Politecnico di Torino

Computer Engineering

A.y. 2023/2024

April 2024 Graduation Session

Master's Degree Thesis in

Developing an AI-Powered Voice Assistant for an iOS Payment App

Supervisors:

Luigi De Russis

Andrea Loffredo

Candidate:

Mario Mastrandrea

Abstract

This thesis investigates the potential of an AI-powered voice assistant in simplifying and securing financial transactions in a Peer-to-Peer (P2P) payments application, integrating cutting-edge Machine Learning technologies on-device such as BERT for text classification and a custom Dialogue State Tracking mechanism. Through a meticulous design and development process, the study explores the integration of voice commands into a real iOS app to facilitate in-app operations, prioritizing user privacy through local data processing, and enhancing user interaction and accessibility via an intuitive UI/UX design leveraging the novel SwiftUI framework. The research conducts a thorough evaluation of the system's performance, focusing on user experience, security, and the accuracy of voice and language recognition capabilities. The analysis reveals significant results in operational efficiency and user satisfaction, highlighting the voice assistant's role in advancing digital payment solutions. The work concludes with a critical analysis of the findings, discussing the implications for future AI applications and proposing avenues for further research in voice-enabled technologies for financial services.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the digital age, **mobile applications** have become crucial tools, transforming the way we communicate, work, and manage our daily lives. With the rise of smartphones, the demand for increasingly sophisticated and user-friendly apps has resulted in considerable advances in the mobile computing field.

With its revolutionary impact on industries and daily lives, the emergence of **Artificial Intelligence (AI)** is marking another turning point in the history of technology, characterizing an era of unprecedented innovation and transformation. It includes a wide range of abilities from simple pattern recognition to intricate decision-making processes, which have quickly evolved into a variety of practical applications that are present in all aspects of our life. AI's impact is extensive, transforming businesses, improving efficiency, and facilitating novel interactions between humans and technology, but its significance is amplified by its versatility and adaptability, finding applications in diverse fields such as healthcare, education, and entertainment, thereby establishing its role as a key aspect of contemporary technological progress.

The combination of AI and mobile technology enhances the utility and the effectiveness of applications, while also enabling the development of unique services previously only seen in science fiction, therefore making a substantial difference in the digital market for the uniqueness and the potential of a software product.

The Financial Technology (FinTech) sector, for example, is revolutionizing financial transactions through the introduction of **Peer-to-Peer (P2P) services**, a model enabling direct financial transactions between people, thus avoiding traditional intermediaries, usually by means of tailored mobile applications. P2P payments are, in fact, significantly enhancing the speed, the effectiveness, and the costs of the transactions. However, still some challenges exist, and Machine

Learning (and more in general Artificial Intelligence) technologies can be used to address them. Specifically, AI can provide algorithms for high-quality anomaly detection, ensuring higher *security* and more sophisticated *fraud detection*, and can also make significant advances in *privacy* by better preserving user data confidentiality. But more importantly, AI can also enhance **User eXperience (UX)** by making interactions more *intuitive*, *natural* and customized to meet the individual user's needs, promoting increased engagement and involvement.

Therefore, integrating AI solutions into the P2P payments applications addresses significant issues and provides new opportunities for advancement and expansion in the field. It plays a crucial role in providing more **effective** and **user-friendly** payment services by improving *security*, data *privacy*, and enhancing the user experience.

1.1 Goal

As we approach this AI-driven revolution in mobile computing, it is crucial to fully understand its potential to drive a progress that not only aligns with society's changing needs but also outlines and promotes positive usages, creating a future where technology and human interaction merge harmoniously.

Indeed, the scope of this Thesis work is to provide a detailed overview of the creation and integration of a specific AI product within a P2P payment application, examining its impact on the app's features and User eXperience. In particular, the project deals with the creation of an **AI-powered voice assistant**, from design to software integration, which converts user speech into actual in-app operations, like sending some money or checking the last transactions, and processes all user data **on-device**, with no need of network connection.

