

Automating the Development of Task-oriented LLM-based Chatbots

Jesús Sánchez Cuadrado
Universidad de Murcia
Murcia, Spain
jesusc@um.es

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

Sara Pérez-Soler
Universidad Autónoma de Madrid
Madrid, Spain
sara.perezs@uam.es

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

ABSTRACT

Task-oriented chatbots are increasingly used to access all sorts of services – like booking a flight, or setting a medical appointment – through natural language conversation. There are many technologies for implementing task-oriented chatbots, including Dialogflow, Watson, and Rasa. They rely on an explicit definition of the user intents, conversation flows, and chatbot outputs, which is costly to specify, and sometimes results in suboptimal user experiences and artificial conversations with limited diversity of chatbot responses.

Recently, the advances in generative artificial intelligence fostered by Large Language Models (LLMs) have enabled a new range of open-domain chatbots, like ChatGPT, able to converse fluently on any topic. However, they are general-purpose, and therefore not directly usable to solve specialised tasks reliably.

In this paper, we study the power of LLMs to build task-oriented chatbots, resulting in lighter specifications – no intent definition required – and more natural conversations than in intent-based approaches. To this end, we propose a lightweight domain-specific language based on YAML to specify chatbots using modules of different types (e.g., menus, question-answering, data gathering). These specifications are compiled into structured LLM prompts that use the ReAct framework to inform our runtime how to interpret the user input and coordinate the tasks that the chatbot must perform. The paper presents the design and realisation of our framework, and an assessment that encodes a set of existing intent-based chatbots using our approach, showing its benefits in terms of specification size, conversation flexibility and output diversity.

CCS CONCEPTS

- **Human-centered computing** → **Natural language interfaces;**
- **Software and its engineering** → **Context specific languages;**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CUI '24, July 8–10, 2024, Luxembourg, Luxembourg

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0511-3/24/07...\$15.00

<https://doi.org/10.1145/3640794.3665538>

KEYWORDS

Task-oriented Chatbots, Large Language Models, Domain-Specific Languages

ACM Reference Format:

Jesús Sánchez Cuadrado, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2024. Automating the Development of Task-oriented LLM-based Chatbots. In *ACM Conversational User Interfaces 2024 (CUI '24), July 8–10, 2024, Luxembourg, Luxembourg*. ACM, New York, NY, USA, Article 1, 11 pages. <https://doi.org/10.1145/3640794.3665538>

1 INTRODUCTION

Task-oriented chatbots are conversational agents designed to perform specific tasks. They are used as front-end for all sorts of services, from customer support to online shopping. They are popular thanks to their seamless integration with social networks (e.g., Telegram, Slack), web sites and intelligent speakers. Moreover, there are many frameworks for their construction, like Google's Dialogflow [10], Amazon Lex [17], IBM Watson Assistant [33] or Rasa [24]. Using these frameworks, developing a chatbot entails the explicit definition of its intents by means of extensive sets of training phrases (in all languages the chatbot should understand), designing all paths a conversation between the chatbot and users may take, and defining the chatbot outputs. This is time-consuming and requires substantial specification effort [27, 29]. Moreover, being ready-made, the chatbot outputs sometimes feel artificial or repetitive.

Recently, the advances in generative artificial intelligence via transformers [32] and Large Language Models (LLMs) [39] have triggered the construction of *open-domain* chatbots, like OpenAI's ChatGPT [19] or Google's Gemini [12]. These chatbots can converse with users on many topics, and despite their so-called *hallucinations* [5], they are heavily used today as assistants in many disciplines like marketing, law, computer science or media content creation [25]. However, they are general-purpose, and therefore not directly usable to solve specialised tasks – like those required from task-oriented chatbots – reliably.

Given the capabilities of LLMs, a natural question is whether they can be used as the underlying technology for task-oriented chatbots. This would make it possible to use the generative nature of LLMs to produce realistic outputs in natural language, and could reduce the chatbot specification effort, avoiding the need to craft large sets of training phrases and complex conversation paths. However, a

direct use of LLMs alone – via monolithic, complex prompts – is not feasible [37]. Although some frameworks to build applications atop LLMs have emerged (e.g., LangChain [14]), they generally require high programming skills and are not specifically designed to build task-oriented chatbots.

To fill this gap, we present a declarative approach to specify task-oriented chatbots that run atop LLMs. Our chatbots are specified via different types of modules (menus, question-answering, actions, data-gathering), from which appropriate prompts are generated, and which are coordinated using a state-machine model. We use ReAct-style prompts [36] to let the LLM inform our runtime how to interpret the user input and coordinate the tasks that the chatbot is expected to perform. Chatbots are specified using a textual domain-specific language with YAML syntax for ease of use. This paper presents the design of the framework, a supporting prototype tool called `TASKYTO`, and an initial evaluation that compares the construction of 4 chatbots using both our approach and intent-based technologies for chatbot development, showing benefits in terms of specification size, conversation flexibility and output diversity.

Paper organisation. Section 2 introduces background and related works on task-oriented chatbots and LLMs. Section 3 motivates our approach by introducing a running example chatbot and some requirements for its construction. Section 4 describes our proposal. Section 5 presents the tooling, and Section 6 an evaluation. Finally, Section 7 concludes and proposes lines for future research.

2 BACKGROUND AND RELATED WORK

This section provides background and reviews related research on techniques for building chatbots. We focus on the two main approaches for building chatbots, based on intents (Section 2.1) and LLMs (Section 2.2).

2.1 Intent-based chatbots

Task-oriented chatbots provide support for specific tasks using Natural Language (NL) conversation. Many technologies – including Dialogflow, Lex, Watson or Rasa (cf. [22] for a survey) – allow their specification by the definition of the user *intents* that the chatbot aims at recognising. Each intent declares *training phrases* exemplifying how users could express the intent. When the user states an utterance, the chatbot matches the most likely intent for the utterance with a certain probability. If no intent matches above a threshold, the agent applies a default *fallback intent* (if defined) to ask the user to rephrase the utterance. Intents may also have *parameters*, which are pieces of information extracted from the user utterances. When the chatbot matches an intent, it performs the *actions* associated to the intent (e.g., accessing an external service, or producing a text response). Finally, full-fledged *conversations* are designed by interleaving expected user intents and agent actions, and may bifurcate depending on the user responses.

Many commercial technologies (e.g., Dialogflow, Watson, Amazon Lex) are cloud-based, low-code platforms that simplify the creation of intent-based chatbots via forms or graphical notations. Instead, other approaches require programming using general-purpose programming languages, and so they have a steeper learning curve. For example, Rasa is a framework to develop chatbots using Python, markdown and YAML files. Midway between both

options, some domain-specific languages (DSLs) aim at reducing the construction effort by providing concepts like intent or conversation as first-class citizens of the language. For instance, CONGA [21], Xatkit [8] and Pandorabots [20] have textual DSLs tailored to define intent-based chatbots. CONGA is a compiled language (i.e., the chatbot definitions are compiled into other technologies such as Dialogflow or Rasa for execution), and Xatkit and Pandorabots are interpreted and provide their own execution engine. While these DSLs are textual, Tilbot [9] provides a visual DSL (a block diagram that can be split in layers) to create conversational interfaces. Tilbot is open-source and targets students and researchers as its main users.

