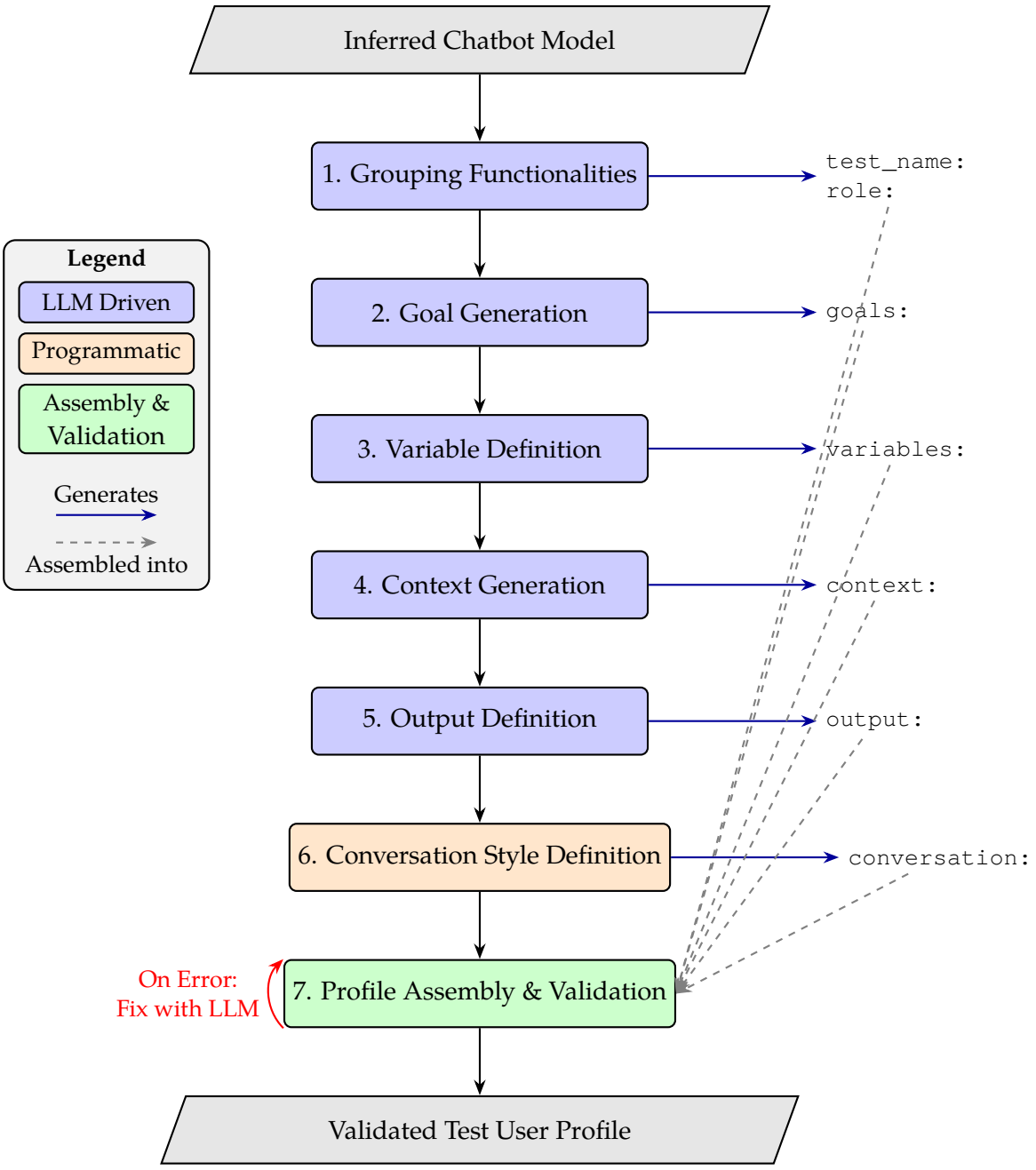



Figure 4.1.: Process to generate profiles from the model to the final user profiles.

outputs. Then, the definition of the conversation style and, finally, the profile assembly and validation step. Each of these steps is detailed in the following subsections.

### 4.2.1. Grouping Functionalities into Profiles

The inferred model is sent to an LLM along with conversation fragments, with this information the LLM is prompted to group the functionalities into realistic and logical user scenarios. Each group will gather functionalities that are semantically related, frequently used together or performed sequentially. Ideally, each functionality is only assigned to one group to ensure that all functionalities are used while avoiding redundancy; this is not always achieved since in cases where a functionality branches into others it is required to go through it at least twice. From these functionality groups, the LLM will generate the `test_name` along with its `role`.

### 4.2.2. Goal Generation

Once the functionalities are grouped, the group of functionalities is sent to a new LLM call that will create a set of functionalities that when fulfilled will ensure that the functionalities have been activated. The goals may or may not have variables, it will depend on the nature of the goal, for example, a query about opening hours will not require variables but if the goal is about asking for a pizza, then the goal will likely have variables.

### 4.2.3. Variable Generation

This step is only needed when variable `{{placeholders}}` have been defined. To understand how the variables are generated first we need to define what a data source is. A data source comprises the values of either a parameter or an output from a functionality node, we decided to use both outputs and parameters since there are occasions where the extracted functionality is for example ”list available pizza types” and the values that we can use for the variables are in the outputs instead of the more obvious parameters. So the data sources are the combinations of all the values from all outputs and parameters.

With this defined, the procedure is as follows for each individual variable. First, we pass all these data sources to the LLM and prompt it to match the variable to one of the data sources emphasizing that it is not necessary to force a match unnecessarily. Then, if one match is given, we request the LLM to generate another extra value based on the content of the returned data source to test the chatbot on values outside of the ones that it suggests. For example, with the pizza sizes small, medium, large, it tends to generate extra large as the extra value, or instance, `pineapple` was generated as a topping in this case (see Listing 4.1)

In the case that no data source is matched, either because the chatbot didn't provide values (for example with dates or numbers) or because the data source matching didn't correctly match the variable, the LLM will generate the data for the variable given the goals, functionalities, role and conversations. It is important to note that variables are not only restricted to be strings, in the Listing 4.2 the drink quantity is a great example of variables generated with numeric values instead of being strings or matched data sources.

The generated variables always use the `forward()` function since it allows us to test every option. We chose this over the nested approach since it has a great balance between completeness and number of conversations. For example, having 8 pizza types with 3 different pizza sizes with nested forwards would require $8 \cdot 3 = 24$ conversations, while using the regular forward would require only 8 conversations to go through all the options.

### 4.2.4. Context Generation

A simple context of two or three sentences is generated by the LLM based on the functionalities, the previously generated content and the conversations. The idea of the generated context is to create a more realistic scenario where the user is not simply testing the chatbot but in a realistic case where for example the user is at a get-together and is looking to order pizzas, or is an Erasmus student requesting help to a university chatbot.

### 4.2.5. Output Definition

The output definition along with the goal and variables is one of the key steps. This allows the chatbot to be tested and see if the chatbot is returning the information that it should and thus completing its functionalities. The outputs are generated by the LLM taking into account the functionality and the goals, seeking to identify elements that confirm their achievement, like for example the order ID or the order price; and by also looking at the outputs from the functionality nodes. The Listing 4.2 contains great examples of outputs of different types such as money, int or string.

These outputs are as granular as possible to allow for a better testing of the chatbot, for example, instead of a more vague output like "order confirmation", it would be split into two granular outputs "order ID" and "total order price".

### 4.2.6. Conversation Style Definition

The `conversation` section does not require LLM calls. First, the number of conversations is chosen based on the variable with the largest data set, for instance, if we have

four variables, and pizza type, with 8 options, is the one with the larger set of options, the number of conversations will be 8 to ensure that all the variables are tested.

For the length of the conversation, a limit is set on a linear combination of the number of goals and outputs, so that the greater the number of goals and outputs, the longer the conversation will be, but still with a hard limit of 30.

Lastly, the interaction style always is set to `single_question` since otherwise the user simulator tries to fulfil all the goals at the same time and results in awkwardly complex sentences.

### 4.2.7. Profile Assembly and Validation

Once all the fields have been generated the profile is assembled into a YAML file. This profile is then passed through a validation script that will check that all the required fields are complete, that every placeholder has a variable defined, and that each variable is correctly defined amongst other checks, and if any error appears it will return a verbose description of the issue. Then, the description of these errors along with the YAML file are sent to the LLM with a prompt to fix the issue.

These seven steps allow us to turn the inferred model from the deployed chatbot under test, into a set of realistic and comprehensive user profiles that will act as test cases when combined with the user simulator SENSEI. This profile generation stage bridges the gap between black-box model inference and automated testing.

# 5. Tool Support

To ensure that the previous methodology can be reproduced we have implemented Task Recognition And Chatbot ExploreR (TRACER) in an open-source Python package to allow users to execute it from a Command Line Interface (CLI). In addition, we have developed a web application that allows users to execute both TRACER and SENSEI without requiring knowledge of how to operate the command line.

## 5.1. Implementation and Architecture

### 5.1.1. Core Framework: LangGraph

As explained during the Section 4.2 (User Profile Generation) TRACER relies on LLMs, this is why we used LangGraph [6] as our framework for development. We chose LangGraph because it allows to manage and orchestrate complex agentic workflows with states, it is also an industry standard with extensive documentation. LangGraph allows us to orchestrate the different stages and to keep complex states where we store the inferred model and the fields generated for the profiles. Currently, TRACER allows OpenAI and Gemini models, but thanks to LangGraph it would be straightforward to add other LLM providers.

### 5.1.2. Modular Architecture

TRACER can infer a chatbot model as long as the chatbot is accessible through an interface, typically a REST API. Currently, it provides access to communicate with chatbots made with different technologies, such as Taskyto [8], Rasa [5] or 1MillionBot [64]. In addition, new connectors could be added by extending the current implementation.

Apart from these connectors, TRACER is divided into three modules, each corresponding to a phase of the methodology:

- **Explorer Module:** Contains the Explorer Agent and implements the logic for the Exploration Phase (see Section 3.2), managing the conversational sessions and the initial extraction of Functionality Nodes.

- **Refinement Module:** Implements the logic for the Refinement Phase (see Section 3.3), responsible for consolidating functionalities, classifying the chatbot, and inferring the final workflow structure.
- **Profile Generation Module:** Implements the seven-step synthesis process (see Section 4.2), taking the final chatbot model and generating the YAML user profiles.

### 5.1.3. Generated Artefacts

Upon completion, TRACER generates the following artefacts containing the results from the full analysis performed on the target chatbot:

- A set of user profiles representing realistic users that would use the application and that will act as test cases (see Listing 4.1 and Listing 4.2)
- A markdown report containing the inferred model information such as the discovered functionalities, fallback message, language and other information such as token usage, number of LLM calls or estimated cost.
- A graph representing the inferred model's workflow (see Figure 3.5)
- A JSON file containing the same workflow but in a textual format.

## 5.2. Distribution and Development Workflow

Before detailing the CLI's functioning, this section will describe TRACER's packaging, distribution, and the software engineering practices used to maintain its quality. TRACER is packaged and distributed as a package on the Python Package Index (PyPI) repository (`https://pypi.org/project/chatbot-tracer/`), facilitating its installation by running `pip install chatbot-tracer`. This approach not only simplifies its use, but also its implementation into other projects such as the web application that was developed, or other projects that could use TRACER since it can be added as another package requirement.

To ensure code quality and automate the release process TRACER makes use of GitHub Actions for the Continuous Integration / Continuous Deployment (CI/CD) pipeline. For the Continuous Integration (CI) we used Ruff [65]. Ruff is Python linter and formatter written in Rust that combines tools like Flake8 or Black into a single and faster tool. We used of Ruff not only to enforce a consistent code style, but also to enforce code quality standards, such as ensuring proper documentation and managing code complexity by setting thresholds for metrics like McCabe's cyclomatic complexity. For the Continuous

Deployment (CD) side, we implemented a pipeline that whenever a tag with the format `v*.*.*` is published automatically builds the package, publishes it to PyPI, and creates the corresponding GitHub release. All the TRACER source code can be accessed in `https://github.com/Chatbot-TRACER/TRACER`.

## 5.3. Chatbot Connectors

A key component of TRACER is its connector system, developed as part of this work and packaged as the `chatbot-connectors` library. The `chatbot-connectors` library is available on PyPI (`https://pypi.org/project/chatbot-connectors/`) and its source code can be accessed at `https://github.com/Chatbot-TRACER/chatbot-connectors`.

The purpose of this package is to provide a unified interface to interact with different chatbots, having the same method for sending messages but different implementations for each chatbot technology. For this we have an abstract base class from which every implementation inherits. This base class standardizes the core methods for sending messages and receiving responses.

### 5.3.1. Available Connector Technologies

TRACER currently supports four connector technologies, each designed to interface with specific chatbot platforms: Custom (a flexible YAML-configured connector for any chatbot API), MillionBot (specialized for the MillionBot platform), Rasa (interfacing with Rasa chatbots through Representational State Transfer (REST) webhooks), and Taskyto (dedicated connector for Taskyto chatbot servers).

