



**Politecnico
di Torino**

Politecnico di Torino

Ingegneria Informatica

A.a. 2022/2023

Sessione di laurea Aprile 2023

Veronica: Multi-Channel Conversational Agent to support the programmer's reasoning

Relatori:

Luigi De Russis

Juan Pablo Saenz

Candidato

Gianluca Filippo Fasulo

Summary

Conversational agents also known as chatbots are more than ever part of our everyday life from some years. Depending on the application's environment it is possible to find those kinds of agents more or less involved to aid the user. Their spreading has led to a continuous research to improve even more their utility and to help the automation of more complex tasks. The major objective of this work is to analyze the capabilities of the chatbots inside a different and less common environment and to understand if there is some potential application for the future. Specifically, during this thesis work a prototype of a chatbot, named Veronica, has been created. Its purpose is to help the programmers, on a daily basis, to improve their quality of life in tasks such as code development and keeping track of reasoning documentation. The tool implemented consist of a conversational agent integrated as an extension for Visual Studio Code. The core components of Veronica are the following: the Visual Studio Code IDE in which the chatbot resides, a small headless google chrome browser to perform Speech Recognition and Speech Synthesis and a small google chrome extension to track the user navigation. First, a literature review has been done to understand the existing proposals that rely on conversational agents to support code development without much success. Because of that scarce adoption of conversational agent on that environment, we decided to think about useful tasks that could have been performed for our use case. Then, a second wave of researches has been done concerning the general users preferences when using conversational agents and the best type of chatbot that could have fit for our needs. Follow the reasoning behind the design of the prototype and the relative implementation. Both has been performed by following the most significant insights from the previous step. Subsequently, the description of the evaluation phase performed on the tool is presented. It has been conducted on a group of university students with the final objective to evaluate the capabilities and usefulness of the chatbot abilities and to listen to possible new implementations to improve it even more in future. The thesis work is concluded with a discussion on the results obtained from the evaluation phase.

Acknowledgements

Un altro traguardo è stato raggiunto.

Innanzitutto voglio ringraziare l'amore della mia vita ed ispirazione per il nome del chatbot, Veronica. Grazie per essermi ancora accanto dopo tutti questi anni e per il continuo supporto che mi dai ogni giorno, senza di te non saprei come fare, ti amo tanto. Mi raccomando, mettimi pure per primo quando tocca a te!

Grazie mamma, papà e fratellone per il supporto in questi anni, vi voglio bene anche se non ce lo diciamo mai.

Grazie a tutti gli amici che mi sono stati vicini durante questo percorso, sia i vecchi sia quelli conosciuti in questo periodo universitario. Una menzione speciale per Dani, in quanto amico e coinquilino per qualche anno, grazie per i consigli e la compagnia, anche se sei un po' apatico con gli anni stai migliorando e lo dimostri a modo tuo! Poi volevo ringraziare i miei colleghi e amici con cui ho condiviso parte di questo percorso. Alex per essere stato un punto di riferimento per la vita a Torino, nonostante le nostre differenze caratteriali sei un grande amico. Claudio e Fabio per la vostra disponibilità e l'aiuto in questi anni dentro e fuori l'università, anche se non ci sentiamo spesso come vorrei! Grazie anche a Patti e Franco per l'ospitalità che non è mai mancata, ormai la vostra è una seconda casa per me. Un grazie ai miei gattini, Rick (in arte Checchio) e Michonne, siete bellissimi. Voglio anche dedicare questo traguardo ai miei nonni, specialmente a nonna Lilli che mi manca tanto. Infine, un ringraziamento anche al professore Luigi De Russis e Juan Pablo Saenz per il supporto ed i molteplici consigli durante lo sviluppo e la stesura della tesi, anche nei weekend! Insomma, grazie a tutti (citati o meno), in un modo o nell'altro avete contribuito in parte al raggiungimento di questo obiettivo! Adesso mi fermo prima di sfiorare anche la pagina dedicata ai ringraziamenti!

Gianluca

Table of Contents

| | |
|---|-----|
| List of Figures | VII |
| 1 Introduction | 1 |
| 1.1 Thesis goals | 3 |
| 1.2 Thesis structure | 4 |
| 2 Background and State of Art | 6 |
| 2.1 Chatbot | 6 |
| 2.1.1 History of chatbots | 6 |
| 2.1.2 Essential concepts | 12 |
| 2.1.3 Types of chatbot | 16 |
| 3 Design | 22 |
| 3.1 Design choices | 22 |
| 3.1.1 Interest around Conversational Agents | 23 |
| 3.1.2 Conversational Agent type and logic | 25 |
| 3.1.3 User preferences | 25 |
| 3.1.4 Conversational Agent and programming | 27 |
| 3.2 Design Goals | 28 |
| 3.3 General Architecture | 29 |
| 3.4 Graphical Interface | 32 |
| 3.5 Veronica’s functionalities | 35 |
| 3.5.1 Memo creation | 36 |
| 3.5.2 Memo deletion | 38 |
| 3.5.3 URL tracking | 40 |
| 3.5.4 Memo navigation | 40 |
| 4 Implementation | 42 |
| 4.1 Components | 42 |
| 4.1.1 Visual Studio Code | 43 |
| 4.1.2 Dialogflow | 48 |

| | | |
|----------|--|-----------|
| 4.1.3 | Local Node.js Server | 52 |
| 4.1.4 | Google Chrome Extension | 54 |
| 4.2 | Libraries | 55 |
| 4.2.1 | Selenium | 55 |
| 4.2.2 | WebSpeech API | 56 |
| 4.2.3 | Socket.io | 56 |
| 5 | Experimental Evaluation | 57 |
| 5.1 | Description | 57 |
| 5.1.1 | Participants | 58 |
| 5.1.2 | Preparation and Setup | 60 |
| 5.1.3 | Execution | 63 |
| 5.2 | Results | 63 |
| 6 | Conclusions and Future improvements | 68 |
| 6.1 | Conclusions | 68 |
| 6.2 | Future improvements | 69 |
| | Bibliography | 72 |

List of Figures

| | | |
|------|--|----|
| 1.1 | Search results by year for “chatbot” or “conversation agent” or “conversational interface” as keywords from 2000 to 2020 made by Scopus. | 2 |
| 2.1 | Example of a conversation with ELIZA | 8 |
| 2.2 | Example of a conversation with A.L.I.C.E. | 9 |
| 2.3 | Example of a conversation with Cleverbot | 11 |
| 2.4 | Example of a discussion with ChatGPT | 12 |
| 2.5 | Example of a AIML code | 13 |
| 2.6 | Example of Entity recognition of a sentence | 16 |
| 2.7 | Chatbot subdivision based on mode of communication | 17 |
| 2.8 | Chatbot subdivision according to action capabilities | 17 |
| 2.9 | Chatbot subdivision based on applicational purpose | 18 |
| 2.10 | Example of the response selection of a Rule-based chatbot | 19 |
| 3.1 | Example of a general chatbot architecture | 22 |
| 3.2 | The high-level architecture of Veronica | 30 |
| 3.3 | Initial draft of the interface when starting Veronica | 33 |
| 3.4 | Positions in which the chat can be placed | 33 |
| 3.5 | Draft of the memo creation by chatting with Veronica | 34 |
| 3.6 | GUI after the extension is started | 34 |
| 3.7 | Veronica’s status button that notifies that the listening is disabled . | 35 |
| 3.8 | Veronica’s status button that notifies that the listening is enabled . | 35 |
| 3.9 | Example of the creation of a memo thanks to Veronica | 37 |
| 3.10 | Example of the creation of a memo from the GUI | 37 |
| 3.11 | Example of the deletion of a memo performed by Veronica | 39 |
| 3.12 | Example of a memo deletion failed | 39 |
| 3.13 | Snapshot of a memo creation after that a navigation online has been detected | 41 |
| 3.14 | The panel in which all the documentation created is stored, it is possible to navigate through the memo by clicking on each entry. . | 41 |

| | | |
|-----|--|----|
| 4.1 | Veronica's detailed architecture | 43 |
| 4.2 | The list of intents defined for Veronica | 49 |
| 4.3 | The list of intents defined for Veronica | 51 |
| 4.4 | The list of intents defined for Veronica | 52 |

Chapter 1

Introduction

Conversational agents have been gaining prominence in the past years. The improvements in Natural Language understanding and Speech Recognition have enabled their use in several and different scenarios. Those technological progresses, made especially in entertainment and educational environments allowed to provide the user with an increasingly customized and interactive experience. This phenomenon has probably to be addressed to the needs to offer a more qualitative and relevant content to the users. The goal is to augment the user engagement and improve their experience (in case of entertainment) and their commitment (in case of educational topics). It was not common to find conversational agents around us during the past decade.

In recent years, instead, with the commodification and the increase of computational power and the sharing of open source technologies and frameworks, chatbots implementations have become increasingly common [1]. Now they are way more involved into our life and a lot of people use them in a wide number of ways.

In fact, it is possible to see conversational agents used as simple clocks or more complex chatbot that reminds appointments and helps the users to manage their daily routine. Conversational agents such as Siri or Cortana are some of the most used and well-known, but still there are a lot of different ways in which a conversational agent can be used that are yet to be discovered, discussed or improved.

The main limitation in relying on rules and pattern matching in chatbots is they are domain dependent, which makes them inflexible as they rely on manually written rules for specific domains [1].

By today, the amount of researches and data generated, related to the chatbots and the conversational agents for different applicational topics is very wide. The advances in AI, Machine Learning and Natural Language Processing tools previously mentioned, are now combined with the availability of an improved computational power. Thanks to that, the possibility to implement “advanced” chatbots which are

not relying so heavily as before on rules and pattern matching techniques is greatly increased. Despite that, the journey to be able to ensure that a single chatbot is able to fulfill different tasks on many different domains without an heavy human presence behind it is still long. Yet, the interest is still raising 1.1 and because of that it is possible to find online a lot of essays or documentation related to different domains or environments in which are discussed the possible implementation for future use of conversational agents. Is important to notice that there is a strong correlation between the chatbot evolution and the improvement of the technologies.

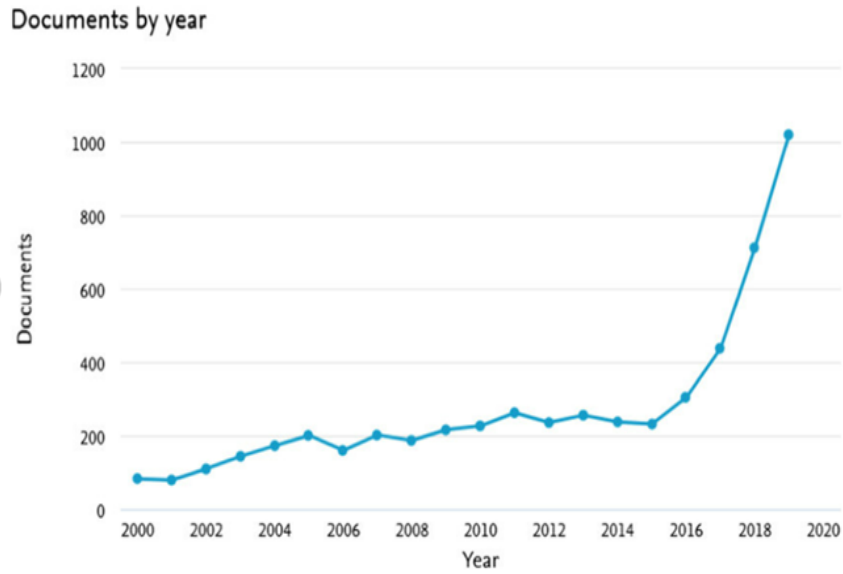


Figure 1.1: Search results by year for “chatbot” or “conversation agent” or “conversational interface” as keywords from 2000 to 2020 made by Scopus.

Some research has explored the suitability of these agents for developing programs. However, despite the great number of papers related to that topic, the documentation related to conversational agents aimed to help the programmer is really poor. We consider that a conversational agent might help programmers seamlessly and quickly document the problem-solving strategy associated with their code while implementing it. Moreover, since their usefulness to support the documentation of the code has not been explored yet. Our idea is to analyze the internal mechanisms that are retrievable by observing the users in contact with the prototype itself.

It is important to point out that this is an exploratory work and will follow these guidelines:

- A deep dive in the topic after the initial analysis.

- Evaluate it in terms of design and the conversational interface of the chatbot within the environment in which is placed.
- Define the usability and usefulness of the chatbot with the functionalities implemented to understand if it's possible to improve it even more for future applications and further studies.

1.1 Thesis goals

The final objective of this thesis work is to propose a prototype of a chatbot. The main purpose will be focused on the improvement of the programmer's proficiency related to the documentation production, by aiding him on some small tasks and by interacting with him directly during the programming session.

We want to identify, based on the literature, the features that a tool of this kind should offer, design and implement it, and evaluate the usability of its first version. The following main steps has been followed:

- Conduct a literature review to understand the existing proposals that rely on conversational agents to support code development and documentation and investigate whether these prior tools have already exploited concepts related to the “documenting the reasoning” subject.
- Design and implementation of the tool following the most significant insights from the previous step. The tool consist of a conversational agent integrated into Visual Studio Code as an extension, devoted to a given domain and programming language(s).
- Evaluation of the tool's usability and assess to what extent it effectively supports programmers documenting their reasoning and looking for past used strategies

Talking briefly about the application, it is a chatbot, named Veronica, developed as an extension for Visual Studio Code. Its purpose is to help and assist the users that want to program, with different skill levels, with some built in functionalities.

The result is a usable prototype, working on computers, that eventually could be published on the Visual Studio Code market place if defined stable.

Then, we want to define if it's possible to make some progresses in the context between conversational agents and the world of programmers. Specifically, if the actual technologies allow us to proceed even further in that direction to evolve from a small prototype to an actual application (even multi-platform) that is able to aid the programmers in different coding languages and at different layers of knowledge.

Talking about the prototype itself, the main functionalities of the extension to help the programmers that we are going to see are the following:

- The possibility to take customized memo that are natively merged into the code. The memos are usually underutilized by the programmers, the possibility to write or dictate them faster and to find them in an easier way increase the order of the code and the efficiency of the user.
- The tracking of the navigation done by the user online to allow him to remember important pages. The possibility to have a link of an important page related to a portion of code is incredibly important. Especially if the programmer had trouble to resolve it the first time.
- The change of the interface to be as much intuitive as possible to be used. The design itself has to be taken into account, since it should be understandable at a first glance and allow the user to focus on his work and not on how to make it works.

1.2 Thesis structure

The thesis is structured as it follows:

In the chapter 2 there is a description of the State of Art. Specifically, it contains the story of the conversational agents, the different evolution and paths that has been generated during their development and a brief description of the different types that can be found by now. Moreover, there will be some data regarding the growth of interest related to the topic and some important definitions. There will also be some examples of famous conversational agents that is possible to find around and that can be taken as example.

Inside the chapter 3 is possible to see the Design of the application and the elements related to the visual compartment. Additionally, it will be explained in general how the extensions was made on Visual Studio Code and how the integration appears inside the environment. After that, there will be an high level description regarding the application and how every piece is connected to make Veronica work.

The chapter 4 will be focused on the implementation, inside it there will be the technical explanations about how every piece of the engine work, and we will dive deeper into the code and how each element is connected to each other.

The chapter number 5 contains all the information related to the usability testing performed in the laboratory, such as the kind of users, the metrics used, the setup of the environment and so on.

During the chapter number 6 we are going to discuss about the results of the usability testing, the attended and desired ones. The final thoughts of every user gained after an exit interview will be taken into account to understand their feeling about the prototype and the topic itself.

The last chapter is dedicated to the conclusion and future improvements that could be applied to the application, together with some personal thoughts.

Chapter 2

Background and State of Art

In this chapter the story of the conversational agents will be discussed, from their origin to the present days. Moreover, the essential concepts related to this topic will be deepened to have a better understanding of how each kind of chatbot works. Subsequently, the different type of chatbots will be described to understand their functionalities and fields of application based on their capabilities.

2.1 Chatbot

2.1.1 History of chatbots

Chatbots, surprisingly, have been around since the mid-1960s. The English computer scientist and pioneer Alan Turing, known as the father of computer science, which was also a crypt analyst, mathematician and logician gave us the first evidence of this kind of computer programs. In 1950, with his famous publication “Computing Machinery and Intelligence (Turing 1995)” [2] he cemented the basis around the computer intelligence. Specifically, he posed the question of whether a computer program could talk to a group of people without realizing that their interlocutor was artificial by proposing a criteria to determine if a computer program was able to have thought or not [3]. This is known as the "Turing Test", which highly depends on the capability of a machine to impersonate a human being in a conversation with at least another human interlocutor. As long as the human that is judging cannot distinguish, reliably on the basis of the conversational content alone, between the program and a real human. This essentially would be the genesis of chatbot technology [4].

The history of chatbots can be divided into several stages:

- **Early beginnings (1960s-1980s):** The first chatbot, ELIZA [5], was created in the 1960s and used simple pattern matching to respond to user inputs. In

the 1980s, chatbots such as PARRY [6] simulated conversations with people suffering from schizophrenia.

- **Rise of the Internet (1990s-2000s):** With the rise of the Internet, chatbots became more widespread (i.e.:A.L.I.C.E.) and started to be used for customer service and online shopping. In 1997, the first commercial chatbot, SmarterChild [7], was launched.
- **Advancements in AI (2010s-present):** With advancements in machine learning and AI, chatbots became more sophisticated and capable of understanding and processing natural language. Today, chatbots are used in a variety of industries, including healthcare, finance, and education, to provide quick and convenient assistance to users.
- **The Future of Chatbots:** Chatbots continue to evolve and new developments are being made all the time. In the future, chatbots are expected to become even more intelligent and capable of carrying out complex tasks, such as booking appointments and providing personalized recommendations.

Early beginnings (1960s-1980s)

The Turing Test of Alan Turing inspired the German-American computer scientist Joseph Weizenbaum, which was a professor at Massachusetts Institute of Technology (MIT). In 1966, Weizenbaum created in the Artificial Intelligence (AI) laboratory of the MIT a program called ELIZA 2.1 [5].

The program ELIZA was named after Eliza Doolittle, which was one of the main characters from Pygmalion. ELIZA was built with the purpose to simulate human conversations, to give to the other interlocutor an illusion to think they were interacting with a real human. His algorithm, which would be used by all the chatbot makers in the near future, examined the keywords received as user input and then triggered its pre-programmed output. Then it would pair those output with a list of possible scripted responses. ELIZA, however, did not possess a framework to understand intents and contexts of a conversation like the modern chatbots do. The scripted responses were designed to simulate a psychotherapist.

The script proved to be a significant impact on natural language processing and unnatural intelligence, with copies and variants protruding up at academies around the country.

