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 1, 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 1, 2025 Iván Sotillo del Horno

Acknowledgments

Abstract

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivation . . . 1
1.3 Objectives . . . 2
1.4 Contributions . . . 3
1.5 Thesis Organization . . . 3

2 Background and State of the Art **4**
2.1 Background . . . 4
2.1.1 Conversational Agents . . . 4
2.1.2 Large Language Models . . . 5
2.1.3 Black-box Testing . . . 6
2.1.4 Development Frameworks . . . 7
2.2 State of the Art . . . 9
2.2.1 Model Learning and Black-Box Reverse Engineering . . . 9
2.2.2 Methodologies for Chatbot Testing . . . 10
2.2.3 User Simulation for Automated Testing . . . 11
2.2.4 Summary and Identified Research Gaps . . . 13

3 TRACER: Automated Chatbot Exploration **15**
3.1 Overview . . . 15
3.2 Exploration Phase . . . 16
3.2.1 Initial Probing . . . 17
3.2.2 Iterative Sessions . . . 18
3.2.3 Functionality Extraction . . . 18
3.2.4 Functionality Consolidation . . . 20
3.3 Refinement Phase . . . 20
3.3.1 Global Consolidation . . . 20
3.3.2 Chatbot Classification . . . 21

3.3.3 Workflow Structure Inference . . . 21
3.3.4 Example: Inferred Model of a Pizzeria Chatbot . . . 21

4 User Profile Structure and Generation 24
4.1 User Profiles Structure . . . 24
4.1.1 LLM Configuration (`llm`) . . . 27
4.1.2 User Persona Definition (`user`) . . . 27
4.1.3 Chatbot Settings (`chatbot`) . . . 28
4.1.4 Conversation Control (`conversation`) . . . 28
4.2 User Profiles Generation . . . 29
4.2.1 Grouping Functionalities into Profiles . . . 29
4.2.2 Goal Generation . . . 29
4.2.3 Variable Generation . . . 30
4.2.4 Context Generation . . . 30
4.2.5 Output Definition . . . 31
4.2.6 Conversation Style Definition . . . 31
4.2.7 Profile Assembly and Validation . . . 31

5 Tool Support 33
5.1 Implementation and Architecture . . . 33
5.1.1 Core Framework: LangGraph . . . 33
5.1.2 Modular Architecture . . . 33
5.1.3 Generated Artifacts . . . 34
5.2 Distribution and Development Workflow . . . 34
5.3 The Command Line Interface . . . 35
5.3.1 Conversation Control . . . 35
5.3.2 Connector Configuration . . . 35
5.3.3 LLM Configuration . . . 35
5.3.4 Output and Logging . . . 35
5.4 The Web Application . . . 36
5.4.1 System Architecture . . . 36
5.4.2 Core Technology Stack . . . 36
5.4.3 Asynchronous Task Handling . . . 38
5.4.4 The Nginx Reverse Proxy . . . 38
5.4.5 Deployment and Containarization . . . 39
5.4.6 Security Measures . . . 41

6 Evaluation 42

7 Conclusions and Future Work 43

Abbreviations **44**

List of Figures **46**

List of Tables **47**

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 [**ChatGPT**] or Google's Gemini [**GoogleGemini**] 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 Large Language Models (LLMs), 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 [**Dialogflow**] or Rasa [**Rasa2020**], multi-agent programming environments based on LLMs like LangGraph [**LangGraph**] and Microsoft's AutoGen [**AutoGen**], and Domain-Specific Languages (DSLs) such as Taskyto [**sanchezcuadradoAutomatingDevelopmentTaskoriented2024**], 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 [**cuadradoIntegratingStaticQuality2024**, **canizaresMeasuringClusteringHeterogeneous2024**], they often focus on particular chatbot technologies [**RasaTest2025**], require substantial manual effort including the provision of test conversations [**CyaraBotium**, **RasaTest2025**]

or synchronous human interaction [**renEvaluationTechniquesChatbot2019**], rely on available conversation corpus [**vasconcelosBottesterTestingConversational2017**], or require access to the source code of the chatbot [**canizaresCoveragebasedStrategiesAutomated2024**, **gomez-abajoMutationTestingTaskOriented2024**, **urricoMutaBotMutationTesting2024**], 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 (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 [**delaraAutomatedEndtoEndTesting2025**, **delaraSensei**]. 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 [**delaraAutomatedEndtoEndTesting2025**, **delaraSensei**], 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 [**sotillodelhornoChatbottracerToolModel**] 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 special 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 modeling chatbot functionality?** This question evaluates the capability of our model discovery method to attain high functional coverage in a controlled environment where the ground truth is available.

- **RQ2: How effective are the synthesized profiles at detecting faults in controlled environments?** This question tests the accuracy of our method by applying mutation testing [**gomez-abajoMutationTestingTaskOriented2024**] to estimate the capacity of the created profiles to detect specific, injected faults.
- **RQ3: How effective is the approach at identifying real-world bugs and ensuring task completion in deployed chatbots?** This is the practical, real-world applicability of our framework by calculating the Bug Detection Rate (Bug Detection Rate (BDR)) and Task Completion Rate (Task Completion Rate (TCR)) of the generated profiles against real-world chatbots.