The Custom connector deserves special attention due to its flexibility, providing a solution for integrating with any chatbot API without requiring custom code development. This connector uses YAML configuration files to define how to communicate with a chatbot's API, specifying the API endpoint, request structure, authentication requirements, and response parsing instructions.

### 5.3.2. Connector Discovery and Configuration

The CLI provides built-in commands to discover available connectors and their configuration requirements:

```
$ tracer --list-connectors
Available Chatbot Connector Technologies:
================================================

```

```
  - custom
    Description: Custom chatbot connector configured by a YAML file
    Use: --technology custom
    Parameters: --list-connector-params custom

  - millionbot
    Description: MillionBot chatbot connector
    Use: --technology millionbot
    Parameters: --list-connector-params millionbot

  - rasa
    Description: RASA chatbot connector using REST webhook
    Use: --technology rasa
    Parameters: --list-connector-params rasa

  - taskyto
    Description: Taskyto chatbot connector
    Use: --technology taskyto
    Parameters: --list-connector-params taskyto

Total: 4 connector(s) available
```

Listing 5.1: Listing available connectors.

For detailed parameter information for a specific connector, users can query the requirements:

```
$ tracer --list-connector-params taskyto
Parameters for 'taskyto ' chatbot:
-------------------------------------------------
  - Name: base_url
    Type: string
    Required: True
    Description: The base URL of the Taskyto server.

  - Name: port
    Type: integer
    Required: False
    Default: 5000
    Description: The port of the Taskyto server.

Example usage:
  JSON format: --connector-params '{"base_url": "http://localhost"}'
  Key=Value format: --connector-params "base_url=http://localhost"
```

Listing 5.2: Listing connector parameters for Taskyto.

### 5.3.3. Custom YAML Connector Configuration

The Custom connector deserves special attention due to its flexibility. It allows users to integrate with any chatbot API by creating a YAML configuration file that describes the API communication protocol.

A Custom connector configuration file contains the following key fields:

- `name`: A friendly name for the chatbot (optional)
- `base_url`: The base URL of the chatbot API (required)
- `send_message.path`: API endpoint path appended to base_url (required)
- `send_message.method`: HTTP method (POST, GET, PUT, DELETE; default: POST)
- `send_message.headers`: Custom headers including authentication (optional)
- `send_message.payload_template`: JSON structure with `{user_msg}` placeholder (required)
- `response_path`: Dot-separated path to extract the bot's reply from the JSON response (required)

#### Configuration Examples

A simple echo bot configuration demonstrates the basic structure:

```
name: "Echo Bot"
base_url: "https://postman-echo.com"
send_message:
  path: "/post"
  method: "POST"
  payload_template:
    message: "{user_msg}"
response_path: "json.message"
```

Listing 5.3: Simple Custom connector configuration.

A more complex configuration with authentication:

```yaml
name: "Secure Bot"
base_url: "https://api.mychatbot.com"
send_message:
  path: "/chat/send"
  method: "POST"
  headers:
    Authorization: "Bearer your-api-key"
    Content-Type: "application/json"
  payload_template:
    query: "{user_msg}"
    session_id: "user123"
response_path: "response.text"
```

Listing 5.4: Custom connector configuration with authentication.

The `response_path` field uses dot notation to navigate through JavaScript Object Notation (JSON) responses:

- `"message"` accesses `response["message"]`
- `"data.text"` accesses `response["data"]["text"]`
- `"results.0.content"` accesses `response["results"][0]["content"]`

## 5.4. The Command Line Interface

The primary method for using TRACER is through the CLI. The entire TRACER pipeline can be executed, from chatbot exploration to profile generation, through a single command. This approach enables users who prefer terminal-based workflows such as developers, to execute TRACER easily. It also enables the integration of TRACER within other projects.

TRACER is run by one main command: `tracer`. To see in more detail its options, users can run `tracer --help`. Some of the key arguments are as follows:

### 5.4.1. Conversation Control

`--sessions` or `-s` that controls the number of conversations that TRACER will have with the chatbot under testing and `--turns` or `-n` for the number of turns or steps per conversation.

### 5.4.2. Connector Configuration

The arguments `--technology` or `-t` allow users to select from the available connector technologies (custom, millionbot, rasa, taskyto). Additional connector-specific parameters can be provided using `--connector-params` or `-cp` in either JSON format or key=value pairs separated by commas. For the Custom connector, users specify the YAML configuration file path.

### 5.4.3. LLM Configuration

Using `--model` or `-m` allows the user to select the Large Language Model used for the exploration and analysis, then `--profile-model` or `-pm` is an optional argument that if set will make the generated user profiles' LLM be the one specified, otherwise the LLM that will appear in the profiles will be the same one used for TRACER, it is recommended to use a more capable model for exploration and analysis since will infer a more comprehensive model with more realistic profiles, and then a more economical model to run the profiles since there will be many more LLM calls and the chosen model will not have as significant an impact.

### 5.4.4. Output and Logging

Three verbose levels are also available: the basic one where only key information, such as the discovered functionalities, the current session or phase, and any warnings. The verbose level, activated with `-v`, which displays the conversations in addition to the previous information, and the debug `-vv` which displays the same information as the verbose level, in addition to details such as prompts sent to the LLM, its responses, and other logs that may have been generated during the development and debugging of the program. Lastly, the `--output` or `-o`, which controls where all the generated artefacts will be placed.

### 5.4.5. TRACER execution commands examples

```
$ tracer \
    --technology taskyto \
    --connector-params "base_url=http://localhost" \
    --sessions 12 \
    --turns 8 \
    --model gemini-2.5-flash \
    --profile-model gemini-2.0-flash \
    --output ./pizzeria_results \
    -v
```

Listing 5.5: TRACER command example with Taskyto connector.

The command in Listing 5.5 demonstrates a typical execution of TRACER against a Taskyto-based pizzeria chatbot. Line 2 specifies the connector technology as Taskyto, line 3 provides the connector parameters with the chatbot server's base URL, lines 4 and 5 configure the exploration to run 12 sessions of 8 turns each, line 6 sets the exploration and analysis model to the more advanced Gemini 2.5 Flash, line 7 defines the profile model as Gemini 2.0 Flash for cost optimization, line 8 specifies the output directory for all generated artefacts, and line 9 enables verbose mode to monitor conversations between the explorer agent and the chatbot under testing.

```
1 $ tracer \
2     --technology custom \
3     --connector-params "config_path=./my-bot-config.yml" \
4     --sessions 10 \
5     --turns 6 \
6     --model gpt-4o \
7     --output ./yaml_bot_results
```

Listing 5.6: TRACER command example with Custom YAML connector.

Listing 5.6 shows how to use TRACER with a custom chatbot through the YAML connector, demonstrating the flexibility of the connector system in accommodating various chatbot technologies. Line 2 selects the custom connector technology, line 3 specifies the path to the YAML configuration file, lines 4 and 5 set up 10 exploration sessions with 6 turns each, line 6 configures GPT-4o as the model for both exploration and profile generation, and line 7 defines the output directory for results.

## 5.5. The Web Application

To complement the CLI, we developed a web application to provide a user-friendly interface for running the whole end-to-end pipeline, that is, to run TRACER to generate the user profiles, and then to execute these with SENSEI. This, enables a broader audience to use both TRACER and SENSEI without the need of knowing how to use the CLI.

### 5.5.1. System Architecture

The web application is built on a modern, multi-tiered architecture designed for scalability, security, and asynchronous processing. Figure 5.1 provides an overview of this

architecture, illustrating the relationships between the five key layers: the Client, Presentation, Application, Task Processing, and Data layers. The following subsections will provide a detailed explanation of the technologies and design patterns used within each of these layers.

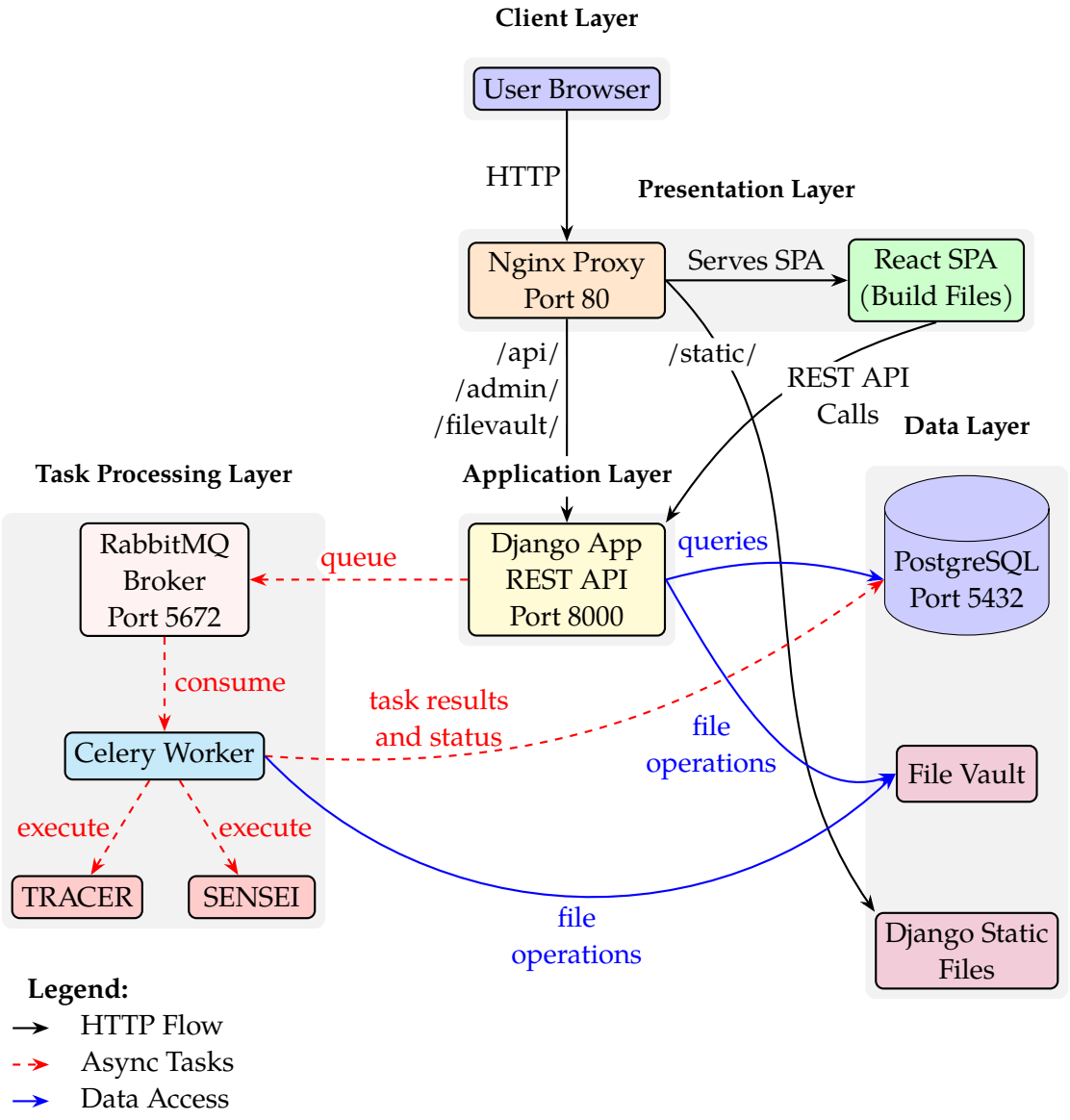



Figure 5.1.: Web Application Architecture and Connections.

### 5.5.2. Core Technology Stack

The backend of the application was developed in Django [66], This framework was chosen because it is Python-based and therefore compatible with TRACER and SENSEI, and also it offers the Django REST Framework [67] which enables the development of a REST API that will be consumed by our frontend.

For the frontend we chose React [68], a JavaScript library to develop Single-Page Applications (SPAs) which consumes data directly from our Django API.

Lastly, to ensure data persistence, we used PostgreSQL [69], we chose a Structured Query Language (SQL) database since Django's Object-Relational Mapping (ORM) supports this type of databases out of the box, also, we preferred PostgreSQL over the default Django's SQLite since the latter is more oriented to development and testing and stores everything in a single file which causes concurrency and performance issues when used in production.

### 5.5.3. Asynchronous Task Handling

TRACER and SENSEI executions both take from a few minutes up to hours, thus, executing this tasks synchronously would leave the user's interface unresponsive for the duration of the whole process. To handle this we implemented asynchronous executions, we did this by using a distributed task queue called Celery [70], which enables these jobs to be handled asynchronously. Then, to communicate Django with Celery we need a broker, for this we used RabbitMQ [71]. These two tools in conjunction will allow the user to execute TRACER or SENSEI and change to a different view, log out, or even turn off the computer and later return to check its progress. It will also prevent server overload since we can limit the number of concurrent jobs and if there are requests to execute more than this number, the jobs will be waiting on the queue instead of saturating the server.