For instance, after receiving a specific sentence ELIZA would perform the following steps:

- Receive the input and save them in the memory for additional analysis.
- Search for specific keywords in the input sentence.

```
Welcome to
EEEEEE LL      IIII ZZZZZZ AAAAA
EE      LL      II     ZZ  AA  AA
EEEEEE LL      II     ZZ  AAAAAA
EE      LL      II     ZZ  AA  AA
EEEEEE LLLLLL IIII ZZZZZZ AA  AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

ELIZA: Is something troubling you ?
YOU:   Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU:   They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU:   Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU:   He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU:   It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

Figure 2.1: Example of a conversation with ELIZA

- Return back the scripted response in case a keyword is matched.
- Give back a default fallback answer if there isn't a match with the expected input.

Although the simplicity of the algorithm, those steps were already enough to make a human judge interlocutor think that they were talking with an intelligent being. Even so, the AI in ELIZA was rudimentary at best yet.

It was until six years later, in 1972 when the natural language program PARRY, constructed by the American psychiatrist Kenneth Mark Colby at Stanford University, became the first program to pass the Turing Test. PARRY was designed to simulate the conversation of a paranoid individual and study the behavior and language patterns of people with mental illness. The chatbot used a rule-based system and a limited knowledge base to respond to user input in a somewhat coherent manner. PARRY marked a significant step forward in the development of artificial intelligence and natural language processing, and it remains an influential work in the field today.

Rise of the Internet (1990s-2000s)

The 1995 marks the development of the first chatterbot that worked with an XML schema also known as artificial intelligence markup language (AIML) [8]. The chatterbot, known under the name of A.L.I.C.E. (Artificial Linguistic Internet Computer Entity) [9] was developed by the computer scientist Richard Wallace which was inspired by the previously cited work of Weizenbaum [5]. The chatbot was designed to simulate human conversation and can respond to a wide range of topics and questions. The idea behind it was to mimic the chat with a real person over the Internet. A.L.I.C.E. is a young-looking woman in human years and tells a user her age, hobbies and other fascinating facts, as well as answering to the user's dialog 2.2.



Chat Bot – ALICE (AI Foundation)

- *Human: hi my name is Carla*
ALICE: How are you doing, Carla.
Human: what is your name?
ALICE: My name is ALICE.
Human: what is your full name?
ALICE: ALICE = Artificial Linguistic Internet Computer Entity.
Human: when were you born?
ALICE: I was connected to the net on November 23, 1995.
- *Human: Have you passed the Loebner Prize?*
ALICE: The Loebner Prize is an annual Turing Test sponsored by New York philanthropist Hugh Loebner. Dr. Wallace and ALICE won it in 2000. The ALICE program was ranked "most human computer" by the panel of judges.
- *Human: who is Dr. Wallace?*
ALICE: He is a famous computer scientist, the author of ALICE, and a graduate of Carnegie Mellon.
- *Human: Where does Dr. Wallace work?*
ALICE: he works from his home in San Francisco.

Carla P. Gomes
INFO372

Figure 2.2: Example of a conversation with A.L.I.C.E.

Thanks to its implementation, A.L.I.C.E. won the Loebner prize three times because it was considered the most human-like conversational agent of its time. However, it never passed the Turing test of being able to think such as humans. It's important to point out that A.L.I.C.E. was an open-source chatbot and because of that the program evolved during the years also thanks to the wide community behind the project. In 1998, the program was edited in Java, and in 2001 Wallace printed an AIML specification of it. From there, other developers drafted free and

open sources of ALICE in different programming languages and a variety of foreign languages. [10]

The introduction of artificial intelligence in the field of conversational agents, attracted the attention of the big companies of the IT world, which have started to release commercial products that use this technology. However, right before the big companies, a good example of a chatbot powered by artificial intelligence was given by Cleverbot in 2008. Cleverbot is the evolution of the Jabbewacky chatbot. It uses machine learning algorithms and a large database of previous conversations to generate responses to user inputs 2.3. This means that the chatbot can learn from its interactions with users and improve its responses over time. Cleverbot has been praised for its ability to simulate human conversation, and many users have reported having long and engaging conversations with the chatbot. The chatbot itself has also been the subject of academic research, and has been used to study the nature of human-AI interaction and the development of more advanced AI systems. Today, Cleverbot continues to be used by a large community of users, and has been integrated into a variety of applications and platforms, including mobile apps and virtual assistants. Despite its age, Cleverbot is still considered one of the most advanced and innovative AI chatbots available, and continues to be used as a platform for research and development in the field of artificial intelligence. Moreover, Cleverbot has reached a score of 59.3% to the Turing Test, compared to the rating of 63.3% human achieved by human participants. A score of 50.05% or higher is often considered to be a passing grade [11].

Advancements in AI (2010s-present)

In this range falls almost all the most famous and known chatbot. In 2010 Apple developed the well-known Siri for iOS. Despite its development has been completed in 2010, it was actually first introduced as an app in 2011 for the iPhone 4S. Siri is a personal assistant and navigator that uses a natural language user interface. It uses natural language processing (NLP) and machine learning algorithms to understand and respond to users' voice commands and questions and to continuously learn from each discussion done. Moreover, studies towards the deep learning has been made to upgrade even further the voice of the assistant to make her voices in different languages to sound more natural, smoother, and allow Siri's personality to shine through [12]. It is possible to say that Siri opened the way for the development of all the subsequent personal assistants (PAs) chatbot made from other companies.

Another one worth to be cited is ChatGPT, a large language model trained by OpenAI and founded in the early 2021. The model is trained on a massive amount of data that enable it to generate human-like text based on the input request.

The ability to range with ease from such a wide number of topics, together with the human-like text based that the bot is able to output had also a massive impact



Figure 2.3: Example of a conversation with Cleverbot

on the general way in which the normal user sees the chatbot.

Some of the world's biggest academic journal publishers have banned their authors from using the advanced chatbot. Since the bot uses information taken from the internet 2.4, there is a general consensus on the idea that that inaccurate or plagiarized work could enter the pages of academic literature [13]. Several times already the chatbot has been entitled as co-author on academic studies. Another recent study [14], has shown that ChatGPT could be used to write a finance paper that would be accepted for an academic journal. Although the bot performed better in some areas than in others, accordingly to the study.

The capabilities of the bot highlighted some clear ethical implications [13]. In short there is a division between two different line of thought. There is who thinks that the problem of plagiarism and faking news might be worsened by the application of these kind of chatbots. On the other hand, there is who thinks that with a human input each research can be improved even more and output acceptable researches.

As we were able to see, the chatbots have come a long way from their first developed to our days.

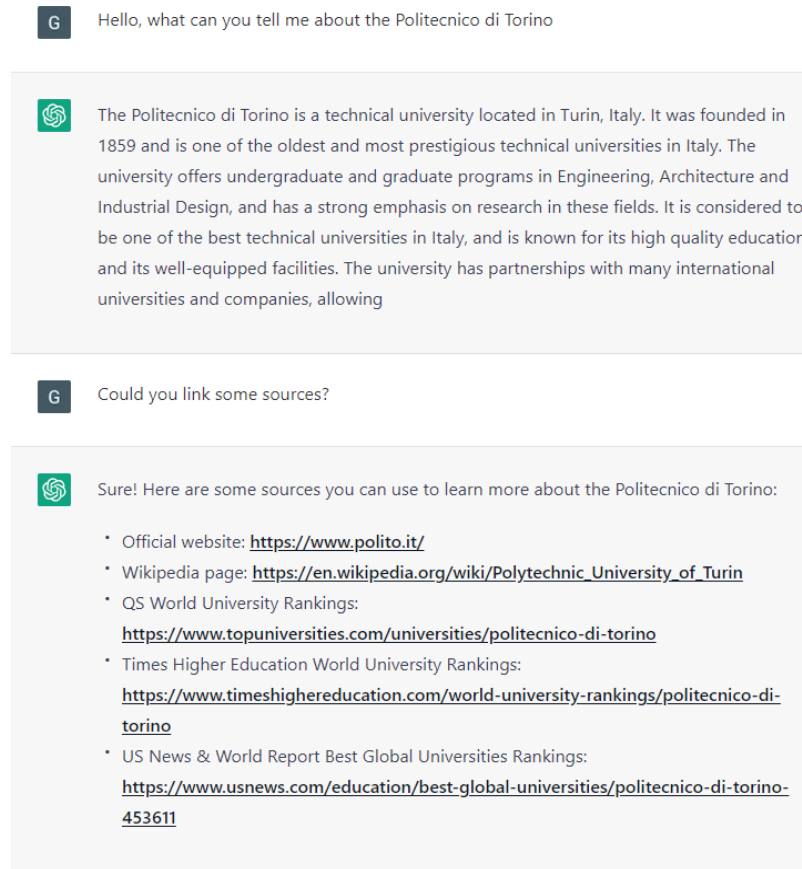


Figure 2.4: Example of a discussion with ChatGPT

2.1.2 Essential concepts

Here are explained some important concepts related to the chatbot technology to understand better how they work. Specifically, the following are going to be seen:

- Pattern Matching.
- Artificial Intelligence Markup Language.
- Natural Language Processing
- Natural Language Understanding
- Intent
- Entity
- Context

Pattern Matching

The Pattern Matching is a method that allows to identify some patterns in the text inputs and to match them with a list of predefined outputs (responses) which are consistent with the user input. Usually, the scripted or Rule-based chatbots fall under this category. Most of the early chatbots used this method, for instance ELIZA and A.L.I.C.E. were the first ones developed using this pattern. Clearly, the ease of development with this method is counterbalanced by the predictability and repetition of the responses, which leads to a lack of credibility. Moreover, the previous discussions are not stored and there is an high chance to have looping conversations [15].

Artificial Intelligence Markup Language

The Artificial Intelligence Markup Language (AIML) it's based on the concepts of Pattern Recognition or Pattern Matching technique. It is one of the main technique that uses markup language, which was created by Dr. Richard S. Wallace. Those kind of chatbots start from a predefined knowledge base which is composed by documents or files that define a particular category. A category is part of the chatbot's knowledge base divided into small sections [16]. These categories contain two tags: patterns and templates 2.5.

