

Enhancing Conversational Troubleshooting with Multi-modality: Design and Implementation

Giulio Antonio Abbo¹[0000–0001–6301–0028], Pietro Crovari¹[0000–0002–6436–4431],
and Franca Garzotto¹[0000–0003–4905–7166]

Department of Electronics, Information, and Bioengineering, Politecnico di Milano,
Milan, Italy

Abstract. Conversational troubleshooting is an increasingly popular technology that consists in utilising dialogue agents to support users of a system through a conversation-based question-answering process, typically through a chatbot. Despite their widespread use, current troubleshooting technologies lack a thorough integration with the applications on which they are overlaid, as they rely only on text to help the users. In this paper, we present TINi, an open-source conversational troubleshooting tool that is multi-modal and multilingual, relies on simple configuration files, and is ready to be deployed in web applications. Users can ask questions to the conversational agent explaining the issue faced; the system analyses it together with the interaction’s context to locate the root problem. Finally, it proposes a solution which engages the user multi-modally: with text in the chat and hints in the graphical interface. A table-based configuration improves system maintainability and enables dialogue designers and field experts to work on the conversation without any coding experience required.

Keywords: Multi-modal · Conversational agent · Troubleshooting

1 Introduction

Assisting users of software systems when they encounter a problem has always been of paramount importance [30]. The diagnostic procedure of individuating the probable causes of an issue and finding an action that could solve it is commonly referred to as *troubleshooting* [18].

In the field of user assistance, troubleshooting has usually taken the form of help desks [8]. At a help desk – physical or virtual – customers of a service can ask for information and guidance. Thanks to this procedure, users do not have to read instruction manuals or lists of frequently asked questions; instead, they are helped by a *system expert* [3].

Cite as: Abbo, G.A., Crovari, P., Garzotto, F. (2023). Enhancing Conversational Troubleshooting with Multi-modality: Design and Implementation. In: Følstad, A., et al. (eds.) Chatbot Research and Design. CONVERSATIONS 2022. Lecture Notes in Computer Science, vol 13815. Springer, Cham. https://doi.org/10.1007/978-3-031-25581-6_7

In recent years conversational agents have been employed to automate troubleshooting services [5, 22] thanks to their availability and low costs associated compared to their human counterparts. These characteristics make chatbot-based assistance optimal for websites, particularly e-commerce applications [20, 31], where 24/7 assistance is considered a key asset for a competitive offer [32].

Today, a wide range of tools for building chatbots is available to automate troubleshooting procedures [14], but they share a fundamental weakness: the definition of the conversation and of the solvable problems are tightly coupled, allowing only a chat-based interaction [5]. Although intuitive in the configuration phase, this choice brings three main unwanted drawbacks.

First, current technologies are not context-aware: they are not designed to complement the information extracted from the conversation with contextual data gathered from the interaction on the graphical interface [21, 25]. Consequently, to have good support, users must describe the details of the problem faced and the context in which the problem is encountered, with increased interaction effort.

Second, chat-based conversational troubleshooting can support users only through textual responses, with the added effort for the users of translating the suggestions received into actions on the interface [10, 39]. A multi-modal response would allow the troubleshooting system to give actionable solutions to users, for example, by highlighting the relevant element on the interface, further improving the quality of the response [9, 11, 27, 35].

Finally, since the conversation data is mixed with the conversational agent’s configuration, it is impossible to edit them separately. A conversation designer or a domain expert can not modify the user experience without minimal programming skills. In the same way, programmers who want to modify the functionalities of the troubleshooting application must have a good understanding of the application domain since they will inevitably intervene in the conversation structure.

To overcome these issues, we propose TINI, a conversational kit for troubleshooting that is natively context-aware and multi-modal. Its table-based configuration allows conversation designers and programmers to work independently on the experience configuration. TINI is an open-source plugin ready to be integrated into a new or existing web application. When TINI is embedded into a website, users can describe the issue faced in a dedicated chat panel in the application interface. TINI analyses users’ messages together with the interaction’s context to get to the root of the problem and proposes a coherent solution multi-modally, describing it in the chat and highlighting the relevant elements on the graphical interface.

In this paper, we improve and extend the work on MCTK published in a poster paper [1]. Having seen the solution’s potential, and thanks to the many comments gathered, we have completely redesigned the system architecture making it a single plug-and-play component – which is lighter, faster, and easier to configure – and introduced multilingual support. On top of that, we formalise the

problem with a mathematical formulation and describe a complete case study to clarify its usage.

2 Related Work

2.1 Dialogue Management in Conversational Agents

Conversational agents are generally structured around a *Dialogue Manager* [33], that handles users' intents and dialogue context to control the agent's subsequent actions, modelling the state of the dialogue, the goals, and the policies [29]. Dialogue management can be performed by following two main kinds of techniques [17]: *handcrafted* and *probabilistic*.