### 5.5.4. The Nginx Reverse Proxy

A reverse proxy is essential for managing the communication between the user, the React frontend, and the Django backend. For this role, we used Nginx [72]. Its primary responsibility is to route incoming HTTP requests to the appropriate service based on the URL path. This also addresses the challenge of accessing each service in a different port, although they are running in a different port we can always access the same URL and Nginx will redirect the call.

In the Figure 5.2 we can see the configuration of the reverse proxy. It recieves all the calls from the users and routes them to the correct service. We have that all calls made to `/api/`, `/admin/` and `/filevault/` get redirected to Django's backend since as it

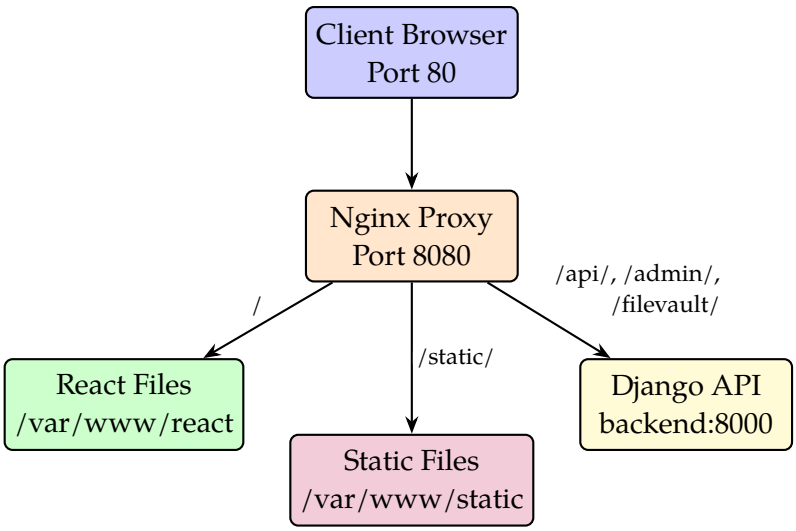



Figure 5.2.: Nginx routing to serve React, Django's static files and Django API.

processes API calls, return the different files and show the admin dashboard. The Django admin dashboard requires its CSS to be rendered correctly, which in turn necessitates serving the Django static files. For this reason, any request made to the `/static/` path is redirected to the location of these files. Lastly, the remaining calls will be redirected to React's build files.

### 5.5.5. Deployment and Containerisation

To facilitate the deployment of these services we containerized the whole web application using Docker [73] along Docker compose. This was done not only to simplify the deployment process, but to ensure that no matter the machine and the dependencies it had installed, it would work if it had Docker. We made two different Docker compose versions, one for development and one for production, each with their own Data Base (DB), message broker, and so on. So that production data was not affected during development. The complete production architecture is shown in Figure 5.3.

In the Figure 5.3 we show how the production containers and volumes are organized. Each square is a container running said service, then each cylinder is volume, where files are stored to ensure data persistance when containers are taken down. Each blue arrow means that the volume is mounted in that container, i.e., the container can access the volume's files, and the red ones show dependencies for the Docker compose, which means that until the container that the arrow is pointing at is not working, the container at the start of the arrow will not be initiated.

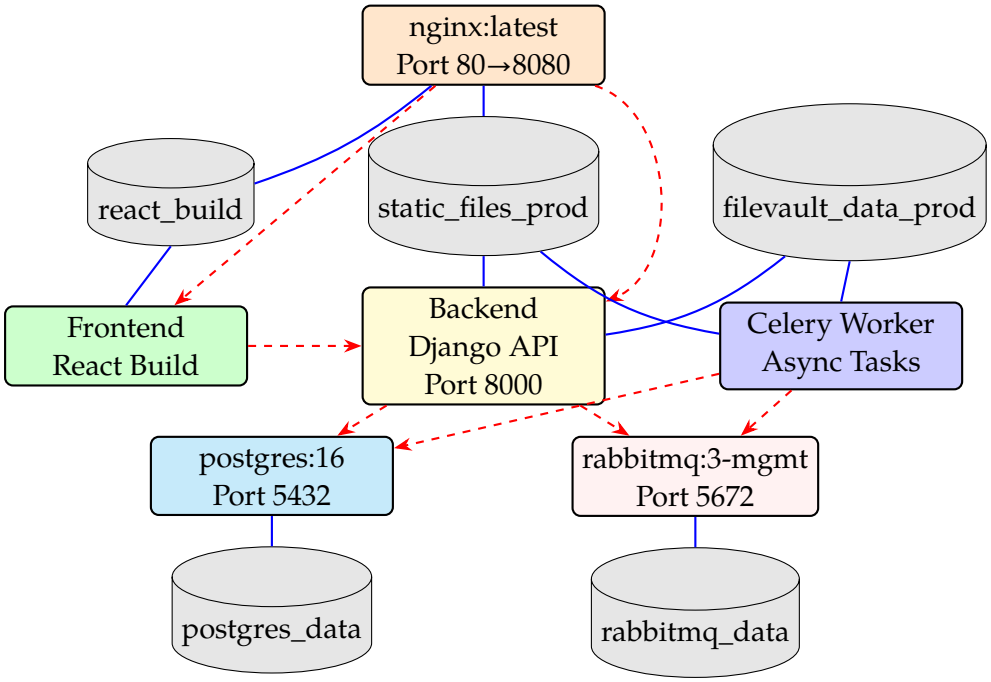



Figure 5.3.: Docker Container Architecture.

This ensures that until we do not have the DB and message broker set the queue and backend will not start, and only then the frontend will be built and lastly the Nginx reverse proxy. In the development version the setup was mostly the same but we had another volume that had the frontend and backend code so that we would have hot reload (i.e., changes in the code were shown live in the web).

### 5.5.6. Security Measures

To ensure the application's security we followed the Django Deployment Checklist [74]. Related to the authentication we implemented a new user model, and to handle the authentication we used Django-Rest-Knox [75] tokens. This is a library that implements an improved token system over the default in the Django REST Framework [67], the main improvements include:

- **One token per device:** instead of having one token per user which would mean that logging in from different devices would mean that they share the same token, causing that for example logging out on one device would terminate the session on all others. Knox solves this by creating one token per device.
- **Encrypted token in DB:** this is a major security flaw from Django REST Framework (DRF) and it is that tokens are stored as plain text in the DB, meaning that if the DB was compromised, the attacker could use those tokens to impersonate the legitimate user.
- **Token expire time:** in DRF, once an account is authenticated, the generated token remains valid indefinitely. Knox, in contrast, allows an expiry time to be set. After this period, the session is terminated as the token becomes invalid.

To store the API keys uploaded by the user to run TRACER or SENSEI we used Fernet Symmetric Encryption [76] to encrypt them before saving them to the DB. Although the DB has a user and password (not the default ones) we determined this was a prudent measure to add further layers of security, the same way that Knox encrypts, so that if the database was compromised, these credentials would remain encrypted.

These measures, combined with strict access controls, ensure the application adheres to robust security practices. Every request to the API requires token validation. Furthermore, models are configured to be readable only by their owner, unless explicitly set to public. Even when a project is public, write access remains restricted to the owner.

### 5.5.7. User Workflow and Features

This section aims to illustrate the web application, describing the typical end-to-end workflow a user follows within the web application. The web application is available at

http://miso.ii.uam.es:8081/.

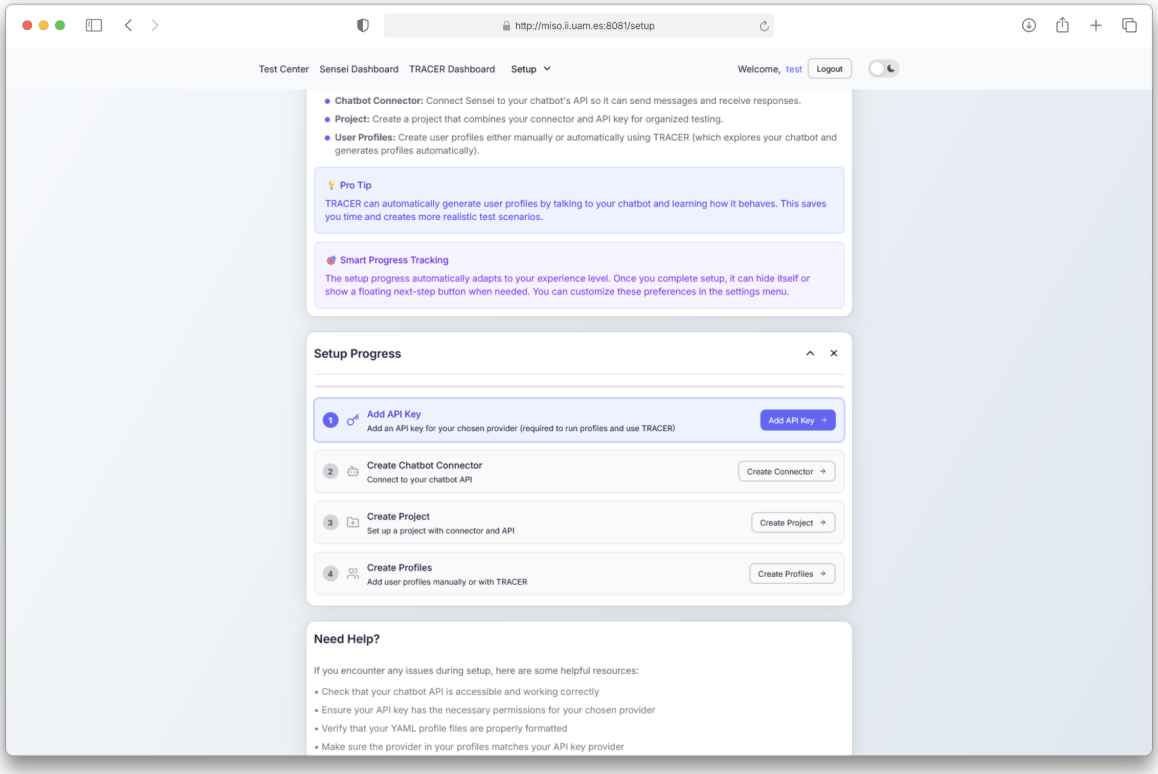



Figure 5.4.: Screenshot of the setup guide.

When users enter the web page they are presented with an authentication page that allows them to log in or register. Once they have logged in they are shown as shown in Figure 5.4 This guide assists the user through the process of configuring the environment before they are able to run TRACER or SENSEI. This guided tour is dynamic, meaning that as the user completes steps it automatically advances to the next step and displays a button to navigate the user directly there. The setup guide can also be closed, and if required, there is a section in the navbar called ”Setup Guide” from which the user can open it again or read it in detail.

The first step in the guided setup is to configure the necessary LLM credentials. As shown in Figure 5.5 this is accomplished from the profile view. A Bring Your Own

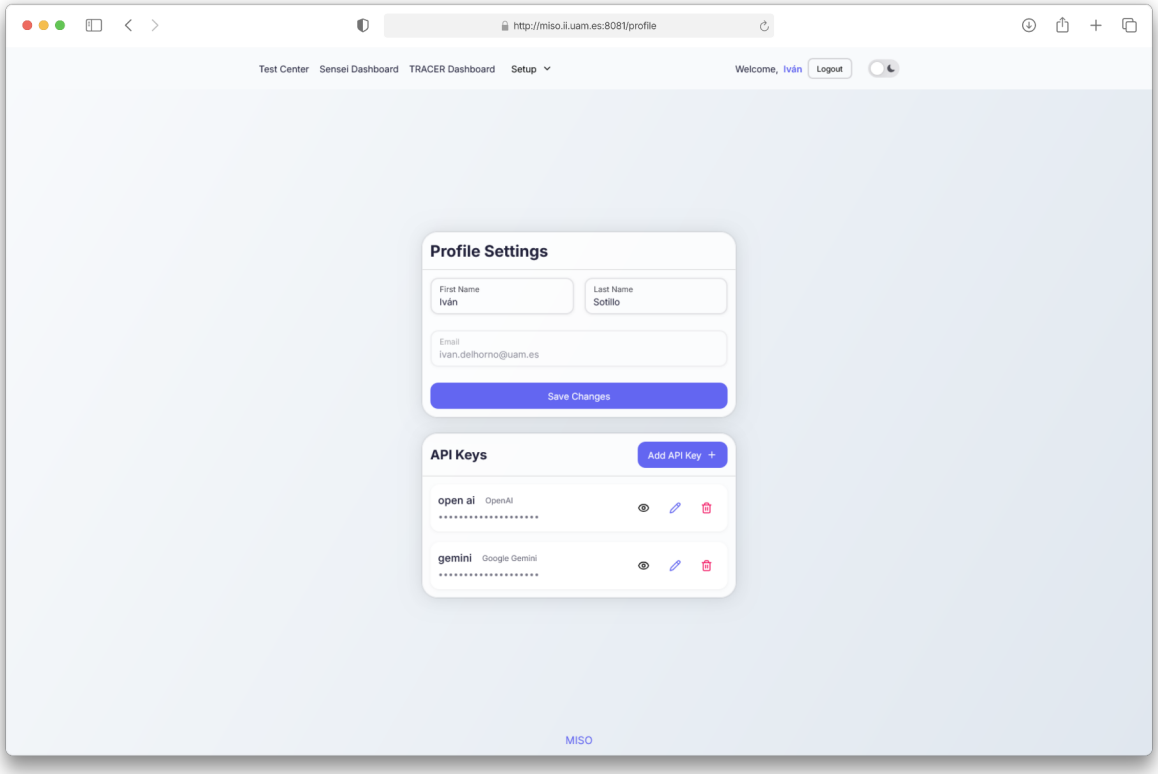
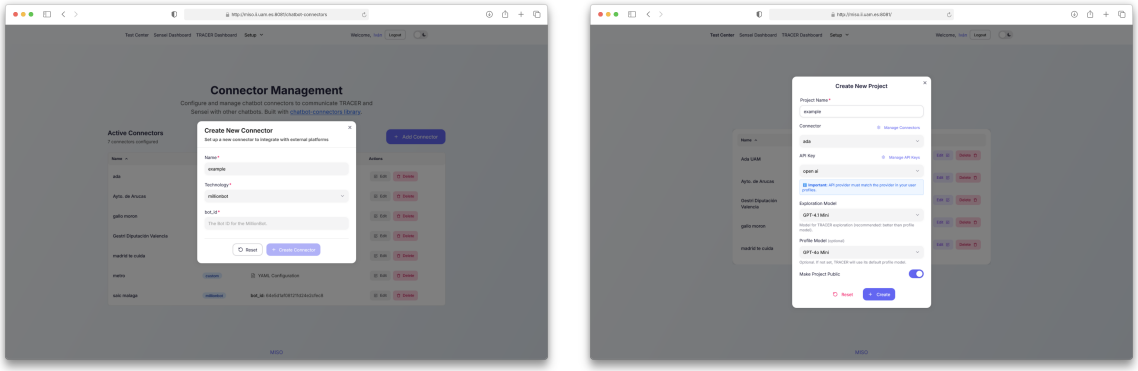