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 synthesizes this model into a set of user profiles that will executed with Sensei [**delaraSensei**] 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 [**AdaUAM**]. or chatbots made with the framework Taskyto [**sanchezcuadradoAutomatingDevelopmentTaskoriented2024**]
- **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 [**ChatGPT**] or Gemini [**GoogleGemini**].

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

- **Intent-based frameworks:** these frameworks such as Google's Dialogflow [**Dialogflow**] or Rasa [**Rasa2020**] 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 [**LangGraph**] or Microsoft's AutoGen [**AutoGen**], 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.

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 [**liEnhancingNaturalLanguage2024**], generation, and reasoning across diverse domains.

Large Language Models are built employing transformers, an architecture proposed by Vaswani et al. [**vaswaniAttentionAllYou2017**]. This architecture's main innovation is the self-attention mechanism, which allows 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. This process has made possible models like OpenAI's GPT

series (e.g., GPT-4) [**OpenAI2025**], Google's Gemini series [**GoogleGemini**], Meta's open-source Llama Series [**touvronLLaMAOpenEfficient2023**], or Anthropic's Claude models [**ClaudeAnthropic**].

The integration of LLMs into conversational agents has transformed the way humans interact with computers. Unlike traditional rules-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, 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. 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

Google's Dialogflow [**Dialogflow**] 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 [**Rasa2020**] offers and open-source [**RasaHQRasa2025**] 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.

Multi-Agent Programming Environments

LangGraph [**LangGraph**] 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 graph where nodes are AI agents or tools and edges control the flow of information between the nodes. This allows for more dyncamic 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 [**AutoGen**] 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.

Domain-Specific Languages

A Domain-Specific Language is a computer language specialized for a particular application domain. 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 [**sanchezcuadradoAutomatingDevelopmentTaskoriented2024**] 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 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 artifacts 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 [**aichernigModelLearningModelBased2018**], system reverse engineering [**hajipourIReEnReverseEngineeringBlackBox2021**, **menguyBlackboxCodeAn**
and network protocol inference [**luoDynPREProtocolReverse**].

Traditional model learning techniques, such as those demonstrated by Muzammil et al. [**shahbazAnalysisTestingBlackbox2014**], 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. [**walkinshawReverseEngineeringSoftwareBehavior2013**] extract behavioural models through dynamic analysis, utilizing techniques like k-tails algorithms [**biermannSynthesisFiniteStateMachines1972**] to infer finite state machines from ex-

ecution 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. [**renEvaluationTechniquesChatbot2019**] 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 [**zhouGastroBotChineseGastrointestinal2024**]. 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 [**vasconcelosBottesterTestingConversational2017**] use existing Q&A corpora to test this. Commercial platforms like Cyara [**CyaraBotium**] and Rasa's testing framework [**RasaTest2025**] 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. [**cuadradoIntegratingStaticQuality2024**] propose static quality analysis techniques that inspect the structural properties of a chatbot's implementation. To assess test adequacy, Cañizares et al. [**canizaresCoveragebasedStrategiesAutomated2024**] 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 [**demilloHintsTestData1978**] 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. [**gomez-abajoMutationTestingTaskOriented2024**] propose mutation operators specifically for task-oriented chatbots like Taskyto [**sanchezcuadradoAutomatingDevelopmentTaskoriented2024**] while Urrico et al. [**urricoMutaBotMutationTesting2024**] introduce MutaBot, a dedicated mutation testing framework for platforms like Dialogflow [**Dialogflow**]. 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. [**griolAutomaticDialogSimulation2013**] employed neural networks trained on dialogue corpora to suggest user utterances. The user simulation capabilities within Bottester [**vasconcelosBottesterTestingConversational2017**] 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. [**ferreiraMultitraitUserSimulation2024**] generate profiles with traits like engagement and verbosity, while Sekulic et al. [**sekulicSimulatingConversa** simulate users with varying levels of patience and politeness for conversational search. Frameworks like CoSearcher [**salleStudyingEffectivenessConversational2021**] also allow for tuning user cooperativeness. These works prove the principle of creating diverse, persona-driven simulated users.

The SENSEI user simulator [**delaraAutomatedEndtoEndTesting2025**, **delaraSensei**], 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. [**kieselSimulatingFollowUpQ** simulate follow-up questions, and the followQG framework [**bImprovingAsynchronousInterview2021**] uses trained models to generate contextually relevant continuations. More advanced frameworks leverage LLMs for even more complex tasks. The Kaucus simulator [**dholeKAUCUSKnowledge** incorporates external knowledge via retrieval augmentation, and Terragni et al. [**terragniInContextLearning** generate user utterances directly from high-level goal descriptions. Bandlamudi et al. [**bandlamudiFrameworkEnableTest2024**] employ a dual-LLM approach where one LLM simulates the user and another judges the chatbot's response. While this cleverly ad-

dresses 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 [**dewitLeveragingLargeLanguage2024**] 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 [**delaraAutomatedEndtoEndTesti**
delaraSensei], 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 input user 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 synthesize 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 [**zhouGastroBotChineseGastrointestinal2024**] Ren et al. [**renEvaluationTechniquesChatbot2019**]	Unscalable, not reproducible
Scripted Testing	No	Yes	Medium	Bottester [**vasconcelosBottesterTestingConversational2017**] Cyara [**CyaraBotium**] Rasa [**RasaTest2025**]	High manual effort, brittle
White-Box Testing	Yes	No	High	Cañizares et al. [**canizaresCoveragebasedStrategiesAutomated2024**] Gómez-Abajo et al. [**gomez-abajoMutationTestingTaskOriented2024**]	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 artifacts 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 Section 2.2 review. This tool addresses the black-box testing challenge mentioned by iteratively discovering functionalities to create a structured model.