```
1 <aiml version="1.0.1" encoding="UTF-8"?>
2 <category>
3   <pattern> HELLO BOT </pattern>
4   <template>
5     Hello my new friend!
6   </template>
7 </category>
8 </aiml>
```

Figure 2.5: Example of a AIML code

The patterns are rules that helps the chatbot to understand what its response should be, while the templates represents how the output should look like. The way these rules are built varies depending on the size and complexity of the knowledge base. Once the chatbot receives a matching input with the pattern, select one of the messages stored as a template and sends out the response. To perform a better Pattern matching, the use of Regular Expressions (regex) is often involved to identify a specific input given by the user. This allows chatbots to handle a wide

range of user inputs and respond in a more natural and conversational manner. For example, if the chatbot receives as an input “Hello”, by using pattern matching it would recognize the message as a greeting and answer back with “Hi! How are you?” 2.5. Based on that, if the questions made by the user don’t fall in line with the manual programming of the chatbot, they won’t be able to find a predefined pattern to match for a template and will respond with fallback phrases. AIML does not require a deep knowledge or skill in a specific programming language. Because of that, it is pretty common to find chatbots that uses this technique, since it facilitates a lot their development.

Natural Language Processing

The Natural Language Processing (NLP) is a sub-field of Artificial Intelligence and Computer Science that studies the way in which the machines can manipulate the natural language, both written and spoken. The knowledge about the understanding and use of human language is gathered to develop algorithms that make computers understand and manipulate natural expressions to perform the desired tasks. NLP’s aim is to extract the meaning from a text, primarily the grammars that specify relationship between parts of a phrase such as verbs, nouns and adjectives. It is also possible to extend the grammar to address natural-language semantics, for example with additional constraints or rules (i.e.: “drink” applies only to drinkable nouns). Unfortunately, the vastly large size and possibilities lead to an unmanageable numerous of rules, often interacting unpredictably, with more frequent ambiguous parses. In general, any kind of NLP algorithm converts words or phrases (which are usually called “tokens”) into vectors of real numbers so that they can be used to train a deep learning algorithm to recognize text patterns [17]. Most NLP techniques are based on machine learning.

Natural Language Understanding

The Natural Language Understanding (NLU) is a subset of NLP that is focused on the task to understand the meaning of the human language inputs. It’s the core of any NLP task. By performing NLU to a sample a data, we basically analyze the human language, which may be unstructured, as text or speech and extract some meaningful insights, ranging from the intention of the speaker to the named entities mentioned till the sentiment expressed by the user [18]. Then, the output should be a reply appropriate accordingly to the original user intention. All the various algorithms used by NLU, such as semantic analysis, syntactic analysis, and word sense disambiguation, are meant to determine the meaning of human language inputs. These algorithms can then classify the inputs into various categories, such as questions, statements, or commands, and extract relevant information, such as named entities and sentiment. NLU is a crucial component in every topic

concerning any conversational AI systems such as chatbots or voice assistants. Not only it allows these systems to understand the meaning behind the users' requests, but also enables them to respond in a way that is relevant and useful. It's also important to point out that NLU is also used in different environments in which can be useful to improve the accuracy of various devices used for text classification, machine translation sentiment analysis.

Intent

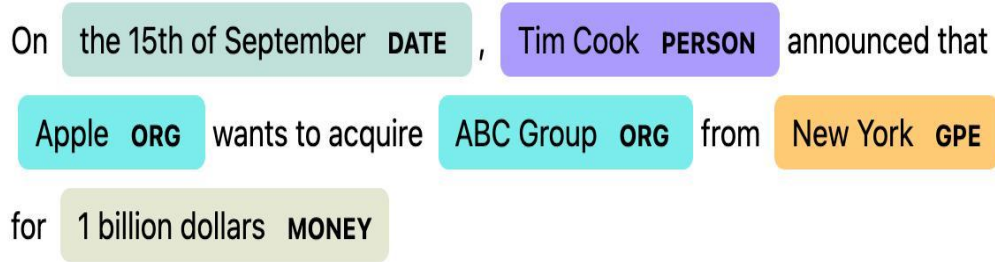
Two fundamental concepts of NLU are intent and entity recognition. The Intent recognition is defined by the process of identification thanks to the user's intentions based on their inputs to determine their objective. It is the first and most important part of NLU because it establishes the meaning of the text. An Intent represents the correlation between what a user says in a sentence and which action the chatbot should take accordingly. An Action correspond to the sequence of steps that the chatbot will take when a specific intent is triggered by a user input. Each sentence of a user may have parameters for specifying detailed information about the steps to be performed. During the Intent detection, single or multiple intent are labels are predicted for each sentence given by the user. Subsequently, from the user intent is possible to extract domain-specific entities.

Entity

The Entity recognition is the other fundamental concept of NLU that focuses on identifying the entities in a message, then extracting the most important information about those same entities. Exists two types of entities divided in "named" and "numeric". The named ones are grouped as categories (i.e.: Food, locations, cars, etc.). Differently, the numeric entities are recognized as numbers, percentages or currencies. When the Entity recognition is performed, the final outcome is to extract some parameters in form of values from the inputs obtained from the Natural language 2.6.

From what we can see in the figure 2.6, the user intent is to inform the interlocutor about a future fact. The entity values are the one highlighted 2.6. Therefore, the user is trying to tell us that the person "Tim Cook", on the date of "15th September" the organization "Apple" wants to acquire the organization "ABC Group" for a money amount of "One billion dollars". There can be entities both system-defined or developer-defined based on the topic in which we want to specialize our conversational agent.

Domain entity extraction usually referred to as a slot-filling problem, is formulated as a sequential tagging problem where parts of a sentence are extracted and tagged with domain entities [3].



On the 15th of September DATE , Tim Cook PERSON announced that Apple ORG wants to acquire ABC Group ORG from New York GPE for 1 billion dollars MONEY

Figure 2.6: Example of Entity recognition of a sentence

Context

In the end we have the contexts, which are strings that store the context of the object the user is referring to or talking about. For example, the user could refer to a previously cited element in the following sentence. He may input a sentence such as “I would like to order a pizza”. Here the context to be saved is the pizza, so that when another sentence such as “I would like to place mushrooms on it” is given as an input, the intent “place mushrooms on it” can be invoked on the context “pizza”.

2.1.3 Types of chatbot

When talking about chatbots, a lot of different criteria for categorizing them exists: the way in which they can communicate, their capabilities, the domain or application in which they operates and finally the way in which they are implemented [19]. We are going to focus our attention on the way in which they are implemented and how they works. Before that a brief insight on the other categories will be given. The first subdivision of conversational agents is based on the mode in which they communicate with the human user 2.7. The communication methods in which a conversation can happens are the following:

- Text-based: The text-based CA interacts with the user only by text, usually from a chat, and it doesn’t utilize any other communication method.
- Voice-based: The voice-based virtual agents are CAs that interact with the users by voice and don’t utilize any other kind of communication method.
- Graphic-based embodied: Graphic-based embodied agents are all the virtual agents that have a virtual body in addition to some voice understanding and speech generation capabilities. The addition of the virtual body enables them to improve the communication with gestures and visual kind of emotions.

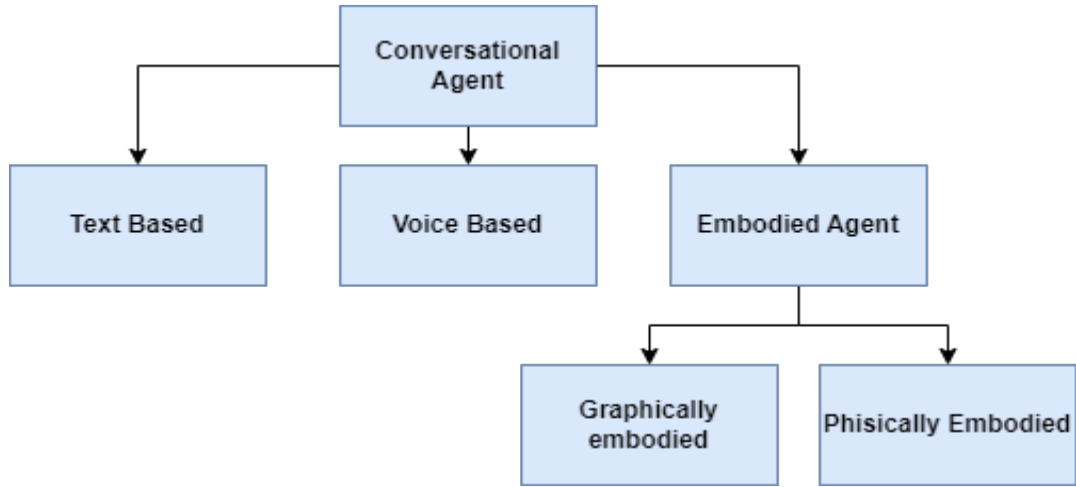


Figure 2.7: Chatbot subdivision based on mode of communication

- Physical-based embodied: Physical-based embodied agents are all the conversational agents that have a physical and tangible body. A good example is given by social robots.

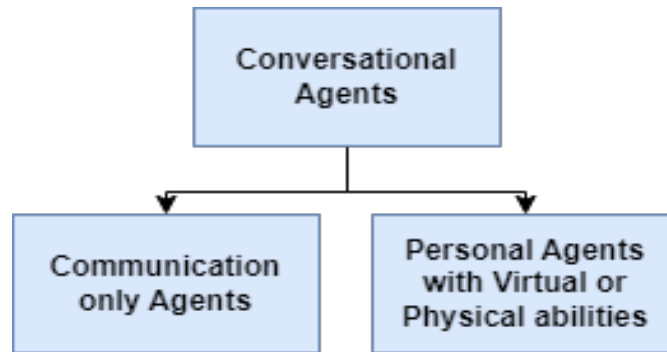


Figure 2.8: Chatbot subdivision according to action capabilities

Another way to distinct the chatbots is given by their capabilities to perform some actions 2.8. Some of them merely communicate with the users and do not perform any action, usually those kind of chatbot are there to answer questions. The virtual or personal assistants, instead, are capable to perform a variety of actions, ranging from turning on something to booking appointments.

Another way in which CAs are often classified according to the applicational purpose 2.9:

- Open domain or general purpose: Those CAs are mainly used to answer

questions ranging in many different domain, usually those are communication-only agents.

- **Goal-oriented:** Specialized chatbots that assist the users in completing complex tasks that requires multiple steps and decisions in a specific topic. These agents may be used both in the business domain or as personal assistants.
- **Social-supporting:** Agents that can support the users in various topic in which they need to improve, for example patients that have a specific medical condition or students that are in a learning process.
- **Social-network:** Those chatbots are also known as influence agents, they are distinct by their intelligence around social media to advertise a product or to influence opinions.

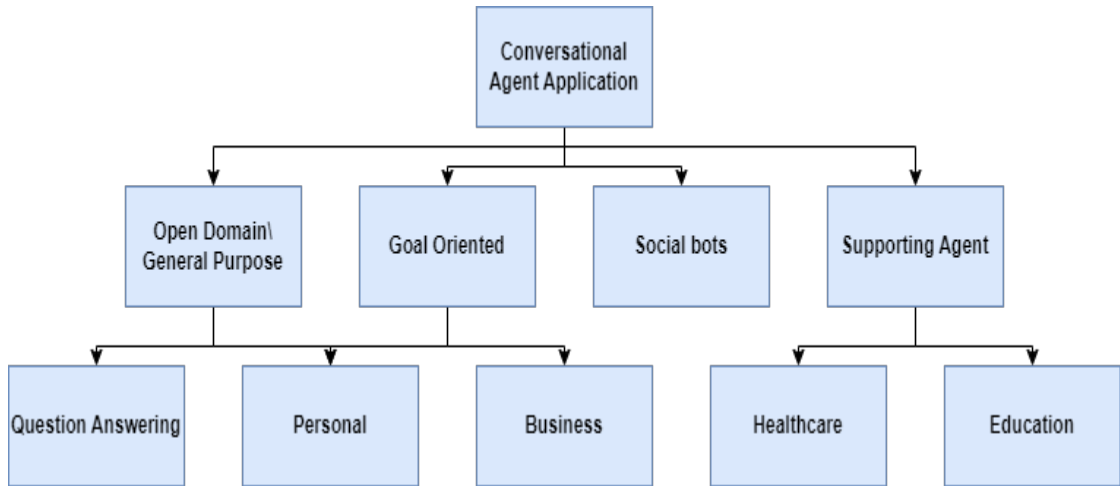


Figure 2.9: Chatbot subdivision based on applicational purpose

Finally, chatbots can be distinguished by the logic of their backend that defines their intelligence level and capabilities.

Rule-based chatbots

The rule-based chatbots follow a set of previously defined rules that enable them to respond to each user's input. They can only follow the rules programmed into it and respond accordingly, leading to some severe and limited conversational capabilities. Those bots, were able to identify some keywords from a given input sentence and be able to match them against a collection of previously defined rules in order to respond with an adequate sentence 2.10. Moreover, the bots of this type abide by the rule-based orientation, in which an algorithm processes every

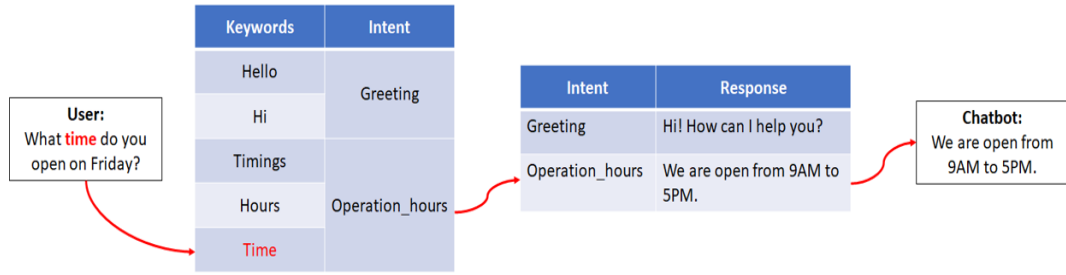


Figure 2.10: Example of the response selection of a Rule-based chatbot

user's utterance and pass them by a defined set of rules. Essentially, after that the scan is completed, those conversational systems proceed to setup a reply with the matching or most similar wording pattern, taken from a linked database or hard coded [20]. In this category falls most of the early developed chatbots such as: ELIZA [5] and PARRY [6]. This kind of chatbot typing allow us to understand that they may not necessarily have to understand human speech, but can also rely on pattern recognition, abiding by the rule-based approach to output acceptable responses. Chatbots have now began understanding human speech as it is spoken, although the technology itself had not turned mainstream. The benefit of such rules is that they are precise, and allow developers to create and remove rules to handle new situations and address bugs with certainty [21].

Retrieval-based chatbots

The mechanism behind the Retrieval-based chatbots is similar, yet different from the rule-based ones. Retrieval-based chatbots work on the principle of graphs or directed flows. The chatbot is trained to provide the best possible response from a database of predefined responses which are based on existing information [22]. The different techniques used by this kind of chatbots to identify the most appropriate response are: keywords matching, machine learning and deep learning. Regardless of the technique, only previously defined responses can be provided by these chatbots and new output can't be generated. Retrieval-based chatbots can have more flexibility in terms of responses compared to the rule-based chatbots, since they can select their responses from a larger pool of entries. However, they are still limited by the previously defined responses in the database.

Generative chatbots

Chatbots that use generative methods, differs from the previously cited Retrieval-based ones because they are not limited to predefined responses. In fact, they can

generate new responses that are given as an output from a large amounts of training data. Generative chatbots use a combination of supervised learning, unsupervised learning and reinforcement learning for multi-step training [22]. Thanks to that, Generative chatbots can also understand the context of a conversation and generate responses that are more appropriate for the given context. The reason to use a combination of the aforementioned techniques is to improve the stability of a human conversation without repeating the same sentences. Alone, the supervised learning mapping of a user inputs to a generated response tends to prioritize high-probability responses such as “I know”.

Intent-based chatbots

These types of chatbots do not relies upon a match between patterns and templates as the previous ones. As the name suggest, they are heavily oriented on the intent recognition and can handle better any kind of text. The result is that any sentence can be used as input by the user and the bot will understand the commands behind it. The way in which the sentence is understood is divided in two part, similar to how it was described previously:

- Intent Recognition: Identification of the intention of the user and the action that has to be performed by the bot.
- Entity Recognition: A drill down on the details of the Intent.

For instance, if the user says to the chatbot that he wants to “Watch an Horror”, the request has to be divided in the aforementioned steps; First the agent have to understand that the user wants to watch a film and then it must catch that he wants to watch an Horror ones. Thanks to the Natural Language Processing (NLP) algorithms, as we have seen before, the Intents can be recognized [17]. A typical implementation is given by a model that consists of an encoder and a decoder that are implemented using recurrent neural networks. The encoder takes a set of tokens as an input and gives as an output a vector. The vector is then given as input into the decoder which generates a set of tokens. Once the decoder generates a stop symbol the process terminate.

The behavior used by those kind of chatbots is often used in language translation.

There are a lot of open source engines available online. One NLP/NLU engine that is also available for free is Dialogflow, it is owned by Google and has been used as a core component inside this project.

Conversational Agents

The Conversational Agents can be considered as an upgrade from the Intent-based agents thanks to the addition of multi-turn dialogue. This is accomplished by

monitoring of the progresses during a conversation and by recognizing when the user wants to switch to another topic. A good example of how to put this idea into practice is given by the RavenClaw framework [23], which makes use of a dialog stack and an expectation agenda. The conversation stack maintains track of everything the chatbot wishes to discuss. The expectation agenda is a data structure that is used by the chatbot to keep track of what input it should receive before hearing it. For example, the expectation agenda is filled with the response “White” when the chatbot asks “What color are the clouds?”. However, if the user responds with “I want to buy food” the bot needs to change the context of the conversation. As a result, the dialog stack is pushed with “Order groceries” and the expectation agenda is updated with potential responses and queries about placing an order for food. A study demonstrates that also reinforcement learning combined with machine learning is another implementation option that is feasible [24]. Briefly, the concept is that the chatbot has states made up of what it knows (questions it has responded to), what it just said, and what the user just said. The objective of the agent is to use a rule given to him that gives the best outcome for being in a specific state. In this case, the problem is related to the rule’s learning, because it requires a lot of training and is often hard to know exactly in which state the agent is.

Chapter 3

Design

This chapter is devoted to outline the architectural work that has been done prior to the chatbot's actual implementation. A first high-level perspective is given, demonstrating how components are linked together to function and communicate, before diving into the specifics of each individual component, including their description, function, and justification for selection.

3.1 Design choices

The design and development of a chatbot involve a variety of techniques [25].

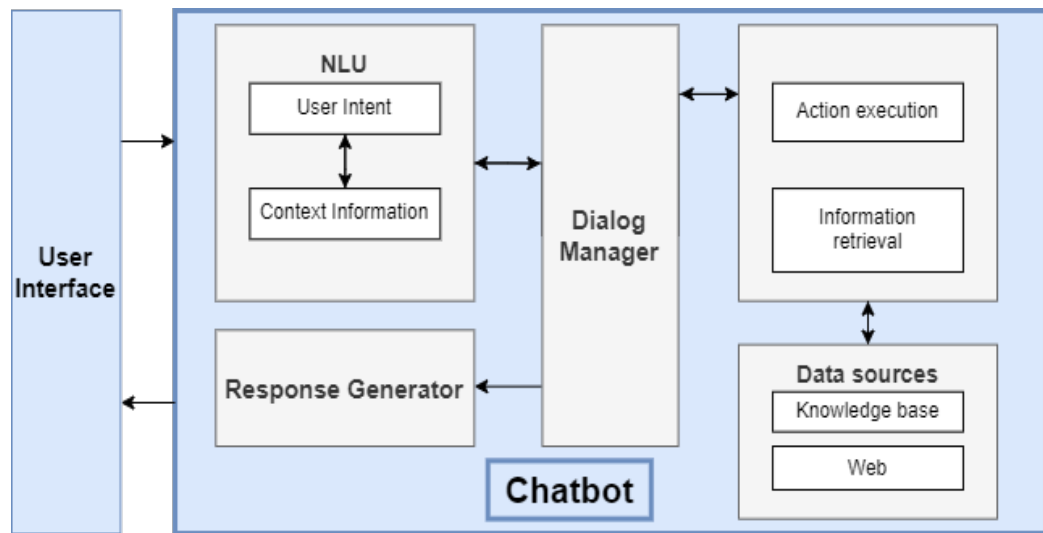