Figure 5.5.: Screenshot of the user profile view where users can set up their API key.

Key (BYOK) approach was chosen, that is, instead of charging per usage each user brings their own API key therefore, they only pay for what they use. The user can save OpenAI and Gemini API keys. These will be used both by TRACER and SENSEI.

The next step is to define the chatbot connectors. As explained in Section 5.3, these are reusable configurations that enable connection to chatbots via an API. There are two main methods for creating a connector:

- **Built-in connectors:** for technologies that are already configured in the `chatbot-connectors` library, such as Taskyto, Rasa or MillionBot, the interface presents a dynamic form that prompts the user for the specific parameters required for that platform (e.g., a `bot_id` for MillionBot, or `url` and `port` for Taskyto).
- **Custom YAML Connector:** for other chatbots that are not available in the list of




(a) Create new connector modal with the connectors dashboard behind it.

(b) Create new project modal with the project dashboard behind it.

Figure 5.6.: Screenshots of the create new connector and create new project modals.

options, but that offer a REST API, the user can select the custom option. This option opens a YAML editor where the users declaratively define the connection. To facilitate this process, a set of helpful features was added to the editor, such as documentation and examples.

Once the API keys and connectors are configured the user is ready to create a project. A project is the central workspace when working on a chatbot, it will contain, in addition to the already mentioned API key and connector, the TRACER and SENSEI executions. To create a project the user must complete a form, assigning it a name, selecting one of the connectors and API keys and also two LLMs, one for the exploration and another for the generated profiles, the list of LLMs is dynamically updated based on the provider of the API key. Lastly, the users select whether the project should be public. If a project is public, other users will be able to freely view the results in the SENSEI and TRACER dashboards. This menu is shown in Figure 5.6b.

After a project is created, the user can access the test center (shown in Figure 5.7), this is the main view to managing TRACER and SENSEI executions. Here the user can create and manage the user profiles, there are two ways to create them:

- **Automated Generation with TRACER:** the primary method for creating user profiles is employing TRACER. To do so, the users must click the Auto-Generate Profiles button and there choose the number of sessions and turns per session, and also the level of verbosity for the logs since the user will be able to review them. Once the button is clicked, the task is sent to the queue and will be executed as

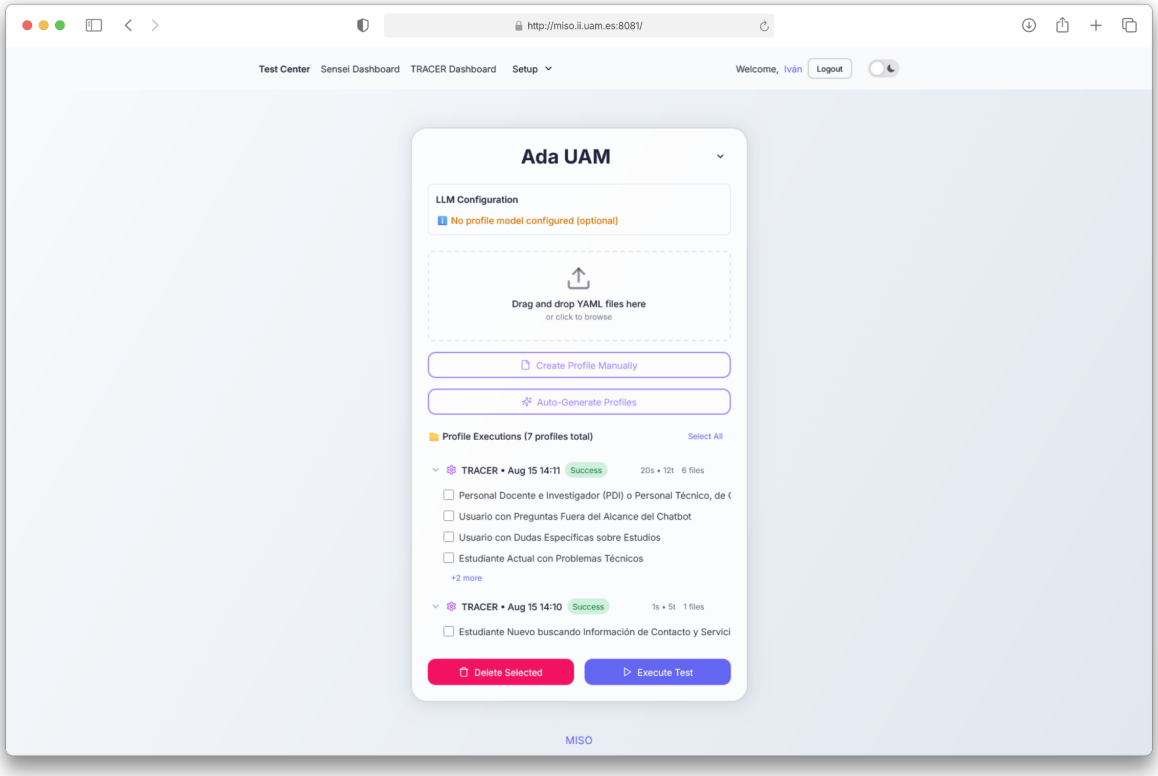


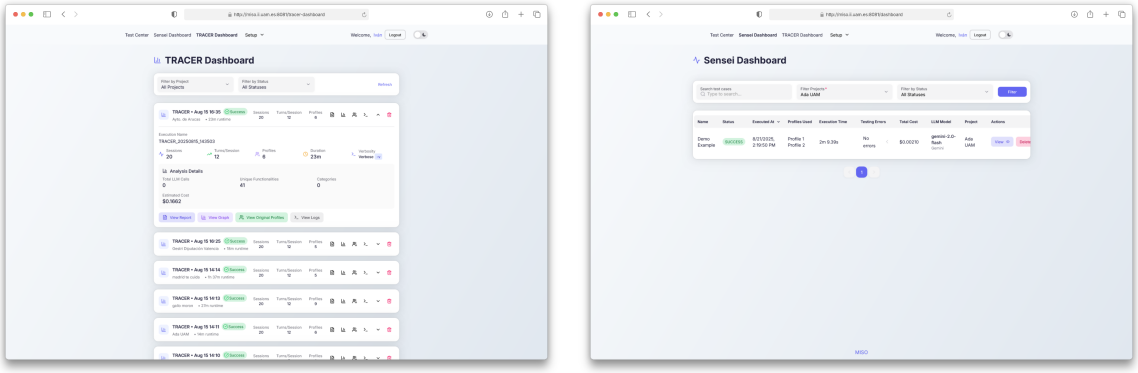

Figure 5.7.: Screenshot of the test center.

soon as possible.

- **Manual Creation:** users can also upload their profiles, or create them from scratch using an advanced YAML editor. This editor contains documentation, autocompletion and real-time validation to verify that the YAML syntax is correct and that the user profile matches the expected profile schema.

With these profiles created, the users can execute them with SENSEI by checking the boxes of the profiles they would like to execute and clicking the `Execute Test` button. No further configuration is required. Once the button is clicked, similar to TRACER, the task is sent to the queue.

Once TRACER and SENSEI have been executed the results can be explored in in their respective dashboards.




(a) TRACER dashboard. (b) SENSEI dashboard.

Figure 5.8.: Screenshots of the TRACER and SENSEI dashboards.

The TRACER dashboard, shown in Figure 5.8a, contains a list of the executions where the users can filter by project and status (running, success, failure, etc.). Each of these executions has a dropdown arrow to view more information, the users can also click to see the generated report containing all the functionalities plus other information in text format, view the graph where the users can observe the inferred workflow model, view the original user profiles generated (note that these are the originals since in the Test Center the users can modify the profiles to their liking), and to view the logs, to debug in case of failure, or to read the conversations.

The SENSEI dashboard is shown in Figure 5.8b. The SENSEI dashboard provides a similar interface where they can filter SENSEI executions and then click on one of them to see the results including the execution times, errors, cost, and other information.

# 6. Evaluation

This section aims to evaluate how TRACER performs at modelling and synthesising user profiles from a chatbot under testing. To achieve this, the following research questions are addressed:

- **RQ1: How effective is TRACER in modelling chatbot functionality?** This question evaluates the capability of the proposed model discovery method to attain high functional coverage in a controlled environment where the ground truth is available.
- **RQ2: How effective are the synthesised profiles at detecting faults in controlled environments?** This question tests the accuracy of the proposed method by applying mutation testing [16] to estimate the capacity of the created profiles to detect specific, injected faults.
- **RQ3: How accurately does TRACER model the functionalities of real-world, deployed chatbots?** This question addresses the practical, real-world applicability of TRACER. To answer it, we run TRACER against a set of deployed chatbots, and then perform a manual verification of every functionality inferred. This allows us to measure the precision of the model, that is, the percentage of discovered functionalities that are correct and valid.

RQ1 evaluates the coverage of the proposed approach in terms of activating chatbot functionalities. To measure the coverage, TRACER will be executed against the chatbot under testing and the generated synthesised profiles will be executed with SENSEI against the same chatbot. This process will be performed with chatbots created with the Taskyto framework because this framework enables the reporting of the activated modules. With these reports, it will be possible to measure the coverage obtained during the TRACER and SENSEI executions independently.

Higher coverage is anticipated during SENSEI profile execution, as profiles are designed to systematically test all discovered parameters and their valid combinations. For instance, TRACER may identify that a pizzeria chatbot offers six pizza types, but does not try them all during exploration. Instead, the generated profiles are designed to test all options.

Evaluating RQ1 is crucial for understanding how thoroughly TRACER exercises the chatbot's functionality. High coverage does not guarantee high quality, but it implies that

a significant part of a system has been tested, increasing the confidence in its reliability [32].

RQ2 assesses the practical effectiveness of the proposed approach in detecting chatbot errors. To this aim, mutation testing is employed [52] by introducing defects into correct chatbots to create mutants, and testing whether the generated profiles detect these defects. A profile successfully detects a mutation (i.e., a fault) when the simulated user fails to complete a goal (e.g., ordering a pizza type that was removed in the mutant) or when an expected output is missing or incorrect. Therefore, assessing RQ2 is important for providing evidence that our approach produces profiles that are sufficiently comprehensive to reveal real-world chatbot errors.

RQ3 assesses TRACER's ability at modelling and synthesising real-world deployed chatbots. Unlike RQ1 and RQ2, which involves chatbots developed with the Taskyto framework specifically for this evaluation, RQ3 evaluates chatbots deployed for real-world applications such as, universities, town halls or transportation companies. To evaluate this, the model inferred by TRACER will be manually validated to measure the precision of the discovered functionalities.

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

## 6.1. RQ1: Coverage of Chatbot Functionality

The primary goal of this first experiment is to quantitatively measure the coverage of TRACER during its execution and the subsequent execution of the generated user profiles with SENSEI. To achieve this white-box evaluation we ran the experiment in a controlled environment using a chatbot whose source code is known, which allowed us to measure the number of modules activated during each conversation.

### 6.1.1. Experiment Setup

To assess the effectiveness of the proposed approach in discovering chatbot functionality, we applied TRACER to four deployed Taskyto chatbots [8]. We chose chatbots built this technology because their code is open-source [77], which allowed us to instrument them to trace the modules activated during conversations, as required to assess RQ1. Moreover, the declarative structure of the chatbots facilitated injecting faults into them, as required to assess RQ2.