The main purpose is to improve user interactions through **voice-activated commands**, thus enhancing the app's accessibility and offering a more natural way of human-machine interaction, which might be useful for example for users with lower technological expertise. This is achieved without sacrificing the user **privacy**, given that no data is sent on the network, and this is of utmost importance since often a major limitation of Machine Learning applications is the need of outsourcing the information due to resource constraints. The pursuit of this goal implies exploring new frontiers of the **Natural Language Processing (NLP)** technologies which are currently disrupting the Information Technology (IT) area, understanding the foundation of ML models behind our current Search Engines (like Google) and the new breakthrough Generative AI products (like ChatGPT), offering not only an overview of the state-of-the-art technologies in the field but also an example of practical usage for such innovative models.

The project focuses exclusively on the implementation for iOS devices, exploring the capabilities of the new *SwiftUI* framework and leveraging crucial programming tools offered by the iOS operative system for the *Speech-to-Text (STT)* and *Text-to-Speech (TTS)* functionalities, which are fundamental blocks of a Voice Assistant. This choice of the platform derives mainly by the technological context in which the project was carried out by the author, which will be better described in the next section.

1.2 Method of Work

This Thesis work was born in collaboration with *Pay Reply*, an IT consultancy company based in Turin and Milan, and specialized in the deployment of Mobile Applications in the **Digital Payments** sector. The project has been carried out as a Research and Development (R&D) activity during an internship experience in the company, during which I had the opportunity to work on a P2P Payment Mobile Application as iOS developer.

The entire task was accomplished following a cyclic development process, each round starting from the Machine Learning part first and then caring about its in-app integration:

- the first round was characterized by a wide research in the Natural Language Processing area and the state-of-the-art of *Chatbots*. In parallel, multiple technologies have been investigated for the creation and training of such ML models, as well as their integration into an iOS application. The final model was built in a Python notebook on the *Google Colab* cloud platform, leveraging *TensorFlow* and *TensorFlow Lite* frameworks. Thereafter, a set of programming tools offered by the iOS platform in Swift have been studied and tested for the Voice Assistant features on-device, like Speech-to-Text and Text-to-Speech, and SwiftUI for the visual part;
- in the second round, after a careful design of the Voice Assistant functionalities, the mentioned tools have been used all together to bring the system to life, from the ML model to its employment inside the Dialogue State Tracker (DST) in Swift, up to the UI implementation;
- a final step included an evaluation phase, with a pool of test users trying the Voice Assistant through a test application released on *TestFlight*; and finally

its integration into the actual P2P Payment app of the company, taken as a case study for this Thesis work and better described in section 1.3.

The entire process was managed using *Git* Version Control System (VCS) and a remote *GitHub* repository.

1.3 Case Study: example of P2P Payment App

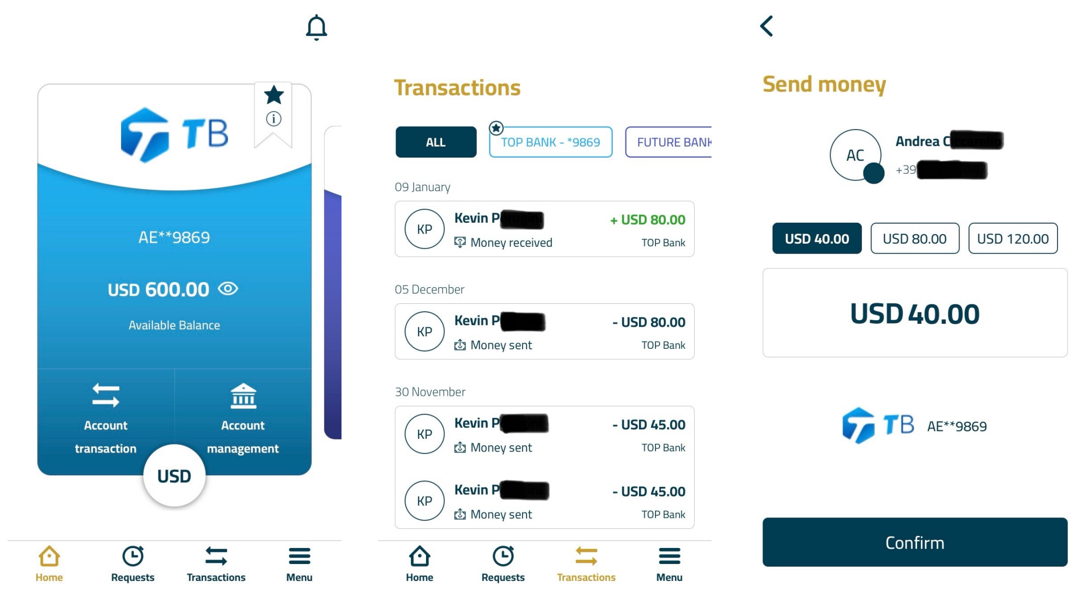
This brief section will present the mobile payment application used as a case study for this work, thus the actual app that inspired all the design choices described in the following Chapters and into which the resulting Voice Assistant was integrated to eventually fully test its functionality. Its name will not be disclosed in this work, to protect the intellectual property of its owners.