Figure 3.1: Example of a general chatbot architecture

Understanding what the chatbot should be able to perform and, in what category falls into, was important to help us pick the right choices and tools to build it [26]. Usually, the main requirements to design a chatbot involves [27]:

- A good strategy for answer generation.
- A set of predefined answers, usually neutral, to utilize when an user's utterance is not caught by any conversational path.
- The selection of the right typology of chatbot.
- The focus on the topic that this specific bot should cover.
- Who controls the direction of the interaction, the user or the conversational agent?
- What freedoms do users have through the interface and what must be prevented from being done.
- How to handle synonymous expressions.

The first step while designing any prototype is to divide it into modules. Each module should have its own task to be performed. In 3.1, a general chatbot architecture is introduced.

3.1.1 Interest around Conversational Agents

To decide the design of the prototype a literature review has been done to find the best fitting options for our use case. First, a study to understand the interest around conversational agents has been conducted. Some studies [28] highlighted the increase of the interest of the conversational agents in many domains. This interest grown was highly correlated with the technology improvement that overcame the previous limitations and difficulties. The use of chatbots seems to be highly focused on specific environments [28]3.1.

| Domain | Chatbot Role | Reference |
|------------|---|----------------------|
| Education | <ul style="list-style-type: none"> • An intelligent tutor for a university level course. • Providing education system information and services on behalf of the academic staff. • Improving the security and automation of a lab by the voice-based agent | Prasad et al. [29] |
| Finance | <ul style="list-style-type: none"> • Providing financial-product sales. • Online customer support in banking industry. • Answering a question about customers account, bill payment, credit card payments, and schedule meetings. | Khan et al. [30] |
| Healthcare | <ul style="list-style-type: none"> • Providing guidance for consumers or their carers when they have medical problems. • Providing diabetics with diets and information regarding foods to be avoided. • Providing information for preventing COVID-19 pandemic. | Martin et al. [31] |
| Tourism | <ul style="list-style-type: none"> • Providing information or services through conversation-like interactions for tourism and travel. • Consumers use travel chatbots to book a trip, plan a vacation, discover new experiences, and make reservations at hotels. | Alotaibi et al. [32] |
| E-commerce | <ul style="list-style-type: none"> • Offering features to brands, such as sending advertisement messages, asking for customer feedback, and collecting customers' preferences. • Providing an online experience and customer service through social media sites. | Zarouali et al. [33] |

Table 3.1: The main domains in which chatbots are used and their main roles.

Because of that, the information about conversational agents that support the programmer were really scarce. That led us to find different sources to make our architectural decisions, such as inter-domain best practices and users preferences.

3.1.2 Conversational Agent type and logic

Then, we moved on to see the best chatbot type to implement the prototype. The choice was to lend toward the Intent-based chatbot, to handle extended discussions and to allow for mimicking context switching and unstructured conversation dialogues of a human-to-human dialogue. Moreover, unlike other chatbot types, with this one is possible to utilize the power of AI in order to respond. We took advantage of the fact that the visual structure adopted is similar to the common chat-like interfaces of which almost every user is used to.

Subsequently, we decided if the chatbot should have been able to support a 1-to-1 scenario or a collaborative one with many users involved [34] [35].

Each conversational agent can be used to support a 1-to-1 scenario in which the parties involved in the discussion are just the user and the agent. The collaborative scenario is way different, cause it requires from the agent to be able to carry on a conversation in which many users can be involved. The ease of implementation of the first one, compared to the latter, was a key factor that influenced our decision to pick the 1-to-1 scenario. Moreover, we assumed that both parties would alternate during the conversation, in that way there won't be mistakes related to comprehension very often.

Another important aspect to consider was related to the ability of the bot to process user utterances only when the agent is in the input phase or constantly monitor user interaction and perform interventions when necessary. To support the programmer's reasoning we opted for the constant monitoring of the user interaction.

3.1.3 User preferences

During this phase the focus was mainly on the communication channel that had to be adopted by the agent. Conversational Agents can be divided in three categories:

- Text-based: the conversational agent is only able to communicate by text, usually from a chat.
- Voice-based: the conversational agent is only able to communicate by voice.
- Multi-Channel: the conversational agent is able to communicate both by text and voice, allowing the user to use his favorite channel.

By gathering some studies [34] [35] [36] we discovered that the results obtained after a user testing were similar despite the different kind of conversational agent used and the different domains. The most relevant aspects that we discovered were:

- Novice users highly prefer to use the voice over text in general.
- Expert users prefer to stick to the text and use the voice to improve what they already can do.
- A voice-only conversational agent seems to be unfeasible and more frustrating. For example, both expert and novice users relied on text to overcome harder problems or when the voice recognition was not working well.
- The hybrid setup seems to be the most appreciated even when using voice, a visual text feedback after or during an action decrease dramatically the cognitive load and difficulty of utilization.
- It is wise to explain to the users how to talk or interact with the application when they use it for the very first time, an on boarding design that influence first-time interaction with users is a critical factor that affects user experience.
- It's important to see if the number of channels of communication has an influence on the user. Cognitive theory says that the human mind can process different information concurrently from different channels, such as visually and verbally.
- A step-by-step interaction could be good, where the chatbot takes the role of the educator or another one, leading to the final objective gradually.
- The use of a conversational agent maintains the users engaged in conversation for prolonged time.
- Every agent that adopts a natural language system can free the user eyes from the phone or the computer, common examples are given by those agents that integrates elements for navigation and exploration options.
- Dialogue is considered a medium through which nearly all learners are familiar. Also, familiarity with chatting media seems to be a feasible tool since everyone is familiar with that kind of communication.

Moreover, Nielsen's heuristic could be applied also to conversational agent interface, a lot of usability problems and interaction design failures can be discovered thanks to them.

Following the results of those studies we decided to opt for a multi-channel conversational agent. In this way it is able to interact both textually from the chat and verbally with the user to allow him to choose for his favorite communication channel.

Moreover, the chat is used to show also the verbal interaction, in this way it is possible to give a feedback and to track the history of the conversation. Another important aspect of the chat design is that almost all the users are used to the chatting application, in this way we give them a sense of familiarity with the tool.

A step-by-step interaction has been implemented, our intention was to help the user to unstuck in case of wrong inputs and to make the bot able to suggest what he expected in specific scenarios.

An interactive tutorial has not been implemented in this phase, however it is possible to ask to the conversational agent what he is able to do thanks to some specific commands.

3.1.4 Conversational Agent and programming

As we previously said, this domain is still a niche in terms of conversational agent application. Some studies has been found in the later phases of our research. Specifically, a conversational agent used to help the programmers to interact with GitHub [37] and another one used to help during the programming phase with an embodied human-like representation of the chatbot [38]. However, despite being studies aligned with our domain, the final goal was slightly different. The most relevant info gathered about that topic are the following:

- Programming by voice takes a huge training time, usually 1 to 2 months, during this time it can be frustrating. After the initial learning curve, users often create new custom prompts for commonly used commands following their needs [39].
- It is useful to rest hands and have a good alternative to the normal text programming. You are not forced to sit close to the table. You think more about what to say, improving the code overall [39].
- On the other hand, you can't listen music or be in a crowd place, silence is required.
- The use of the English language is often tricky, because some words are part of the code, so it's impossible to use them to dictate something and it's required to find shortcuts or alternatives to that.

- A constrained natural language is good to reduce the ambiguity of the normal language and improve the understanding of the programming system. However it increases the cognitive load of the user that must think about precise commands [34].
- A continuous conversation helps on the debugging part, allowing to have a more solid structure of the code even before the termination of the program [36].
- The conversational agent prevents the user from doing logical errors by notification.

Because of that, we decided to pick functionalities to help the programmers that wouldn't collide with the code. In this way the training time related to the coding is nullified and at the same time we are able to receive a fast feedback from the usability testing done in the later phase of the thesis. Moreover, the chatbot is able to listen continuously to the users to support their reasoning.

3.2 Design Goals

Given the previous analysis, we decided to define three goals that Veronica should be able to perform, relieving the programmers to actively thinking about those tasks. Here are presented the main goals that we wanted to achieve with the Veronica's prototype:

- **Support in real time user's documentation management directly from Visual Studio Code:** The goal of Veronica is to inspire the programmers to record their development process while writing code as fluidly as possible. Because of that we have chosen to keep a graphical interface visible next to the code 3.6. In that way, the user can activate Veronica by clicking on a button to the right of the Visual Studio Code status Bar. Once Veronica is activated, the conversation appears on the right sidebar. A recording icon is also added next to the label "Veronica listening", meaning that starting from that very moment, Veronica will continuously listen to the microphone and type everything into the chat. In addition, the text color changes to red 3.8. In a similar way, the user can click the same button to stop the listening. Moreover, there are three ways to make a memo, which will be described later on in this chapter. In the same way it is possible to delete a memo in some different ways 3.9. The multiple ways in which is possible to manage the documentation, together with the visual hints given by the interface, makes Veronica very important to lighten up and stimulate the user's to generate more data on their reasoning.

- **Automatic information retrieval from the web browser and subsequent integration into the code:** In order to obtain documentation and solutions to the problems that can be encountered during the development process, programmers frequently switch between the IDE and the browser. As a result, each time the programmer opens the browser and returns to Visual Studio Code, Veronica automatically collects the URL of the most recent websites visited and prompts the developers to link those URL to specific sections of the code. The focal point that serves this purpose is that the proposal to save the URL is given by Veronica. In that way the user can focus totally on his work, while this task is relieved by the chatbot itself that takes the initiative to advise the user. Thanks to this functionality that we wanted to achieve, the programmer is prevented from forgetting which documentation served as the source of their solution. In this way, also if another developer would access the code, the reasons behind the implementation choices would be clear. As a result, a memo is attached to a particular line of code or block of code, and it contains self-explanatory annotations and, eventually, links to the consulted documentation 3.13.
- **Visualize and navigate across the memos created in all the files:** To enable the developers to navigate through the memos they created, we decided to add a panel listing all the memos in the file currently opened in the editor, as shown in 3.14. In this manner, the developer has a consolidated view of the self-explanatory comments. Naturally, these memos do not aim at replacing the comments on the code file. While the comments in the code commonly follow some conventions and intend to provide technical explanations of how the code works, the memos account for explanations, in the developers' words, regarding why they implemented a given function in a certain way and which documentation inspired their decision.

3.3 General Architecture

Starting from the analysis conducted in the previous sections we identified all the design choices to be used in the tool to support the programmers' reasoning. The proposed solution is the chatbot Veronica, developed specifically for this thesis work. Citing the title of the thesis, Veronica is a multi-channel conversational agent developed to support the programmers and their reasoning. Multi-channel, because just like the property itself, the chatbot is able to communicate by the means of different channels through text or voice. The second part, conversational agent, denotes the nature of the application. This aspect is covered in part by the chatbot builder Dialogflow for routing user requests to correct responses thus allowing to perform the right conversational path. The bot architecture is the following 3.2:

- The core is given by the extension on Visual Studio Code, which contains all the code and the bigger part of the logic that the chatbot follows, together with the graphical settings.
- A middleware, in this case also known as Chatbot Builder that can handle both conversational logic and the multi-channel aspect.
- A Google Chrome extension that runs in background and is used to enable one of the functionalities of the chatbot.
- A local headless server that is functional to connect all the components and to enable the use of the Web API to perform the Speech Recognition.

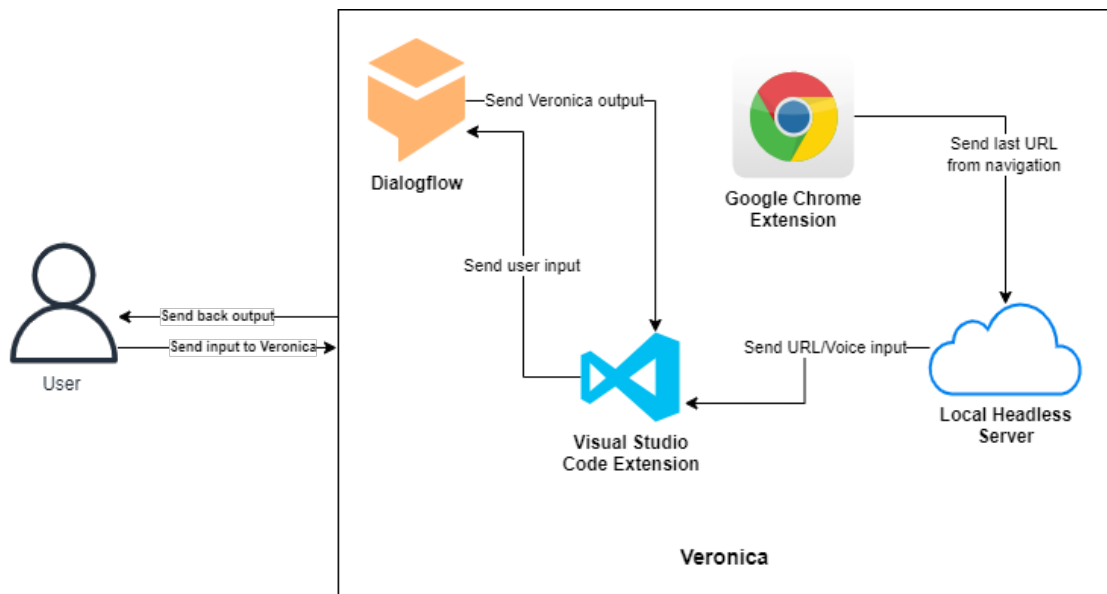


Figure 3.2: The high-level architecture of Veronica

Depending on how the user decides to give any input, the process followed by the tool varies a little. Just to give a brief description of the figure 3.2 and what each arrow means.

When a user writes a sentence in the chat, the process to generate a response is started from the Visual Studio Code extension. Specifically, the input is sent as a query to Dialogflow using its APIs. Then, inside Dialogflow a research is done to understand the intent of the user and to perform the associated action, if any, otherwise a default fallback intent is triggered. Once a response is selected, it is sent back and stored into the extension. Finally, the answer is reproduced vocally by the bot thanks to the Speech Synthesis library given

by the Web Speech API and concurrently printed in the chat inside the IDE to show a visual feedback.

If a vocal input is given from the user, the command is intercepted by the local headless server that starts the process to produce an output. Specifically, the server is listening continuously to the user and is able to translate every utterance into a “text” format thanks to the Speech Recognition library given by the Web Speech API. Then, the text is sent to the Visual Studio Code extension through a socket.io connection that has been established during the start of the components. Once the utterance is received inside the extension, the result is sent to Dialogflow like in the previous case through the APIs. Subsequently, the intent of the user is matched and the related response is sent back to the extension. Finally, the output from Veronica is reproduced both vocally and textually as before.