In handcrafted dialogue managers, developers must specify the rules to react to each user's intent. These approaches are realised mainly through slot-filling techniques [15] – where the system collects information from the user's messages and asks back for the missing data – or employing finite-state machines [28, 41] – where the dialogue status is represented as states in a finite-state automaton [43]. Handcrafted dialogue managers offer developers complete control over the interaction without having to encode all the possible interactions by hand.

The probabilistic strategies for dialogue management consist in training an algorithm from large data sets of samples [42]. These dialogue managers require a large base of conversation examples to be trained, but, based on the samples provided, they can respond to unforeseen user messages that were not considered at design time. Some solutions adopt a mixed approach. For example learning from a corpus with an external knowledge base or ontology [2, 26].

In general, many of these solutions are strongly dependent on the conversation that the user will have [4], and make difficult the separation of the conversation from the application context. This is obvious when learning from a corpus, where the samples inevitably mix contents and conversation flow, but this is also true for state machine approaches, because the conversation steps are modelled in a graph and are inextricable from the process on which the conversation is based. Consequently, the behaviour of the software is forcibly linked with the dialogue, making it difficult to support additional features, such as multiple interaction modes and multiple languages.

These approaches are largely uni-modal: the user can interact using only the conversation, be it in written or spoken form, and they require other external components to achieve a multi-modal interaction [24]. Yet, it is proven that multi-modality plays a significant role in the interaction with conversational agents [27]; as reported by Oviatt et al. [35], people tend to prefer to interact multi-modally as the task complexity increases. Multi-modal solutions are being developed to be applied in many fields from theatre [6] to the automotive system [23, 36], in most cases with a custom solution. However, multi-modality in chatbots for user assistance in troubleshooting remains largely unexplored.

2.2 Conversational Troubleshooting

Guided troubleshooting is the process of aiding users in a specific field, assisting in the solution of certain problems; usually, the user asks a question, which is then validated to find the root cause and execute a fixing procedure [18].

Creating programs that autonomously guide users in the troubleshooting process requires extracting the intent and specific information from the question, finding an appropriate response, and performing the required actions. There are many approaches to solving this problem [38], and – as in the case of dialogue managers – they can be divided into two groups: those that require manual rules configuration and those that learn from data.

The rule-based approaches, such as taxonomic case-based reasoning [16] or complex slot-filling techniques [44], permit more customisation of behaviour and policies compared to automated learning, for instance, with the integration of external services to improve performance. However, large sets of rules are difficult to manage, and the scope of these troubleshooting frameworks is usually very narrow: they are typically custom solutions for a specific application in a single domain [37]. Subramaniam et al. [37] try to overcome this limitation with a multi-bot system to cover multiple application domains at the cost of a more complex configuration.

The automated approach avoids manually setting up a complex knowledge base by employing automated learning techniques [40]. The problem however is that domain-specific training data sets are difficult to produce, and the quality and consistency of the agent’s answers can vary considerably [13].

3 Preliminary Analysis

TINI is a framework to provide users with multi-modal conversational troubleshooting on web applications. Embedding it into a new or existing website allows users to receive suggestions on how to solve the issues they encounter.

3.1 Design Principles

Before proceeding with the description of the system, we present the principles we applied to the design – namely multi-modality, separation of concerns, and extensibility – illustrating the rationale guiding their consideration and how they are pursued in TINI.

Multi-modality A system is multi-modal if it supports more than one means of interaction: text, graphics, gestures, and more [34]. Multiple research studies prove that the response to multi-modal stimuli is better compared to uni-modal [27, 35]. In addition, since the primal interaction ground for a web application is the graphical interface, we want to intervene and guide the user in that environment as much as possible, instead of asking the user to switch to a completely separate conversational one [7].

A thorough integration between multiple modalities also supports the interpretation of users' requests. Indeed, the same way a human field expert asks for details about the state of the system to answer a question, an automated solution should leverage the context of the request to have a better understanding of it. Additional information such as past user's actions on the graphical interface could help provide a solution tailored for each specific instance.

Contrarily to most used chatbot frameworks, TINI will natively support multi-modality. Indeed, the configuration will be based on a Configuration Table that combines the context – here defined as what is currently shown on the screen – with the problem-solution pairs, and allows including in the response the interface elements relevant to the solution, together with the answer. In this way, different views might produce different suggestions tailored to what is happening on the screen.

Separation of Concerns In software engineering, separation of concerns is the practice of separating the main algorithm from the parts that pertain to specific tasks [19]. We can extend this concept beyond software modules and apply it to the professional figures involved in the different aspects of a system.