The mentioned application constitutes an advanced **Peer-to-Peer digital payment system** developed for a leading financial institution in the Extra-EU market. It stands out for providing a smooth, safe, and exceptionally quick payment processing experience, with transactions being finalized in a few seconds. The platform facilitates payments not just for Peer-to-Peer transactions but also for businesses, corporations, and government agencies, offering an exceptional digital payment service 24/7.

The app's main feature is its capability to *process transactions* just based on the recipient's mobile number or email, improving customer ease. In addition to traditional money transfer functions, it offers advanced features like *"request money"* and *"split bills"*, as well as *QR code payment* compatibility, which is especially advantageous for businesses. Upcoming updates will provide new features such as real-time direct debit, Electronic Direct Debit Authorization (EDDA), and fully digitalized invoices.

The platform may be accessed via the mobile application, which is available on both Google Play and App Store, making it convenient for users on different devices. The app's latest upgrades have concentrated on improving the user experience through enhancements to the UI and bug fixes.

In figure 1.1 some examples of the app's screens are reported, in different scenarios. Image 1.1a shows the app's *home* screen, where the user can visualize all their registered bank accounts and the corresponding information (e.g. the balance). In the *Transactions* tab (subfigure 1.1b) all the most recent user's transaction are shown, either incoming or outgoing. The third image (subfigure 1.1c) shows instead a screen involved in the *send money* operation process, where the user can type



(a) Home. **(b)** Transactions. **(c)** Send money.

Figure 1.1: Some of the main screens of the case study P2P payment app.

the right amount to be sent after having selected the recipient and the source bank account.

1.4 Overview of the document

After this introduction, in Chapter 2 we are going to introduce the main theoretical topics involved in the project, from the Artificial Intelligence state-of-the-art models in the field, including the two foundational Language Models BERT and GPT, to the Peer-to-Peer payments domain; Chapter 3 contains an analysis of the user requirements at the foundation of the adopted technical choices, together with the functional and non-functional requirements arising from them; Chapters 4 and 5 meticulously describe the design and the implementation of the Voice Assistant module respectively, characterized by the composition of a multitude of software components, each specialized in a specific sub-task (e.g. Speech Recognition, Text Classification, Dialogue State Management, etc.), with an high focus on the custom Machine Learning model created for text classification as well as on its incorporation into the iOS platform, and finally discuss the in-app integration process of the proposed system module; Chapter 6 illustrates the test app release on TestFlight and the evaluation phase carried out by means of an extensive testing based on an online form; and in Chapter 7 a critical analysis of both the adopted process and the resulting findings is carried out in the context of the relevant literature topics, providing some insights for future improvements.

Chapter 2

Background and State of the Art

This Chapter explores the complex development of **Voice Assistants**, tracing their progression from basic voice commands to advanced AI-powered interfaces. It delves into how *Artificial Intelligence* and **Natural Language Processing** played a crucial part in improving voice assistant skills, resulting in the emergence of **Large Language Models** that support their comprehension and interactivity. The Chapter delves more into *Dialogue State Management*, which is essential for ensuring consistent interactions. It also explores the adoption of these technologies in the **Peer-to-Peer Payments** industry, emphasizing the obstacles and possibilities. Finally, it discusses **iOS development**, highlighting the platform-specific factors involved in implementing voice assistants in Apple's ecosystem.