In summary, there are many approaches to building intent-based chatbots for specific tasks, some based on DSLs. However, they still require considerable effort to define enough training phrases for ensuring an accurate recognition of the user intents, as well as to foresee all conversations the chatbot can have. This may result in conversations that are artificial, not flexible, and overly sensitive to small variations in how users express their intents. Next, we show how LLM-based chatbots alleviate some of these problems.

2.2 LLM-based chatbots

LLMs are transformer-based neural networks trained on vast amounts of text data. Many general LLMs have emerged, like those of the GPT family [23], Llama [31], Claude [1], BLOOM [28], and others [39]. Given the generic nature of LLMs [4], chatbots built with them – like ChatGPT or Gemini – are typically general-purpose and open-domain.

To narrow down the output of LLMs towards specific interactions, some researchers propose creating task-oriented chatbots atop LLMs via *prompting* (i.e., by prepending textual instructions and examples of the desired interaction to the LLM input with the aim of improving its output). For example, BotDesigner [37] is a no-code tool to create LLM-based chatbots solely through prompt templates, and observe the effects by conversing with the chatbot. A prompt template includes NL instructions for the chatbot, messages with which the chatbot will start any conversation, and a reminder to be included in the LLM prompt of the next conversation turn. However, crafting effective prompts is challenging, especially for non-AI-experts. The reported challenges [38] include difficulty on choosing and formulating the right instructions to achieve the desired effects, systematic testing, or understanding the effect of prompts. In the health domain, Wei *et al.* [34] explore different prompt styles to build chatbots for self-reporting data atop LLMs. In particular, they analysed slot-filling performance and conversation styles. Even if task-oriented, these chatbots perform simple data-gathering tasks. Hence, more complex, multi-step tasks would benefit from dedicated support.

To address these challenges, the community is demanding development tools (akin to traditional programming tools) that assist prompt programmers. In [11], the authors extract the structure of NL prompts from prompt metadata annotations, which prompt editing environments can exploit. Another attempt is Impromptu [7], a DSL to define platform-agnostic prompts and generate prompt versions for specific platforms. But, perhaps, one of the best-known

tools is LangChain [14]. It is a Python library for LLM-based application development, with facilities to build prompts and agents. However, it requires substantial programming expertise, and the execution model focusses on linear chains of procedures. LangFlow [15] permits defining LangChain flows graphically, but they need to be exported to LangChain for refinement. Finally, PromptSapper [6] permits building LLM-based applications with an execution model similar to LangChain, but uses a graphical block-based language and has a chatbot to assist in their construction. Its focus is also on chains, with no first-class support for typical features of task-oriented chatbots like fallbacks, parameter declaration and passing, or allowed conversation languages. In general, agent-based technology is not adequate for building task-oriented chatbots since agents are intended to solve multi-step tasks autonomously. Instead, task-oriented chatbots require that the user interaction drives the execution of the agent. Thus, the analysed tools either create chatbots that are general-purpose [7, 11], require programming expertise [14, 15], or miss primitives for task-oriented chatbots [6]. Actually, a recent work [2] proposes a new Rasa engine based on LLMs, showing that the construction of task-oriented chatbots with LLMs needs techniques different from the ones currently available.

Another technique for obtaining narrower chatbots is *fine-tuning* on specialised data, like code [35]. This enables repurposing an LLM pretrained on generic data for specific downstream tasks (e.g., question-answering) or domains (e.g., programming), but it is very expensive, since it requires a training phase. Although some techniques permit reducing the cost, like LoRA [13], they require large amounts of (perhaps unavailable) data and the computational cost remains high. A less expensive approach is retrieval-augmented generation (RAG), which has been used to enable chatbots to converse on proprietary data [16]. Still, these chatbots are not task-oriented, and would require coordination to accomplish multi-step tasks.

In conclusion, LLM-based chatbots overcome the artificiality of task-oriented chatbots by enabling realistic conversations, but they are general-purpose. Some techniques allow conversations to be tailored to restricted domains (e.g., fine-tuning, RAG), but to the best of our knowledge, there are no practical tools specific for the declarative creation of task-oriented chatbots using LLMs. In the rest of this paper, we present our proposal that does exactly that.

3 MOTIVATION AND RUNNING EXAMPLE

To motivate our work and illustrate the following sections, let us consider the development of a chatbot for a photography shop specialised in art galleries. The chatbot should help the visitors of the shop website in the following tasks:

- Answer questions about the shop. For instance, it should make clear that the shop does not do wedding photographs.
- Gather contact information for having a personal call before a photo session is scheduled.
- Give price estimates for the most common types of works.

For instance, Figure 1 shows a potential conversation for the running example. The implementation of this chatbot poses several challenges:

- (1) *Ease of implementation.* Many chatbots have a small scope [3], so it is not possible to devote many resources to their development. However, using an intent-based approach requires a








I0		Chatbot: Hello
I1		User: Hi
I2		Chatbot: Welcome to ArtClicks, we are specialised in photographs for art galleries. What can I do for you?
I3		User: I need a photographer for my gallery, for tomorrow
I4		Chatbot: Of course, we'd be happy to assist with your gallery photography. Could you please provide your name and phone number for our records? Also, if you wish, you can provide your email address, although this is optional
I5		User: My name is John Smith, and my phone is 212-998-1212
I6		Chatbot: Thanks John Smith! We'll contact you very soon for a session on 10/02/2024

Figure 1: Potential conversation for the example chatbot.

non-trivial effort to provide training phrases, likely in several languages, which correctly capture the domain vocabulary and trigger the adequate conversation flows.

- (2) *Smart conversation flow.* Users would like to converse with the chatbot in a way that feels natural to them, where the chatbot remembers past interactions. For instance, in interaction *I1*, the user might have said: "Hi, I'm John Smith". In such a case, in interaction *I4*, the chatbot should ask the appointment date but not the user name. In general, users may provide relevant information to the chatbot at different points in time, and the chatbot should be flexible to respond properly in all cases. For instance, the user has not provided the email address in interaction *I5*, but since it is optional, the chatbot does not insist on asking for it. In intent-based chatbots, these conversation possibilities need to be defined in advance.
- (3) *Realistic, engaging chatbot answers.* Users may be more willing to engage with natural-sounding chatbots [18, 26, 30], which may use the (perhaps unsolicited) information that the user provides to converse in a natural way. For example, if the user says "Hi, I'm John Smith" in interaction *I1*, the chatbot may reply "Hi John! Welcome to ArtClicks [...]", using the user name in the response. In the example interaction, in step *I4* the chatbot refers to the gallery the user just mentioned ("Of course, we'd be happy to assist with your gallery photography [...]"). Hence, our chatbot should avoid ready-made responses, since they may become repetitive and lead to user frustration [18, 26].
- (4) *Scoped conversation.* Unlike open-domain chatbots, task-oriented chatbots are designed to solve particular tasks in an effective manner. Therefore, our chatbot must not engage in conversations outside its scope. For example, if the user says "What are the best football players in the world?", the

chatbot should redirect the conversation to the objective tasks, answering “I’m sorry, I can help you setting an appointment for a session, provide price estimates, and answer issues related to our shop. Is there anything related to our services that I can help you with?”.