We want to empower the conversation designer and the domain expert to customise the behaviour of TINI independently of the developer. To achieve so, we locate the domain-specific data in a single place, on which these experts can directly operate, modifying the conversation as desired. In addition, the structure containing this data must be accessible to those with little programming knowledge.

By operating on the Configuration Table – using any CSV reader – domain experts can insert new problem types, modify the wording of non-effective utterances, or modifying the solution for an existing issue.

Extensibility An extensible solution allows adding new behaviours and modifying the existing ones with minimal intervention in its internal structure. Indeed, if the field-specific data is embedded within the application, as the knowledge base grows, introducing changes will require interventions in multiple sites, reevaluation of all the existing rules to understand whether each of them is affected by the changes, and the extension of the features quickly becomes unfeasible.

The table-based configuration fulfils this requirement: all the field-specific knowledge and the data related are detached from the code that implements the logic.

3.2 Usage Scenario

To explain how the proposed system works, we exemplify its usage in a concrete example. Andy is a user of a web application developed by Barb. Barb's app is a simple online image editor with many tools and controls organised in panels. In her system, Barb uses TINI to add a troubleshooting agent to support her

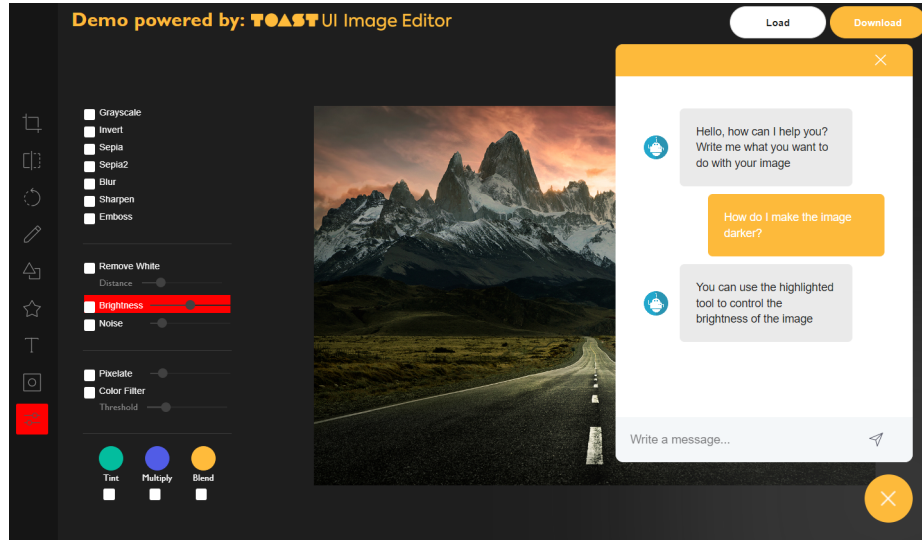


Fig. 1. A photo editing application, enhanced with the TINI tool. On the right, the chat panel displays the conversation with the chatbot; on the left, the available tools are displayed in a panel, with those relevant to the solution highlighted in red. The developer can customise every aspect of the interface.

users. When Andy needs help, he can click on a button added by TINI to start chatting with the conversational agent.

Andy visits the web app and loads the image he wants to edit, then proceeds to use the available tools to crop and rotate the image. At this point, Andy would like to reduce the image’s brightness but does not know how. He clicks on the TINI button and writes in the chat: “How do I make the image darker?”; the system elaborates the message with the available information about the active modules and answers: “You can use the highlighted tool to control the brightness of the image”. At the same time, the system highlights in the interface the relevant tool and the panel in which it can be found, as shown in Figure 1. Andy reads the answer, finds the correct tool, and can continue editing the image.

3.3 Problem Formalisation

From a high-level abstraction standpoint, the proposed system has to take as input the question provided by the user and information about the context, consult a knowledge base, and provide an answer indicating the relevant elements that must be notified to the user.

Modules and Parameters Definition In order to represent the context, we can consider the options available to the user and group them into *modules*.

The modules can be active or not, depending on whether they are available to the user. For example, in a photo editing application, the modules could be various tools, such as the brush, crop, or stamp tool. In addition, each module can have one or more *parameters*, representing the settings of its module. In the image editor, these could be the size and transparency levels of the brush tool and the proportions of the area in the crop tool. The system will then receive the information about which modules are active, as inactive modules are not available to the user and cannot be used to solve the issue the user is facing.

Mathematical Formulation In mathematical terms, the problem can be formalised as follows. Let M be the set of all the modules in the system and P the set of parameters; we suppose that each parameter is associated with one and only one module. Let q be the question of the user. Given A , a set of active modules, $A \subseteq M$, we describe the system as the relation $\sigma(A, q) = \langle \overline{M}, \overline{P}, \mathbf{r} \rangle$ that, given in input the question and the active modules, returns the sets of highlighted modules $\overline{M} \subseteq A$ and parameters $\overline{P} \subseteq P$, and the response $\mathbf{r}$ explaining the actions to perform. A description of the execution will be given in Section 4.3.