The chapter will be structured with first a high-level overview of the tool's two phase implementation Section 3.1. Then we will detail the exploration phase Section 3.2, followed by the refinement phase 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 [**delaraSensei**, **delaraAutomatedEndtoEndTesting2025**] user simulator to test the chatbot. An scheme of the proposed end-to-end testing can be seen in Figure 3.1.

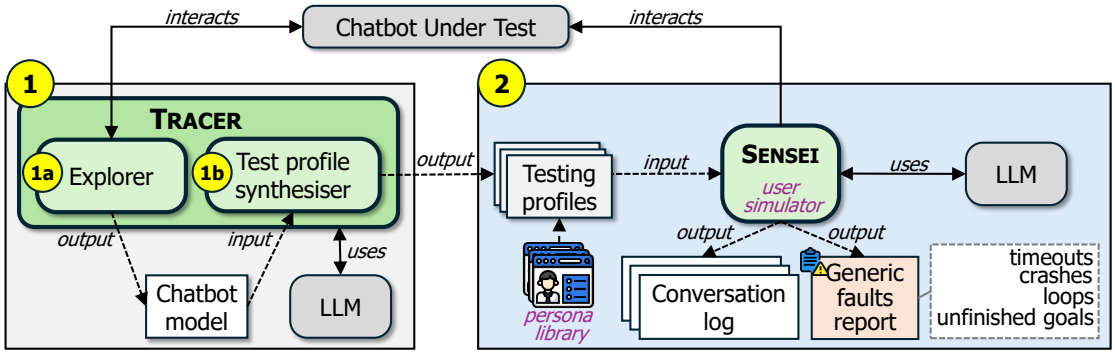

Figure 3.1: Scheme of our approach and its main components. (1a) Chatbot's functionality explorer. (1b) Synthesiser of test conversation profiles. (2) User simulator.

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:

- Language(s) that the chatbot understands.
- The chatbot's default fallback sentence (e.g., "I'm sorry, I can't undertand 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 pase 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.

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 to extract 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 synthezing 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.

The explorer agent, inspired by SENSEI [**delaraSensei**, **delaraAutomatedEndtoEndTesting2025**], mimics a human interacting with the chatbot thanks to the use of LLMs.

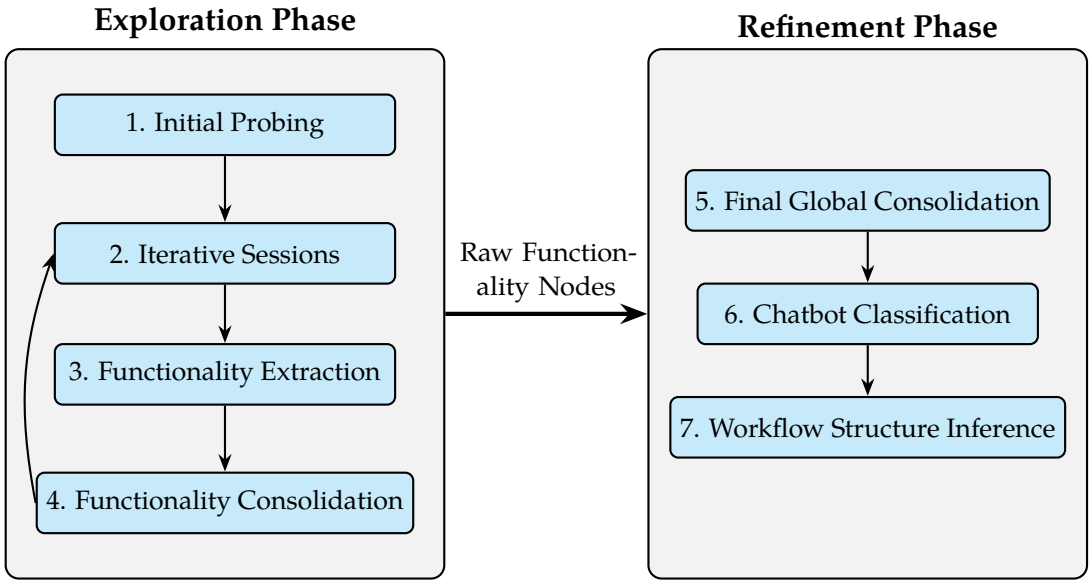

Figure 3.2: Flow-chart of TRACER's two phase methodology. The Exploration Phase (left) iteratively discoveres functionalities while the Refinement Phase (right) consolidates and structures them into the final model.