Taskyto is a declarative framework to develop LLM-based task-oriented chatbots using a YAML-like DSL. Chatbot definitions consist of any number of modules of five possible types:

- **Menu:** to define conversation alternatives.
- **Sequence:** to define a sequence of conversation steps.
- **Action:** to execute Python code and produce a verbatim or rephrased response upon receiving some input data.
- **Data gathering:** to request user input data.
- **Question answering:** to declare a FAQ.

Table 6.1 displays size metrics of the chatbots used for this experiment. We show the number of modules, lines of code (YAML lines), input fields or parameters, the number of possible values for these inputs in case that they are an enumeration, and the number of question-answer pairs for the FAQ.

*Bike-shop* schedules bike repair appointments and answers bike maintenance questions. *Photography* is a chatbot for a photography shop, which answers questions about the shop, gathers contact details of clients, and gives price estimates. *Pizza-order* handles pizza orders, including their size, toppings and drinks. *Veterinary* sets appointments and answers questions about a veterinary clinic.

Table 6.1.: Size metrics of the four Taskyto chatbots used in the evaluation.

| Chatbot | Modules | YAML lines | Inputs | Values | Q&A Pairs |
|---|---|---|---|---|---|
| Bike-shop | 3 | 65 | 3 | 2 | 4 |
| Photography | 5 | 140 | 9 | 12 | 5 |
| Pizza-order | 10 | 282 | 22 | 78 | 6 |
| Veterinary | 3 | 71 | 3 | 4 | 5 |

We modified Taskyto to generate logs for each conversation containing each activated module, expected input data from the user (e.g., pizza type), values provided for the input data with a limited set of options (e.g., Carbonara), and performed questions from the chatbot FAQ. Subsequently, a Python script was used to unify these logs, first by merging them into a unified log, and then by generating a Markdown report with the coverage percentage obtained for each chatbot.

To account for the non-deterministic nature of LLMs, we repeated the experiment three times. To ensure that TRACER finds as many functionalities as possible we ran TRACER for 20 conversations with 12 turns each to allow sufficient time for functionalities to branch out. These parameters provide a balance between discovery and time. The LLM used by TRACER was Google's Gemini 2.0 Flash with the default model's parameters (1.0

tempereature), which we did not modify since it allowed creativity in the conversations while still following TRACER's instructions.

SENSEI and Taskyto used OpenAI's GPT-4o-mini, we used an OpenAI model since these programs did not yet support a different provider For Taskyto, the temperature was set to 0 to ensure the chatbot's behaviour was deterministic. SENSEI used the temperatures defined in the generated profiles, which as explained during the profile generation in Section 4.2 is set randomly.

The overall procedure then, is as follows. First, we executed TRACER three times with the aforementioned settings measuring the Taskyto coverage during each execution. Using TRACER's generated profiles, we then ran SENSEI. Since TRACER was run three times, this resulted in three different sets of profiles. Consequently, we measured the coverage for each of these three executions and also reported an aggregated coverage.

### 6.1.2. Results and Discussion

Table 6.2 summarizes the results obtained throughout this experiment. It contains the median and aggregated coverage obtained during TRACER's exploration alone and during the execution of the generated profiles within SENSEI. The coverages are split into modules, input parameters, values of these inputs and FAQ questions. The table highlights in green the highest median and aggregate coverage obtained per chatbot and metric (i.e., module, input, value or question).

Regarding activated modules both TRACER and SENSEI achieved a 100% aggregate coverage across all chatbots, and similarly for the median coverage, except for one case where TRACER achieved an 83.33%. This shows that the core functionalities of all chatbots were successfully discovered during the exploration and exercised during the profile execution.

In terms of the input parameters and the values that these input fields allow, we also obtained high values, with a minimum value of 62.50% aggregated value for TRACER and a 85.71% for SENSEI, with both reaching a 100% for the *Pizza-order* chatbot. For the values that these fields can take, we obtained coverages as low as 48.81% or 55.56% in the aggregate TRACER coverage, but then for SENSEI we obtained higher values between 77.78% and 94.12%. This was expected since during the exploration TRACER might find values or options and not explore them further but they are subsequently used in the profiles, which means that SENSEI will use these values. A clear example of this is the aggregate coverage in in the *pizza-order* values, with only a 48.81% during exploration but a 90.48% for the profiles, This example shows a greater jump since as shown in Table 6.1 this chatbot has the highest number of values (78 values, while the other chatbots have 2, 12 and 4). This shows that the profile creation is comprehensive.

Lastly, aggregate question coverage was somewhat lower, with SENSEI oscillating

Table 6.2.: Coverage of TRACER (chatbot exploration) and SENSEI (profile execution). In green there is the greatest coverage obtained median and aggregate coverage per chatbot and metric (i.e., module, input, value or question).

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

between a 50 and a 100%. This is the only metric where the aggregate coverage of TRACER was higher than SENSEI's, which is contrary to the expected outcome. This occurred because Taskyto sometimes struggles with FAQs; if the question is not asked exactly as it is written, Taskyto often fails to identify it. TRACER asked it as it was written in the FAQ "Which is the price of a new tire?" while SENSEI used "What is the price of a new tire?" and Taskyto did not proceed with the answer, thus, not adding it to the coverage logs.

### 6.1.3. Answer to RQ1

We conclude that TRACER is able to effectively identify most of a chatbot's functionalities, input parameters and values for these inputs. Furthermore, we can conclude that TRACER effectively models the chatbot since the inferred model used to generate the user profiles for SENSEI managed to achieve aggregated coverages in the 77.78% to the 100% range except for the outlier in the *Bike-shop* FAQ.

## 6.2. RQ2: Effectiveness in Detecting Faults

After establishing TRACER's effectiveness at discovering chatbot functionalities (RQ1), this second experiment aims to assess the practical fault detection capability of the synthesised user profiles. To this end, we will use mutation testing [52] to create faulty versions of the chatbots (mutants) and measure the percentage of these faults that are detected when executing the generated user profiles with SENSEI.

### 6.2.1. Experiment Setup

We applied mutations (i.e., injected errors) to the four chatbots used in the previous RQ. To choose the user profiles, we selected the set that achieved the highest coverage out of the three sets generated during RQ1.

To create the mutants we used a Python script that generates as many mutant chatbots as possible from a given original, introducing a single error in each mutant. It achieves this by systematically injecting single faults into the chatbot's YAML definition. The following mutation operators are based on previous work on mutation testing for task-oriented chatbots [16, 17]:

- **Delete Enum Value:** deleting a possible value from an input parameter (e.g., removing `small` from `pizza_size`).
- **Change Optionality:** making an optional parameter required, or vice-versa.

- **Delete QA Pair:** removing a question from a FAQ module.
- **Swap QA Answer:** swapping the answers of two questions.
- **Delete Menu Alternative:** removing a conversational choice from a menu module.
- **Delete Fallback:** removing the fallback response.
- **Delete Sequence Step:** removing a necessary step from a workflow.
- **Swap Sequence Steps:** swapping the order of two steps in a sequence.
- **Delete Output Data:** omitting a piece of data from a chatbot's response (e.g., the price).
- **Change Rephrase Behavior:** altering the LLM's rephrasing mode.
- **Change Memory Scope:** changing the chatbot's ability to access data from previous turns.

The script systematically injects faults in every possible location of the chatbots. However, due to the time and cost of running SENSEI, we randomly selected two mutants of each type for each chatbot. If only one or zero mutants of a type existed, all were selected.

A mutant is considered "killed" if, during the execution of the SENSEI profiles, at least one of the following conditions is met:

- The simulation results in a crash, timeout, or conversation loop.
- A specified conversation `goal` is not achieved (e.g., the user fails to order the pizza).
- An expected `output` is missing or incorrect.

### 6.2.2. Results and Discussion

Table 6.3 shows the results for each chatbot. The column `# Mutants` represents the total number of mutants generated following the rule of selecting a maximum of 2 per mutation type. `# Profiles` shows the amount of profiles used followed by `# Convs. Per Chatbot`, which represents the total number of conversations that occur by using the set of profiles from the previous column. `Total Convs.` is the product of `# Mutants` and `# Convs. Per Chatbot` since each mutant is a new chatbot. Finally, `Mutation Score (%)` and `# Live Mutants` show the results from the executions.

Table 6.3.: Summary of the mutation analysis results for each chatbot.

| Chatbot | # Mutants | # Profiles | # Convs. Per Chatbot | Total Convs. | Mutation Score (%) | # Live Mutants |
|---|---|---|---|---|---|---|
| Bike-shop | 13 | 3 | 9 | 117 | 91.0 | 1 |
| Photography | 18 | 3 | 12 | 216 | 76.9 | 3 |
| Pizza-order | 20 | 4 | 22 | 440 | 75.0 | 3 |
| Veterinary | 13 | 5 | 21 | 273 | 91.7 | 1 |
| **TOTAL** | **64** | **15** | **64** | **1046** | **84.6** | **8** |

Mutation Score (MS) is the percentage of the mutants that were killed, while the number of live mutants represents the number of errors that were not found.

We observe a high overall Mutation Score with an average MS of 84.6%. All chatbots achieved at least 75.0%, and peaking at 91.0%. The cases with the lowest MS (75.0% and 76.9%) correspond to the most complex chatbots (see Table 6.1).

Although most of the errors were detected automatically (58%), some cases required manually inspecting the conversation, e.g., when the chatbot omitted a specific piece of data (such as the location of the photography shop) but included other requested information. For instance, the Photography chatbot outputs a summary of the shop information, and some but not all information is set in this output, therefore, there is no error of an unachieved goal, even if the response is incomplete. To detect such errors automatically, detecting such errors automatically would require testing rules that set oracles and analyse the expected chatbot output values, as explained in [19].

### 6.2.3. Answer to RQ2

The high mutation scores achieved (84.6% on average) provide strong evidence that the profiles synthesised by TRACER are effective at detecting faults in controlled environments. However, full automation would require complementing the profiles with (manually created) testing rules that search for specific data in the profile outputs.

## 6.3. RQ3: Accuracy of Modeling Deployed Chatbots

The goal of RQ3 is to evaluate TRACER in a realistic environment, using a black-box chatbot for which the ground truth is unknown. Unlike the controlled environments of RQ1 and RQ2, this evaluation uses publicly deployed, real-world chatbots. The goal is to measure the precision of TRACER's inferred model, defined as the percentage of

discovered functionalities that are correct and valid. We chose this metric over others, such as recall, because we do not have access to the complete model. Therefore, we cannot determine if the inferred model is missing any functionality.

### 6.3.1. Experiment Setup

For this experiment, we selected five different real-world deployed chatbots. Ada-UAM [21], Gallo de Morón de la Frontera [78], Madrid te Cuida [79], Gestri (Diputación de Valencia) [80], and Ayuntamiento de Arucas [81].

Ada-UAM is a university support chatbot. Gallo de Morón de la Frontera and Ayuntamiento de Arucas are citizen services chatbots for town halls. Gestri offers a chatbot that assists with the payment of taxes. Lastly, Madrid te Cuida offers sanitary assistance.

These chatbots were chosen because they represent genuine production systems from various public sector domains. All of them were easily accessible through an API.

The first stage of the protocol was to execute TRACER against each of these five chatbots. To maintain consistency with the previous research questions, we ran TRACER with 20 sessions, each with 12 turns, and chose Gemini 2.0 Flash as the LLM.

The second stage was the manual verification. The verification was performed for each functionality and it involved interacting with the chatbot and validating that the chatbot exhibits that functionality. We also checked for duplicate functionalities. Based on this verification, each of TRACER's discovered functionalities was classified as either a True Positive (TP) or a False Positive (FP). With the results, we computed the precision (see Equation 6.1), a metric that measures the correctness and accuracy of TRACER's output.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{6.1}$$

- **True Positive (TP):** The number of functionalities discovered by TRACER that were confirmed to be real, valid, and distinct.
- **False Positive (FP):** The number of functionalities discovered by TRACER that were either incorrect (not actually present in the chatbot) or were duplicates of another functionality.

### 6.3.2. Results and Discussion

The inferred models by TRACER can be found at `http://miso.ii.uam.es:8081/tracer-dashboard`. The screenshot in Figure 6.1 shows the executions that contain the results of this experiment. The projects are set to public so that the results can be consulted without needing to sign up on the web application.


Figure 6.1.: Screenshot of the web application showing the TRACER executions that contain the results of the experiment.

Table 6.4 summarizes the results of the manual verification for each of the five chatbots. The table is divided into five columns; the first is the chatbot under testing. `# Func. (TRACER)` is the number of functionalities found by TRACER. `TP` is the number of those functionalities that have been manually validated, i.e., the chatbot exhibits those functionalities and are not duplicated. Meanwhile, `FP` represents those that have been verified to be incorrect or duplicated. Lastly, we have the `Precision` computed as shown in Equation 6.1.