4 System’s Overview

The central component of TINI is the *Configuration Table*, a data structure representing connections between problems and solutions in a specific field of application whose aspect is presented in Section 4.1. We can describe the action of TINI in four steps, as shown in Figure 2. The system receives an issue from the interface, described by the user in the chat, for example: “How do I make the image darker?” with the active modules on the screen; in the example, these are the active tools. The sentence is used to identify the problem faced, through intent extraction. This is used in the Configuration Table to retrieve the list of possible solutions for the active modules. These solutions are finally communicated to the users through a sentence in the chat that guides them through the resolution, and some visual hints, such as the tools to use and their parameters to edit, that suggest to users where to operate on the interface.

4.1 Configuration

Before presenting in detail the system’s structure, we propose an overview of the data models and configuration files. TINI can be configured using three files: the *training*, the *configuration table*, and the *utterances* files.

training.json This file provides examples of user sentences for each type of problem; it is used to train an NLU engine to extract the intent – representing the problem type – from the users’ questions. The format follows the specifications

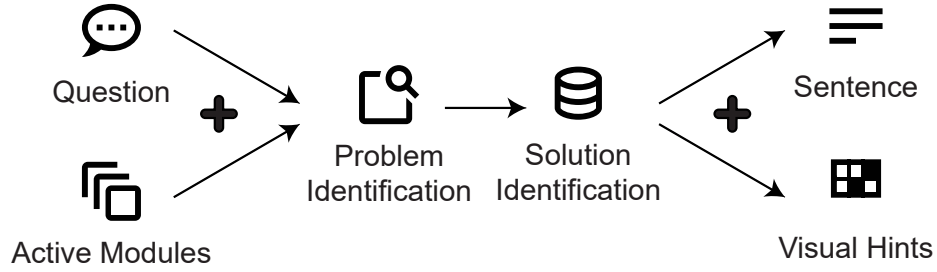


Fig. 2. Sequence of operations performed by the system.

of NLP.js¹ as displayed below. Multiple languages can be supported by providing multiple files with the language locale name: for example, training-en, training-it.

```

{
  "name": "training",
  "locale": "en-US",
  "data": [
    {
      "intent": "alignment",
      "utterances": [
        "The image is tilted",
        "The picture is not aligned",
        ...
      ]
    },
    ...
  ]
}

```

configuration-table.csv This table, as reported in Table 1, contains the connections between problem types, modules, parameters, and answers. This structure balances maintainability, relevant when reading and updating the table, with extensibility, which is important when expanding the table with new problem types and solutions.

utterances.json A file mapping the answers' identifiers with their actual text. It is a simple JSON file with the structure reported below. As before, multiple files can be included with the locale name to support multiple languages: for example, utterances-en, utterances-it.

¹ www.github.com/axa-group/nlp.js

Table 1. Structure of the Configuration Table. Each row corresponds to a module parameter, and every column is a problem TINI wants to address. The presence of an utterance identifier in a cell means that the corresponding parameter is a possible solution to the problem in the column. The conversation designer can directly operate on this table to add problems and utterances and create new connections between problems and parameters.

module	parameter	composition	style	focus
crop	tool	comp.crop		
filter	grayscale		style.generic	
filter	sepia		style.generic	
filter	vintage		style.generic	
filter	blur			focus.blur
filter	sharpen			focus.sharp
filter	pixelate		style.generic	focus.pixel

```
{
  "align.rotate": "To correct the picture's alignment ...",
  "focus.blur": "If you are looking for a more soft look, ...",
  "focus.sharp": "To increase the sharpness of the image ...",
  ...
}
```

The proposed system has the necessary extensibility requisites. Adding a module or a parameter is as easy as adding a row to the table, adding a new problem type can be done by adding a column. Adding or changing a connection between problem types and parameters is done by modifying an utterance identifier in the corresponding cell, and the use of identifiers, instead of putting the utterances themselves in the cell, allows reusing the same utterance multiple times without duplication and support multiple languages; these operations do not require interventions on the code, only on the data structure and on the NLU configuration, fulfilling the separation of concerns.

These three files are provided during the initialisation of the frontend component. In this phase, the developer can also customise the interface's appearance and provide the active modules, as described in Section 4.4.

4.2 Architecture

The system is structured in five modules, as presented in Figure 3. It consists of and can be used as a self-contained Vue.js² library, which is available as open-source.