- (5) *Trustworthiness*. Task-oriented chatbots need to perform the tasks they are designed for in a trustworthy manner. This includes preventing the chatbot from having hallucinations or providing false or made-up data to the user or to other components the chatbot interacts with.
- (6) *Maintainability*. Chatbots are often designed as a monolith [3], which makes their extension and evolution difficult. Instead, we would like our chatbot to be defined with a modular structure to allow the independent development and modification of the modules, making the chatbot easier to maintain and extend.

Table 1 shows the degree in which intent-based technologies and a direct use of raw LLMs meet these challenges.

Requirement	Intent-based	LLM-based (pure)
<i>Ease of implementation</i>	×	~
<i>Smart conversation flow</i>	~	✓
<i>Realistic, engaging chatbot answers</i>	×	✓
<i>Scoped conversation</i>	✓	×
<i>Trustworthiness</i>	✓	×
<i>Maintainability</i>	×	×

Table 1: Coverage of requirements for chatbot construction with current technologies.

First, intent-based chatbots are hard-to-build, as they require an explicit design of intents and conversations, with a provision of a potentially large number of training phrases per intent, in each language supported by the chatbot (e.g., English, Spanish), and with an explicit identification of parameters within each phrase. In contrast, LLMs are conversation-ready and do not need a training phase beforehand. The conversation abilities of LLMs – maybe enhanced by prompts encoding the conversation history – contrast with the need for an explicit conversation design of intent-based approaches. LLMs are open-domain, and hence able to converse on any topic. Instead, intent-based approaches have a scoped conversation by design. The generative nature of LLMs makes them able to emulate realistic, engaging conversations that may use the information provided by the user as context, even if unexpected. Their “creativity” or determinism can be modulated by setting a *temperature* to the LLM. Instead, answers of intent-based chatbots are typically pre-defined. This fact makes them trustworthy, while LLMs have the problem of hallucinations. Finally, even if the design of intent-based chatbots is organised in intents, there is no notion of module [22]. Similarly, a large monolithic prompt to create an LLM chatbot is hard to maintain and extend, and can be problematic to ensure the chatbot’s correct behaviour [37].

In this paper, we propose a new method for building task-oriented chatbots, motivated by the need to leverage the advantages of intent-based and LLM-based approaches. On the one hand, we exploit LLMs’ support for smart conversation flows, realistic and engaging answers, and the avoidance of training phrases and an

explicit design of all possible conversation paths. On the other hand, we adopt intent-based features for scoping the conversation topics and obtaining trustworthiness. Our method promotes modularity to improve the maintainability of chatbots, and reduces the implementation effort through a declarative YAML-based DSL that reduces the need for coding.

4 OVERVIEW OF OUR APPROACH

Figure 2 shows a schema of our approach. First, the developer specifies the chatbot using declarative modules that encapsulate independent parts of the chatbot behaviour, and may refer to other modules. There must be a *top-level* module, which is the entry point to the chatbot conversation. Each module will be *run* using an LLM (e.g., of the GPT family, or others like Llama2). The developer can configure the parameters of the LLM, as well as other general chatbot aspects, like the conversation languages. Then, the defined chatbot is parsed into a state-machine model, where each module becomes a state that defines a structured prompt generated from the module information. These prompts contain, among other things, the purpose of the module, and the format of the chatbot response. An interpreter takes care of executing the state-machine, keeping track of the active state, and storing a working memory with the previous conversation history and the data provided by the user. The latter is employed to provide context information to the LLMs to handle the user requests.

Next, Section 4.1 provides more details on how to specify chatbots, and Section 4.2 explains how these are executed.

4.1 Chatbot specifications

In our approach, a chatbot is specified as a set of modules that can be of five types: menu, action, data gathering, question answering, and sequence. Moreover, as we explain next, some types of modules can refer to other modules in the specification.

- A *menu module* allows defining a set of conversation alternatives (exclusive choice). These can either capture the answer to a specific user question, be a complex conversation flow handled by other module, or be a sequence module (explained later). Menu modules may also specify a *fallback* response, to be used when the chatbot does not know how to handle the user message. Typically, a chatbot will define a menu module as its top-level module.
- An *action module* executes an action upon receiving some input data. Actions are defined by code in some language (currently Python) and produce a response for the user. This response can be displayed verbatim, or be rephrased by an LLM using the conversation context of the caller module. This latter option enables more natural and less repetitive conversations.
- A *data gathering module* is used to request some data from the user, and may then execute some action.
- A *question answering module* declares a set of questions that users may ask, and their answers. Unlike intent-based chatbots, which need to provide many training phrases to recognise the user intents, questions in our approach do not need to be reformulated in several ways, but an LLM is used to

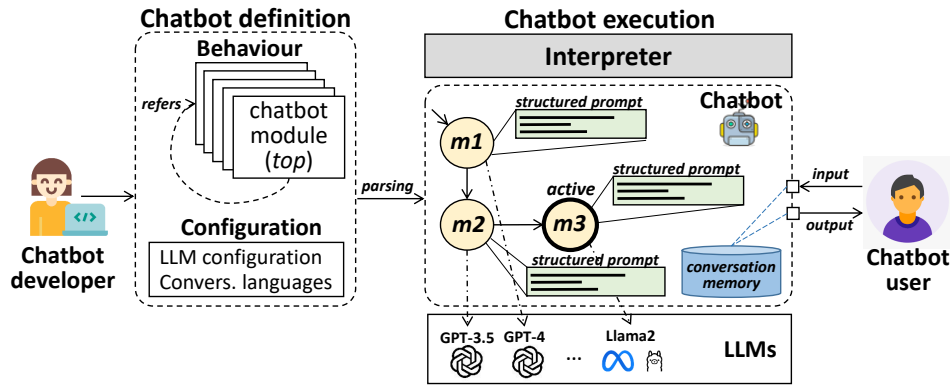


Figure 2: Overview of our approach.

identify which question in the module corresponds to the user utterance.

- A *sequence module* defines a chain of conversation steps, each of them defined in another module.

To simplify the specification, we provide a DSL with YAML syntax to define chatbots. As an example, Listing 1 uses this DSL to define the top-level module of our running example. It is a menu module (as stated in line 2) and contains a presentation section to instruct the LLM on the conversation scope and the assistance to provide (line 3). As we will see in Section 4.2, this will be injected as part of the prompt for the LLM. Then, there is a fallback message (line 4) and a set of conversation possibilities given by other modules: `photo_qa` (lines 7–9, which provides a FAQ on common questions), `call_appointment` (lines 10–12, to make an appointment), a sequence of two modules (lines 13–17, to calculate the estimate price of a work), and a welcome message (lines 18–20, to greet the user). In the menu module, each conversation option has a descriptive *title*, a *kind* (module, sequence, or answer), and a *reference* to the name of another module (or modules in the case of *references*).