If the user that is using Veronica goes online using Google Chrome, his navigation is tracked by the extension, which has been created specifically for that browser. Each time a new URL is reached, the link is sent through socket.io to the local headless server that is continuously listening for new values. Once a new URL is received, the server takes it and then forwards it to the Visual Studio Code extension. When the user gets back to the extension, by opening it, it is notified by Veronica that subsequently asks him if he wants to save the last URL to any memo.

Talking about the functionalities included in Veronica, we opted for the ones described in the table 3.2. Those functionalities has been decided taking into account the difficulty of the implementation and the actual usefulness that we wanted to test in the last phase of the thesis, during the user evaluation.

| Functionality | Description |
|-----------------|--|
| Memo creation | It allows to create a new memo inside the interface on the actual file. |
| Memo deletion | Delete a previously created memo from the file. |
| URL tracking | Track the user navigation and asks if he wants to save the URL inside a memo. |
| Memo navigation | Move directly to one of the existing memo, by selecting it on a specific panel in which are all tracked. |

Table 3.2: Proposed functionalities in the chatbot Veronica.

3.4 Graphical Interface

An important aspect that has gone through many phases is the graphical interface of Veronica. At the beginning, a draft of the interface has been drawn on Balsamiq [40]. In the draft were shown all the functionalities thought for the prototype together with a graphical example 3.3 3.5. However, most of the designed drafts had to be changed during the actual implementation. The environment of Visual Studio Code has a lot of constraints in terms of item creation and graphic freedom, so the design of our drafts changed accordingly to those constraints. In general, the developer has to adapt to what already exists or ask for a new implementation in the community page [41]. If the new request in the community post receives a lot of feedback from other users, the idea is taken into account from the Visual Studio Code developers to be implemented inside the environment. Given the specific functionalities that we wanted to implement and the time span of this thesis work, we couldn't afford to wait for a new implementation after one of our request. Hence, we decided to change the outcome of our initial draft and use what was already present. The first decision was related to the chat position inside the tool 3.4. The default icons and layouts weren't fitting enough to our idea, so we decided to implement a Webview to customize the chat interface. The available spots where the following in figure 3.4:

1. **Primary sidebar** This is the default place where every new webview is placed. The most common and used sidebar, however it is usually used to display tree structures such as directories or Github Branches. Usually is closed or kept small by the users, so we decided to not pick this place.
2. **Panel** The common place where the terminal and the debug console are placed. Usually the panel section is used for those addons that have a supportive feature. Also in this case the decided to not use this part of the IDE, mainly because of its graphical structure.
3. **Secondary sidebar** This part of the IDE is usually underutilized unless the user moves something there manually. The dimension is closer to what we imagined for a chat-like interface, moreover it is possible to close it whenever we want. In the end we opted for this place for our interface.

It is also important to point out that, in any case, the chat can be manually moved by the user where he wants to. However, the graphic is designed specifically for the secondary sidebar and the outcome in case of a movement could be different from the attended result. The final result is given by the figure 3.6, here we decided to opt for a combination of colors that recall the main Visual Studio Code palette together with a mobile chat-like design.

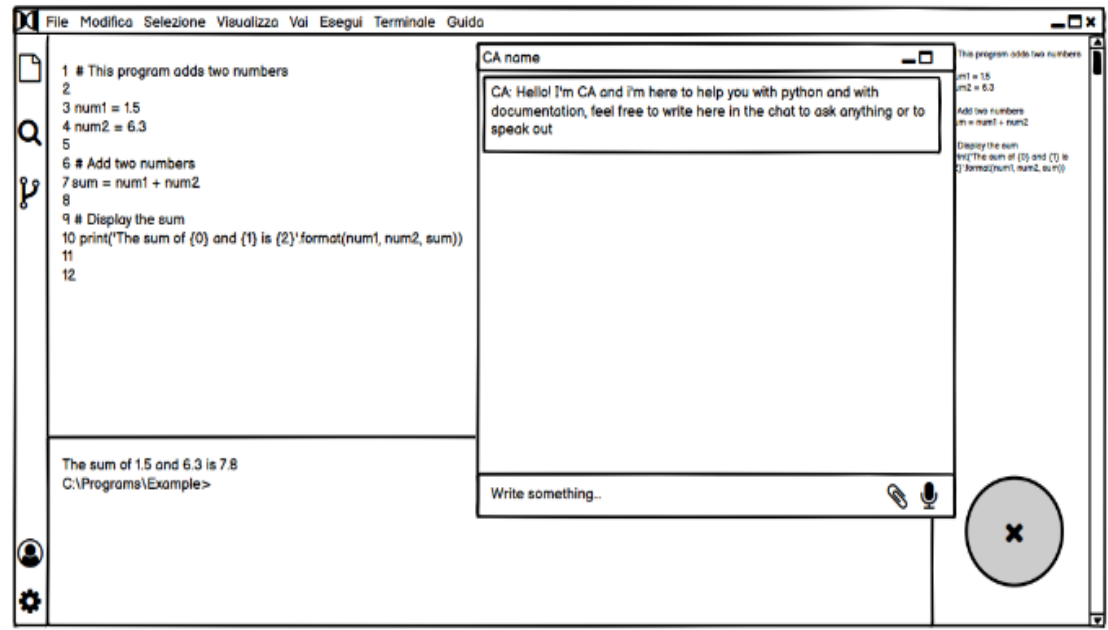


Figure 3.3: Initial draft of the interface when starting Veronica

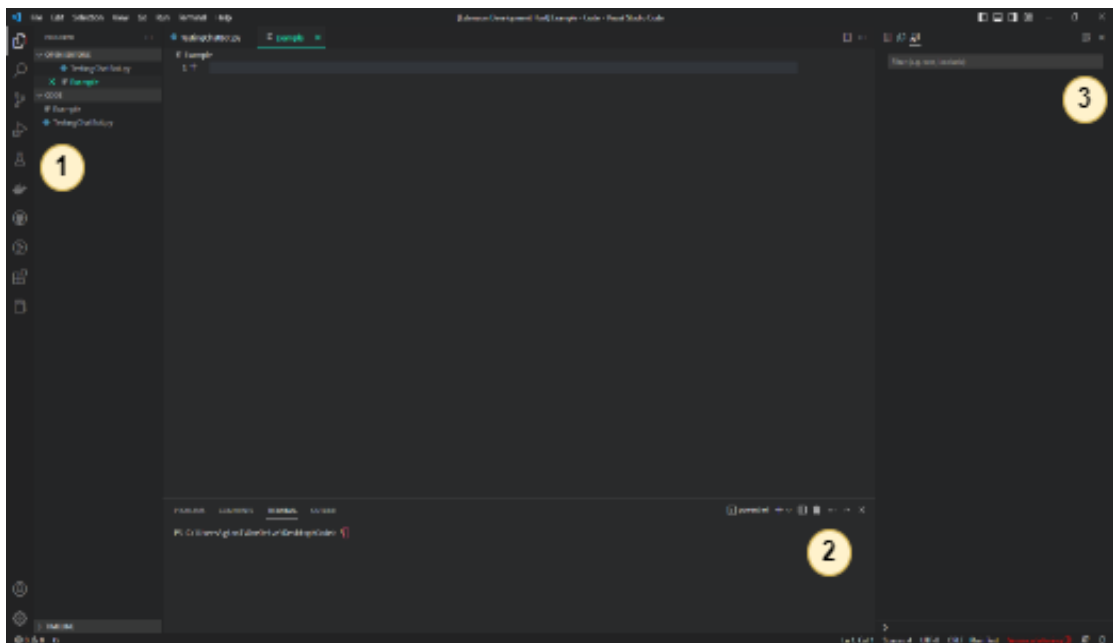


Figure 3.4: Positions in which the chat can be placed

The second important decision to take was concerning the display of the bot

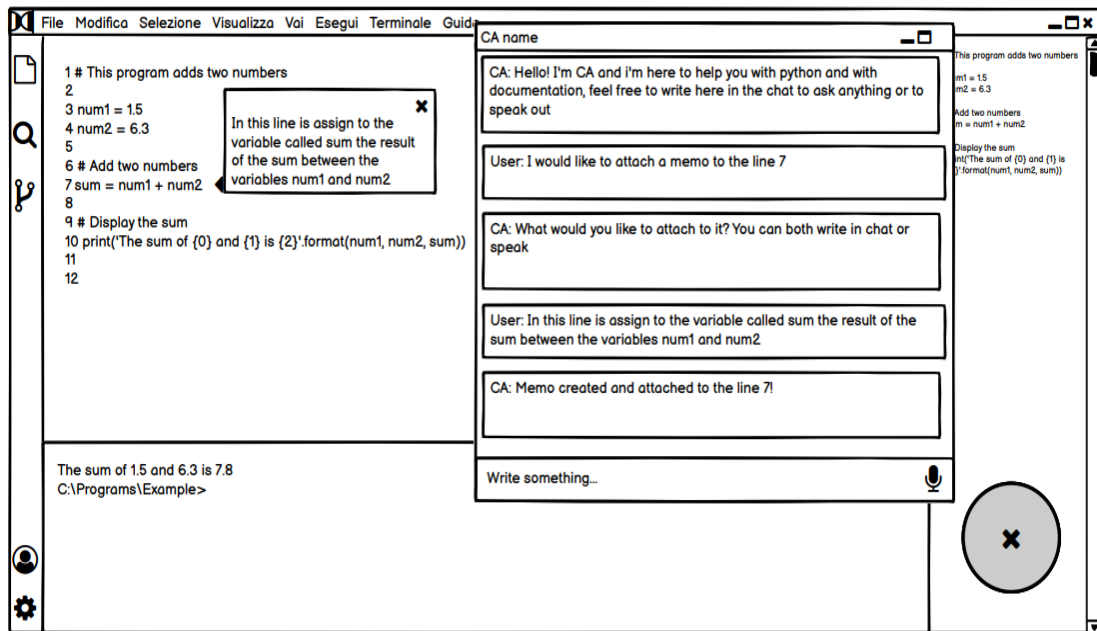


Figure 3.5: Draft of the memo creation by chatting with Veronica

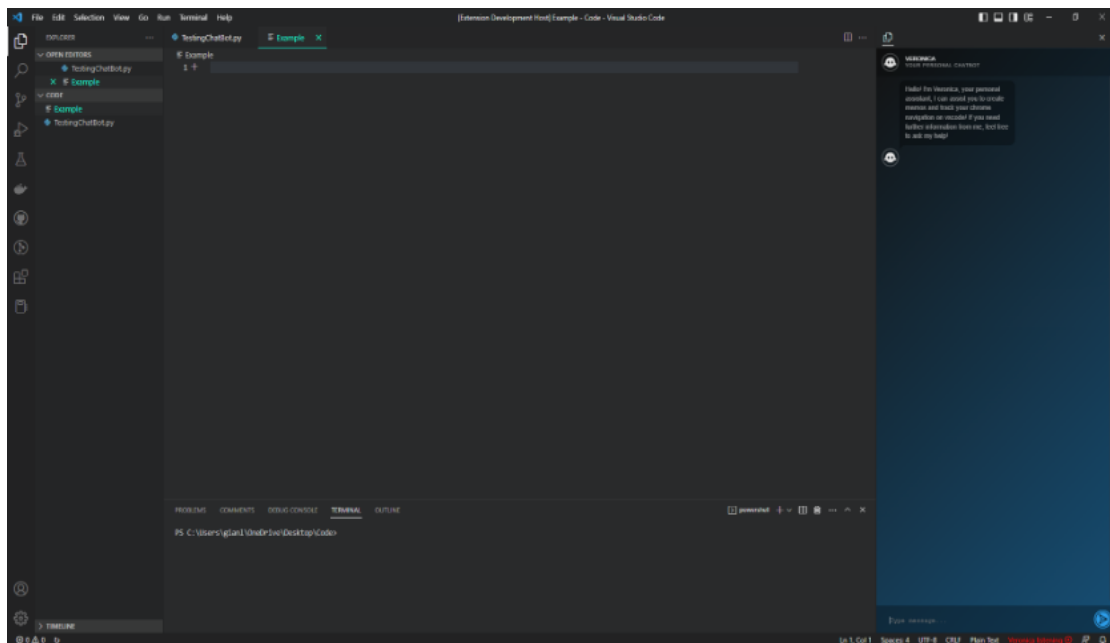


Figure 3.6: GUI after the extension is started

state. As stated before, Veronica can be in a state of active and continuous listening.

However, it is also possible to disable the listening. Differently from our starting draft, where a floating button where designed, we had to opt for another solution. We placed a small button in the bottom bar, together with a text describing the actual state of Veronica. The result can be seen in the figures 3.7 3.8.

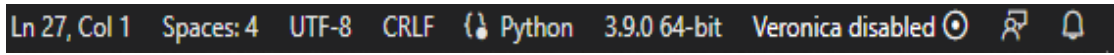


Figure 3.7: Veronica’s status button that notifies that the listening is disabled

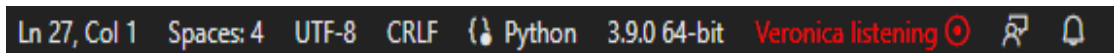


Figure 3.8: Veronica’s status button that notifies that the listening is enabled

3.5 Veronica’s functionalities

In this section we will go through all the functionalities implemented, together with some images to show how they are designed inside Visual Studio Code. The functionalities available in the tool are the following:

- The possibility to create a memo directly from the interface or using the chatbot, both via text and voice (**Design goal number 1**).
- Deletion of the memos created with the tool, also in this case is possible to delete them manually from the graphical user interface or thanks to the help of the chatbot (**Design goal number 1**).
- Tracking of the user navigation to enable the chatbot to be aware of when the user goes online to search for specific information while he is programming (**Design goal number 2**).
- A feature that enables the navigation through the memo created in the code, it is possible to click on each entry and the correlated file will be opened and focused on the selected memo (**Design goal number 3**).

The objective of the the tool is to improve the quality of life of the programmers and their reasoning concerning the creation of documentation with the application of the functionalities implemented. Moreover, being this an exploratory work, we would like to understand if the tool can be taken into consideration for future applications or further improvements.

3.5.1 Memo creation

This is the first functionality designed for Veronica. The initial idea was to help the programmer on taking a new annotation of the code that he was writing by the means of the chatbot. However, after our researches the decision was to enable that feature and all the following ones both from the conversational agent and the graphical user interface. During the creation of a new annotation, Veronica guides the user over each step that needs to be performed, in order to gain enough information to complete the task. A scaffolding strategy has been adopted like in the educational domain [35]. Thanks to that approach, it is possible to retrieve each information without giving to the user a bad feeling during the conversation. Each step has a specific goal and some default fallback answers has been defined to help the user to not get stuck inside a conversational pattern. Let's analyze three possible scenarios that could happen during a memo creation with the help of Veronica:

The user says: "I want to create a memo on the line number fifteen", in this case Veronica understands that the memo has to be created on a "line", specifically the number "fifteen".

The user says: "I want to create a memo", from that simple utterance Veronica is not able to understand where the memo should be created, so it starts to help the users to fill those missing data. At that point Veronica would say: "You want to attach it to a line or a block of code?" to induct the user to specify where this memo should be placed. Finally, after the user's answer, Veronica would ask "To which line (or block of code)", to understand the exact location of the memo 3.9.

The user says again "I want to create a memo", but this time, instead of following the rest of the conversation without problem, it answers in an unexpected way to Veronica's questions. Since the scaffolding pattern fails, a fallback answer like: "I didn't get what you mean, could you please say that again?" is given by Veronica to the users. Then, if the users gives as a new input what Veronica expects to receive, then the conversation can continue, otherwise some keywords such as "exit" or "quit" are given to the user to quit the conversation and get unstuck.

Finally, if the user doesn't want to create the memo with the help of Veronica, there is always a secondary way, which is given by the graphical user interface (GUI) itself 3.10. In this way it is possible to make a new annotation without having to comment manually inside the code.

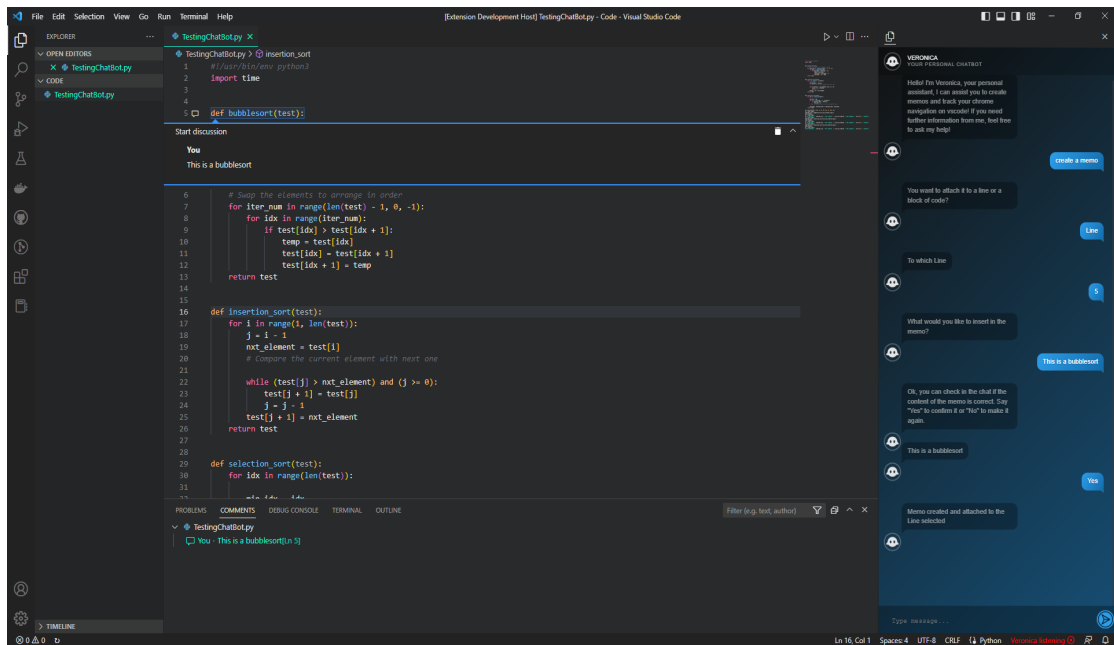


Figure 3.9: Example of the creation of a memo thanks to Veronica

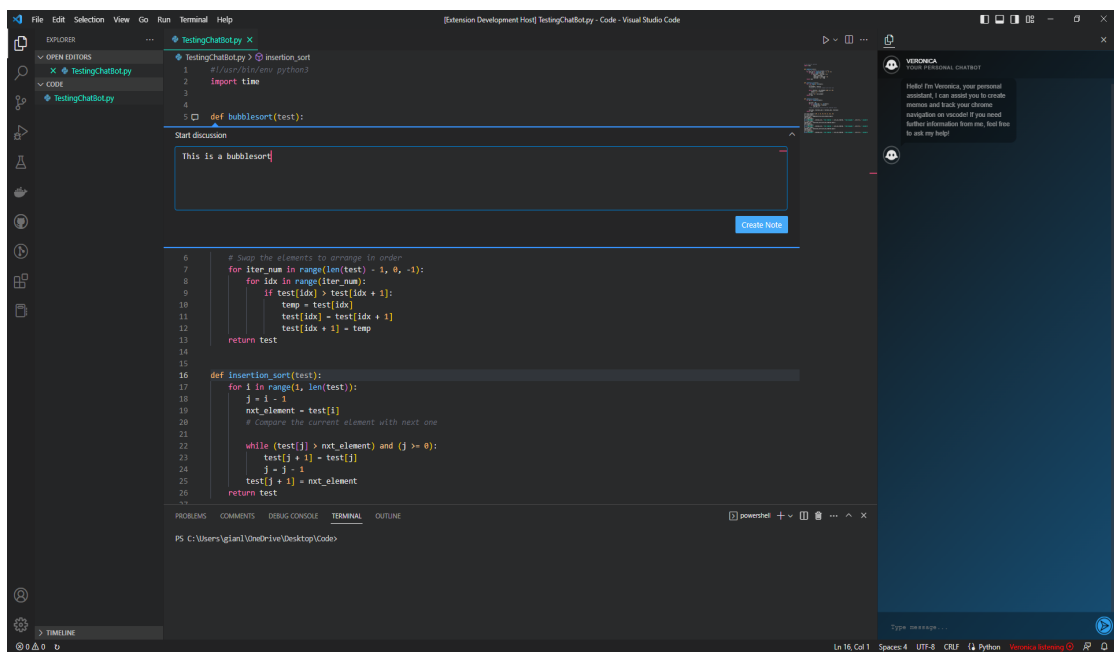


Figure 3.10: Example of the creation of a memo from the GUI

3.5.2 Memo deletion

This functionality is the complementary of the previous one. Like before, initially the programmer could only delete his annotations only by using the voice or the text channel with Veronica. Similarly to the complementary functionality, Veronica guides the user over each step for which is requested a specific element in order to complete the task smoothly. The scaffolding strategy has been used also in this case. However, differently from the creation of a memo, in this case the importance to retrieve the intent and the entities pointed by the user was more sensitive. For example, a user could try to delete a memo in an empty position or in a wrong file. Because of that we implemented these functionalities by keeping in mind a possible human error. Let's analyze the possible conversational pattern that could happen during a memo deletion with the help of Veronica:

The user says: "I want to delete the memo on the block of code between number eleven and seventeen", in this case Veronica understands that the memo has to be deleted on a "block of code", specifically the one between the lines "eleven and seventeen".

The user says: "I want to delete a memo", from that simple command Veronica is not able to locate the memo that has to be deleted, so it starts to help the users to fill those missing data. At that point Veronica would say: "On which line does the memo starts?" to induct the user to specify where is the annotation that has to be removed. Lastly, after the user's input, Veronica is able to understand the location of the memo to be deleted and can perform the removal 3.11.

The user performs the same actions from before, the only difference is given by the input of the location of the memo. The location given by the user is empty, hence no memo are available there. In this case, Veronica answers that "There are non memo available to delete on that line" 3.12.

It is important to notice that, independently from the scenario, Veronica is designed to always give a feedback to the user. The utilization of a continuous feedback is crucial, both to help the user in certain situation in which it might get stuck or to notify him about the success of a defined operation.

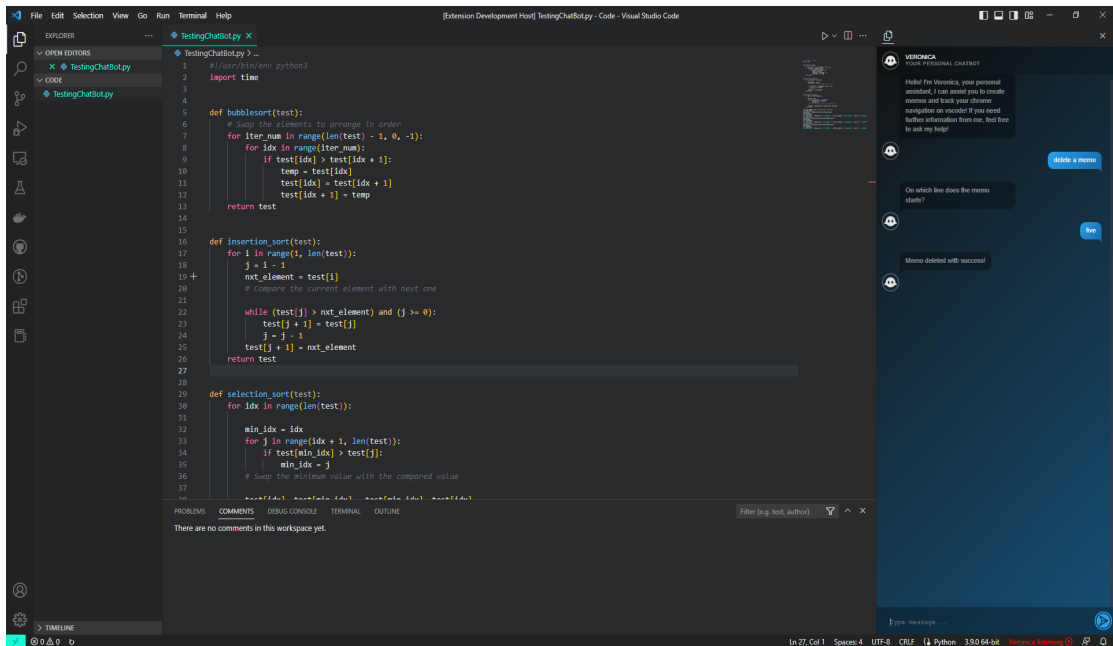


Figure 3.11: Example of the deletion of a memo performed by Veronica

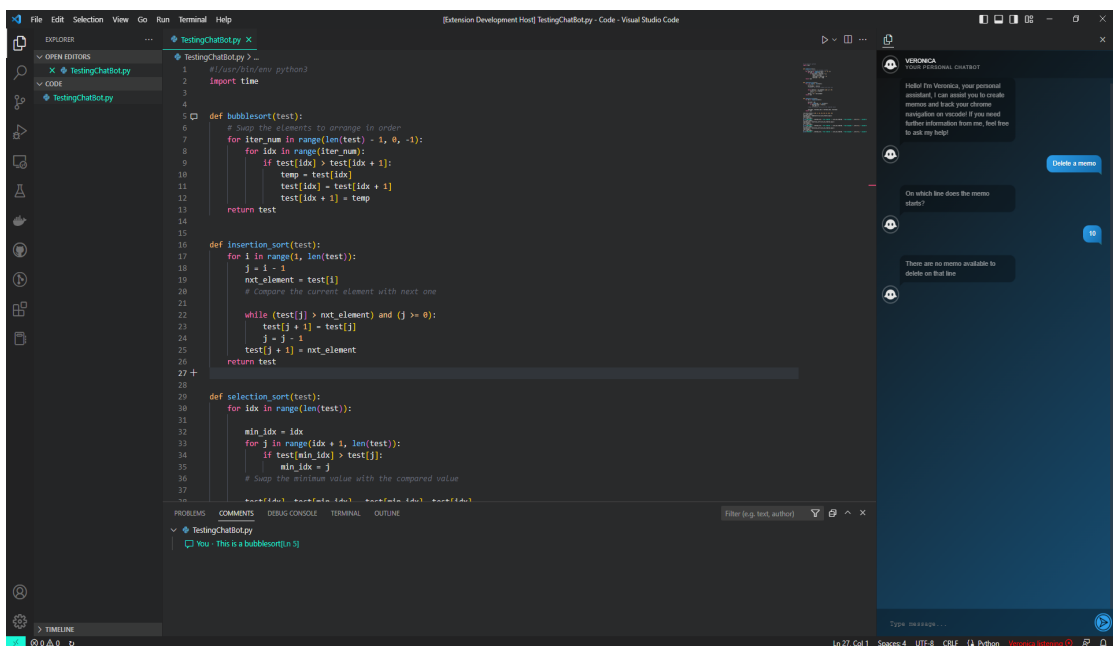


Figure 3.12: Example of a memo deletion failed

3.5.3 URL tracking

This functionality has been implemented to enable Veronica to take the initiative and ask a user's feedback after his navigation online. From what has been shown in the figure 3.2, the tracking is made from the Google Chrome extension, outside of the Visual Studio Code extension itself. This functionality is triggered when a user decides to navigate online. The steps executed are the following 3.13:

- The user open Google Chrome, navigates to a specific page in search for something.
- Once he finds it, he gets back to the programming IDE.
- Veronica detects that the user searched something and after his return to the IDE says something like: "I saw that you navigated online, do you want to save your last URL?", by sending in the chat also the URL she is talking about.
- At that point, the user can choose to save it, leading to the memo creation functionality. Instead, if he doesn't want to save it, the conversation is closed with an ending feedback and the choice memorized.

In this sense, if another developer opens the code file, he would understand the reasoning behind the implementation decisions just by checking the memo or the associated URL. Therefore, a memo is pinned to a specific line or portion of code, and its content consists of self-explanatory annotations and, eventually, links to the consulted documentation.

3.5.4 Memo navigation

The navigation has been implemented to allow whoever gets his hands on the code to see immediately all the documentation that has been generated and reach them without difficulties. It consists of a panel, placed at the same level of the terminal and the debug console. Inside the panel are inserted all the memo created and it is organized with a tree-like configuration 3.14. All the memo are divided by file, and for each one is shown the start of the text contained and the line (or block of code) in which is placed. By clicking on a specific entry is possible to navigate directly on the file containing it and have a view of the opened comment. The decision to implement this comment panel, was mainly to ease the tracking of all the documentation inside a single source. Moreover, if another programmer works on the same project, he could be able to gather fast all the reasoning of another person or where did his implementations came from.

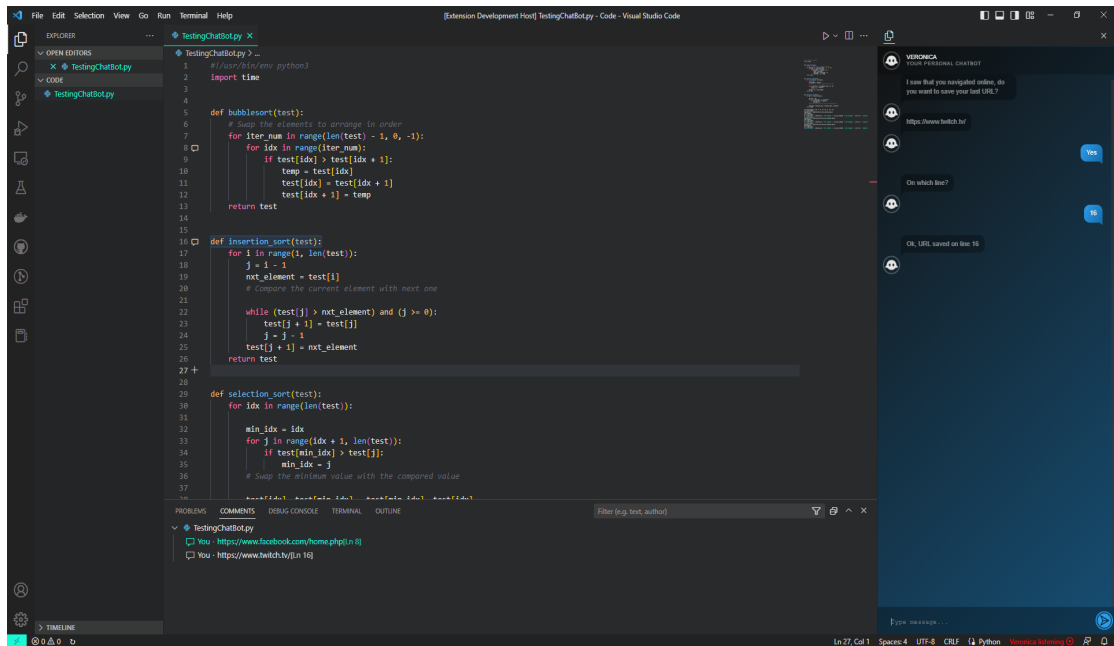


Figure 3.13: Snapshot of a memo creation after that a navigation online has been detected

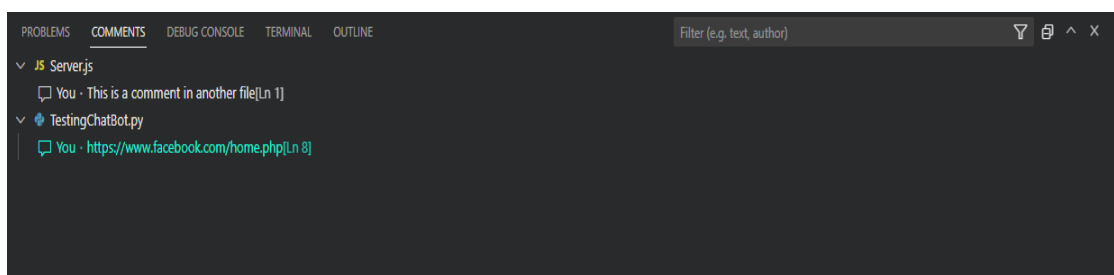


Figure 3.14: The panel in which all the documentation created is stored, it is possible to navigate through the memo by clicking on each entry.

Chapter 4

Implementation

This chapter contains the details of how the application works from a technical perspective. The description follows the flow of a typical user experience. Moreover, an in-depth description of the various components is given.

4.1 Components

Here we give a brief introduction to the components that have been used in the tool, that will be analyzed later on in this chapter:

- **Visual Studio Code:** This is the IDE chosen in which we developed the conversational agent. This is the component in which most of the work has been done.
- **Dialogflow:** The natural language understanding platform that we implemented for Veronica. This is the key component to understand the user intents.
- **Google Chrome extension:** A Google Chrome extension that runs in background and is used to enable one of the functionalities of the chatbot (URL tracking).
- **Local server:** A local headless server that is functional to connect all the components and allow communication between them.

The whole project has been managed through Visual Studio Code. Ideally, the goal is to find a way to condense every component inside the Visual Studio Code extension if this exploratory work returns good feedback in terms of interest and utility. At the moment of the writing, to make the tool work is necessary to install separately the Local server, the Google Chrome extension and the Visual Studio Code extension. By doing so, the tool will work locally inside the user's computer.

In the figure 4.1 is possible to see more in detail how each component is connected and from which portion of code. Each file will be then described further in its own subsection.

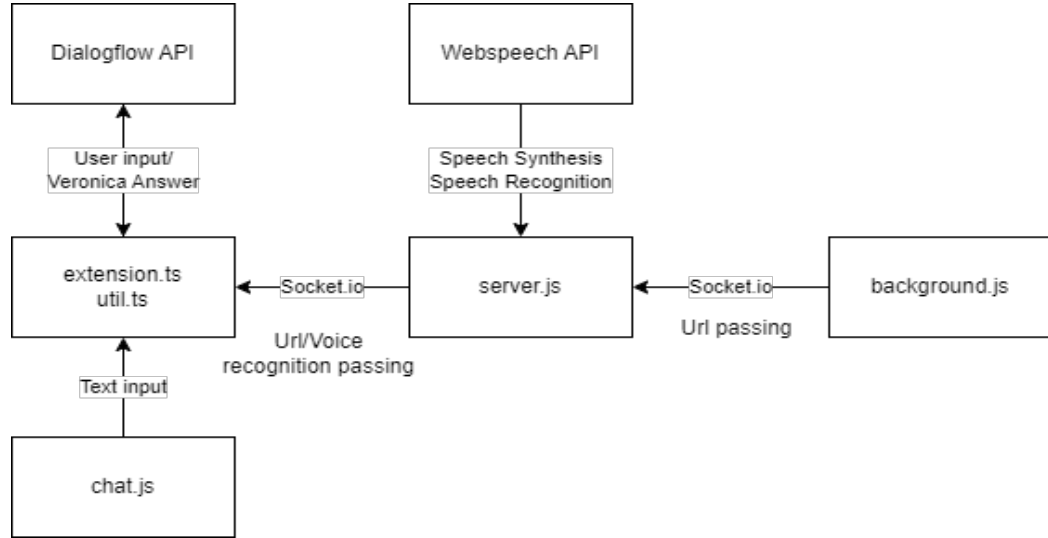


Figure 4.1: Veronica’s detailed architecture

4.1.1 Visual Studio Code

Visual Studio Code (VS Code) is a lightweight, cross-platform source code editor developed by Microsoft. It has become increasingly popular among developers due to its flexibility and powerful features. VS Code is designed to be user-friendly and highly customizable. It includes features like syntax highlighting, code completion, debugging, version control, and code refactoring. One of the main features that sets VS Code apart from other editors is its extensions library. This allows developers to customize their editor with add-ons that add new functionality, themes, and language support. There are thousands of extensions available for VS Code, ranging from hinters and debuggers to snippets and icons. Veronica has been designed as an extension for VS Code.

Visual Studio Code is the core component used to implement Veronica and in which the most part of the code is located.

Extension Anatomy

Here is given a brief explanation of how a Visual Studio Code extension should be implemented. Let’s take a closer look at a sample’s source code and see how a general extension is structured.