Interface Component This component is available to the developer and can be embedded in the web application. It displays a circular overlay button on the

² www.vuejs.org

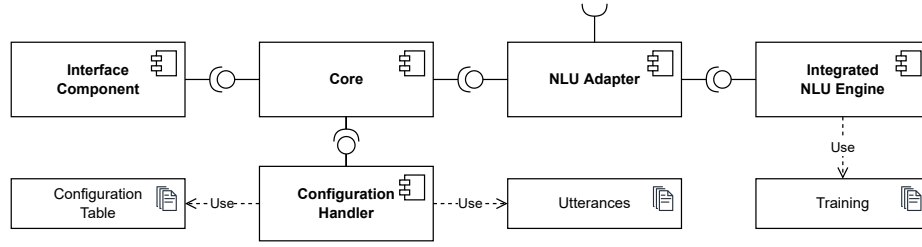


Fig. 3. The components of the system and the configuration files.

bottom right of the screen. The button allows toggling the visibility of a chat panel, which handles the communication with the conversational agent. When the user sends a message, the interface will forward it to the *Core* component, and when it receives a response, it will display it in the chat; in addition, it will update the information on which modules and parameters should be highlighted in the application. The component is implemented in Vue.js; when used, it accepts several configuration parameters, including the configuration files' URLs and the list of active modules, and fires an event when the highlighted elements change.

Core The *Core* implements the logic of the system. It is initialised with information about which modules are currently active. When it receives the user's question, it uses the *NLU Adapter* and the *Configuration Handler* to obtain the response, consisting of an answer and a list of elements to highlight in the interface. The details about the algorithm are reported in Section 4.3. The response is returned to the *Interface Component*. This part is implemented in JavaScript and is independent of the interface: in the future, other interfaces can be added, extending the library to other frontend frameworks.

NLU Adapter and Integrated NLU Engine The *NLU Adapter* is in charge of extracting the intent, which represents the problem encountered by the user, from the user's question. To achieve this, it employs an *Integrated NLU Engine*, which is a NLP.js instance, a JavaScript library implementing machine learning models for natural language understanding. We chose NLP.js because it can run in the browser and natively supports 41 languages, 104 with the BERT [12] extension. When the web app is loaded in the client browser, it is trained with the examples in the *Training* file and is ready to identify the intent from the user sentences. In addition, the *NLU Adapter* detects the language of the user's question.

Alternatively, the adapter can be configured with an external, pre-trained NLU service: in this second case, the *Training* configuration file can be omitted.

Configuration Handler Finally, the *Configuration Handler* retrieves and parses the *utterances* and *configuration table* documents and provides their contents to the *Core* component. The documents can be provided through a URL or as strings directly.

4.3 Runtime Behaviour

The behaviour of the *Core* component can be portrayed using the mathematical notation introduced in the previous section. Given the users' question q , the NLU Adapter extracts the intent, representing the problem type t . For example, from the sentence "How do I make the image darker?" the NLU Adapter would extract the intent representing the problem type *image.dark*. The system can extract only one intent from the given question q . For this reason, if the conversation designers want to accept composite questions (e.g. "I want to cut and make the image bigger") they have to create specific intents.

Let A be the set of active modules as before. The algorithm scans the Configuration Table – where the rows contain the parameters p_i (with the corresponding module m_i), the columns contain the problem types t_j , and the corresponding cell is denoted with u_{ij} , representing an utterance identifier – and finds all the $\langle m_i, p_i, u_{ij} \rangle$ that satisfy $t_j = t$ and $m_i \in A$; in other words, it finds all the parameters of the active modules connected with the problem and the corresponding sentences. The language of the input (e.g., English) is recognised in the user sentence and used to retrieve the actual utterances constituting the answer from the *utterances* document. These are then concatenated to produce a single final response $\mathbf{r}$.

In practice, the sequence of interactions of the user with the system can be modelled as in Figure 4, which represents its behaviour from the moment the application is initialised. During the initial phase, the *Integrated NLU Engine* – if used – is quickly trained. The user can interact with the system and ask a question. The system interprets the question, keeps track of the active modules, and produces a response. When the operation is completed, an event is fired to inform the application that it is necessary to update the highlighted elements.

4.4 Deployment

To use TINI, it is sufficient to import the package, which is publicly available, into a new or existing Vue application. When adding the provided component to the application, the developer will be able to specify the configuration files' URLs, provide a list of active modules – to be kept updated during the execution – and use the provided props to customise the behaviour of the system. Specific CSS classes can be used to refine the graphical appearance. To use the multi-modal response, the developer will listen to a specific event that provides the list of modules relevant to the answer: this can be used to update some CSS classes of other components to highlight them.