An average precision of 96.0% can be observed, with a minimum of 90.2% in the Arucas Town Hall chatbot. A perfect score of a 100% was achieved on the simple chatbots Ada-UAM and Gestri. And a 96.7% was obtained in Madrid te Cuida, the most complex chatbot tested with 91 discovered functionalities, which 88 of them were verified to be valid and unique.

All but one of the FPs were duplicates. This means that out of the 210 discovered functionalities, only one was hallucinated, i.e., 99.52% of the functionalities are real

Table 6.4.: Precision of TRACER's Inferred Models for Deployed Chatbots.

| **Chatbot** | **# Func. (TRACER)** | **TP (Verified)** | **FP (Incorrect/Duplicate)** | **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** |

functionalities exhibited by the chatbot. The hallucinated functionality was a claim that Gallo de Morón offered a service to claim lottery prizes, which it does not. The remainder of the FPs were all duplicates in which the LLM identified two very similar functionalities. For example, in the 'Arucas' chatbot, TRACER created separate nodes for 'Present Electronic Office Options' and 'Provide Electronic Office link'. This indicates that while TRACER's discovery process is robust, the automated consolidation logic can be further improved.

### 6.3.3. Threats to Validity

The primary threat to this experiment is the inability to calculate the recall. This is due to the nature of black-box systems, which makes obtaining a ground-truth model of the chatbot infeasible. Therefore, while the correctness of the inferred model (precision) was effectively measured, its completeness (recall) could not be quantitatively assessed and remains an open question.

### 6.3.4. Answer to RQ3

The results of this experiment allow us to positively answer RQ3. TRACER demonstrated its ability to accurately model the functionalities of real-world, deployed chatbots, achieving an average precision of 96.0%. The qualitative analysis revealed that the errors were not due to the hallucination of false functionalities but rather to an over-granularity in distinguishing between semantically similar features. This confirms that TRACER is a precise tool to automatically model chatbots in a black-box and real-world scenario.

# 7. Conclusions and Future Work

This final chapter synthesises the contributions of the thesis, summarises the key findings from the experimental evaluation, and outlines promising directions for future research.

## 7.1. Conclusions

The proliferation of complex, heterogeneous, and often black-box conversational agents has created a pressing need for advanced, automated testing methodologies. As established in this thesis, existing approaches to chatbot quality assurance are frequently limited by their reliance on manual effort, access to source code, or pre-existing conversation corpora, hindering their applicability in real-world scenarios.

This thesis presented TRACER, a two-phase methodology for the automated exploration and profiling of conversational agents. The approach begins with an Exploration Phase, where an LLM-powered agent systematically interacts with a chatbot to automatically infer a functional model of its capabilities. This is followed by a Profile Generation process that synthesizes a comprehensive suite of detailed, executable test user profiles directly from the inferred model. The entire methodology is implemented as an open-source tool, available via a CLI or through a web application.

The effectiveness of this approach was rigorously validated through three research questions. The evaluation first demonstrated that in a controlled, white-box environment, TRACER is highly effective at achieving high functional coverage of a chatbot's capabilities (RQ1). Furthermore, the synthesised profiles proved capable of detecting faults, successfully identifying an average of 84.6% of injected errors in a comprehensive mutation testing analysis (RQ2). Finally, in a practical, black-box setting against five real-world deployed chatbots, the models inferred by TRACER were shown to be highly accurate, achieving an average precision of 96.0% upon manual verification (RQ3).

Taken together, these findings confirm that the methodology presented in this thesis provides a practical and effective solution that significantly advances the state of the art. By successfully bridging the gap between automated model inference and test case synthesis, TRACER offers a fully automated, framework-agnostic, and black-box approach to the quality assurance of modern conversational agents.

## 7.2. Future Work

While this thesis presents a complete and validated methodology, this research opens up several promising avenues for future work. The following directions could build upon the foundation established by TRACER:

- **Improving the Consolidation Logic:** The evaluation of RQ3 revealed that while TRACER's discovery process is robust, the automated consolidation of semantically similar functionalities remains a challenge, especially for highly complex chatbots. Future work could explore more sophisticated semantic clustering algorithms.
- **Enhancing Profile Synthesis with Advanced Testing Strategies:** The current profile generation process creates comprehensive tests for discovered functionalities. A promising extension would be to incorporate more advanced and adversarial testing strategies. This could include synthesizing profiles that specifically test for security vulnerabilities like prompt injection, check for social biases in the chatbot's responses, or simulate users who are deliberately uncooperative or attempt to derail the conversation.
- **Extending to Multimodal and Voice-Based Agents:** The current implementation of TRACER focuses on text-based conversational agents. A significant future direction would be to extend the methodology to support multimodal chatbots that interact using images, interactive buttons, and voice. This would require extending the 'chatbot-connectors' library to handle different input/output formats and leveraging multimodal LLMs for both the Explorer Agent and the user simulator.
- **Quantitative Measurement of Recall in Black-Box Settings:** As identified in the threats to validity for RQ3, a major open challenge in the field is the quantitative measurement of recall (completeness) for black-box systems. Future research could investigate the application of statistical methods, such as capture-recapture models from ecology, to estimate the total number of functionalities in a system. This could provide a quantitative estimate of recall, even without access to the source code, further strengthening the evaluation of black-box discovery tools.

These future directions highlight the potential for the TRACER methodology to serve as a foundational platform for a new generation of intelligent, automated quality assurance tools for the rapidly evolving landscape of conversational AI.

# A. Prompts

```
You are a meticulous Workflow Dependency Analyzer AND Thematic
    Categorizer. Your primary task is to analyze the provided
    functionalities (extracted interaction steps) and conversation
    snippets to model the **precise sequential workflow** a user
    follows to complete a transaction or achieve a core goal. Your
    secondary task is to suggest a high-level thematic category for
    each functionality.

Input Functionalities (Extracted Steps):
{func_list_str}

Conversation History Snippets (Context for Flow):
{conversation_snippets}

**TASK 1: Determine Functional Dependencies to Assign
    `parent_names`**

For EACH functionality provided in the input, you MUST determine its
    `parent_names`. A functionality (Child F) should have a parent
    functionality (Parent P) in `parent_names` ONLY IF Parent P meets
    **ALL** of the following strict criteria:
1. **Immediate & Necessary Prerequisite:** Parent P must be a step
    (an action or prompt by the chatbot) that is **DIRECTLY AND
    IMMEDIATELY NECESSARY** for Child F to occur or make logical
    sense in the conversation. Ask: "Is Child F impossible or
    nonsensical if Parent P did not *just* happen?" If Child F could
    happen without P, or if other steps intervene, P is NOT an
    immediate parent.
2. **Chatbot-Driven Sequence:** The conversation flow must clearly
    show the *chatbot* initiating Parent P, which then directly leads
    to the chatbot initiating Child F.
3. **Closest Functional Link:** If A -> B -> C, then C's immediate
    parent is B, not A. Focus ONLY on the *single closest* necessary
    preceding step performed by the chatbot.
4. **Core Task Progression:** Assume the conversation aims to
    complete ONE primary user goal (e.g., order an item). Steps
```

```
   listed as parents must be essential for progressing this core
   task.

**RULES FOR `parent_names` ASSIGNMENT:**
* **Unique Functionalities:** The output JSON list should contain
   each unique functionality name from the input ONCE. Your primary
   task is to assign the correct `parent_names` to these unique
   functionalities.
* **Root Nodes (`parent_names: []`):**
   * Functionalities that initiate a core task or a distinct sub-flow
      (e.g., `provide_welcome_message`, `start_new_order_flow`,
      `request_help_topic_selection`).
   * General meta-interactions (e.g., `greet_user`,
      `explain_chatbot_capabilities`) are typically roots.
   * The first *task-specific* prompt by the chatbot in a clear
      sequence (e.g., `prompt_for_item_category_to_order`) is often
      a root if not forced by a preceding meta-interaction.
* **Sequential Steps (Common Patterns):**
   * `prompt_for_X_input` is a strong candidate to be a parent of
      `confirm_X_input_details`.
   * `prompt_for_X_input` can be a parent of `prompt_for_Y_input` if
      Y is the immediate next piece of information solicited by the
      chatbot in a sequence (e.g., `prompt_for_size` ->
      `prompt_for_color`).
   * `present_choices_A_B_C` is a parent of
      `prompt_for_selection_from_A_B_C` if the prompt immediately
      follows the presentation of choices by the chatbot.
* **Branches:** If `offer_choice_path1_or_path2` leads to either
   `initiate_path1_action` or `initiate_path2_action`, then
   `offer_choice_path1_or_path2` is a parent to both.
* **Joins:** If distinct paths (e.g., `complete_path1_final_step` and
   `complete_path2_final_step`) BOTH can directly lead to a common
   subsequent step (e.g., `display_final_summary`), then
   `display_final_summary` would have `parent_names:
   ["complete_path1_final_step", "complete_path2_final_step"]`.
* **AVOID Conversational Fluff as Parents:** Steps like `thank_user`,
   `acknowledge_input_received`, or general empathetic statements
   are RARELY functional parents. Do NOT list them as parents if a
   more direct data-gathering step or action-enabling step is the
   true prerequisite.

**TASK 2: Suggest a Thematic Category**
```

```
For EACH functionality, assign a `suggested_category` - a short,
    descriptive thematic label that groups similar functionalities
    together. Aim for a small, consistent set of broad categories.

Examples of good categories for transactional chatbots:
* "Order Placement" - For steps related to selecting and specifying
    an order
* "Payment" - For steps related to payment methods, billing, etc.
* "User Authentication" - For steps related to login, verification,
    etc.
* "Order Confirmation" - For steps confirming or summarizing an order
* "Chatbot Meta" - For general chatbot interaction like greetings,
    explanations
* "Customer Support" - For troubleshooting or help-related
    functionalities

 Maintain consistency by reusing categories when appropriate rather
     than creating too many unique ones.

 **ANALYSIS FOCUS:**
 For every functionality, meticulously trace back in the
     conversation: What was the *chatbot's very last action or
     prompt* that was *essential* for this current functionality to
     proceed? That is its parent. If no such single essential step
     exists, it's likely a root node or its parent is misidentified.

 **OUTPUT STRUCTURE:**
 Return a JSON list where each object represents one of the unique
     input functionalities, augmented with the determined
     `parent_names` and `suggested_category`.
 - "name": Functionality name (string).
 - "description": Description (string).
 - "parameters": (Preserve EXACTLY from input).
 - "outputs": (Preserve EXACTLY from input).
 - "parent_names": List of names of functionalities that meet ALL
     the STRICT criteria above. Use `[]` for root nodes.
 - "suggested_category": A short string for the thematic category
     (e.g., "Order Placement", "Payment").

 **FINAL INSTRUCTIONS:**
 - Preserve ALL original details (name, description, parameters,
     outputs) for each functionality.
 - The list should contain each input functionality name exactly
     once.
```

```
  - Focus entirely on deriving the most accurate, functionally
      necessary `parent_names`.
  - The `suggested_category` is for organizational purposes and does
      not influence `parent_names`.
  - Output valid JSON.

  Generate the JSON list:
```