3.2.1 Initial Probing

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

- **Language Detection:** The agent determines the language by sending some basic messages to the chatbot and analyzing the response.
- **Fallback Message Detection:** The fallback message is the message that chatbots give when they cannot understand the user's intent. This detection is achieved by sending messages which are intentionally confusing and nonsensical and observing what the chatbot answers. 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?"

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 his words when the chatbot is not understanding him.

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 this 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, he will engage in a natural conversation by first greeting the chatbots, and then if the chatbots doesn't give away what he can do, the explorer will directly ask.
- **Functionality Branch Exploration:** in the case that there are functionalities in the queue, they will get popped and fed to the explorer agent. Then the explorer will have a conversation where he will try to find branches and variations of this functionality. 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.

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 enf of each conversation, the Explorer Agent looks at the conversation history and tries to look for 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)

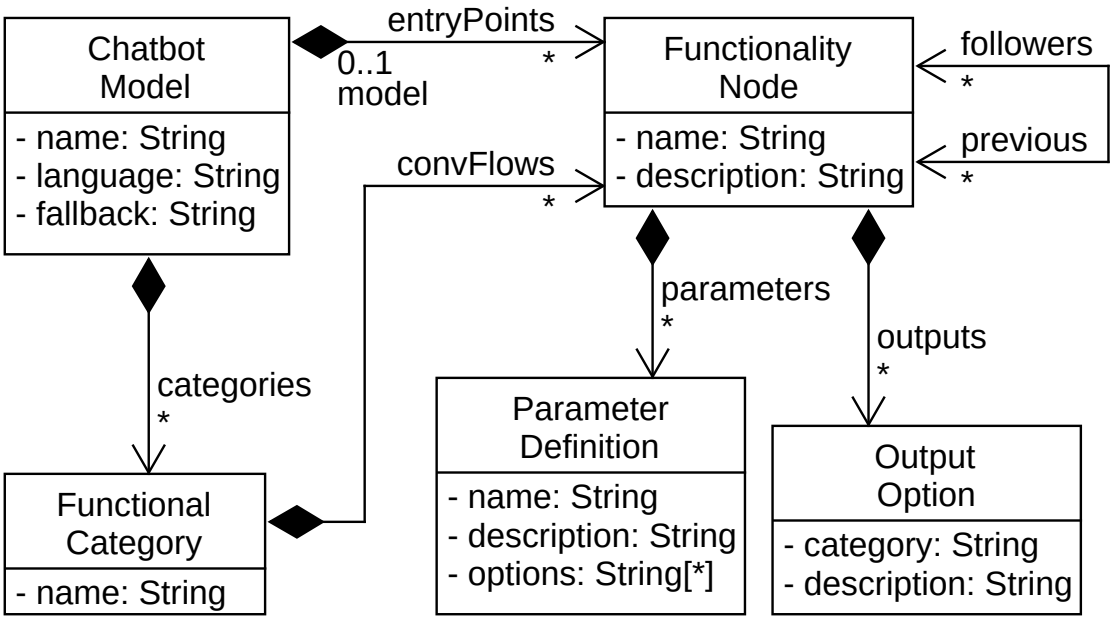

Figure 3.3: Chatbot model schema.

- **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.
- **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 aranged as a DAG, the nodes have children and parent that mimic the workflow of the chatbot. The idea of this workflow graph, is to order functionalities in the order that one will encounter them, for example, the chatbot will asks for the drinks always after asking for the pizzas so then the drink functionality should be a children of the pizza one.

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 look for semantically similar functionalities and merges them into one.

To better understand this, we will give an example. 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 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.

3.3 Refinement Phase

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

3.3.1 Global Consolidation

While during the exploration phase we have a consolidation phase, variations of the same functionalities may still appear. This consolidation step solves this by checking again all the functionality nodes.

3.3.2 Chatbot Classification

Next, based on the discovered functionalities and the conversation history the chatbot is classified into informational or transactional.

- **Informational:** these are chatbots that mainly answer questions and provide information. Examples of these are unversity or bank chatbots that only give you information or if you need to do any paperwork they will give you a link and redirect you but they will not assist you with the tramit directly.
- **Transactional:** these are chatbots that do guide the user through a task or workflow. For example, the pizzeria example we have been using or a hotel chatbot that helps you book a room.

3.3.3 Workflow Structure Inference

The final step is to define the Directed Acyclic Graph 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 set 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 on the identifications of 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.
- **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.

3.3.4 Example: Inferred Model of a Pizzeria Chatbot

Figure 3.4 shows the workflow graph resulting from a TRACER execution against a pizzeria chatbot built with Taskyto taken from [**sanchezcuadradoAutomatingDevelopmentTaskoriented2024**]. The entrypoint, 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 the categories and functionalities.