2.1 Evolution of Voice Assistants

A **Voice Assistant** is a software that interprets and responds to voice commands to perform various digital tasks, leveraging technologies like Speech Recognition and Natural Language Processing. The growth of voice assistants is closely connected to the historical evolution of virtual assistants and conversational agents in general, all of which rely on advancements in NLP and Natural Language Understanding (NLU) approaches. Apart from rudimentary experiments in the early decades of the 1900s involving voice-activated machines, the journey started with the *ELIZA* experiment in the 1960s, the first real attempt to simulate human communication using a computer, which established fundamental notions for conversational agents.

ELIZA, however rudimentary, illustrated the possibility of machines imitating human conversation [1]. This idea had been developed by the mathematician Alan Turing in 1950, who invented a test (the *Turing Test*) which assesses a machine's capability to display intelligent behavior that is indistinguishable from a human's [2]. *"Can machines think?"* was the question raised by Turing, implicitly laying the foundation of the *Natural Language Processing* and inspiring the next generations of research in the field.

Advancements in technology led to the development of conversational interfaces, progressing from basic text interfaces to complex Voice User Interfaces (VUIs), which transformed the way humans engage with machines. This progression resulted in the development of different virtual assistants, each representing a significant advancement in technological capacities. For example, in the 1960s, IBM's *Shoebox* could recognize spoken numerals and mathematical orders [3]; while in the 1970s, Carnegie Mellon University's *Harpy* speech recognizer could interpret over a thousand words [4].

As these technologies advanced, the difference between **open-domain** and **task-oriented** conversational bots became more evident. *Open-domain* agents are created for ongoing, unrestricted interactions like human conversation, while *task-oriented* agents prioritize carrying out specified tasks according to user instructions. The emergence of Conversational User Interfaces (CUIs) combined conversational agents' functionality with voice commands' intuitiveness, improving user accessibility and interaction with digital systems. However, the deployment of voice assistants in different fields, including smart homes and healthcare, has revealed various obstacles, such as providing responses that are contextually appropriate, safeguarding user privacy, and addressing language and dialect differences.

Some significant advancements in the digital assistants were made from the 1990s up to the present day, starting with IBM which introduced Speech Recognition (SR) on their Personal Computers (PCs). But the era of modern voice assistants started with Apple's **Siri** [5], the first integrated into a smartphone, later followed by Amazon's *Alexa* [6] and *Google Assistant* [7], all leveraging advanced AI algorithms. These assistants utilize Deep Learning and extensive datasets to comprehend and analyze human speech more accurately, hence enhancing accessibility and convenience in daily activities. Nonetheless, Voice Assistants still encounter technological constraints in comprehending intricate queries, recognizing human emotions, and delivering contextually suitable responses in all scenarios, despite their progress. As these technologies advance, they are expected to increasingly blend human and machine interaction, enhancing the accessibility and user-friendliness of digital services worldwide.

2.2 Artificial Intelligence in Voice Assistants

Artificial Intelligence (AI) plays a key role in the voice assistant technology evolution, especially in its area of *Natural Language Processing (NLP)*. AI has significantly improved virtual assistants by enabling them to better understand and engage with people through methods that imitate cognitive abilities, especially leveraging the advancements in the latest decades regarding *Machine Learning* models and *Deep Learning* in particular.

Machine Learning (ML) is the subset of AI which enables machines to learn from data and improve over time from the experience. **Deep Learning (DL)**, instead, is a ML approach involving complex Neural Networks and is at the core of these developments thanks to the extremely high performances of those models. It has greatly enhanced features including text classification, text production, Speech Recognition, and Text-to-Speech conversion. The advancements have enhanced voice assistants, making them more skilled, intuitive, and versatile tools that can facilitate intricate human-machine interactions with exceptional ease and efficiency.