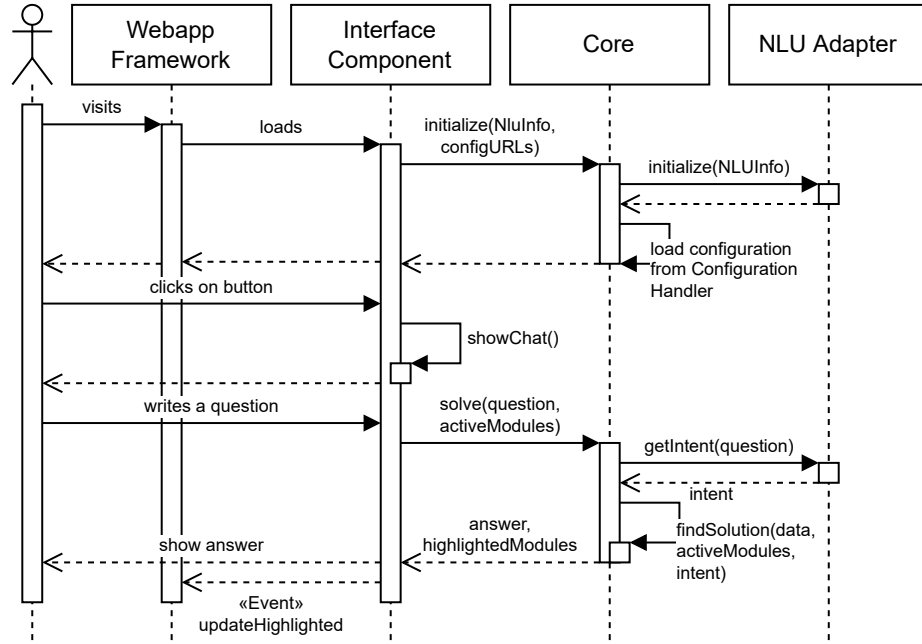


Fig. 4. The system’s behaviour; the Configuration Handler and the language detection are not shown for simplicity.

5 Conclusion

We propose TINI, a novel open-source plugin for conversational troubleshooting chatbots. The system is multi-modal: it suggests the elements of the interface that are relevant to the solution of the issue together with a textual answer. In addition, it is context aware: when responding, it takes into consideration the context of the application together with the question of the user.

TINI builds on the results of previous research and improves its ease of use, as it is now a self-contained frontend plugin instead of requiring multiple external services while maintaining the simplicity of its configuration and extensibility. The separation of concerns ensures that a person without programming experience can autonomously change the configuration. Finally, the tool now supports multiple languages.

In the future, we want to extend TINI’s capabilities in two directions: including multi-step interactions to guide the user when the issue is not immediately clear and computing the solution considering not only which modules are active but also the values of the available parameters. In addition, the plugin can be ported to other frontend frameworks to make it more available. Finally, we plan to evaluate different alternatives to the Configuration Table, including graph-based approaches and a custom interface to edit the configuration.

Bibliography

- [1] Abbo, G.A., Crovari, P., Pidò, S., Pinoli, P., Garzotto, F.: MCTK: A Multi-modal Conversational Troubleshooting Kit for supporting users in web applications. In: *Proceedings of the 2022 International Conference on Advanced Visual Interfaces*. pp. 1–3. AVI 2022, Association for Computing Machinery, New York, NY, USA (Jun 2022)
- [2] Agarwal, S., Dusek, O., Konstas, I., Rieser, V.: A knowledge-grounded multimodal search-based conversational agent. *arXiv:1810.11954 [cs]* (Oct 2018)
- [3] Al-Hawari, F., Barham, H.: A machine learning based help desk system for IT service management. *Journal of King Saud University - Computer and Information Sciences* 33(6), 702–718 (Jul 2021)
- [4] Baez, M., Daniel, F., Casati, F.: Conversational web interaction: Proposal of a dialog-based natural language interaction paradigm for the web. In: Følstad, A., Araujo, T., Papadopoulos, S., Law, E.L.C., Granmo, O.C., Luger, E., Brandtzaeg, P.B. (eds.) *Chatbot Research and Design*. pp. 94–110. *Lecture Notes in Computer Science*, Springer International Publishing, Cham (2020)
- [5] Bavaresco, R., Silveira, D., Reis, E., Barbosa, J., Righi, R., Costa, C., Antunes, R., Gomes, M., Gatti, C., Vanzin, M., Junior, S.C., Silva, E., Moreira, C.: Conversational agents in business: A systematic literature review and future research directions. *Computer Science Review* 36, 100239 (May 2020)
- [6] Bhushan, R., Kulkarni, K., Pandey, V.K., Rawls, C., Mechtley, B., Jayasuriya, S., Ziegler, C.: ODO: Design of multimodal chatbot for an experiential media system. *Multimodal Technologies and Interaction* 4(4), 68 (Sep 2020)
- [7] Boshwabadi, A.M., Biria, R.: The efficacy of multimodal vs. print-based texts for teaching reading comprehension skills to iranian high school third graders. *International Journal of Language Learning and Applied Linguistics World* 5, 17 (Jan 2014)
- [8] Corea, C., Delfmann, P., Nagel, S.: Towards Intelligent Chatbots for Customer Care - Practice-Based Requirements for a Research Agenda (Jan 2020)
- [9] Crovari, P., Pidó, S., Garzotto, F., Ceri, S.: Show, Don't Tell. Reflections on the Design of Multi-modal Conversational Interfaces. In: Følstad, A., Araujo, T., Papadopoulos, S., Law, E.L.C., Luger, E., Goodwin, M., Brandtzaeg, P.B. (eds.) *Chatbot Research and Design*. pp. 64–77. *Lecture Notes in Computer Science*, Springer International Publishing, Cham (2021)
- [10] Crovari, P., Pidò, S., Pinoli, P., Bernasconi, A., Canakoglu, A., Garzotto, F., Ceri, S.: GeCoAgent: A conversational agent for empowering genomic data extraction and analysis. *ACM Transactions on Computing for Healthcare* 3(1), 1–29 (2021)
- [11] De Cicco, R., Iacobucci, S., Aquino, A., Romana Alparone, F., Palumbo, R.: Understanding Users' Acceptance of Chatbots: An Extended TAM Ap-