Listing 2 shows the `photo_qa` question answering module, which compiles frequently asked questions. Even if the chatbot is to operate on several languages (e.g., English, Spanish), the questions and answers only need to be provided in one language (English in this case).

Module `call_appointment` in Listing 3 is a data-gathering module to request the name, phone number, email and appointment date from the user, as lines 5–11 declare. Our DSL supports both classic computer language data types (date, line 11) and “soft” types (e.g., person name in line 5, or countries). In the latter case, the chatbot will use an LLM to validate whether a user-provided value is appropriate for the type. The DSL has formatters for some data types, which convert the user-supplied string into the specific data type. For example, the field `appointment` should be a date, so if a user states *in two days*, and today is January 22nd 2024, the chatbot will store the value `24/01/2024` in `appointment`. The data are mandatory by default, but can be explicitly declared optional, like `email` in line 10. The chatbot will ask for the value of the mandatory data that the user does not provide, but not for the optional data. Moreover, users can provide all required data in a single phrase, or after multiple interactions. Finally, the on-success section (lines 13–19) is executed

```

1 name: top-level
2 kind: menu
3 presentation: You are a chatbot which helps customers of a photography shop
  specialised in photographs for art galleries. The shop also works on other
  media like video and 3D rendering.
4 fallback: I'm sorry, I can help you setting an appointment for a session, provide
  price estimates, and answer issues related to our shop.
5
6 items:
7   - title: Q&A
8     kind: module
9     reference: photo_qa
10  - title: Sets a follow-up appointment
11    kind: module
12    reference: call_appointment
13  - title: Calculates a price estimate of the work
14    kind: sequence
15    references:
16      - session_details
17      - estimate_price
18  - title: Welcome. To say hello to the customers.
19    kind: answer
20    answer: Welcome to ArtClicks, we are specialised in photographs for art
  galleries. What can I do for you?
  
```

Listing 1: Top menu module of the photography chatbot.

```

1 name: photo_qa
2 kind: question_answering
3 description: Useful for answering general questions about ArtClicks' photo shop.
4 questions:
5   - question: When are you open?
6     answer: We shoot art on location and on our studio at SoHo. We are open
7       Monday to Friday from 9:00am to 17:00pm.
8   - question: Do you charge per hour or by number of photos?
9     answer: We charge 200$ per hour
  
```

Listing 2: Question answering module.

once the chatbot has obtained all mandatory data. This section contains the code to *execute* (lines 14–16) and the chatbot *response* (lines 17–19). The former can be a Python code snippet (as in the listing), or a reference to an external Python file. For simplicity, the illustrated snippet prints a line, but one could provide a webhook to connect with an existing information system. Note that the code and the response can include the value of the gathered data by their name (e.g., `name`, `phone_number`). As for the response, it can be

```

1 name: call_appointment
2 kind: data_gathering
3 description: Useful for asking the contact information, so that the shop can
  contact the client for the photo session.
4 data:
5   - name: Person name
6   - phone_number:
7     type: Phone number
8   - email:
9     type: email
10    required: false
11   - appointment: date
12
13 on-success:
14 execute:
15   language: python
16   code: print(f"Saving appointment for {name} ({phone_number}) at {
  appointment} into data base")
17 response:
18   text: Thanks {name}! We'll contact you very soon for a session on {
  appointment}.
19   rephrase: simple
    
```

Listing 3: Data gathering module to obtain the contact information.

displayed either verbatim (the default) or rephrased (as indicated in line 19). The *rephrase* option allows the chatbot to answer slightly differently each time, so that the conversation does not seem prefabricated. In the example, the response could be rephrased as *Thank you, John! We'll be reaching out to you shortly to schedule a session on January 24th, 2024*. The DSL supports two rephrasing modes: *simple* (as in the example), in which an LLM is asked to rephrase a fixed response, and *in-caller*, in which the LLM automatically rephrases the answer when the module execution ends and the module response is pasted in the caller module history. The latter mode is particularly suitable for adapting a module response to the previous conversation history.

For space reasons, we omit the definition of the other example's modules, which are available at <https://github.com/satori-chatbots/taskyto>.

Finally, a configuration file permits defining the default LLM used by all modules and its temperature, which particular modules can override. It also permits configuring global chatbot options, like the conversation languages. This forces the chatbot to converse only on those languages, informing the user of this fact in the language the user employed. For example, if a chatbot is set to converse only in English and Spanish, and the user says in French: *Pouvons-nous parler en français?*, the chatbot replies *Désolé, je ne parle pas français. Pouvez-vous s'il vous plaît parler en anglais ou en espagnol?*

4.2 Chatbot execution model

The chatbot execution model is based on the idea that, at each conversation step, the chatbot attempts to accomplish some task. For instance, in the top-level module of the running example, the task of the chatbot is to determine the user intention. If the user types *I would like to know the pricing of your services*, then the task switches to gathering the data required to price a service (module *estimate_price*). Thus, executing the chatbot requires the ability to "jump" from one module to another when this other module is best suited to handle the user request.

To achieve this effect, our approach first analyses the chatbot specification to compute a dependency graph, where each node refers to a chatbot module (i.e., to a task), and the edges represent possible transitions between tasks. Figure 3 shows the graph for the running example.

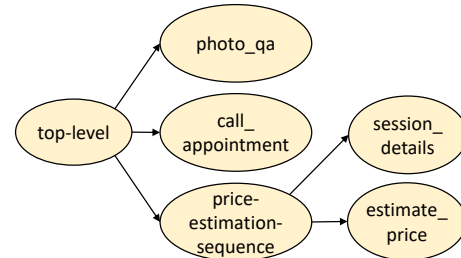


Figure 3: Dependency graph of the photography chatbot.

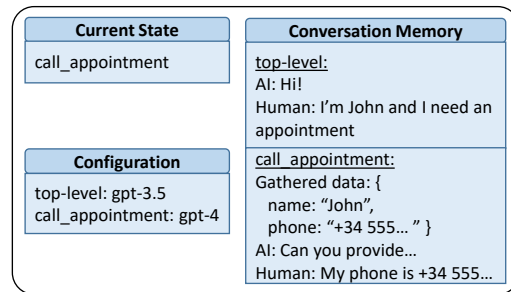


Figure 4: Interpreter structure.