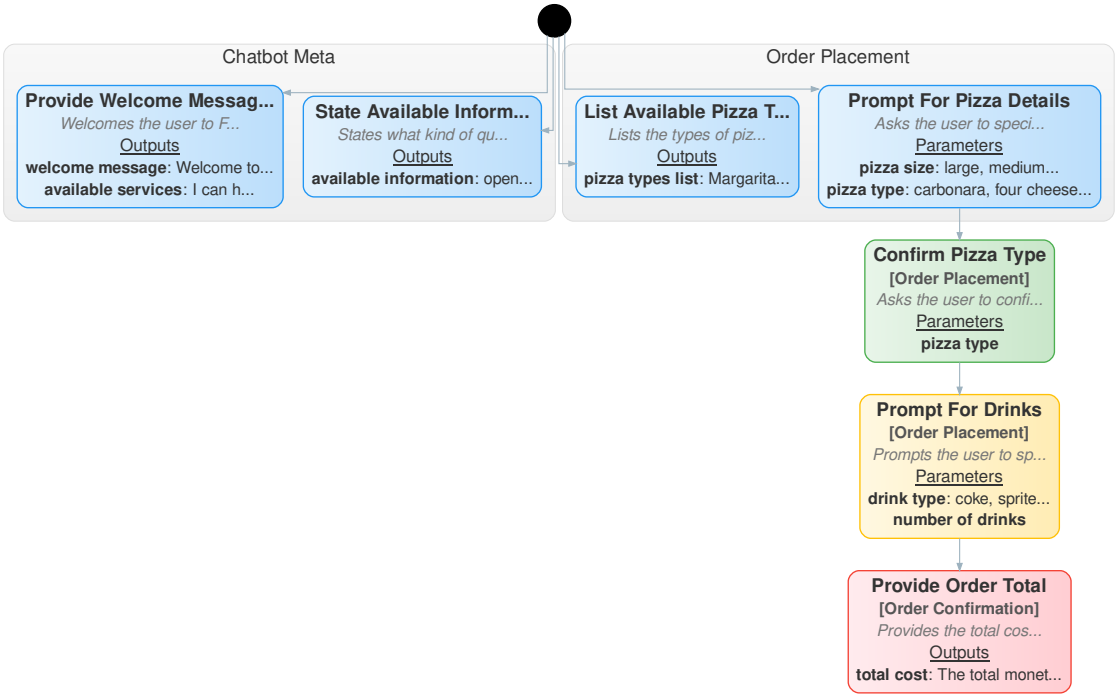

Figure 3.4: Workflow model inferred by TRACER from a pizzeria chatbot.

- **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 he wishes.
- **Order Confirmation:** 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 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. In the next section we will show how TRACER uses this model to generate user profiles for SENSEI.

4 User Profile Structure and Generation

Once the model has been finished, 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 value, and to find 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 synthesized 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 turn per conversations. 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 made up 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`)

In this section we can choose the Large Language Model that we are going to use along with its temperature. The chosen `model` will have an impact on the achieved results since each model has its own strengths and limitations and there will be models that will perform better than others, usually at the cost of being more expensive. In this field we will just input the name of the model (e.g. `gpt-4o-mini`). Then we have the `temperature`, this parameter controls the randomness of the LLM, that is, when the model is going to choose the next word how randomly it does it. A value of 0 means that the model is deterministic while 1 means that it will be more creative.

4.1.2 User Persona Definition (`user`)

In the `user` section, we will define the persona, its role, context and goals. We start with the `language` that the user will employ, followed by the `role` of the simulated user, that is, a sentence representing who the user is and what he ought to do (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. In it, we will describe 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 we have the list of parameters, which is the placeholders we left in goals. Each of these parameters will have assigned a function:

- `random()`: A random value of the data below will be taken.
- `random(x)`: Takes x different random values.
- `forward()`: Selects one value sequentially from the data list.
- `forward(x)`: Instead of one by one, takes the x next values.
- `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 chesee, ..., medium pepperoni, medium four chesee, ..., large pepperoni, large four chesee...

After the function, we can specify its `type`, with values like int, float, string or date. Right after that, we have the data field, where we will input a list of values or if it is a numeric value, a range with min and max and the step to go from the min to the max.

4.1.3 Chatbot Settings (`chatbot`)

Here, expected behaviour of the system under test will be specified. The first field is the `fallback`, this is the sentence given by the chatbot when he cannot understand what the user is saying, or what he is saying is outside of the chatbot's scope. Next we have 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 we will give a name 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.

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, specially 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 making sure 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. This styles are predefined and include some like *make spelling mistakes*, *use short phrases* or *single question*.

4.2 User Profiles 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 begins by grouping functionalities into profiles, followed by the LLM-driven generation of goals, variables, context, and 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 send 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 but 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 this functionality groups, the LLM will generate the `test_name` along with its `role`.

4.2.2 Goal Generation

After the grouping has been done, 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 if asking about the opening hours it will not

have 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 see waht a data source is. A data source are 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.

Now that we now this, 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 just for the sake of making one. Then, if one match is given, we request the LLM to generate another extra value based on what the returned data source to test the chatbot on values outside of the ones that he suggests. For example, with the pizza sizes small, medium, large, it tends to generate extra large as the extra value, or in the toppings pineapple was generated 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 are 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 one 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 content generated before 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 in a get together 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. It will let us test the chatbot 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, trying to look for things that will ensure that they have been achieved, 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, we went for a limit that is set based 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 if not the uer simulator tries to fulfil all the goals at the same time and ends up making weird complicated sentences.