- proach. In: Følstad, A., Araujo, T., Papadopoulos, S., Law, E.L.C., Luger, E., Goodwin, M., Brandtzaeg, P.B. (eds.) *Chatbot Research and Design*. pp. 3–22. Lecture Notes in Computer Science, Springer International Publishing, Cham (2022)
- [12] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (May 2019)
 - [13] Følstad, A., Nordheim, C.B., Bjørkli, C.A.: What Makes Users Trust a Chatbot for Customer Service? An Exploratory Interview Study. In: Bodrunova, S.S. (ed.) *Internet Science*. pp. 194–208. Lecture Notes in Computer Science, Springer International Publishing, Cham (2018)
 - [14] Følstad, A., Taylor, C.: Conversational Repair in Chatbots for Customer Service: The Effect of Expressing Uncertainty and Suggesting Alternatives. In: Følstad, A., Araujo, T., Papadopoulos, S., Law, E.L.C., Granmo, O.C., Luger, E., Brandtzaeg, P.B. (eds.) *Chatbot Research and Design*. pp. 201–214. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020)
 - [15] Goddeau, D., Meng, H., Polifroni, J., Seneff, S., Busayapongchai, S.: A form-based dialogue manager for spoken language applications. In: *Proceeding of Fourth International Conference on Spoken Language Processing. ICSLP '96*. vol. 2, pp. 701–704 vol.2 (Oct 1996)
 - [16] Gupta, K.M.: Taxonomic Conversational Case-Based Reasoning. In: Aha, D.W., Watson, I. (eds.) *Case-Based Reasoning Research and Development*. pp. 219–233. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2001)
 - [17] Harms, J.G., Kucherbaev, P., Bozzon, A., Houben, G.J.: Approaches for dialog management in conversational agents. *IEEE Internet Computing* 23(2), 13–22 (Mar 2019)
 - [18] Heckerman, D., Breese, J.S., Rommelse, K.: Decision-theoretic troubleshooting. *Communications of the ACM* 38(3), 49–57 (Mar 1995)
 - [19] Hürsch, W.L., Lopes, C.V.: Separation of Concerns. Tech. rep. (1995)
 - [20] Janssen, A., Rodríguez Cardona, D., Breitner, M.H.: More than FAQ! Chatbot Taxonomy for Business-to-Business Customer Services. In: Følstad, A., Araujo, T., Papadopoulos, S., Law, E.L.C., Luger, E., Goodwin, M., Brandtzaeg, P.B. (eds.) *Chatbot Research and Design*. pp. 175–189. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021)
 - [21] John, R.J.L., Potti, N., Patel, J.M.: Ava: From data to insights through conversation. *CIDR* p. 10 (2017)
 - [22] Kvale, K., Freddi, E., Hodnebrog, S., Sell, O.A., Følstad, A.: Understanding the User Experience of Customer Service Chatbots: What Can We Learn from Customer Satisfaction Surveys? In: Følstad, A., Araujo, T., Papadopoulos, S., Law, E.L.C., Luger, E., Goodwin, M., Brandtzaeg, P.B. (eds.) *Chatbot Research and Design*. pp. 205–218. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021)