We have implemented an interpreter that coordinates the execution of the tasks (see Figure 4). The interpreter keeps the conversation state, which is given by the current state (i.e., a node of the dependency graph) and the conversation history. Upon receiving an input from the user, the interpreter assembles a prompt tailored to the current state. This prompt uses information from the module specification associated to the current state, and the stored conversation history. The prompt is then sent to an LLM to obtain a response. As an example, Figure 5 shows the prompt compilation for the top-level module, which contains the following eight sections:

- (1) *Presentation*. It includes a high-level description of the chatbot purpose. It is extracted from the top-level module, and applied to all modules to ensure that the chatbot always knows its working context.
- (2) *Tool formatting*. It provides instructions on how to obtain the chatbot response. This depends on whether the LLM has all the information to answer directly, or needs to delegate the task to another module. We use ReAct prompting [36] to know when the conversation should flow to another module, using a similar style to LangChain which employs the term *tool* to refer to modules. In the prompt, the tag *Thought* indicates whether to use a tool or answer directly, *Action* defines the name of the tool to use, and *Action Input* permits passing parameters to the tool. For example, given the user input *I want an appointment for tomorrow*, the control will flow to module *call_appointment* passing the parameter *appointment*

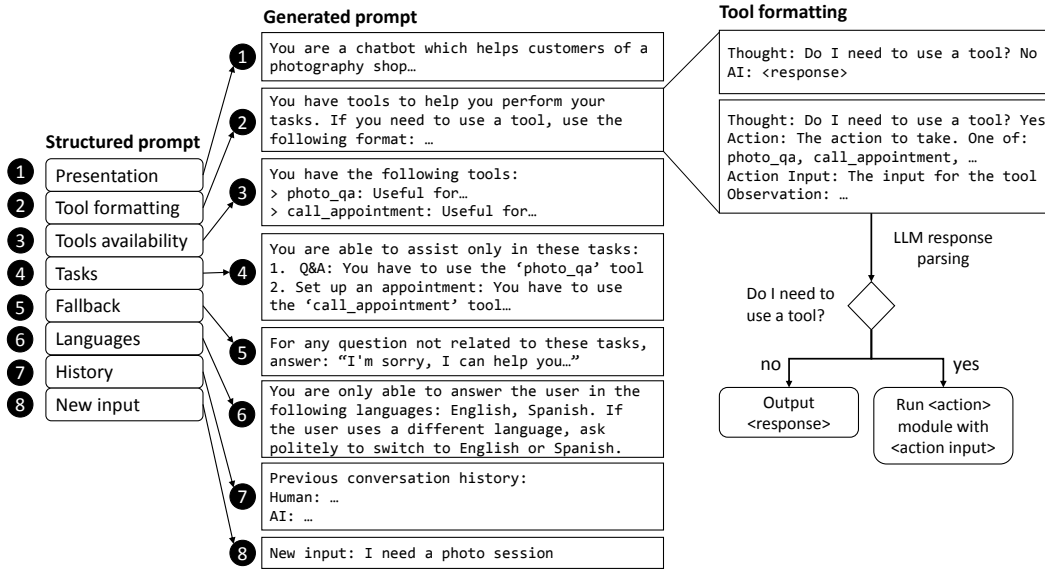


Figure 5: Example of how the prompt for the top-level module is generated from the structured prompt.

- = tomorrow. The interpreter will parse the output of the LLM according to this format to determine the next action (either showing the AI response or delegating to another tool).
- (3) *Tool availability.* It lists the tools that the LLM can use, including their name (so that the LLM can refer to them) and an automatically generated description of their usefulness to perform a certain task. For instance, the description of a question answering module includes the questions (but not the answer), and the description of a data gathering module includes information about the data to be collected. This section and the previous one are only present when the module may not be self-sufficient.
- (4) *Tasks.* It describes the task that this module performs. In Figure 5, the task is to decide which sub-task must be invoked (since it is a menu). Instead, the task of the call_appointment data gathering module is collecting data, and its prompt would be “Your task is collecting the following data from the user: name, phone number [...]”.
- (5) *Fallback.* It is a predefined response to be used when the user input is unrelated to the current task.
- (6) *Languages.* It instructs the LLM to converse only in the specified languages, asking the user to switch to any of them if he/she uses a different one.
- (7) *History.* This section is a placeholder for the interpreter to paste the current conversation of the module.
- (8) *New input.* This is a placeholder for the user input.

As an example of the execution model, when the user is in the top-level menu and says *Hi, my name is John*, the interpreter records this interaction in the history and lets the LLM answer without executing any other module. Then, if the user says *I need a photo session tomorrow*, the LLM detects the need to invoke the call_appointment module to satisfy the user request. Thus, the interpreter moves the chatbot to the call_appointment state. It is worth noting that, since *John* is in the history, this information is passed

to the call_appointment module, in addition to the appointment date, so that the module does not need to ask the user for them.

As a final note, the prompt for the question answering modules contains the whole compiled list of frequently asked questions in the module (cf. Listing 2). For large modules (with many queries or lengthy answers), we plan to make available a RAG architecture. This way, the query list is split and stored in a knowledge base, and a retrieval step is used to obtain the relevant information and inject it into the prompt.

5 TOOL SUPPORT

Our proposal is backed by a tool implemented in Python, called TASKYTO. It has an extensible design that enables the seamless integration of LLMs as plug-ins (e.g., a custom server deploying a Llama2 model). Currently, TASKYTO has direct support for LLMs of the OpenAI’s GPT family using their REST API, but we plan to support other LLMs such as Claude. The tool can be easily used in the command line. In addition, to facilitate the deployment of chatbots, we provide a web server able to keep several independent conversation states simultaneously.

To facilitate the incremental development of chatbots, TASKYTO allows recording user-chatbot interactions as a YAML file, and replaying recorded user utterances. In addition, TASKYTO provides facilities to describe and run test cases. A test case comprises a set of user-chatbot interactions, which may include assertions to check the correctness of the conversation flow, the gathered data, and the chatbot responses. Since chatbots may rephrase their responses, assertions can specify several possible chatbot answers which are checked for an exact match, as well as define a tolerance for approximate string sequence alignment¹.

The source code and usage instructions are available at <https://github.com/satori-chatbots/taskyto>.

¹ <https://github.com/eseraygun/python-alignment>

Name	Rasa						TASKYTO				
	Intents	Training Phrases	Words	Files (YAML/md)	LoC (YAML/md)	LoC (Python)	Modules	Words	Files (YAML)	LoC (YAML)	LoC (Python)
<i>Photography</i>	10	98	1451	6	332	207	5	381	6	140	14
<i>Veterinary</i>	8	94	1179	6	292	148	3	244	4	71	0
<i>Pizza-shop</i>	5	52	627	6	256	277	4	201	5	92	0
<i>Bike-shop</i>	5	70	781	6	212	200	3	208	4	65	0

Table 2: RQ1: Comparing the specification sizes of Rasa and TASKYTO chatbots.

6 EVALUATION

In this section, we evaluate the chatbots built with TASKYTO under three perspectives: their specification size (as a proxy for effort), the flexibility of their conversation flows, and the degree in which they produce realistic, varied responses. These aspects correspond to requirements R1–R3 stated in Section 3. For space constraints, we leave the evaluation of requirements R4 (scoped conversation), R5 (trustworthiness) and R6 (maintainability) for future work.

To frame our evaluation, we compare TASKYTO’s approach with an intent-based technology, such as Rasa. We aim at answering the following research questions (RQs):

RQ1 What is the specification size of TASKYTO chatbots in comparison with intent-based technologies?

RQ2 How flexible are the conversation flows of TASKYTO chatbots in comparison with intent-based technologies?

RQ3 How varied are the answers of TASKYTO chatbots in comparison with intent-based technologies?