4.2.7 Profile Assembly and Validation

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

This 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). On top of that, we have developed a web application that allows users to execute both TRACER and Sensei without the need of knowing 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 Profiles Generation) TRACER relies on LLMs, this is why used LangGraph [**LangGraph**] as our framework for development. The reason why we chose LangGraph was because it allows to manage and orchestrate complex agentic workflows with states, it is also an industry standard and it has an 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. Right now TRACER allows OpenAI and Gemini models, but thanks to LangGraph it would be easy to add other LLM providers.

5.1.2 Modular Architecture

TRACER can infer a chatbot model as long as the chabots is accessible through an interface, typically a REST API. Right now it provides access to communicate with chatbots made with different technologies, such as Taskyto [**sanchezcuadradoAutomatingDevelopmentTaskoriente**
Rasa [**Rasa2020**] or 1MillionBot [**1MillionBot**]. 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 Artifacts

Upon completion, TRACER generates the following artifacts 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 like the discovered functionalities, fallback message, language and also other information like token usage, number of LLM calls or estimated cost.
- A graph representing the inferred model's workflow (see Figure 3.4)
- A JSON file containing the same workflow but in text.

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/`), making it easy to install by running `pip install chatbot-tracer`. This not only makes it easy to use, but also makes it easy to implement into other projects such as the web application done, or other projects that could use TRACER since it can just 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 made use of Ruff [**Ruff**]. Ruff is Python linter and formatter written in Rust that combines tools like Flake8 or Black into a single and faster tool. We made use 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 The Command Line Interface

The primary way of using TRACER is through the CLI. We can execute the whole TRACER pipeline, from chatbot exploration to profile generation, through a single command. This way enables users who prefer terminal-based workflows such as developers, to execute TRACER easily. It also allows TRACER to be integrated 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 the following ones:

5.3.1 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.

5.3.2 Connector Configuration

The arguments `--technology` or `-t` along with `--url` or `-u` allow to configure the connector by choosing the chatbot technology along with the API endpoint.

5.3.3 LLM Configuration

Using `--model` or `-m` lets one decide 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 appaer in the profiles will be the same one used for TRACER, it is recommended to use a better model for exploration and analysis since will infer a more comprehensive model with more realistic profiles, and then a more economic model to run the profiles since there will be many more LLM calls and the chosen model will not have that much impact.

5.3.4 Output and Logging

We also have the verbose levels which can be three levels: the basic one where we will just see things like the discovered functionalities, which session or phase are we in and

warnings. The verbose level, activated with `-v`, which allows us to see the conversation too, and the debug `-vv` that will show what the verbose shows plus information like prompts sent to the LLM, its responses, and other logs that may have been set during the development and debugging of the program. Lastly, we have `--output` or `-o`, which simply controls where all the generated artifacts will be placed.

```
$ tracer -t taskyto -u http://localhost:5000 -s 12 -n 8 -m
  gemini-2.5-flash -pm gemini-2.0-flash -o ./pizzeria_results
  -v
```

Listing 5.1: Tracer command example.

The command in Listing 5.1 demonstrates a typical execution of TRACER against a Taskyto-based pizzeria chatbot. It is configured to run 12 exploration sessions of 8 turns each. For the exploration, analysis and profile generation, the smarter model Gemini 2.5 Flash will be used, then for the model that is defined in each user profile, Gemini 2.0 Flash will be used since it is faster and cheaper. We used the `-v` to be able to monitor the conversations that are happening between the explorer agent and the chatbot under testing. Finally, all the artifacts will be stored in a directory called `pizzeria_results`.

5.4 The Web Application

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

5.4.1 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.

5.4.2 Core Technology Stack

The backend of the application was developed in Django [**Django**], the reason why this framework was used is because it is made for Python so it suits both TRACER and Sensei,