```
├── .vscode
│   ├── launch.json
│   └── tasks.json
├── .gitignore
├── README.md
├── src
│   └── extension.ts
├── package.json
└── tsconfig.json
```

From the tree, the most important files are:

- **Package.json:** This is the extension manifest file. Each VS Code extension must have a “package.json” as its Extension Manifest. The “package.json” contains a mix of NodeJS fields such as scripts and “devDependencies” and VS Code specific fields such as “publisher”, “activationEvents” and “contributes”.
- **Extension.ts:** This is the extension entry file. The extension entry file exports two functions, “activate” and “deactivate”. The function “activate” is executed when a registered Activation Event happens. “deactivate” gives a chance to clean up before the extension becomes deactivated. For many extensions, explicit cleanup may not be required, and the deactivate method can be removed. However, if an extension needs to perform an operation when VS Code is shutting down or the extension is disabled or uninstalled, this is the method to do so. In our specific case, the “deactivate” function has not been used, while the “activate” one has been used largely.

Most of the other files are either configuration files to adjust the setup or files to define dependencies and version control.

Extension capabilities utilized

Here the core functions and capabilities used for Veronica will be analyzed more in-depth. All the portions of code that will be used are part of the “extension.js” file unless specified differently.

- The first function, from where everything is created, is the “activate” one as previously cited. Inside this function all the elements of the extension are initialized, which are the status bar button to activate Veronica 3.7 3.8, the comment controller to manage the memo creation and deletion, the chat view provider that manages the interface of Veronica and all the commands that are triggered during the use of the agent. Moreover, the array “context.globalState” that contains all the existing memos is checked and every annotation is placed

in the right file and line. The format of the element inside the array is composed by an object containing all the needed data to recreate the memo. The main quality of this array is that it maintains the data stored inside it even when the extension is closed, in this way we were able to not utilize any database. From the code it is possible to see how the comments are settled back up when a user restart the program. As said previously, this portion of code is inside the “active” function which is ran at the beginning of the code and everytime a new command is executed. At the beginning the boolean “threadLoad” is checked to run this code only at the first activation each time. Then, all the values stored in the array are taken for each entry to repopulate the memos thread. The variables refers to the starting and ending line of the memo, the file in which they are contained, the context of the message and the reference to the comment controller in which the memos have to be placed.