In the following, Section 6.1 presents the experiment set-up, Sections 6.2–6.4 report on the results, and Section 6.5 discusses threats to validity. The raw data of the experiments, including the created chatbots and the user-chatbot interactions in test suites, are available at <https://github.com/satori-chatbots/taskyto-evaluation>.

6.1 Experiment set-up

To evaluate our approach, we compare four chatbots built with both TASKYTO and Rasa 1.10 open source². The first chatbot is the running example (*Photography*). The other three were taken from the literature [3, 22], so they were originally implemented in Rasa, and we created TASKYTO versions trying to mimic their behaviour. *Veterinary* is a chatbot for setting appointments and asking the opening hours of a vet clinic [22]. *Pizza-shop* assists in ordering pizzas and selecting their size, toppings and drinks [3]. Finally, *Bike-shop* helps arranging appointments for different types of bike repairs, and answers to common bike maintenance questions [3].

6.2 Experiment results: Specification size (RQ1)

Table 2 shows size metrics of the chatbot specifications in both technologies. Columns 2–7 correspond to Rasa, where chatbots define intents and training phrases. Rasa specifications comprise a markdown file with the definition of the intents and their training phrases; additional YAML and markdown files to configure the NLU component, conversation flows, chatbot answers, entities, execution policies, credentials and endpoints; and Python files to define

² We chose this version because three of the four chatbots were originally developed using this version.

custom actions (e.g., for data validation). Thus, for each Rasa chatbot, the table reports the number of intents, training phrases, words in the training phrases and chatbot answers, YAML/markdown files, YAML/markdown lines of code (LoC), and Python LoC. Typically, Rasa chatbots require 6 YAML/markdown files and one Python file.

Columns 8–12 show similar metrics for TASKYTO. These chatbots are specified via modules, which can embed Python code (or refer to a Python file) to define custom actions. The number of words in this case considers prompts and answers.

We observe that, in all cases, specifications are larger in Rasa than in TASKYTO, which may suggest a greater effort to define Rasa chatbots. In particular, the Rasa specifications have between 3,12 and 4,83 times more words in inputs and outputs, and between 2,37 and 4,11 times more YAML LoC, than the TASKYTO specifications. Similarly, the number of Python LoC is between 0 and 14 in the TASKYTO chatbots, but around a couple of hundreds in the Rasa chatbots. This code was used in Rasa, e.g., to handle parameter gathering and validation, while using TASKYTO, only the *Photography* chatbot needed Python code to estimate the price for a photo session. Moreover, Rasa chatbots sometimes require defining custom entity types – like phone numbers or emails – via regular expressions, which can be cumbersome and error-prone. This is not needed in TASKYTO, as it uses an LLM to validate the values for these data types. Finally, the metrics in Table 2 are for chatbots supporting one language (English). If they had to support several languages (e.g., Spanish, French), then the size of the Rasa specifications would be roughly multiplied by the number of languages, while the size of the TASKYTO specifications would not change.

Answering RQ1: For the chatbots in our evaluation, the specifications are smaller for TASKYTO than for Rasa. The number of words needed to specify the understanding and answering aspects of the TASKYTO chatbots is lower (sometimes by one order of magnitude), and similarly for the LoC of the YAML files. Finally, the TASKYTO chatbots only required Python code to implement some business logic, but not to specify the chatbot behaviour.

6.3 Experiment results: Conversation flexibility (RQ2)

To answer RQ2, we created test suites for each chatbot, which contained user interactions and indications on how the chatbot should react for an engaging experience. We run each test on the Rasa and TASKYTO chatbots, and assessed if the chatbots reacted as expected (column Pass), if the answers or the conversation flow were wrong (column Fail), and if the conversation had some faulty interaction but was overall correct (column Defects). Table 3 summarises the results.

Name	Tests	Rasa			TASKYTO		
		Pass	Fail	Defects	Pass	Fail	Defects
<i>Photography</i>	8	4	1	3	7	0	1
<i>Veterinary</i>	5	2	2	1	4	0	1
<i>Pizza-shop</i>	9	4	4	1	9	0	0
<i>Bike-shop</i>	11	4	6	1	8	2	1

Table 3: RQ2: Comparing conversation flexibility in Rasa and TASKYTO chatbots.

We can see that the Rasa chatbots had more failures (13 vs 2) and defects (6 vs 3). Some were due to selecting the wrong intent, and others can be attributed to a rigid treatment of parameters. For example, in the *Photography* chatbot, if the user’s first phrase is *Hello, I’m John Smith*, the Rasa chatbot triggers the appointment intent (which has the user name as parameter), and asks for the missing mandatory parameters as follows: *Could you share the phone number where we can contact you?*. However, greeting the user by name would be more engaging, as TASKYTO does when replying *Welcome to ArtClicks, we are specialised in photographs for art galleries. What can I do for you, John Smith?*. Later, if the user wants an appointment, the history records the user name so the TASKYTO chatbot does not ask it again.

TASKYTO also handles optional parameters more flexibly. For example, the email is optional when setting an appointment in *Photography*. TASKYTO correctly mentions this to the user. If the user says *Hello, my name is Peter Parker and I need a photo session tomorrow*, the TASKYTO chatbot replies *Hello Peter Parker, I’m glad to hear you’re interested in a photo session. Could you please provide your phone number for us to confirm the appointment? Also, if you wish, you could share your email address with us. This is optional but it would help us to send you updates and special offers*. Instead, the Rasa chatbot replies the same as before (*Could you share the phone number where we can contact you?*), which is the prompt for the only mandatory parameter not entered.

Regarding TASKYTO, it currently lacks data validation for parameters, which caused defects in the tests. For example, the TASKYTO chatbots allow scheduling appointments for past dates. Instead, the Rasa chatbots can include Python code to validate the user data and flag those inputs as invalid. We plan to incorporate data validation in future versions of TASKYTO. Another case of failing tests in TASKYTO occurred when the user asked information defined in a question answering module, while another module was active. This caused troubles in the *Bike-shop* chatbot, e.g., when asking the price of a tire while in the process of setting an appointment.

On the positive side, TASKYTO is able to go back to previous modules in a sequence of correct inputs. For instance, *Pizza-shop* concatenates three data gathering modules asking the pizza size, toppings and drinks in that order. If the user wants to change the pizza size when the drinks module is active, it can do so with a sentence like *I’d like to change the pizza size to a large one*. TASKYTO automatically returns to the pizza size module, stores the new pizza size, and continues the conversation asking for drinks. While this is also possible in Rasa, it requires an explicit coding of the conversation flow. The handling of pizza toppings was also very natural in TASKYTO, but caused three failures in Rasa. In TASKYTO, the user

can request toppings in very flexible ways (e.g., *I’d like all toppings that are vegetarian*), can eliminate or replace toppings (e.g., *Oh sorry, I don’t want olives anymore, but instead ham*), or ask details about toppings (e.g., *Which of the toppings are vegan?*). This level of flexibility would be difficult to achieve in Rasa, as users have many alternatives for expressing a choice, which would need to be anticipated and coded.