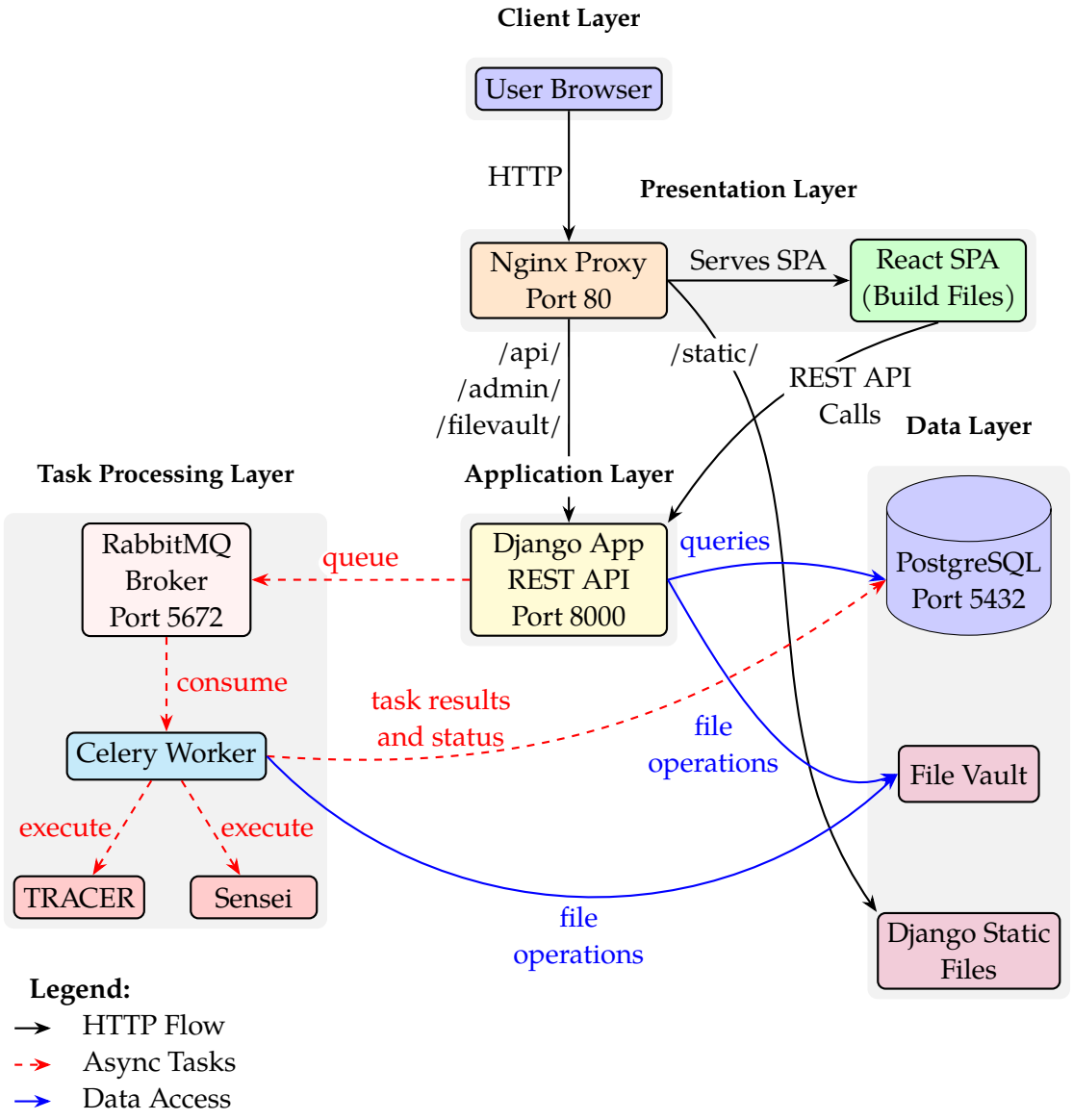

Figure 5.1: Web Application Architecture and Connections.

and also it offers the Django REST Framework [**DjangoRESTFramework**] that will allow us to develop a Representational State Transfer (REST) API that will be consumed by our frontend.

For the frontend we went with React [**React**], a JavaScript library to develop Single-Page Applications (SPAs) that will consume directly from our Django API.

Lastly, to ensure data persistance, we used PostgreSQL [**PostgreSQL2025**], 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.

5.4.3 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 frozen 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 [**Celery**], which lets us handle these jobs asynchronously. Then, to communicate Django with Celery we need a broker, for this we used RabbitMQ [**RabbitMQ**]. 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 then come back to check the progress. It will also help the server to not get saturated 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.

5.4.4 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 [**Nginx**]. Its primary responsibility is to route incoming HTTP requests to the appropriate service based on the URL path. This also solves the issue of accessing each service in a different port, although they are running in a different port we can just access always 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 it is who will process API calls, return the different files and show the admin dashboard, although to show the admin dashboard's CSS we need to collect and serve the Django static files,

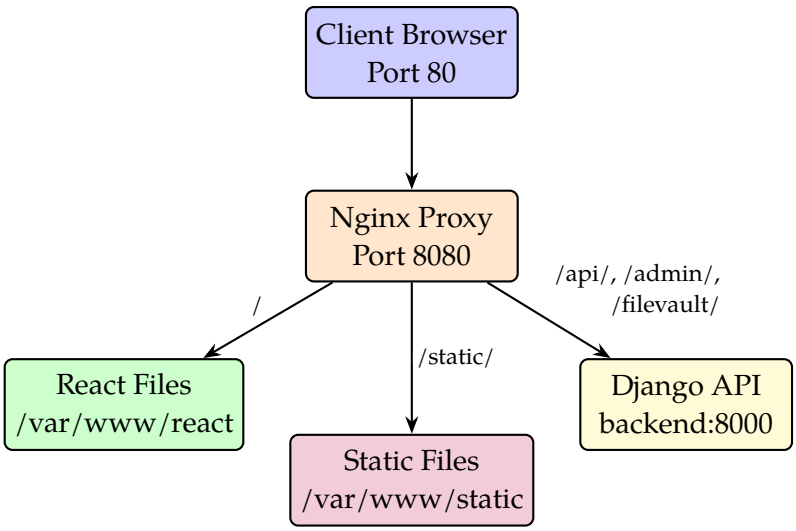

Figure 5.2: Nginx routing to serve React, Django's static files and Django API.

which is why calls made to `/static/` will get redirected to Django's static files. Lastly, the remaining calls will be redirected to React's build files.