```
if(!threadLoad){
    nElement = context.globalState.get<number>("nElement", 0);
    for(i=0; i<context.globalState.get<number>("nElement"); i++){
        newPos1 = vscode.Position(context.globalState.get<any>
            (context.globalState.keys()[i]).newPos1, 0);

        newPos2 = ...

        range = new vscode.Range(newPos1, newPos2);

        userUri = context.globalState.get<any>
            (context.globalState.keys()[i]).userUri;

        body = context.globalState.get<any>
            (context.globalState.keys()[i]).body;

        thread = commentController.createCommentThread
            (vscode.Uri.file(userUri), range, []);

        newComment = new NoteComment(body, vscode.CommentMode.Preview, { name:

        thread.comments = [...thread.comments, newComment];

        myThreads.set(thread.range.start.line, thread);
    }
    threadLoad = true;
}
```

- Talking about the comment controller, it is utilized to create inside the IDE the interface to manually generate the memos. Moreover, it's called by all the commands defined in the “activate” function to generate all the annotations that are not created manually. The commands related to the comments are four: create a note, delete a note, save the created note to the “context.globalState” array and the one to remove them from the array once deleted as last.
- The “chatViewProvider” is utilized to manage the functionalities related to the chat interface, specifically to retrieve the inputs given by the user to forward them through the Dialogflow API to the identify the user's intent. Additionally, the HTML structure of the chat is defined here. It's important to notice that to let communicate the “chat.js” file which is the one that manages the chat itself with the “chatViewProvider” some specific API given by VS Code have to be used.
- The most used function related to the direct communication user-Veronica is the one that has been utilized to send the input text given by the user to Dialogflow. Thank to that function it is possible to analyze the user intent and answer accordingly, since the result is returned in a text format to be printed on the chat subsequently. In the portion of code below, the main pieces have been highlighted, Specifically, the text input is inserted into a JSON format variable, then the Dialogflow API is called to request the detection of the intent. As it is possible to see, the function is asynchronous, in this way annoying behaviors related to the communication with the bot, that otherwise should await for a response, are reduced.

```
async function userInput(projectId = 'project-name',value: text)
{
  const request = {
    session: sessionPath,
    queryInput: {
      text: {
        text: value,
        languageCode: 'en-US',
      },
    },
  };
};
const responses = await sessionClient.detectIntent(request);
const result = responses[0];
```



```
    return result;
}
```

- Once the intent detection returns a result from the previous function, an answer from Veronica is received. Depending on the type of intent there are different tasks to be performed within the code. As it is possible to see from the code below, for each different case a unique action is performed. There are way more cases inside this function, for ease of reading just some of them have been included.

```
public async actionToDo(result: any){
  if(result.queryResult.cancelsSlotFilling){
    return;
  }
  switch(result.queryResult.intent.displayName){
    case'Memo Creation':
    {
      if(result.queryResult.allRequiredParamsPresent){
        ...
        if(this.elem === "Block"){
          ...
        }else{
          ...
        }
      }
    }
    break;
  }
  case'Memo Context':
  {
    this.userText = result.queryResult.queryText;
    this.caUserMessage(result);
    break;
  }
  case'Memo Confirm':
  {
    vscode.command.executeCommand("saveCommentState", ...);
    break;
  }
  ...
}
```

- All the graphical and live updates of the chat interface are managed from a different file, which is the “chat.js”. Depending on the way in which the input is generated (from the user or from the chatbot), the listener receives a different input that defines the origin of the message. Given the origin, the right class name between two different “iDiv” is chosen, generating the message in the right color and position in the chat. This is the default behavior of the chat, which covers most of the possible interactions. Some exceptions are made from specific messages that has to be shown from Veronica to the user (i.e.: the input given for a memo or the URL during the navigation), in those cases the origin is different from the format that the message should have, hence different functions manages those cases. Finally, in the file is present a small function to update automatically the scroll bar, to always show the most recent messages.

The core functions has been analyzed in the previous list. However, it is important to point out that many others are present in Veronica’s code. Those are used to manage specific cases or define the agent behavior depending on the biggest possible scenarios (that diverge from the common use-case defined) that we tried to handle.

4.1.2 Dialogflow

Dialogflow is a natural language understanding platform that makes it easy to design and integrate a conversational user interface into your mobile app, web application, device, bot, interactive voice response system, and so on. Dialogflow can analyze multiple types of input from the user, including text or audio inputs (like from a phone or voice recording). It can also respond in a couple of ways, either through text or with synthetic speech [42]. There are two versions of Dialogflow, the essential one (ES) which is the one used for Veronica and the CX one, which is the premium version and has way more capabilities used for more complex agents. Dialogflow has been the middleware used to handle both conversational logic and the multi-channel aspect. From the main page it is possible to use the Graphical User Interface (GUI) to setup all the intents needed.

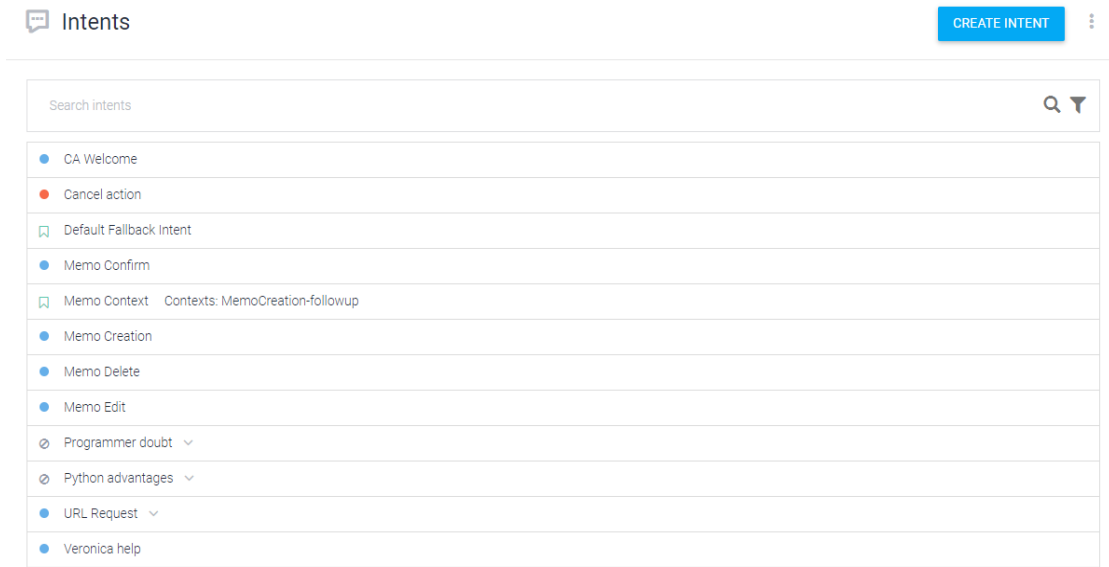


Figure 4.2: The list of intents defined for Veronica

| Intent | Description |
|----------------------|--|
| CA Welcome | This intent is triggered at the start of the extension. Veronica briefly introduces itself and explains on an high-level what it is able to do. |
| Memo Creation | Started when the will to create a new memo is detected. It contains intermediate questions to fill the required fields and other fallback intents to proceed with the conversation. |
| Memo Delete | Similar to the previous one, it is activated when a user wants to delete and existing memo. Also here are given some intermediate scaffolding questions mandatory to locate the memo to delete. |
| Memo Confirm | Once the memo that has to be created is typed or dictated, Veronica shows it in the chat and asks for a confirmation. Depending on the user's answer this intent is triggered to give a visual feedback about the creation by inheriting the values previously given to the chatbot. |
| Veronica Help | Simple intent enabled by a request of help from a user. Veronica explains more in detail what she can do and tries to hint better how each functionality can be performed. |
| URL Request | Proactive intent which is activated thanks to the communication between the Google Chrome extension and the Visual Studio Code one. When a user navigation is detected, once he gets back to the IDE, the Google Chrome extension instantly sends an hard-coded message to trigger this intent and proceed with the conversation to save the last URL visited. |

Table 4.1: Description of the main intents.

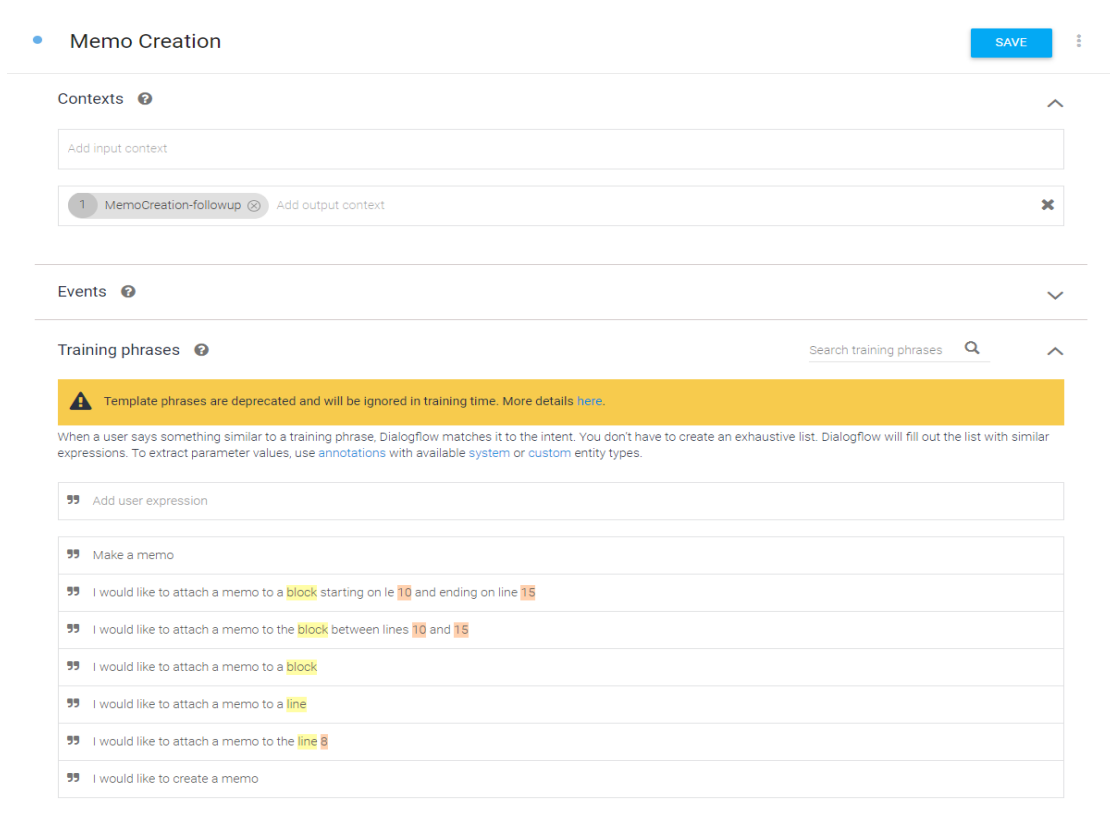
From the figure 4.2 and the table 4.1 it is possible to see the list of intent designed for Veronica, together with their priority (the colored dots) and the initially discarded ones. All the intents are defined to catch a specific intention of the user, ranging from the needs of help to the creation of a new memo. Going more in depth, inside a single intent we can see all its structure 4.3 4.4:

- First, we have the **Contexts** field, which is divided into input and output contexts. The input ones are used to define a sort of “entry rule” to the intent, meaning that this conversational pattern will be opened after certain action (i.e.: another intent that has the same context as an output one is triggered before). On the other hand, the output context defines the conversational background in which the user and Veronica are, together with the context also the entities and stored values are carried on. The small number near to the context defines its “time to live”, meaning that if the number is “x”, after “x” interactions the context will be removed from the stack 4.3.
- Then there is the **Events** entry. Here it is possible to define alternative ways to trigger the intent without textual or vocal inputs. For example, if the “Welcome” event is settled up, every time that Veronica is executed the correlated intent will be triggered without any user’s utterance detected 4.3.
- The **Training phrases** is an important field. It is where all the trigger phrases are defined by the developer. While there is no limit to the amount of phrases that can be inserted inside it, we decided to insert a small amount of them to test the agent and the self-training feature, meaning that Veronica can be improved further with more phrases in the future. Once a user’s utterance is detected, Veronica researches the matching intent by watching at the training phrases or by filtering the important keywords. Usually, the most valuable keywords are the recurring ones in most phrases or the ones associated to entities (the colored words) 4.3.
- Thanks to the use of **Action and parameters** it is possible to define a scaffolding procedure to fill some data that are required after that a specific intent is triggered. From the figure 4.4 it is possible to see that each value, related to a specific entity, can be defined as required or not. If a value is required and at the same time is missing from the phrase that triggered the intent, the associated prompt is executed. In that way Veronica tries in a more specific way to receive that value from the user. Once all the required parameters have been provided, Veronica proceeds with the final response.
- from the **Responses** field it is possible to define the final answer given by Veronica. To increase the interactivity and reduce the redundancy of the conversations, it is also allowed to setup multiple answers that are randomly

selected by the agent. Moreover, it is possible to define this interaction as the end of the conversation, in that way most of the resources stored such as the context can be deleted 4.4.

- Finally, we can see the **fulfillment** options in which is possible to decide if we want to enable the webhook calls for slot filling or for the intent 4.4.

By mixing all those functionalities together, we were able to define specific conversational pattern to manage and implement the functionalities defined in the Design chapter.



Memo Creation SAVE

Contexts ?

Add input context

1 MemoCreation-followup ? Add output context ✕

Events ?

Training phrases ? Search training phrases Q

⚠ Template phrases are deprecated and will be ignored in training time. More details [here](#).

When a user says something similar to a training phrase, Dialogflow matches it to the intent. You don't have to create an exhaustive list. Dialogflow will fill out the list with similar expressions. To extract parameter values, use [annotations](#) with available [system](#) or [custom](#) entity types.

” Add user expression

” Make a memo

” I would like to attach a memo to a **block** starting on le **10** and ending on line **15**

” I would like to attach a memo to the **block** between lines **10** and **15**

” I would like to attach a memo to a **block**

” I would like to attach a memo to a **line**

” I would like to attach a memo to the **line 8**

” I would like to create a memo

Figure 4.3: The list of intents defined for Veronica

Action and parameters

Enter action name

| REQUIRED | PARAMETER NAME | ENTITY | VALUE | IS LIST | PROMPTS |
|-------------------------------------|----------------|--------------|-------------|-------------------------------------|--------------------|
| <input checked="" type="checkbox"/> | element | @Element | Selement | <input type="checkbox"/> | You want to att... |
| <input checked="" type="checkbox"/> | number | @sys.number | Snumber | <input checked="" type="checkbox"/> | To which Seleme... |
| <input type="checkbox"/> | Enter name | Enter entity | Enter value | <input type="checkbox"/> | - |

[+ New parameter](#)

Responses

DEFAULT +

Text or SSML Response

- What would you like to insert in the memo?
- Enter a text or SSML response variant

[ADD RESPONSES](#)

☐ Set this intent as end of conversation

Fulfillment

☒ Enable webhook call for this intent

☒ Enable webhook call for slot filling

Figure 4.4: The list of intents defined for Veronica

4.1.3 Local Node.js Server

This is the local server that has been utilized to take advantage of the WebSpeech API, which by default are not utilizable inside the Visual Studio Code extension since it has no internal connection. Since the main functionality of the server is to be a bridge between the other components and to perform the SpeechRecognition, the implementation is quite simple. The server has been developed in Node.js. The component is composed by the following elements:

- A **server.js** file in which the server itself is started. At the start all the needed dependencies are executed synchronously. The port on which the server awaits is defined manually.

```
const port1 = 3001;
const server = app.listen(process.env.PORT || port1, () =>
{console.log('Server listening on port %d in %s mode',
server.address().port, app.settings.env);
});
```

Then with Selenium a local headless Chrome instance is started.

```
(async function headlessCreation() {  
  let driver = await new Builder()  
    .forBrowser('chrome')  
    .setChromeOptions(new chrome.Options().setUserPreferences({  
      "profile.default_content_setting_values.media_stream_mic":1  
    }))  
    .addArguments("--headless=chrome")  
  )  
  .build();  
  await driver.get('http://localhost:3001');  
}())
```

From the function is possible to see that we decide the type of browser (Chrome), that it has to be headless, the URL and port in which it listens (localhost:3001) and that by default the microphone of the PC is enabled. An headless instance represents a normal browser (like Google Chrome), the main difference is that no pages are shown, since it works in background. In that way the SpeechRecognition described in the file below is in place without the necessity to have a permanent page on the browser opened.

- A **package.json** file in which all the dependencies user are installed, specifically we used express to start the server, selenium to run the headless browser and socket.io to have a communication channel between the components.
- A **script.js** file in which all the SpeechRecognition is implemented. Here a brief portion of the code:

```
const SpeechRecognition = window.SpeechRecognition ||  
  window.webkitSpeechRecognition;  
const recognition = new SpeechRecognition();  
recognition.lang = 'en-EN';  
recognition.continuous = true;  
recognition.interimResults = false;  
const socket = io();  
recognition.start();  
recognition.addEventListener('result', (e) => {  
  let last = e.results.length - 1;  
  let text = e.results[last][0].transcript;  
  socket.emit('chat message', text);  
});
```

```
});  
  
...  
  
recognition.addEventListener('end', (e) => {  
    recognition.start();  
});
```

As we can see from that snapshot of the code, the SpeechRecognition APIs, which are part of the WebSpeech API library is started inside the window. The SpeechRecognition works only within a window, that's the reason that led us to start the headless browser. Once started it is possible to decide the language to detect, if the recognition has to be continuous or not and if we want to store interim results. Then a list of event listener are present to define what to do in specific cases, we included in the snapshot the two most important. The “result” one is the event listener that transcribes the user's utterance and after converting it into a text, send it via socket to the Visual Studio Code extension. The other one is the “end” event listener that had to be added to ensure the permanent listening of the bot. By default in Google Chrome the listening of the microphone is designed to be interrupted after some minutes of silence, so we defined this event to restart the listening in case of prolonged silence by the user. All the other events are less important or have been implemented to manage some specific cases.

4.1.4 Google Chrome Extension

Google Chrome as browser has been chosen for two specific reasons. First, because it is the most utilized accordingly to the latest revelations made in January 2023 [43], having 66,39% of the users utilizing it. Second, we decided to remain in the same domain used for the other components such as Dialogflow.

Usually, a Google Chrome extension is composed by three different categories of script:

- **Content Script:** This kind of script is utilized to modify directly the web pages.
- **Background script:** Also known as service worker (depending on the manifest version utilized). As the name suggests, this type of script is opened in background and doesn't require a graphical interface. Its main role is to manage all the events in the browser that are external to the extension.

- **Options script:** A Google Chrome extension that runs in background and is used to enable one of the functionalities of the chatbot.

The extension of Google Chrome for Veronica is really small. It is composed by a manifest file in which the service worker is defined together with some permissions related to the web navigation and the Chrome tabs. The other important file is the one concerning the service worker itself. We can see from this brief extract of code the main functionality of the “background.js” file:

```
chrome.webNavigation.onCompleted.addListener(function(details){
  if(details.url != "about:blank"){
    if(details.frameId==0){
      socket.emit("Url", details.url);
    }
  }
});
```

It is shown a function that with a listener that is triggered after the loading of a new page in the browser is completed. The first “if” is needed to avoid the sending of the “about:blank” URL, which is saved on the first opening of the browser. Instead, the second “if” is needed to take the data regarding the main frame which is the one loaded at the beginning of the navigation. In the end, the URL is sent through socket to the local server that serves the purpose of forwarding it to the Visual Studio Code extension.

4.2 Libraries

Inside the extension, some important open source libraries has been utilized to make every component work smoothly. Those libraries has been chosen thanks to their capabilities compared to other ones which were lacking some important functions.

4.2.1 Selenium

Selenium is an open-source framework that enables automated testing of web applications. It provides a suite of tools that allow developers to write tests in various programming languages, and simulate user interactions with web browsers. Selenium consists of several components that work together to automate web testing. The most important component is the WebDriver, which is a browser automation API that allows you to programmatically control a web browser and simulate user interactions. WebDriver supports multiple browsers including Chrome, Firefox,

Safari, Edge, and Internet Explorer. In general, Selenium is a powerful tool when talking about testing of web applications.

Selenium was used to start a local headless server to enable the use of the WebSpeech API described below. Without Selenium the user should have kept a Google Chrome instance opened continuously during the use of Veronica. Instead, in that way it is possible to use Veronica without having to rely on an open tab, keeping the general interface and requirements simpler.

4.2.2 WebSpeech API

The goal of the Web Speech API is to make it possible for web developers to offer speech-input and text-to-speech output functions in web browsers that are generally unavailable when using screen reader or regular speech recognition software. The API itself supports both server-based and client-based/embedded recognition and synthesis and is independent of the underlying speech recognition and synthesis implementation. Both quick (one-shot) speech input and ongoing speech input are supported by the API. The web page receives speech recognition results as a list of hypotheses, along with other pertinent data for each hypothesis. For our tool we used both the SpeechRecognition API to understand the user utterances and the SpeechSynthesis API to reproduce by voice the output given by Veronica from the chat.

4.2.3 Socket.io

It has always been quite difficult to create a chat application using well-known web application stacks. Because of that, the majority of real-time chat systems have historically been built using sockets. The sockets offer a bi-directional communication channel between one or more clients and a server. In other words, the server is able to push messages to clients and concurrently to receive them. Usually, the principle behind a chat is that whenever you write a message, the server will receive it and send it to every other client who is connected. However, in our case we used sockets only to implement the real-time communication, while having a uni-directional communication from the server to the Visual Studio Code client. At the same time another uni-directional communication has been implemented from the Google Chrome extension client to the server.

Chapter 5

Experimental Evaluation

This chapter contains an initial description of the evaluation phase of this thesis work.

5.1 Description

The test phase of the extension had the final objective to understand the utility of the functionalities proposed for Veronica to support the programmers. Given the context of this exploratory work, a simpler analysis has been conducted. Because of that, the final decision to perform an usability testing instead of a more complex A/B testing was taken. The test has been guided by the following questions:

1. Are the functionalities designed to complete the tasks useful or they need to be revisited/improved?
2. Does the experience of the user in the programming environment affects the final results of the test?
3. If a user is more experienced with a conversational agent, will he adapt to the tool in a easier way?

To answer these questions, we provided to the testers a survey both at the beginning and at the end of the test. Moreover, the screen has been registered during the testing to analyze afterward the behavior of each user and a brief interview has been done at the end to retrieve important thought right after the usage of Veronica. In this way it is possible to gain important data both from the user perspective and from the analysis of the exercises performed.

5.1.1 Participants

The choice about the type of user to perform the usability testing was straightforward. We opted for a group of university students, because they are easier to find for the testing purpose. Moreover, Veronica has been thought for programmers of different expertise, which are easy to find within the same university, based on their years. In fact, the students from these kinds of courses pass most of their time on the computer, usually to study or for entertainment. It is hard to find a computer engineer student that is not familiar with Visual Studio Code or a chatbot of any type. All the participants selected are between 21 and 28 years old and have been contacted mainly from the teacher's courses or within personal contacts. We defined 3 levels of expertise: beginner, intermediate and expert. The number of participants wanted was based on an equal division of these expertise ranks, hence a number multiple of 3 (i.e.: 3 3 3 or 4 4 4). During the selection we tried to select the users from different courses based on the year of the course, without any specific attention to other elements. In the end, 9 participants were recruited. Eight of them self-identified as male, and one self-identified as female. Their average age was 22.7 years (min = 21, max = 27).

The study has been described to each participant as an experiment to verify the usefulness of the chatbot as a valid supportive tool for the programmer.

In the table 5.1 is shown the entry survey together with the answers from the participants. Here a brief explanation of the reasoning behind each question:

- **Q-01:** Question to understand the background of the user, programmatically talking. We wanted to understand if the perceived knowledge of the user could have some impact on the usability of the prototype. For example, we thought that if a user had to focus more on the code he could have found the functionalities less intuitive. Most of the participants self-evaluated themselves as “intermediate”, even if part of three different academic years with a different amount of time of experience.
- **Q-02:** This one was asked to know if each user was comfortable with the chosen IDE. In this way we could understand from the usability testing if some difficulties could also have been linked to the inexperience of the user with the environment (and not related to a specific functionality directly). As expected, all the users knew VS Code and utilized it at least once.
- **Q-03:** For the users that used Visual Studio Code, we asked additionally if they ever used any kind of extension within the IDE. Most of them were familiar with background extensions (55%), the 33% used both background and active extensions and one user never used any. By asking this question we wanted to understand the familiarity of the students with the different types of extensions. A user that is used to addons with a different graphical interface,

for instance, should likely be less prone to errors compared to another one that has never seen them.

- **Q-04:** With this question we wanted to understand the users knowledge about any kind of interaction with a chatbot. In fact, a conversation can be totally different depending on the user's experience, because the logic behind most of the conversational agents is similar despite the different environments of application. From the answers it appears that the knowledge of conversational agents is still not too popular in this environment (44% had previous experience).
- **Q-05:** Since the language that Veronica understands is constrained to the English at the moment, and the voice recognition is quite precise on the mother language accent, this question was important to have a forecast of the outcome of the conversations. We did not want to test users that could have been blocked by a language barrier, since this was not the aim of the usability testing. Hence, it was important to select the right users for the purpose. Overall all the users had at least the B1 level.
- **Q-06:** The question around the programming language was used to provide the best possible code language for the user to perform the tasks. The knowledge of the coding language should have not increased the cognitive load of the user, so to pick a familiar language we followed also the answers to this question. In the end, all the users selected Python as known language, so all the tests have been made on a Python code.
- **Q-07:** The browser question was made to understand which browser the user uses usually. It was important to notify immediately at the start of the usability testing to use Google Chrome because one of Veronica's functionalities was based on a Chrome extension. The decision to choose Google Chrome was lead by the decision to stay within the same domain with part of the other components (i.e.:Google Dialogflow). Moreover, as the site [43] states, the predominance of users is already accommodate to use Chrome (66,39%), while the rest is evenly divided between the other browsers.

| ID | Questions | Answers |
|------|--|--|
| Q-01 | How would you assess your level of programming? | beginner (1) - intermediate (4) - expert (4) |
| Q-02 | Have you ever used Visual Studio Code? | no (0) - sometimes (3) - yes, I use it often (6) |
| Q-03 | If yes to the previous question, have you ever used an extension on Visual Studio Code? Did any of the extensions you used have a graphical interface? | no (1) - yes, someone in background (5) - yes, someone with visible graphical interface (0) - yes, both types (3) |
| Q-04 | Have you ever interacted with a chatbot? | yes (4) - no (5) |
| Q-05 | What is your level of English? | A1/A2 (0) - B1/B2 (6) - C1/C2 (3) |
| Q-06 | What programming languages do you know, at least at a basic level? | Python (9) - C (8) - JavaScript (6) - C++ (4) - Java (2) - Assembly (2) - Bash (1) - HTML (2) - Kotlin (2) - PHP (2) |
| Q-07 | What browser do you usually use? | Google Chrome (5) - Microsoft Edge (2) - Brave (1) - Mozilla Firefox - (1) Safari (0) |

Table 5.1: Answers to the entry survey.

5.1.2 Preparation and Setup

Before performing any test, we thought about the best approach to analyze the prototype. The main decisions were related to: Which type of test (“A/B” or usability testing), how to perform it (live from a single computer or asynchronously by giving to each user a guide to install the tools). Regarding the type, the A/B testing was thought at the beginning thanks to its usefulness. However, given its time consumption compared to a usability testing and that the functionalities of the prototype would have not differ a lot the two configurations, it has been discarded. Moreover, the usability testing seemed to fit better our case, where we wanted to validate the functionalities in general and the overall usability. Concerning the performance of the tests, we opted for a live session, both to leverage the users from all the passages to install the various components and to follow them closely during the tasks execution. The usability testing has been performed in the university laboratory. To haste the setup process and avoid users’ problems related to the installation, a previously configured computer has been used to perform all the tasks. In the computer were installed the following tools: Visual Studio Code with the extension ready to be started, the local headless server to enable the WebSpeech API and the Google Chrome extension. Moreover, OBS Studio has been installed to record the screen and the audio to analyze subsequently the users tests and personal thoughts. Together with the entry survey shown on 5.1, a debriefing questionnaire (the SUS) has been provided as well. The System Usability Scale (SUS) [44] provides a “quick and dirty”, reliable tool for measuring the usability. It consists of a 10 item questionnaire with five response options for each entry; from

Strongly agree to Strongly disagree. The questions are shown on the table 5.2. As it is possible to see, all the questions positioned on odd numbers are the “positive” ones, meaning that an higher vote would result in a better score for the SUS survey. On the other hand, all the even questions are the “negative” ones, meaning that a lower vote would improve the final score obtained.

| ID | Questions |
|------|---|
| S-01 | I think that I would like to use this system frequently |
| S-02 | I found the system unnecessarily complex |
| S-03 | I thought the system was easy to use |
| S-04 | I think that I would need the support of a technical person to be able to use this system |
| S-05 | I found the various functions in this system were well integrated |
| S-06 | I thought there was too much inconsistency in this system |
| S-07 | I would imagine that most people would learn to use this system very quickly |
| S-08 | I found the system very cumbersome to use |
| S-09 | I felt very confident using the system |
| S-10 | I needed to learn a lot of things before I could get going with this system |

Table 5.2: Questions of the SUS questionnaire

Additionally, in the Visual Studio Code were presents some files written in different coding languages (to match the skills defined by each user during the entry survey) where the tasks should have been performed. The code provided was intentionally easy to understand, since any level of user should have been able to understand it and focus mainly on the tasks. Each file contained 3 functions defined to perform different sorting algorithms: “bubbleSort”, “insertionSort” and “selectionSort”. In the end, as previously stated, only the Python one was enough since all the users were confident with that coding language. Talking about the tasks defined 5.3 to be performed, those has been decided to analyze the intuitiveness of the prototype and the ease of utilization. Each task was designed to be as much as possible independent from the others, so that it could have been possible to proceed without remaining stuck during the test.

| ID | Assigned Task | Success Criteria | Max Time |
|-------------|---|---|----------|
| T-01 | Start Veronica and ask for help | Veronica has been started and the “help intent” has been triggered | 5 min |
| T-02 | Create a memo attached to the function “bubbleSort” | The memo has been created (manually or with the bot) | 8 min |
| T-03 | Search online the algorithms computational complexity and save the page in a memo | The URL related to the page found has been saved by the bot (not manually) | 8 min |
| T-04 | Delete the memo that contains a wrong description of the attached code | The memo has been found and deleted with success (manually or with the bot) | 8 min |
| T-05 | Disable Veronica listening | The listening has been disabled, leaving the rest of the functionalities online | 5 min |

Table 5.3: Answers to the entry survey.

To better understand the aim of each task, they will be briefly described one by one:

- **T-01:** The objective of the first task is to understand if Veronica is easy to activate from the designed button. Moreover, it is also required to ask for help, in this way after the activation it is possible to trigger the intent correlated and understand all the functionalities available.
- **T-02:** With the second task the information collected are useful for different feedback. First, we can understand if the defined ways to create a new element of documentation in the code is easy or too complex. Second, it is interesting to see if the users, depending on a common factor (the skill level for example), prefer to create them manually from the comment controller or thanks to the help of Veronica.
- **T-03:** In the third task it is request to search for the computational complexity. The idea is to force the user to search it online since knowing the computational complexity might not be banal. In this way it is possible to trigger the URL tracking functionality.
- **T-04:** Similarly to the second task, here we want to understand the user approach and the difficulty of the task based on the functionalities given. Moreover, the wrong memo has been attached to a different and closed file, in this way also the comment navigation from the bottom panel is tested.
- **T-05:** Since Veronica listens continuously to the user, it is important that is clear how her listening can be disabled. There are many reasons to disable the listening, for example the user could be in a crowd place or wants to listen

to some music. Here we want to test the difficulty related to disabling the listening and understand if there could be better ways to implement it.

5.1.3 Execution

Talking about the execution of the test itself. The following process has been followed and has been the same for every user that tested the prototype:

Initially, a brief introduction was given to explain the function of the usability testing.

Right after, the entry survey was submitted to gather some data about the user 5.1.

Then, a paper containing the tasks was provided to the user 5.3 without any explanation of each of them.

Then the user was left with the computer to perform all the task given in order. As mentioned in the previous section, we decided to record each user, because we left during their testing. In this way they would not be tempted to ask for hints and we were able to evaluate the functionalities without bias. The only interruption by our side was to unlock the users on specific task after a certain time threshold specified on the table 5.3.

Once the testing was finished, the SUS questionnaire 5.2 has been provided to each user to evaluate with another tool the usability of Veronica.

In the end, a final interview has been conducted to listen the users' thought, ask for a personal feedback regarding the tool and possible further improvements that they would have liked to see. We opted for the final interview to allow each user to describe more freely their own ideas, without having to rely on a written open question that could constrain all their thoughts.

5.2 Results

Thanks to the usability testing we were able to collect some important data related to the functionalities implemented and contextually to the graphical interface of the tool. Before starting to watch the recorded video for each user, we defined some metrics to understand the actual usability of each functionality. Those metrics can be applied to each task and the final result is shown in the table 5.4.

The metrics taken into account are:

| ID | Critical errors | Non-critical errors | Completion rate | Error-free rate |
|------|-----------------|---------------------|-----------------|-----------------|
| T-01 | 0 | 7 | 55% | 22% |
| T-02 | 0 | 6 | 100% | 66% |
| T-03 | 1 | 1 | 88% | 88% |
| T-04 | 0 | 1 | 100% | 88% |
| T-05 | 0 | 6 | 88% | 44% |

Table 5.4: Tasks’ metrics and results

- **Critical Errors:** Those are the errors that prevent the completion of the task. Meaning that if the user makes this kind of error, he won’t be able to fulfill the task anymore. For each task has been inserted the total amount of errors done by the users.
- **Non-critical errors:** Mistakes that enable the fulfillment of the task but incompletely or erroneously. Moreover, has been inserted under this category all the errors made by the users during the accomplishment of a task. Also in this case, each number represents the total amount of errors done.
- **Completion rate:** Percentage of the students that did not make any critical errors. In this category falls all the users that completed a specific tasks (both with or without errors). Differently from the previous metrics, here the value represents the % of users that managed to complete a task, where “1” represents the 100% (9 users).
- **Error-free rate:** Percentage of students that did not make any mistake. This is a subset of the previous category, meaning that the results here are always minor or equal to the Completion rate metric.
- **Task completed in time:** This metric is defined behind a “on/off” logic. Meaning that we analyzed if each user was able to perform each task within a predefined maximum time defined in the table 5.1. This time was defined by calculating and rounding the times of two additional users that were aware of the functionalities and performed the tasks with some help by our side. In the table the % is equal to the “Completion rate” metric.

By analyzing this table, is already possible to make some assumptions on the various functionalities. While the table 5.3 can be used as a reference to see the description of each task. Here the results from the table 5.4 are discussed and integrated with the information gathered from the video recording:

T-01: From the Completion rate and the Error-free rate it is instantly possible to see that this task was the hardest to complete. Although there were no critical errors, only 5 of the participants were able to complete it without our help. All the non-critical errors are related to different attempts of activation without success. Most of the users tried to activate it through a command in the integrated terminal or by using the normal terminal in the panel section. Some of them tried to activate it vocally trying some kind of verbal commands. Additionally, just 2 participants completed it smoothly, while the other 3 completed it almost at the end of the maximum time that we defined. For sure this was the least understandable task, the button was hard to find and did not stand out within the environment. The constraints given from the VS Code IDE made the button position design and hard task to be accomplished.

T-02: In the second task, 6 users managed to complete it without errors. It is interesting to see that, 4 of them used directly the interface given from the comment controller to create new memo. The other 2, instead used the chat directly given in the interface. The rest of the users made some non-critical errors, concerning attempts to create the memo with commands that were not understandable by Veronica. Then, they completed it manually from the comment controller as a backup plan. The behavior of one user was quite interesting, he tried to use the vocal function to create the memo, when Veronica didn't recognize some of his utterances he swapped to the chat to complete the sentences. Despite of some small errors, all the participants managed to complete the assigned exercise.

T-03: The task related to the online research was the most successful in terms of error-free rate. All the users were able to complete it without problems thanks to the proactivity of Veronica. The critical error happened when Veronica asked the participant if he wanted to save the URL and by mistake he answered negatively. After trying to get back to the browser and then again to Visual Studio Code, Veronica did not ask him again if he wanted to save it because the choice had already been saved. Apart from this case, all the participants completed the task successfully. The high success rate seems to be addressable to the starting input given from the agent, instead of coming from the user.

T-04: Similarly to the second task, all the users were able to complete it without incurring in critical-errors. One participant made one non-critical error because instead of deleting the memo, he manually deleted the content. However, he noticed his mistake and deleted the memo afterwards. 6 users deleted the memo manually from the comment controller, without interacting with Veronica. The rest exploited the bot to perform the task, switching to the manual deletion in case of difficulties during the conversation. Overall, probably because of

the similitude with the memo creation task, all the user performed this one smoothly.

T-05: Finally, no critical errors were made to disable Veronica’s listening. However, participants made six non-critical errors, all related to trying to disable Veronica by writing a command in the chat instead of clicking the same button that was used to activate the extension. Nevertheless, 8 out of the 9 participants succeeded in disabling Veronica without many problems. The key aspect is that, even if the majority of the users managed to succeed in the task, all of them tried to deactivate the listening in other ways, before using the implemented one.

From the task analysis already it was possible to identify valid improvements for the functionalities implemented and understand what was working fine by itself. The second element used to define the quality of the prototype was the SUS questionnaire that helped us to rank its usability. Interpreting the SUS scoring can be complex. The participant’s scores for each question are converted to a new number, added together and then multiplied by 2.5 to convert the original scores of 0-40 to 0-100. Though the scores are ranged from 0 to 100, these are not percentages and should be considered only in terms of their percentile ranking [44].

Based on the research [45], a SUS score above 68 would be considered above average and anything below 68 is below average. In the table 5.6 it is possible to see the correlation between the SUS score obtained and the adjective rating. Regarding the results, the scores that has been obtained from the users can be seen in the table 5.5.

| User | Score | User Perceived level |
|-------------|-------|----------------------|
| U-01 | 67.5 | Intermediate |
| U-02 | 80 | Intermediate |
| U-03 | 57.5 | Intermediate |
| U-04 | 72.5 | Beginner |
| U-05 | 65 | Intermediate |
| U-06 | 75 | Expert |
| U-07 | 40 | Expert |
| U-08 | 75 | Expert |
| U-09 | 82 | Expert |

Table 5.5: SUS scores obtained by the users

From the list it is possible to see that the scores were not affected by the expertise of each user (**Guiding question 2**). One score of 40 is present, meaning that for that user the usability of the tool has been “awful”. Then, 2 other results fell into

the “poor” rating while a third one was borderline between the “poor” and the “okay” tier. Only one user SUS questionnaire returned an “excellent” score and the remaining 4 corresponded to the “Good” one. Despite the expertise of the users was not a key factor during the testing, it is clear from the results that there is margin to improve Veronica.

| SUS Score | Grade | Adjective Rating |
|-----------|-------|------------------|
| > 80.3 | A | Excellent |
| 68 – 80.3 | B | Good |
| 68 | C | Okay |
| 51 – 68 | D | Poor |
| < 51 | E | Awful |

Table 5.6: SUS score meaning

To give a brief recap of the results collected:

- **The expertise of the users** does not appreciably affect the utilization of the prototype. As it is possible to see from the table 5.7 the average SUS score is similar for the “Intermediate” and the “Expert” users. While the “Beginner” is slightly higher, it has to be taken into account that only one user self-identified himself as beginner.

| Expertise level | Average SUS score |
|-----------------|-------------------|
| Beginner | 72.5 |
| Intermediate | 67.5 |
| Expert | 68 |

Table 5.7: Average SUS score for expertise level

- **The 4 users that had already interaction experience with a chatbot** scored the higher SUS scores, meaning that a previous experience could improve the intuitiveness of the tool.
- Concerning the functionalities, there is margin for improvements, especially for the ones used in the first and last task of the usability testing. The possible changes are discussed in the following chapter.

Chapter 6

Conclusions and Future improvements

6.1 Conclusions

The goal of this thesis work was to experiment the usage of a multi-channel conversational agent as an extension for a programming IDE to support the programmer's reasoning. The first step made was to understand if similar projects have been explored in this domain. However, despite the great amount of researches related to conversational agents in different domain, the findings concerning the programmers and a way to help them were really scarce. Because of that, the decision was to develop this as an exploratory work, to understand if there could be a future for this kind of chatbot in this environment.

Then, a study to outline the components to define an high-level architecture has been performed. The individual components that best suited the solution defined were: Dialogflow, a conversational agent builder designed by Google for the chatbot to take on charge the logic management and multi-channel deployment. Visual Studio Code was used as the IDE in which implement the extension, the language used was Typescript by guideline suggestion. For the speech recognition and synthesis the WebSpeechAPI has been used together with a local headless server written in JavaScript. Moreover, a small Google Chrome extension has been utilized to track the user navigation on internet.

The bot validation was done with a usability test made in laboratory. The usability test taken was composed by four different elements to gather useful data: an entry survey, the list of tasks performed itself, a SUS questionnaire and a debriefing interview. Test results have revealed the chatbot strength and weaknesses together with important feedback and appreciation from the user.

In conclusion, chatbots interest is growing year after year and are starting to

represent a good alternative in many domains. They can be flexible enough to be used in conjunction with artificial intelligence, data science, natural language understanding and other fields of technology. They find their place in a wide variety of applications, from entertainment to customer care, from e-commerce to even psychotherapy. Yet, there is still a great margin of improvement in the programming field, both to write directly the code and to help the quality of life of every programmer. The chatbot developed for this thesis project has aroused interest between the users that used it for the test. Yet there is still some work to do so that it can be considered a valid and reliable tool to be used extensively in the field, both in terms of technology progress and general interest in this specific domain.

6.2 Future improvements

The developed chatbot lends itself to a variety of future improvements. In regards of that, we collected a series of possible upgrades and mandatory steps to be done to make Veronica more user friendly and at the same time less frustrating to interact with. From what was stated, Veronica is working locally by the time being. However, the first step to be done before publishing the extension would be to implement an online database to synchronize in real-time the data of all the users connected. A good candidate could be Google Firebase, since it is easy to exploit his API to be merged together with the others Google tools used in this project.

Some important changes concerns the further enhancement of the already existing features which at the moment presented some margin of improvement after what had been seen during the usability testing. Talking about those possible improvements, some of them were thought already during the usability testing.

The button to start the extension, given the constraints of the VS Code IDE in relation to the visual changes, was hard to detect almost by every user. So, the best idea would be to highlight even more the existing button. Moreover, it could be possible to add a related command from the terminal to perform the same action. The last possible improvement for the activation, based on the users behavior, would be to have a vocal command to activate the extension (but in this case the listening should be enabled from the beginning and triggered only by the right commands or keywords).

The method to deactivate the listening could be improved aswell. While the button was easy to find after the first task, many users tried to use the chat or a vocal command to order to the agent this action. The most natural implementation could be to add an intent that matches the will of the user to disable the listening.

Another important aspect that became way more prominent during the testing was **the needs to have better defined fallback answers** from Veronica. Some output should define better what the agent is expecting without distorting the feel of the conversation. Moreover, there should be additional controls all over the conversational patterns, to prevent the user from being stuck also on easier situations.

However, it is important to point out that many fallback has been defined on purpose simpler, since it is possible to add a lot of phrases to match the user utterances. The management of the unexpected input was not the main focus of this study, hence it is advisable to improve it for a possible use in the field.

Beside the natural upgrades that we thought by watching the usability testing, a debriefing interview has been performed to listen the participants impressions and gather possible good idea to be implemented. Some of those ideas seemed worth to be mentioned because of the recurrence in many users' interviews or because of their suitability with this kind of chatbot:

Many users expressed the **need of a more descriptive help command**, specifying that only knowing that Veronica can do, without an explanation on how to proceed isn't enough for a newbie. A similar request was to adapt more detailed help commands for each feature, in order to allow the chatbot to cover a specific functionality in a broader way. Another user asked to provide a sort of menu in the chat after the activation (similar to the menus present in the telegram's bot) listing all the commands. While this feature could be very helpful, at the same time is distant from the idea of this conversational agent, because it would make the interaction feeling different from the one that we tried to achieve.

Another highly requested feature was to **enable the activation of Veronica thanks to a specific keyword**. This is also covered by our assumptions after the usability testing as expected. While it wasn't initially implemented, we also think this feature could be useful, especially if the users are not too confident on where the button could be the first times.

A subgroup of users suggested some graphical adjustment. The most recurring one was to **add a microphone icon in the activation button**, to help its detection easily and because some of them did not linked the deactivation task to this button logically. One change worth to mention is also the addition of a setting icon in the chat to allow a sort of customization of the interface, together with the possibility to activate/deactivate the bot or the listening.

The need to **edit each comment** directly also came up, without having to delete them and then recreate it in the same place.

Finally, a user suggested a new feature in line with the philosophy of the chatbot, which is to **understand what the user is searching online** and provide a prototype of it directly in the code together with the URL. While the request seems to be useful, it has to overcome the constraints given by the Visual Studio Code IDE and a deeper level of Machine learning should be involved.

Given all our assumptions and the ideas given by the users, there is a great margin to improve even more Veronica.

Bibliography

- [1] Kenneth McGarry Guendalina Caldarini Sardar Jaf. *A Literature Survey of Recent Advances in Chatbots*. URL: <https://www.mdpi.com/2078-2489/13/1/41/htm#B6-information-13-00041> (visited on 01/17/2023) (cit. on p. 1).
- [2] A. M. Turing. «Computing Machinery and Intelligence». In: *Mind* 59.236 (1950), pp. 433–460. ISSN: 00264423, 14602113. URL: <http://www.jstor.org/stable/2251299> (visited on 02/17/2023) (cit. on p. 6).
- [3] Eleni Adamopoulou and Lefteris Moussiades. «An Overview of Chatbot Technology». In: *AIAI 2020: Artificial Intelligence Applications and Innovations*. May 2020 (cit. on pp. 6, 15).
- [4] Willem Osuch. *Chatbots: A Brief History Part I - 1960s to 1990s*. URL: [https://www.botsplash.com/post/chatbots-a-brief-history#:~:text=In%5C%201966%5C%2C%5C%20the%5C%20world's%5C%20first,\(MIT\)%5C%20by%5C%20Joseph%5C%20Weizenbaum.](https://www.botsplash.com/post/chatbots-a-brief-history#:~:text=In%5C%201966%5C%2C%5C%20the%5C%20world's%5C%20first,(MIT)%5C%20by%5C%20Joseph%5C%20Weizenbaum.) (visited on 01/17/2023) (cit. on p. 6).
- [5] Joseph Weizenbaum. «ELIZA—a Computer Program for the Study of Natural Language Communication between Man and Machine». In: *Commun. ACM* 9.1 (Jan. 1966), pp. 36–45. ISSN: 0001-0782. DOI: 10.1145/365153.365168. URL: <https://doi.org/10.1145/365153.365168> (cit. on pp. 6, 7, 9, 19).
- [6] Kenneth Mark Colby, Sylvia Weber, and Franklin Dennis Hilf. «Artificial Paranoia». In: *Artificial Intelligence* 2.1 (1971), pp. 1–25. ISSN: 0004-3702. DOI: [https://doi.org/10.1016/0004-3702\(71\)90002-6](https://doi.org/10.1016/0004-3702(71)90002-6). URL: <https://www.sciencedirect.com/science/article/pii/0004370271900026> (cit. on pp. 7, 19).
- [7] György Molnár and Zoltán Szüts. «The role of chatbots in formal education». In: *2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY)*. IEEE. 2018, pp. 000197–000202 (cit. on p. 7).
- [8] et al. Marietto M. «Artificial intelligence markup language: a brief tutorial.» In: (2013), pp. 1–20. URL: <http://www.airccse.org/journal/ijcses/papers/4313ijcses01.pdf> (cit. on p. 9).

- [9] Richard S. Wallace. «The Anatomy of A.L.I.C.E.» In: *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*. Ed. by Robert Epstein, Gary Roberts, and Grace Beber. Dordrecht: Springer Netherlands, 2009, pp. 181–210. ISBN: 978-1-4020-6710-5. DOI: 10.1007/978-1-4020-6710-5_13. URL: https://doi.org/10.1007/978-1-4020-6710-5_13 (cit. on p. 9).
- [10] Ina. *The History Of Chatbots – From ELIZA to ChatGPT*. URL: <https://onlim.com/en/the-history-of-chatbots/> (visited on 01/17/2023) (cit. on p. 10).
- [11] Jacob Aron. *Software tricks people into thinking it is human*. URL: <https://www.cleverbot.com/human> (visited on 01/17/2023) (cit. on p. 10).
- [12] Siri Team. *Deep Learning for Siri’s Voice: On-device Deep Mixture Density Networks for Hybrid Unit Selection Synthesis*. URL: <https://machinelearning.apple.com/research/siri-voices> (visited on 02/02/2023) (cit. on p. 10).
- [13] Brian Lucey and Michael Dowling. *ChatGPT: our study shows AI can produce academic papers good enough for journals – just as some ban it*. URL: <https://theconversation.com/chatgpt-our-study-shows-ai-can-produce-academic-papers-good-enough-for-journals-just-as-some-ban-it-197762> (visited on 02/02/2023) (cit. on p. 11).
- [14] Michael Dowling and Brian Lucey. «ChatGPT for (Finance) research: The Bananarama Conjecture». In: *Finance Research Letters* (2023), p. 103662. ISSN: 1544-6123. DOI: <https://doi.org/10.1016/j.fr1.2023.103662>. URL: <https://www.sciencedirect.com/science/article/pii/S1544612323000363> (cit. on p. 11).
- [15] Kiran Ramesh, Surya Ravishankaran, Abhishek Joshi, and K. Chandrasekaran. «A Survey of Design Techniques for Conversational Agents». In: *Information, Communication and Computing Technology*. Ed. by Saroj Kaushik, Daya Gupta, Latika Kharb, and Deepak Chahal. Singapore: Springer Singapore, 2017, pp. 336–350. ISBN: 978-981-10-6544-6 (cit. on p. 13).
- [16] Amandus Krantz and Petrus Lindblom. *Generating Topic-Based Chatbot Responses*. 2017 (cit. on p. 13).
- [17] Sutskever I. Mikolov T., Corrado G. S. Chen K., and Dean J. *Distributed Representations of Words and Phrases and their Compositionality*. URL: <https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf> (visited on 02/04/2023) (cit. on pp. 14, 20).

- [18] Sangkeun Jung. «Semantic vector learning for natural language understanding». In: *Computer Speech & Language* 56 (2019), pp. 130–145. ISSN: 0885-2308. DOI: <https://doi.org/10.1016/j.csl.2018.12.008>. URL: <https://www.sciencedirect.com/science/article/pii/S0885230817303595> (cit. on p. 14).
- [19] Amos Azaria Merav Allouch and Rina Azoulay. «Conversational Agents: Goals, Technologies, Vision and Challenges». In: (2021). ISSN: 1424-8220 (Electronic). DOI: 10.3390/s21248448. URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8704682/#:~:text=They%5C%20group%5C%20the%5C%20conversational%5C%20agents,oriented%5C%20dialogue%5C%20agents%5C%2C%5C%20and%5C%20chatbots.> (cit. on p. 16).
- [20] Jagdish Singh, Minnu Helen Joesph, and Khurshid Begum Abdul Jabbar. «Rule-based chabot for student enquiries». In: *Journal of Physics: Conference Series* 1228.1 (May 2019), p. 012060. DOI: 10.1088/1742-6596/1228/1/012060. URL: <https://dx.doi.org/10.1088/1742-6596/1228/1/012060> (cit. on p. 19).
- [21] Yuan M. «A developer’s guide to chatbots». In: (Aug. 2016) (cit. on p. 19).
- [22] Fainchtein L. *Generative vs Retrieval Based Chatbots: A Quick Guide*. URL: <https://blog.cloudboost.io/generative-vs-retrieval-based-chatbots-a-quick-guide-8d19edb1d645> (visited on 02/12/2023) (cit. on pp. 19, 20).
- [23] Dan Bohus and Alexander I. Rudnicky. «Ravenclaw: dialog management using hierarchical task decomposition and an expectation agenda». In: *8th European Conference on Speech Communication and Technology, EUROSPEECH 2003 - INTERSPEECH 2003, Geneva, Switzerland, September 1-4, 2003*. ISCA, 2003. URL: http://www.isca-speech.org/archive/eurospeech%5C_2003/e03%5C_0597.html (cit. on p. 21).
- [24] Iulian V. Serban et al. *A Deep Reinforcement Learning Chatbot*. 2017. DOI: 10.48550/ARXIV.1709.02349. URL: <https://arxiv.org/abs/1709.02349> (cit. on p. 21).
- [25] Nahdatul Akma Ahmad et al. *Review of Chatbots Design Techniques*. URL: https://www.researchgate.net/profile/Nahdatul-Akma-Ahmad/publication/327097910_Review_of_Chatbots_Design_Techniques/links/5b77cf3e4585151fd11cd905/Review-of-Chatbots-Design-Techniques.pdf (visited on 02/04/2023) (cit. on p. 22).
- [26] Ketakee Nimavat and Tushar Champaneria. «Chatbots: An overview types, architecture, tools and future possibilities». In: *Int. J. Sci. Res. Dev* 5.7 (2017), pp. 1019–1024 (cit. on p. 23).

- [27] Agnese Augello, Manuel Gentile, and Frank Dignum. «An Overview of Open-Source Chatbots Social Skills». In: *Internet Science*. Ed. by Sotiris Diplaris, Anna Satsiou, Asbjørn Følstad, Michail Vafopoulos, and Thomas Vilarinho. Cham: Springer International Publishing, 2018, pp. 236–248. ISBN: 978-3-319-77547-0 (cit. on p. 23).
- [28] Baki Kocaballi Ahlam Alnefaie Sonika Singh and Mukesh Prasad. *An Overview of Conversational Agent: Applications, Challenges and Future Directions*. URL: <https://www.scitepress.org/Papers/2021/107086/107086.pdf> (visited on 02/15/2023) (cit. on p. 23).
- [29] V. Akshatha Prasad and R. Ranjith. «Intelligent Chatbot for Lab Security and Automation». In: *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (2020), pp. 1–4 (cit. on p. 24).
- [30] Khan S. and M. R. Rabbani. «Artificial Intelligence and NLP-Based Chatbot for Islamic Banking and Finance». In: *International Journal of Information Retrieval Research (IJIRR)* (2021), pp. 65–77 (cit. on p. 24).
- [31] Alistair Martin, Jama Nateqi, Stefanie Gruarin, Nicolas Munsch, Isselmou Abdarahmane, Marc Zobel, and Bernhard Knapp. «An artificial intelligence-based first-line defence against COVID-19: digitally screening citizens for risks via a chatbot». In: (2020). DOI: 10.1038/s41598-020-75912-x (cit. on p. 24).
- [32] Reem Alotaibi, Ahlam Ali, Haya Alharthi, and Renad Almehamdi. «AI Chatbot for Tourist Recommendations: A Case Study in the City of Jeddah, Saudi Arabia». In: *International Journal of Interactive Mobile Technologies (iJIM)* 14.19 (Nov. 2020), pp. 18–30. DOI: 10.3991/ijim.v14i19.17201. URL: <https://online-journals.org/index.php/i-jim/article/view/17201> (cit. on p. 24).
- [33] Walrave M. Zarouali B. Van den Broeck E. and Poels K. «Predicting Consumer Responses to a Chatbot on Facebook». In: (2018). DOI: 10.1089/cyber.2017.0518 (cit. on p. 24).
- [34] Jessica Van Brummelen, Kevin Weng, Phoebe Lin, and Catherine Yeo. «CONVO: What does conversational programming need?» In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Aug. 2020. DOI: 10.1109/vl/hcc50065.2020.9127277. URL: <https://doi.org/10.1109%5C%2Fvl%5C%2Fhcc50065.2020.9127277> (cit. on pp. 25, 26, 28).

- [35] Rainer Winkler, Sebastian Hobert, Antti Salovaara, Matthias Söllner, and Jan Marco Leimeister. «Sara, the Lecturer: Improving Learning in Online Education with a Scaffolding-Based Conversational Agent». In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–14. ISBN: 9781450367080. DOI: 10.1145/3313831.3376781. URL: <https://doi.org/10.1145/3313831.3376781> (cit. on pp. 25, 26, 36).
- [36] Yoonji Kim, Youngkyung Choi, Daye Kang, Minkyong Lee, Tek-Jin Nam, and Andrea Bianchi. «HeyTeddy: Conversational Test-Driven Development for Physical Computing». In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3.4 (Sept. 2020). DOI: 10.1145/3369838. URL: <https://doi.org/10.1145/3369838> (cit. on pp. 26, 28).
- [37] Nick C. Bradley, Thomas Fritz, and Reid Holmes. «Context-Aware Conversational Developer Assistants». In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 993–1003. ISBN: 9781450356381. DOI: 10.1145/3180155.3180238. URL: <https://doi.org/10.1145/3180155.3180238> (cit. on p. 27).
- [38] Peter Robe and Sandeep K. Kuttal. «Designing PairBuddy—A Conversational Agent for Pair Programming». In: *ACM Trans. Comput.-Hum. Interact.* 29.4 (May 2022). ISSN: 1073-0516. DOI: 10.1145/3498326. URL: <https://doi.org/10.1145/3498326> (cit. on p. 27).
- [39] A. Nowogrodzki. «Writing code out loud». In: *Nature* 559 (2018), pp. 141–142 (cit. on p. 27).
- [40] *Balsamiq home*. URL: <https://balsamiq.cloud/> (visited on 11/21/2023) (cit. on p. 32).
- [41] *Visual Studio Code development discussion*. URL: <https://github.com/microsoft/vscode-discussions/discussions> (visited on 11/21/2023) (cit. on p. 32).
- [42] *Dialogflow Documentation Page*. URL: <https://cloud.google.com/dialogflow/docs> (visited on 03/01/2023) (cit. on p. 48).
- [43] *Desktop Browser Market Share Worldwide*. URL: <https://gs.statcounter.com/browser-market-share/desktop/worldwide> (visited on 11/14/2023) (cit. on pp. 54, 59).
- [44] *SUS questionnaire*. URL: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html> (visited on 03/07/2023) (cit. on pp. 60, 66).

- [45] *Interpreting System Usability Scale (SUS) Score*. URL: <https://uiuxtrend.com/measuring-system-usability-scale-sus/#interpretation> (visited on 03/09/2023) (cit. on p. 66).