Answering RQ2: The conversation flows for handling user-supplied data were generally more flexible and engaging in TASKYTO than in Rasa chatbots. One reason is the availability of a conversation history, which modules can provide to the LLM to reason about. This is not possible in intent-based chatbots, as the selected intent only knows the current sentence and the data captured by other intents so far. We have also noticed some current limitations of our framework regarding data validation and navigation between modules, which we will improve in future work.

6.4 Experiment results: Answer diversity (RQ3)

To answer RQ3, we looked at the variety of chatbot answers in repeated executions of the test suites of Table 3. First, the chatbots of Rasa 1.10 open source only provided the answers defined in their specification. Instead, using the option rephrase of TASKYTO with different temperatures for the LLM, we obtained a variety of well-formed, realistic answers. On one extreme, with temperature 0, the rephrases always yielded the same sentence. For example, the appointment module of *Photography* responded *Your appointment has been successfully set for tomorrow, John Smith. We will contact you on the provided phone number 555-123456 for further details*. On the other extreme, with temperature 1, each execution produced a different answer. For instance, the same appointment module produced phrases like *Your appointment has been set successfully for tomorrow, John Smith. We will contact you on the provided phone number for further details*. Moreover, TASKYTO can use the user-supplied information to construct a realistic answer. For example, in *Pizza-shop*, if the user says *I’m Ann Gonzales, and I want pizza!*, the TASKYTO chatbot may answer *Welcome to my pizza shop, Ann Gonzales. [...]*. Instead, the counterpart Rasa chatbot cannot use the user name in its answers since the chatbot does not require this name (i.e., no intent declares parameters for the user name).

Intent-based chatbots could achieve a similar behaviour by connecting an LLM that rephrases the chatbot answers. To test this possibility, we connected the *Photography* Rasa chatbot with an LLM of the GPT family, by performing calls to OpenAI’s API in the chatbot actions. To obtain realistic answers – similar to our *in-caller* rephrase strategy – we needed to obtain the conversation history, to pass it to the LLM. This entailed some 150 lines of Python code.

Answering RQ3: TASKYTO chatbots can produce realistic and varied answers by using the rephrase option, setting a temperature to modules, and providing an example phrase. Instead, Rasa 1.10 requires declaring all possible chatbot answers in advance. In addition, TASKYTO chatbots can use user-supplied data (not necessarily defined as parameters) as context information to compose the answers, which are then adapted to the user input. Although intent-based chatbots can achieve a similar behaviour by the use of an LLM, this would require coding effort to properly handle the context information for rephrasing.

6.5 Discussion and threats to validity

Regarding internal validity (confidence on the cause-and-effect relationships deduced from the results), for RQ1, we counted the number of words and LoC to measure specification size. This can be seen as a proxy for specification effort, and may suggest that building chatbots with TASKYTO requires less effort than with Rasa. Still, a user study comparing Rasa and TASKYTO development should be conducted to assess effort.

Also regarding internal validity, we (the authors) built all the TASKYTO and one of the Rasa chatbots (the others were taken from the literature), and tested them all, which might have introduced a bias. To mitigate this threat, disjoint sets of developers implemented the chatbots of each technology, before creating their test suites. In addition, we (the authors) assessed the output of the chatbots. While three different authors performed this assessment, in future work, we would like to conduct a user study to evaluate the perceived quality of the conversations for each technology.

We developed the chatbots of both technologies to the best of our abilities, however, we do not claim that Rasa 1.10 chatbots cannot pass all test suites in Table 3. In general, we observed a trade-off between the conversation flexibility of Rasa chatbots and their specification complexity. This is so as adding new conversation paths requires their explicit encoding in the chatbot, making its specification more complex. We assessed this via an additional experiment, where we (after-the-fact) improved the *Photography* Rasa chatbot to pass all failing tests. We had to encode the missed conversation paths, which increased the number of training phrases by a 42%, the number of words by a 24%, the LoC of YAML files by a 12%, and the LoC of Python code by a 46%. Still, this improvement does not guarantee the coverage of other realistic conversations that TASKYTO would support.

With regards to external validity (generalisability of results), our evaluation is based on a comparison of 4 chatbots. Thus, we may increase the confidence in our findings by using a larger sample size (more chatbots), more intent-based technologies, more complex and bigger chatbots, and a wider variety of chatbot tasks. We leave this as future work, along with the evaluation of requirements R4–R6 stated in Section 3. Given the reduced amount of question answering information in the evaluated chatbots, we did not need any special architecture in the Rasa or TASKYTO chatbots. However, for large information-intensive chatbots (e.g., conversational FAQs), a RAG architecture may be appropriate. In the case of TASKYTO, it would be straightforward to implement a RAG as an extension of our question answering modules. Regarding R4 (scoped conversation), our preliminary tests have not uncovered any situation where the chatbot replied out of scope, but a systematic evaluation of this aspect is future work. In this line, we have found TASKYTO to be *trustworthy* (R5) in general, but this fact needs to be stressed with more test cases (e.g., written by external users) and by monitoring the runtime behaviour of a concrete chatbot in a real deployment. The main source of hallucinations that we found when developing the chatbots was related to confusing the intention of the user to drive the conversation to a specific conversation flow (e.g., in the menu module, the chatbot may confuse scheduling a session with computing a price estimate). We tried to adjust the prompts while testing the chatbots by making more explicit the purpose

of each module. However, other means are necessary to enhance trustworthiness. We plan to include in the runtime a *supervisor* which double-checks the responses of the LLM to mitigate possible hallucinations by making sure that the answers are in line with the current conversation.

The evaluated chatbots may not represent industrial use cases which could be larger. However, the modular architecture of TASKYTO allows the construction of chatbots in a scalable manner since each task is encapsulated into an independent module. Another question which impacts the usability of the chatbots is the response time. In the case of TASKYTO, the response time currently depends on the OpenAI API. We have measured the time of the interactions in the evaluation, and the average response time between the user input and the chatbot response is around 2.5 seconds. Interestingly, this is not much slower than the average response time for interactions with Rasa chatbots deployed locally with no GPU support, which is around 2 seconds for interactions that do not involve Python actions, and 4 seconds for those using forms from actions. We plan to conduct a systematic evaluation of scalability and performance in future work.

Finally, we have compared TASKYTO with Rasa, but not with LLM technologies (e.g., LangChain or a direct approach based on prompting) which is also future work.

7 CONCLUSIONS AND FUTURE WORK

This paper has introduced a novel approach to develop LLM-based task-oriented chatbots. The approach is modular and declarative, since chatbots are defined by different types of modules (menu, data gathering, action, Q&A, sequences) using YAML files. We have provided an implementation (TASKYTO), which we have evaluated by comparing chatbots built with both Rasa and our approach, obtaining substantial reduction in specification size and benefits in terms of conversation flexibility and answer variety.