Listing A.1: Prompt used to structure the workflow of a transactional chatbot.

```
You are a meticulous Workflow Dependency Analyzer AND Thematic
    Categorizer. Your primary task is to analyze the provided
    functionalities (interaction steps) and conversation snippets to
    model the **interaction flow**.
You MUST recognize that for an **Informational/Q&A chatbot**, most
    functionalities will likely be **independent topics/root nodes**.
    Your secondary task is to suggest a high-level thematic category
    for each functionality.

**CRITICAL CONTEXT FOR THIS TASK:**
- This chatbot appears primarily **Informational/Q&A**. Users likely
    ask about independent topics.
- **Your DEFAULT action MUST be to assign `parent_names: []` (root
    node) to each functionality.**
- ONLY create parent-child links if conversational evidence for a
    *strict, forced functional dependency* is EXPLICIT, CONSISTENT,
    and UNDENIABLE.

Input Functionalities (Name and Description Only):
{func_list_str}
# Note: Full Parameter and Output details for each functionality are
    known but omitted here for brevity.
# YOU MUST PRESERVE the original parameters and outputs associated
    with each
# functionality name when generating the final JSON output. Your task
    is to determine `parent_names` and suggest a thematic category.

Conversation History Snippets (Context for Flow):
{conversation_snippets}

**TASK 1: Determine Minimal Functional Dependencies to Assign
    `parent_names`**

For EACH functionality provided in the input, you MUST determine its
```

```
   `parent_names`. Your **default is `parent_names: []`**.
Assign a parent (i.e., a non-empty `parent_names` list) ONLY IF
   Parent P meets **ALL** of the following extremely strict criteria:
1. **Explicit Forced Sequence:** The chatbot *explicitly states* or
   *programmatically forces* the user from Parent P to Child F
   (e.g., asks a clarifying question P whose answer F is required).
2. **Absolute Functional Necessity:** Child F is *literally
   impossible or completely nonsensical* without the specific
   information or state created by Parent P immediately preceding it.
3. **Consistent Observation (Ideal):** This explicit dependency
   should ideally be observed consistently whenever these
   functionalities appear.
4. **Closest Functional Link:** Even if a rare dependency exists,
   only list the *single closest* necessary parent step.

**RULES FOR `parent_names` ASSIGNMENT (Informational Context):**
* **Unique Functionalities:** The output JSON list should contain
   each unique functionality name from the input ONCE. Your primary
   task is to assign the correct `parent_names` (defaulting to `[]`).
* **OVERWHELMING DEFAULT: Root Nodes (`parent_names: []`):**
   * Assign `[]` to functionalities representing distinct
      informational topics (e.g., `provide_opening_hours`,
      `explain_return_policy`, `describe_product_X_features`).
   * Assume ALL topics are independent unless an EXPLICIT forced
      sequence is proven by the conversation.
   * Meta-interactions (e.g., `greet_user`, `list_capabilities`,
      `request_rephrasing`) are ALWAYS roots.
   * If in ANY doubt, assign `parent_names: []`.
* **RARE Exceptions for Non-Root Nodes (Potential Parents):**
   * A node that presents clarification options for a complex topic
      (e.g., `offer_topic_A_subcategories`) *might* be a parent to a
      node providing details on a chosen subcategory
      (`provide_details_for_subcategory_A1`), but ONLY if the
      chatbot *forces* this selection path.
   * A node asking for essential identifying information *before*
      providing specific data (e.g.,
      `prompt_for_policy_document_name`) *might* be a parent to the
      node providing that specific document
      (`display_policy_document_X`), if the document cannot be
      displayed otherwise.
* **AVOID Inferring Links:**
   * Do NOT link `topic_A` to `topic_B` just because a user asked
      about them sequentially in one conversation.
   * Do NOT link based on simple topical similarity.
```

```
   * Do NOT link just because one piece of information *could* be
      useful before another, unless the chatbot *forces* that order.

**TASK 2: Suggest a Thematic Category**

For EACH functionality, assign a `suggested_category` - a short,
   descriptive thematic label that groups similar functionalities
   together. Aim for a small, consistent set of broad categories.

Examples of good categories for informational chatbots:
* "Account & Access" - For account management, login issues, access
   rights
* "Network & Connectivity" - For network setup, troubleshooting,
   connections
* "Software & Applications" - For software features, updates,
   compatibility
* "Hardware Support" - For device-specific information and
   troubleshooting
* "General Information" - For company info, policies, etc.
* "Chatbot Meta" - For chatbot capabilities, help commands, etc.

Maintain consistency by reusing categories when appropriate rather
   than creating too many unique ones.

**ANALYSIS FOCUS:**
For every functionality, assume `parent_names: []`. Only override
   this default if you find **undeniable proof** in the conversation
   snippets of an **explicitly forced sequence** or **absolute
   functional necessity** linking it directly to an immediate
   predecessor performed by the chatbot.

**OUTPUT STRUCTURE:**
Return a JSON list where each object represents one of the unique
   input functionalities, augmented with the determined
   `parent_names` and `suggested_category`.
- "name": Functionality name (string).
- "description": Description (string).
- "parameters": (Preserve EXACTLY from original data - **DO NOT
   OMIT**).
- "outputs": (Preserve EXACTLY from original data - **DO NOT OMIT**).
- "parent_names": List of names of functionalities that meet ALL the
   STRICT criteria above (this will be `[]` for MOST, if not all,
   nodes).
- "suggested_category": A short string for the thematic category
```

```
   (e.g., "Account & Access", "Network & Connectivity").

**FINAL INSTRUCTIONS:**
- Preserve ALL original details (name, description, parameters,
   outputs) for each functionality in the final JSON.
- The list should contain each input functionality name exactly once.
- Focus entirely on deriving the most accurate `parent_names`,
   defaulting STRONGLY to `[]`.
- The `suggested_category` is for organizational purposes and does
   not influence `parent_names`.
- Output valid JSON. Ensure the entire response is a single,
   well-formed JSON list.

Generate the JSON list:
```

Listing A.2: A prompt used to structure the workflow of an informational chatbot.

# Abbreviations

**AI** Artificial Intelligence

**NLP** Natural Language Processing

**RAG** Retrieval-Augmented Generation

**LLM** Large Language Model

**DSL** Domain-Specific Language

**TRACER** Task Recognition And Chatbot ExploreR

**CLI** Command Line Interface

**API** Application Programming Interface

**JSON** JavaScript Object Notation

**PyPI** Python Package Index

**NLU** Natural Language Understanding

**DAG** Directed Acyclic Graph

**DFS** Depth First Search

**BFS** Breadth First Search

**CI/CD** Continuous Integration / Continuous Deployment

**CI** Continuous Integration

**CD** Continuous Deployment

**REST** Representational State Transfer

**SPA** Single-Page Application

**SQL** Structured Query Language

**ORM** Object-Relational Mapping

**DB** Data Base

**DRF** Django REST Framework

**LUIS** Language Understanding Intelligent Service

**URL** Universal Resource Locator

**HTTP** Hypertext Transfer Protocol

**MS** Mutation Score

**TP** True Positive

**FP** False Positive

**BYOK** Bring Your Own Key

# List of Figures

# List of Tables

# Bibliography


[1] "ChatGPT," Accessed: Jul. 10, 2025. [Online]. Available: https://chatgpt.com.

[2] "Google Gemini," Gemini, Accessed: Jul. 10, 2025. [Online]. Available: https://gemini.google.com.

[3] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao. "Large Language Models: A Survey." arXiv: 2402.06196 [cs], Accessed: Aug. 6, 2025. [Online]. Available: http://arxiv.org/abs/2402.06196, pre-published.

[4] "Dialogflow," Google Cloud, Accessed: Jul. 10, 2025. [Online]. Available: https://cloud.google.com/products/conversational-agents.

[5] "Rasa," Rasa, Accessed: Jul. 10, 2025. [Online]. Available: https://rasa.com/.

[6] "LangGraph," Accessed: Jul. 10, 2025. [Online]. Available: https://www.langchain.com/langgraph.

[7] "AutoGen," Accessed: Jul. 10, 2025. [Online]. Available: https://microsoft.github.io/autogen/stable/.

[8] J. Sánchez Cuadrado, S. Pérez-Soler, E. Guerra, and J. De Lara, "Automating the Development of Task-oriented LLM-based Chatbots," in *Proceedings of the 6th ACM Conference on Conversational User Interfaces*, ser. CUI '24, New York, NY, USA: Association for Computing Machinery, Jul. 8, 2024, pp. 1–10, ISBN: 979-8-4007-0511-3. DOI: 10.1145/3640794.3665538. Accessed: Mar. 19, 2025. [Online]. Available: https://doi.org/10.1145/3640794.3665538.

[9] J. S. Cuadrado, D. Ávila, S. Pérez-Soler, P. C. Cañizares, E. Guerra, and J. De Lara, "Integrating Static Quality Assurance in CI Chatbot Development Workflows," *IEEE Software*, vol. 41, no. 5, pp. 60–69, Sep. 2024, ISSN: 0740-7459, 1937-4194. DOI: 10.1109/ms.2024.3401551. Accessed: Jul. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10533225/.

[10] P. C. Cañizares, J. M. López-Morales, S. Pérez-Soler, E. Guerra, and J. De Lara, "Measuring and Clustering Heterogeneous Chatbot Designs," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–43, May 31, 2024, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3637228. Accessed: Jul. 10, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3637228.

[11] "Rasa Test," Accessed: Jul. 10, 2025. [Online]. Available: https://rasa.com/docs/pro/testing/evaluating-assistant/.

[12] "Cyara Botium," Cyara, Accessed: Jul. 10, 2025. [Online]. Available: https://cyara.com/products/botium/.

[13] R. Ren, J. W. Castro, S. T. Acuña, and J. De Lara, "Evaluation Techniques for Chatbot Usability: A Systematic Mapping Study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, pp. 1673–1702, 11n12 Nov. 2019, ISSN: 0218-1940, 1793-6403. DOI: 10.1142/s0218194019400163. Accessed: Jul. 10, 2025. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218194019400163.

[14] M. Vasconcelos, H. Candello, C. Pinhanez, and T. Dos Santos, "Bottester: Testing Conversational Systems with Simulated Users," in *Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems*, Joinville Brazil: ACM, Oct. 23, 2017, pp. 1–4. DOI: 10.1145/3160504.3160584. Accessed: Jul. 10, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3160504.3160584.

[15] P. C. Cañizares, D. Ávila, S. Perez-Soler, E. Guerra, and J. de Lara, "Coverage-based Strategies for the Automated Synthesis of Test Scenarios for Conversational Agents," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, ser. AST '24, New York, NY, USA: Association for Computing Machinery, Jun. 10, 2024, pp. 23–33, ISBN: 979-8-4007-0588-5. DOI: 10.1145/3644032.3644456. Accessed: Mar. 19, 2025. [Online]. Available: https://doi.org/10.1145/3644032.3644456.

[16] P. Gómez-Abajo, S. Pérez-Soler, P. C. Cañizares, E. Guerra, and J. de Lara, "Mutation Testing for Task-Oriented Chatbots," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24, New York, NY, USA: Association for Computing Machinery, Jun. 18, 2024, pp. 232–241, ISBN: 979-8-4007-1701-7. DOI: 10.1145/3661167.3661220. Accessed: Mar. 19, 2025. [Online]. Available: https://doi.org/10.1145/3661167.3661220.

[17] M. F. Urrico, D. Clerissi, and L. Mariani, "MutaBot: A Mutation Testing Approach for Chatbots," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, Lisbon Portugal: ACM, Apr. 14, 2024,

pp. 79–83. DOI: 10.1145/3639478.3640032. Accessed: Jul. 10, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3639478.3640032.

[18] J. de Lara, A. del Pozzo, E. Guerra, and J. Sánchez Cuadrado, "Automated End-to-End Testing for Conversational Agents," in *Journal of Systems and Software (under Evaluation), 2025*, Journal of Systems and Software (under evaluation), 2025.

[19] J. De Lara, E. Guerra, A. del Pozzo, and J. Sanchez Cuadrado. "Sensei," GitHub, Accessed: Jul. 10, 2025. [Online]. Available: https://github.com/satori-chatbots/user-simulator.

[20] I. Sotillo del Horno, *Chatbot-tracer: A tool to model chatbots and create profiles to test them.* Version 0.2.10. Accessed: Jul. 10, 2025. [Online]. Available: https://github.com/Chatbot-TRACER/TRACER.

[21] "AdaUAM," Accessed: Jul. 11, 2025. [Online]. Available: https://www.uam.es/uam/tecnologias-informacion/servicios-ti/acceso-remoto-red.

[22] "Metro de Madrid empleará la inteligencia artificial para atender a los usuarios a través de un nuevo canal de WhatsApp," Metro de Madrid, Accessed: Aug. 6, 2025. [Online]. Available: http://www.metromadrid.es/es/nota-de-prensa/2025-06-30/metro-de-madrid-empleara-la-inteligencia-artificial-para-atender-a-los-usuarios-a-traves-de-un-nuevo-canal-de-whatsapp.

[23] "Meet Julie: Your Virtual Amtrak Travel Assistant," Accessed: Aug. 6, 2025. [Online]. Available: https://www.amtrak.com/about-julie-amtrak-virtual-travel-assistant.

[24] "@kuki_ai," @kuki_ai, Accessed: Aug. 6, 2025. [Online]. Available: https://www.kuki.ai.

[25] S. Li, Y. Chen, and X. Zhang, "Enhancing Natural Language Instruction Document Comprehension with Large Language Models," in *2024 5th International Conference on Computer, Big Data and Artificial Intelligence (ICCBD+AI)*, Jingdezhen, China: IEEE, Nov. 1, 2024, pp. 622–626. DOI: 10.1109/iccbd-ai65562.2024.00109. Accessed: Jul. 11, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10933784/.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention Is All You Need." version 7, Accessed: Jul. 16, 2025. [Online]. Available: https://arxiv.org/abs/1706.03762, pre-published.

[27] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving. "Fine-Tuning Language Models from Human Preferences." arXiv: 1909.08593 [cs], Accessed: Aug. 6, 2025. [Online]. Available: http://arxiv.org/abs/1909.08593, pre-published.

[28] "OpenAI," Accessed: Jul. 16, 2025. [Online]. Available: https://openai.com/.

[29] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. "LLaMA: Open and Efficient Foundation Language Models." arXiv: 2302.13971 [cs], Accessed: Jul. 16, 2025. [Online]. Available: http://arxiv.org/abs/2302.13971, pre-published.

[30] "Claude \ Anthropic," Accessed: Jul. 16, 2025. [Online]. Available: https://www.anthropic.com/claude.

[31] J. Zamfirescu-Pereira, H. Wei, A. Xiao, K. Gu, G. Jung, M. G. Lee, B. Hartmann, and Q. Yang, "Herding AI Cats: Lessons from Designing a Chatbot by Prompting GPT-3," in *Proceedings of the 2023 ACM Designing Interactive Systems Conference*, Pittsburgh PA USA: ACM, Jul. 10, 2023, pp. 2206–2220, ISBN: 978-1-4503-9893-0. DOI: 10.1145/3563657.3596138. Accessed: Aug. 6, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3563657.3596138.