5.4.5 Deployment and Containarization

To be able to deploy all of these services we containerized the whole web application using Docker [**merkelDockerLightweightLinux2014**] 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, etc. 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 one that is pointing (has the start of the arrow) is not going to get started.

This ensures that until we don't have the DB and message broker set the queue and backend won't start, and only then the frontend will get build and lastly the Nginx

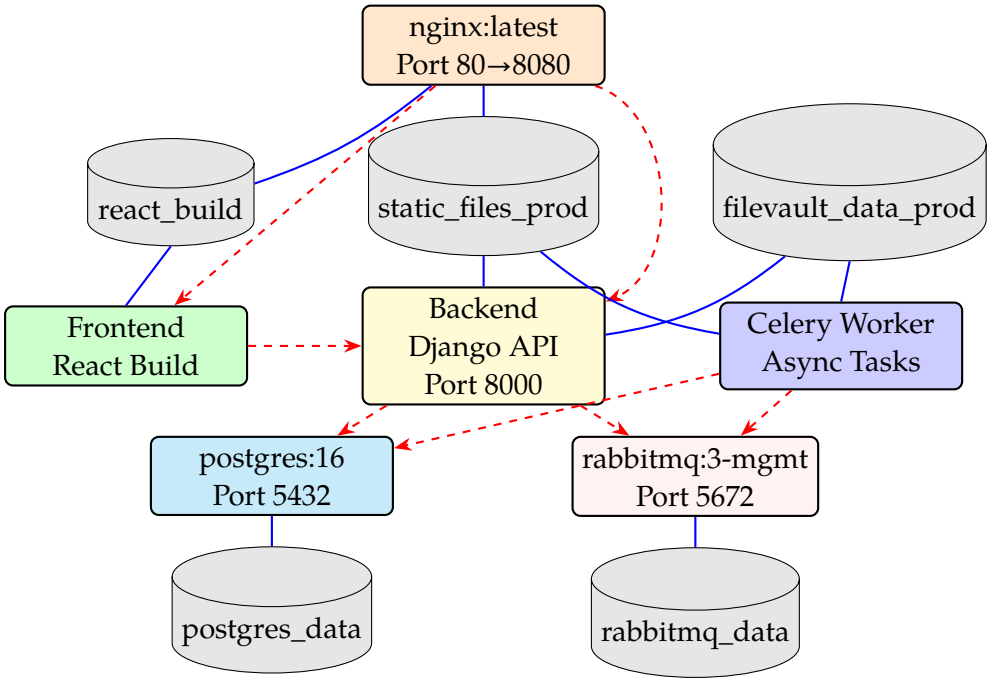

Figure 5.3: Docker Container Architecture.

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).

5.4.6 Security Measures

To ensure the application's security we followed the Django Deployment Checklist [**DeploymentChecklistDjango**]. Related to the authentication we implemented a new user model, and to handle the authentication we used Django-Rest-Knox [**DjangoRestKnox**] tokens. This is a library that implements a improved token system over the default in the Django REST Framework [**DjangoRESTFramework**], the main improvements that it brings are:

- **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 closing session in one device would close it in all the 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 act like he was logged in the account he wanted.
- **Token expire time:** in DRF once you log in with an account that token is valid for ever. Meanwhile, in Knox you can set a expiracy time which will cause that after that time, your session will be closed since the token won't be valid.

To store the API keys uploaded by the user to run TRACER or Sensei we used Fernet Symmetric Encryption [**FernetSymmetricEncryption**] to encrypt them before saving them to the DB. Although the DB has a user and password (not the default ones) we decided this was a good idea to have more layers of security, the same way that Knox encrypts, so that if someone managed to breach the DB those things would still be encrypted.

These things combined with the fact that every time that a petition is made to the API the token is validated, and that the models are configured so that they can only be read by the owner unless it is set to be public (one can set projects as public) but that even if it is public it can only be written by the owner, ensure that the application follows good security practices.

6 Evaluation

7 Conclusions and Future Work

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

BDR Bug Detection Rate

TCR Task Completion Rate

CLI Command Line Interface

API Application Programming Interface

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

List of Figures

3.1 Scheme of our approach and its main components. (1a) Chatbot's functionality explorer. (1b) Synthesiser of test conversation profiles. (2) User simulator. 15
3.2 Flow-chart of TRACER's two phase methodology. The Exploration Phase (left) iteratively discoveres functionalities while the Refinement Phase (right) consolidates and structures them into the final model. 17
3.3 Chatbot model schema. 19
3.4 Workflow model inferred by TRACER from a pizzeria chatbot. 22

5.1 Web Application Architecture and Connections. 37
5.2 Nginx routing to serve React, Django's static files and Django API. 39
5.3 Docker Container Architecture. 40

List of Tables

2.1 Comparison of State-of-the-Art Chatbot Testing Paradigms 13