In the future, we would like to strengthen the results of our evaluation by considering more chatbots and a wider diversity of tasks. In addition, we plan to expand our comparison of TASKYTO with intent-based technologies, e.g., in terms of the usability of the frameworks and the resulting chatbots. For this purpose, we plan to organise a user study. We also plan to perform user studies from the perspective of the chatbot developer. We are currently extending TASKYTO with data validation and a more flexible conversation model. For chatbots with large amounts of data that need to be handled by question answering modules, we plan to include a RAG architecture. We are also improving the testing capabilities of TASKYTO with other types of assertions. To help modernising intent-based chatbots, our next step is to provide automated migration from technologies such as Dialogflow and Rasa into our framework. Finally, we aim at building a web-based low-code environment to build, test and deploy TASKYTO chatbots.

Acknowledgments. Work supported by the Spanish Ministry with the following grants: TED2021-129381B-C21 and TED2021-129381B-C22 (MICIU/AEI/10.13039/501100011033 and UE/NextGenerationEU), PID2022-140109NB-I00 (MICIU/AEI/10.13039/501100011033 and FED-ER/UE), PID2021-122270OB-I00 (MICIU/AEI/10.13039/501100011033 and FEDER/UE) and RED2022-134647-T (MICIU/AEI/10.13039/501100011033).

REFERENCES

- [1] Anthropic. last access in 2024. Claude. <https://www.anthropic.com/news/introducing-claude>.
- [2] Tom Bocklisch, Thomas Werkmeister, Daksh Varshneya, and Alan Nichol. 2024. Task-oriented dialogue with in-context learning. *arXiv preprint arXiv:2402.12234* (2024).
- [3] P. C. Cañizares, J. M. López-Morales, S. Pérez-Soler, E. Guerra, and J. de Lara. 2024. Measuring and clustering heterogeneous chatbot designs. *ACM Trans. Softw. Eng. Methodol.* 22, 4 (2024), 90:1–90:43.
- [4] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie. 2024. A survey on evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.* (jan 2024). Just Accepted.
- [5] Y. Chen, Q. Fu, Y. Yuan, Z. Wen, G. Fan, D. Liu, D. Zhang, Z. Li, and Y. Xiao. 2023. Hallucination detection: Robustly discerning reliable answers in large language models. In *CIKM*. ACM, 245–255.
- [6] Y. Cheng, J. Chen, Q. Huang, Z. Xing, X. Xu, and Q. Lu. 2023. Prompt Sapper: A LLM-empowered production tool for building AI chains. *ACM Trans. Softw. Eng. Methodol.* In press (2023), 22 pages.
- [7] R. Clarisó and J. Cabot. 2023. Model-driven prompt engineering. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 47–54.
- [8] G. Daniel and J. Cabot. 2024. Applying model-driven engineering to the domain of chatbots: The Katkit experience. *Sci. Comput. Program.* 232 (2024), 103032.
- [9] J. de Wit and A. Braggaar. 2023. Tilbot: A visual design platform to facilitate open science research into conversational user interfaces. In *5th International Conference on Conversational User Interfaces, CUI*. ACM, 55:1–55:5.
- [10] Dialogflow. last access in 2024. <https://dialogflow.com/>.
- [11] A. J. Fiannaca, C. Kulkarni, C. J. Cai, and M. Terry. 2023. Programming without a programming language: Challenges and opportunities for designing developer tools for prompt programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems, CHI EA*. ACM, 235:1–235:7.
- [12] Google. last access in 2024. Gemini. <https://gemini.google.com/>.
- [13] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. 2022. LoRA: Low-rank adaptation of large language models. In *10th International Conference on Learning Representations, ICLR*. OpenReview.net.
- [14] LangChain. last access in 2024. <https://www.langchain.com/>.
- [15] LangFlow. last access in 2024. <https://www.langflow.org/>.
- [16] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*. 16 pages.
- [17] Lex. last access in 2024. <https://aws.amazon.com/en/lex/>.
- [18] Robert J. Moore and Raphael Arar. 2019. *Conversational UX Design: A Practitioner's Guide to the Natural Conversation Framework*. ACM Books, Vol. 27. ACM.
- [19] OpenAI. last access in 2024. <https://openai.com/>.
- [20] Pandorabots. last access in 2023. <https://home.pandorabots.com/>.
- [21] S. Pérez-Soler, E. Guerra, and J. de Lara. 2020. Model-driven chatbot development. In *ER (LNCS, Vol. 12400)*. Springer, 207–222.
- [22] S. Pérez-Soler, S. Juárez-Puerta, E. Guerra, and J. de Lara. 2021. Choosing a chatbot development tool. *IEEE Softw.* 38, 4 (2021), 94–103.
- [23] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. 2018. Improving language understanding by generative pre-training. See also <https://openai.com/research/language-unsupervised>.
- [24] Rasa. last access in 2024. <https://rasa.com/>.
- [25] P. P. Ray. 2023. ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems* 3 (2023), 121–154.
- [26] R. Ren, J. W. Castro, S. T. Acuña, and J. de Lara. 2019. Evaluation techniques for chatbot usability: A systematic mapping study. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1673–1702.
- [27] E. Ruane, R. Smith, D. Bean, M. Tjalve, and A. Ventresque. 2020. Developing a conversational agent with a globally distributed team: An experience report. In *15th IEEE/ACM International Conference on Global Software Engineering, ICGSE*. ACM, 122–126.
- [28] T. Le Scao et al. 2023. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv:2211.05100* [cs.CL]
- [29] R. M. Schuetzler, G. M. Grimes, J. S. Giboney, and H. K. Rosser. 2021. Deciding whether and how to deploy chatbots. *MIS Q. Executive* 20, 1 (2021), 4.
- [30] M. Skjuve, A. Følstad, and P. B. Brandtzaeg. 2023. The user experience of ChatGPT: Findings from a questionnaire study of early users. In *5th International Conference on Conversational User Interfaces, CUI*. ACM, 2:1–2:10.
- [31] 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. 2023. LLaMA: Open and efficient foundation language models. *arXiv:2302.13971* [cs.CL]
- [32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems* 30. 5998–6008.
- [33] Watson. last access in 2024. <https://www.ibm.com/cloud/watson-assistant/>.
- [34] Jing Wei, Sungdong Kim, Hyunhoon Jung, and Young-Ho Kim. 2024. Leveraging large language models to power chatbots for collecting user self-reported data. In *CSCW*. *arXiv:2301.05843* [cs.HC] See also <https://arxiv.org/abs/2301.05843>.
- [35] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. 2022. A systematic evaluation of large language models of code. In *MAPS@PLDI*. ACM, 1–10.
- [36] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).
- [37] J. D. Zamfirescu-Pereira, H. Wei, A. Xiao, K. Gu, G. Jung, M. G. Lee, B. Hartmann, and Q. Yang. 2023. Herding AI cats: Lessons from designing a chatbot by prompting GPT-3. In *DJS*. ACM, 2206–2220.
- [38] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang. 2023. Why Johnny can't prompt: How non-AI experts try (and fail) to design LLM prompts. In *2023 CHI Conference on Human Factors in Computing Systems, CHI*. ACM, Article 437, 21 pages.
- [39] W. X. Zhao et al. 2023. A survey of large language models. *CoRR* abs/2303.18223 (2023), 124 pages. <https://doi.org/10.48550/ARXIV.2303.18223> *arXiv:2303.18223*