[32] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge: Cambridge university press, 2017, ISBN: 978-1-107-17201-2.

[33] S. Pérez-Soler, S. Juárez-Puerta, E. Guerra, and J. de Lara, "Choosing a Chatbot Development Tool," *IEEE Software*, vol. 38, no. 4, pp. 94–103, Jul. 2021, ISSN: 1937-4194. DOI: 10.1109/MS.2020.3030198. Accessed: Aug. 6, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9364349.

[34] *RasaHQ/rasa*, Rasa, Jul. 17, 2025. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/RasaHQ/rasa.

[35] "Microsoft Bot Framework," Accessed: Jul. 17, 2025. [Online]. Available: https://dev.botframework.com/.

[36] "LUIS (Language Understanding) - Cognitive Services - Microsoft," Accessed: Aug. 6, 2025. [Online]. Available: https://www.luis.ai/.

[37] "IBM watsonx Assistant Virtual Agent," Accessed: Aug. 6, 2025. [Online]. Available: https://www.ibm.com/products/watsonx-assistant.

[38] "AI Chat Builder - Amazon Lex - AWS," Accessed: Aug. 6, 2025. [Online]. Available: https://aws.amazon.com/lex/.

[39] "CrewAI," Accessed: Aug. 6, 2025. [Online]. Available: https://www.crewai.com/.

[40] *crewAIInc/crewAI*, crewAI, Aug. 6, 2025. Accessed: Aug. 6, 2025. [Online]. Available: https://github.com/crewAIInc/crewAI.

[41] "CrewAI Introduction," CrewAI, Accessed: Aug. 6, 2025. [Online]. Available: https://docs.crewai.com/en/introduction.

[42] Z. Duan and J. Wang. "Exploration of LLM Multi-Agent Application Implementation Based on LangGraph+CrewAI." arXiv: 2411.18241 [cs], Accessed: Aug. 6, 2025. [Online]. Available: http://arxiv.org/abs/2411.18241, pre-published.

[43] A. Wąsowski and T. Berger, *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Cham: Springer International Publishing, 2023, ISBN: 978-3-031-23668-6 978-3-031-23669-3. DOI: 10.1007/978-3-031-23669-3. Accessed: Aug. 6, 2025. [Online]. Available: https://link.springer.com/10.1007/978-3-031-23669-3.

[44] B. K. Aichernig, W. Mostowski, M. R. Mousavi, M. Tappler, and M. Taromirad, "Model Learning and Model-Based Testing," in *Lecture Notes in Computer Science*, Cham: Springer International Publishing, 2018, pp. 74–100, ISBN: 978-3-319-96561-1 978-3-319-96562-8. DOI: 10.1007/978-3-319-96562-8_3. Accessed: Jul. 13, 2025. [Online]. Available: http://link.springer.com/10.1007/978-3-319-96562-8_3.

[45] H. Hajipour, M. Malinowski, and M. Fritz, "IReEn: Reverse-Engineering of Black-Box Functions via Iterative Neural Program Synthesis," in *Communications in Computer and Information Science*, Cham: Springer International Publishing, 2021, pp. 143–157, ISBN: 978-3-030-93732-4 978-3-030-93733-1. DOI: 10.1007/978-3-030-93733-1_10. Accessed: Jul. 13, 2025. [Online]. Available: https://link.springer.com/10.1007/978-3-030-93733-1_10.

[46] G. Menguy, "Black-box code analysis for reverse engineering through constraint acquisition and program synthesis," Ph.D. dissertation, Université Paris-Saclay, Mar. 14, 2023. Accessed: Jul. 13, 2025. [Online]. Available: https://theses.hal.science/tel-04097552.

[47] Z. Luo, K. Liang, Y. Zhao, F. Wu, J. Yu, H. Shi, and Y. Jiang, "DynPRE: Protocol Reverse Engineering via Dynamic Inference," The Internet Society. Accessed: Jul. 13, 2025. [Online]. Available: https://www.bibsonomy.org/bibtex/18ccab56a03c098f0aabeeb15461716a3.

[48] M. Shahbaz and R. Groz, "Analysis and testing of black-box component-based systems by inferring partial models," *Software Testing, Verification and Reliability*, vol. 24, no. 4, pp. 253–288, Jun. 2014, ISSN: 0960-0833, 1099-1689. DOI: 10.1002/stvr.

1491. Accessed: Jul. 13, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/stvr.1491.

[49] N. Walkinshaw, "Reverse-Engineering Software Behavior," in *Advances in Computers*, Elsevier, 2013, pp. 1–58. DOI: 10.1016/b978-0-12-408089-8.00001-x. Accessed: Jul. 13, 2025. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B978012408089800001X.

[50] A. W. Biermann and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, Jun. 1972, ISSN: 0018-9340. DOI: 10.1109/tc.1972.5009015. Accessed: Jul. 17, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/5009015/.

[51] Q. Zhou, C. Liu, Y. Duan, K. Sun, Y. Li, H. Kan, Z. Gu, J. Shu, and J. Hu, "GastroBot: A Chinese gastrointestinal disease chatbot based on the retrieval-augmented generation," *Frontiers in Medicine*, vol. 11, May 22, 2024, ISSN: 2296-858X. DOI: 10.3389/fmed.2024.1392555. Accessed: Jul. 13, 2025. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fmed.2024.1392555/full.

[52] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978, ISSN: 0018-9162. DOI: 10.1109/c-m.1978.218136. Accessed: Jul. 17, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/1646911/.

[53] D. Griol, J. Carbó, and J. M. Molina, "A automatic dialog simulation technique to develop and evaluate interactive conversational agents," *Applied Artificial Intelligence*, vol. 27, no. 9, pp. 759–780, Oct. 21, 2013, ISSN: 0883-9514, 1087-6545. DOI: 10.1080/08839514.2013.835230. Accessed: Jul. 13, 2025. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/08839514.2013.835230.

[54] R. Ferreira, D. Semedo, and J. Magalhaes, "Multi-trait User Simulation with Adaptive Decoding for Conversational Task Assistants," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, Miami, Florida, USA: Association for Computational Linguistics, 2024, pp. 16 105–16 130. DOI: 10.18653/v1/2024.findings-emnlp.945. Accessed: Jul. 13, 2025. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.945.

[55] I. Sekulić, L. Lu, N. S. Bedi, and F. Crestani, "Simulating Conversational Search Users with Parameterized Behavior," in *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*, Tokyo Japan: ACM, Dec. 8, 2024, pp. 72–81. DOI: 10.1145/

3673791.3698425. Accessed: Jul. 13, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3673791.3698425.

[56] A. Salle, S. Malmasi, O. Rokhlenko, and E. Agichtein, "Studying the Effectiveness of Conversational Search Refinement Through User Simulation," in *Lecture Notes in Computer Science*, Cham: Springer International Publishing, 2021, pp. 587–602, ISBN: 978-3-030-72112-1 978-3-030-72113-8. DOI: 10.1007/978-3-030-72113-8_39. Accessed: Jul. 13, 2025. [Online]. Available: https://link.springer.com/10.1007/978-3-030-72113-8_39.

[57] J. Kiesel, M. Gohsen, N. Mirzakhmedova, M. Hagen, and B. Stein, "Simulating Follow-Up Questions in Conversational Search," in *Advances in Information Retrieval*, N. Goharian, N. Tonellotto, Y. He, A. Lipani, G. McDonald, C. Macdonald, and I. Ounis, Eds., Cham: Springer Nature Switzerland, 2024, pp. 382–398, ISBN: 978-3-031-56060-6. DOI: 10.1007/978-3-031-56060-6_25.

[58] P. R. S. B, M. Agnihotri, and D. B. Jayagopi, "Improving Asynchronous Interview Interaction with Follow-up Question Generation," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 6, pp. 79–89, Special Issue on Artificial Intelligence, Paving the Way to the Future 2021, ISSN: 1989-1660. Accessed: Jul. 13, 2025. [Online]. Available: https://ijimai.org/journal/bibcite/reference/2902.

[59] K. D. Dhole. "KAUCUS: Knowledge Augmented User Simulators for Training Language Model Assistants." arXiv: 2401.16454 [cs], Accessed: Jul. 13, 2025. [Online]. Available: http://arxiv.org/abs/2401.16454, pre-published.

[60] S. Terragni, M. Filipavicius, N. Khau, B. Guedes, A. Manso, and R. Mathis. "In-Context Learning User Simulators for Task-Oriented Dialog Systems." version 1, Accessed: Jul. 13, 2025. [Online]. Available: https://arxiv.org/abs/2306.00774, pre-published.

[61] J. Bandlamudi, K. Mukherjee, P. Agarwal, R. Chaudhuri, R. Pimplikar, S. Dechu, A. Straley, A. Ponniah, and R. Sindhgatta, "Framework to enable and test conversational assistant for APIs and RPAs," *AI Magazine*, vol. 45, no. 4, pp. 443–456, Dec. 2024, ISSN: 0738-4602, 2371-9621. DOI: 10.1002/aaai.12198. Accessed: Jul. 13, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/aaai.12198.

[62] J. De Wit, "Leveraging Large Language Models as Simulated Users for Initial, Low-Cost Evaluations of Designed Conversations," in *Lecture Notes in Computer Science*, Cham: Springer Nature Switzerland, 2024, pp. 77–93, ISBN: 978-3-031-54974-8 978-3-031-54975-5. DOI: 10.1007/978-3-031-54975-5_5. Accessed: Jul. 13,

2025. [Online]. Available: https://link.springer.com/10.1007/978-3-031-54975-5_5.

[63] A. Cooper, R. Reimann, D. Cronin, and A. Cooper, *About Face: The Essentials of Interaction Design*, Fourth edition. Indianapolis, IN: John Wiley and Sons, 2014, 690 pp., ISBN: 978-1-118-76657-6.

[64] "1 Million Bot," 1MillionBot, Accessed: Jul. 26, 2025. [Online]. Available: https://1millionbot.com/.

[65] "Ruff," Accessed: Jul. 26, 2025. [Online]. Available: https://docs.astral.sh/ruff/.

[66] "Django," Django Project, Accessed: Jul. 27, 2025. [Online]. Available: https://www.djangoproject.com/.

[67] "Django REST framework," Accessed: Jul. 27, 2025. [Online]. Available: https://www.django-rest-framework.org/.

[68] "React," Accessed: Jul. 27, 2025. [Online]. Available: https://react.dev/reference/react.

[69] "PostgreSQL," PostgreSQL Documentation, Accessed: Jul. 27, 2025. [Online]. Available: https://www.postgresql.org/docs/17/biblio.html.

[70] "Celery," Accessed: Jul. 27, 2025. [Online]. Available: https://docs.celeryq.dev/en/stable/.

[71] "RabbitMQ," Accessed: Jul. 27, 2025. [Online]. Available: https://www.rabbitmq.com/.

[72] "Nginx," Accessed: Jul. 28, 2025. [Online]. Available: https://nginx.org/en/.

[73] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2:2, Mar. 1, 2014, ISSN: 1075-3583.

[74] "Deployment checklist | Django documentation," Django Project, Accessed: Jul. 28, 2025. [Online]. Available: https://docs.djangoproject.com/en/5.2/howto/deployment/checklist/.

[75] "Django-Rest-Knox," Accessed: Jul. 28, 2025. [Online]. Available: https://jazzband.github.io/django-rest-knox/.

[76] "Fernet (symmetric encryption) — Cryptography 46.0.0.dev1 documentation," Accessed: Jul. 28, 2025. [Online]. Available: https://cryptography.io/en/latest/fernet/.

[77] *Satori-chatbots/taskyto*, satori-chatbots, May 14, 2025. Accessed: Aug. 12, 2025. [Online]. Available: https://github.com/satori-chatbots/taskyto.

[78] "El Ayuntamiento de Morón de la Frontera da el salto a la inteligencia artificial con "El gallo de Morón"," Ayuntamiento de Morón de la Frontera, Accessed: Aug. 17, 2025. [Online]. Available: https://www.morondelafrontera.es/es/noticias/22/el-ayuntamiento-de-moron-de-la-frontera-da-el-salto-a-la-inteligencia-artificial-con-%E2%80%9Cel-gallo-de-moron%E2%80%9D.

[79] "Chatbot "Madrid te cuida"," 1MillionBot, Accessed: Aug. 17, 2025. [Online]. Available: https://1millionbot.com/chatbot-madrid-salud/.

[80] "La Diputación atiende más de 85.000 consultas sobre gestión tributaria con su chatbot 'Gestri'," Valencia Plaza, Accessed: Aug. 17, 2025. [Online]. Available: https://valenciaplaza.com/valenciaplaza/diputacion-atiende-85000-consultas-gestion-tributaria-chatbot-gestri.

[81] "Ayuntamiento de Arucas - ASISTENTES VIRTUALES DE ATENCIÓN CIUDADANA," Accessed: Aug. 17, 2025. [Online]. Available: https://www.arucas.org/modules.php?mod=portal&file=ver_gen&id=T0RZek1RPT0=.