- [23] Lemon, O., Georgila, K., Henderson, J., Stuttle, M.: An ISU dialogue system exhibiting reinforcement learning of dialogue policies: Generic slot-filling in the TALK in-car system. *Demonstrations* p. 4 (2006)
- [24] Liao, L., Ma, Y., He, X., Hong, R., Chua, T.S.: Knowledge-aware Multi-modal Dialogue Systems. In: *Proceedings of the 26th ACM International Conference on Multimedia*. pp. 801–809. MM '18, Association for Computing Machinery, New York, NY, USA (Oct 2018)
- [25] Liu, W., Li, X., Huang, D.: A survey on context awareness. In: *2011 International Conference on Computer Science and Service System (CSSS)*. pp. 144–147 (Jun 2011)
- [26] Madotto, A., Wu, C.S., Fung, P.: Mem2Seq: Effectively incorporating knowledge bases into end-to-end task-oriented dialog systems. *arXiv:1804.08217 [cs]* (May 2018)
- [27] Massaro, D.W.: A framework for evaluating multimodal integration by humans and a role for embodied conversational agents. In: *Proceedings of the 6th International Conference on Multimodal Interfaces - ICMI '04*. p. 24. ACM Press, State College, PA, USA (2004)
- [28] Mathur, V., Singh, A.: The rapidly changing landscape of conversational agents. *arXiv:1803.08419 [cs]* (Mar 2018)
- [29] McTear, M., Callejas, Z., Griol, D.: *The Conversational Interface*. Springer International Publishing, Cham (2016)
- [30] Morana, S., Pfeiffer, J., Adam, M.T.P.: User Assistance for Intelligent Systems. *Business & Information Systems Engineering* 62(3), 189–192 (Jun 2020)
- [31] Moriuchi, E., Landers, V.M., Colton, D., Hair, N.: Engagement with chatbots versus augmented reality interactive technology in e-commerce. *Journal of Strategic Marketing* 29(5), 375–389 (Jul 2021)
- [32] Nursetyo, A., Setiadi, D.R.I.M., Subhiyakto, E.R.: Smart Chatbot System for E-Commerce Assistance based on AIML. In: *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. pp. 641–645 (Nov 2018)
- [33] O'Shea, J., Bandar, Z., Crockett, K.: Systems engineering and conversational agents. In: Kacprzyk, J., Jain, L.C., Tolk, A., Jain, L.C. (eds.) *Intelligence-Based Systems Engineering*, vol. 10, pp. 201–232. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
- [34] Oviatt, S.: Multimodal Interfaces. In: *The Human-Computer Interaction Handbook*. CRC Press, second edn. (2007)
- [35] Oviatt, S., Coulston, R., Lunsford, R.: When do we interact multimodally? Cognitive load and multimodal communication patterns. *Proceedings of the 6th international conference on Multimodal interfaces* p. 8 (2004)
- [36] Pieraccini, R., Dayanidhi, K., Bloom, J., Dahan, J.G., Phillips, M., Goodman, B.R., Prasad, K.V.: Multimodal conversational systems for automobiles. *Communications of the ACM* 47(1), 47–49 (Jan 2004)
- [37] Subramaniam, S., Aggarwal, P., Dasgupta, G.B., Paradkar, A.: COBOTS - A Cognitive Multi-Bot Conversational Framework for Technical Support. In: *Proceedings of the 17th International Conference on Autonomous Agents*

- and MultiAgent Systems. pp. 597–604. AAMAS '18, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (Jul 2018)
- [38] Thorne, C.: Chatbots for troubleshooting: A survey. *Language and Linguistics Compass* 11(10), e12253 (2017)
 - [39] Turk, M.: Multimodal interaction: A review. *Pattern Recognition Letters* 36, 189–195 (Jan 2014)
 - [40] Vinyals, O., Le, Q.: A Neural Conversational Model (Jul 2015)
 - [41] Wilks, Y., Catizone, R., Worgan, S., Turunen, M.: Some background on dialogue management and conversational speech for dialogue systems. *Computer Speech & Language* 25(2), 128–139 (Apr 2011)
 - [42] Williams, J.D., Asadi, K., Zweig, G.: Hybrid Code Networks: Practical and efficient end-to-end dialog control with supervised and reinforcement learning (Apr 2017)
 - [43] Zeigler, B., Bazor, B.: Dialog design for a speech-interactive automation system. In: *Proceedings of 2nd IEEE Workshop on Interactive Voice Technology for Telecommunications Applications*. pp. 113–116 (Sep 1994)
 - [44] Zhao, G., Zhao, J., Li, Y., Alt, C., Schwarzenberg, R., Hennig, L., Schaffer, S., Schmeier, S., Hu, C., Xu, F.: MOLI: Smart Conversation Agent for Mobile Customer Service. *Information* 10(2), 63 (Feb 2019)