2.2.1 Natural Language Processing

Natural Language Processing (NLP) is an evolving discipline that combines *Linguistics* and *Computer Science* to empower machines to comprehend and analyze human discourse, thus focusing on the interaction between computers and humans through natural language [8]. The advancement from basic rule-based algorithms to complex Machine Learning and Deep Learning models is a notable technological progression which happened at the end of the previous century and evolved in the last years, thanks to the emerging *Large Language Models* (subsection 2.3). This progress resulted in the rise of the **Natural Language Understanding (NLU)** sub-area, which concentrates on understanding the context and purpose of the text for a certain language.

NLP originally included just subfields like syntax analysis, semantic analysis, and pragmatics, which focus on language structure, meaning, and use. The incorporation of NLU in the last period has enabled more sophisticated analysis of language, leading to progress in fields such as sentiment analysis, machine translation, question answering, text summarization, *intent classification*, *text generation* and in general voice-activated assistants capabilities, like *Speech Recognition* and *Speech Synthesis*

[9]. The following subsections will examine some important aspects of those subfields, offering insights into relevant factors characterizing the structure of the Voice Assistant in this project.

2.2.2 Speech Recognition

Speech Recognition, also known as *Speech-to-Text*, is a crucial element of human-computer interaction that involves a computer system's capacity to understand human speech by processing and interpreting it, translating it into text [10]. The speech is a sound wave, that is a time series of pressure values over time and characterized by amplitude and frequency. In the case of a human speech, multiple types of sound can be distinguished with respect of the specific language's phonemes, which can be hierarchically grouped in words, phrases and sentences, each with a grammatical and syntactical meaning. A Speech Recognizer is a system capable of capturing sound waves and identify all those groups, reconstructing the speaker utterance.

The evolution of **Deep Learning** in speech processing has been marked by several key models and architectures that have significantly advanced the field, like *Hidden Markov Models (HMMs)*, *Recurrent Neural Networks (RNNs)* and in general different kind of *Deep Neural Networks (DNNs)*. The newest solutions resort to **transformer-based** models, encoder-decoder architectures which can produce an output of different length than the input, and based on *attention* mechanisms (see 2.3.1).

There are some systems that require "training" (sometimes called "enrollment"), in which a speaker reads text or specific words to the machine. This technology utilizes the individual's unique voice to enhance the accuracy of Speech Recognition, and it is referred to as **"speaker dependent"**. Systems that operate without the need for training are referred to as **"speaker-independent"** systems. They are considered more advanced due to their increased versatility in assistant usage, but they also pose security problems as their services can potentially be accessed by anybody, including malicious users.

This field has revolutionized user-device interaction by allowing hands-free control and assisting with activities like dictation and command execution in applications like virtual assistants, customer service automation, and **accessibility** solutions for individuals with physical disabilities. However, Speech Recognition has also some limitations. Accents, dialects, and ambient noise might hinder accuracy, resulting in misunderstandings and thus compromising the user experience. Furthermore, the technology's dependence on extensive data and computational resources may lead to **privacy** and resource consumption issues. Despite obstacles, the continuous progress in Deep Learning and Natural Language Processing is improving systems' performance, ensuring more smooth and comprehensive human-computer interactions.

2.2.3 Text Classification

A key task in Natural Language Processing is **text classification**, which in the context of Machine Learning refers to the process of classifying text into predetermined groups or classes. This method is used in various applications, including sentiment analysis, spam identification, subject labeling, and *intent classification*. The latter is essential for comprehending user commands or requests in conversational agents and voice assistants.

When describing text classification, it is crucial to differentiate between *sequence-to-sequence (seq2seq)* models and *sequence-to-vector (seq2vec)* models. Seq2seq models, like the ones employed in **Named Entity Recognition (NER)**, function at by converting a sequence of text into a different sequence, where each input token (e.g., word or letter) corresponds to an output token and is accordingly labeled. NER, also referred as *Entity Extraction*, is a NLP task aiming at isolate and categorize specific *entities* inside a sentence falling into specific classes, like people names, organizations, places, and so on, and it is usually accomplished with seq2seq models classifying each single token. This is especially beneficial for tasks that necessitate detailed annotation at the token level, including recognizing and categorizing elements within a sentence.

Seq2vec models, commonly used in **intent classification** tasks, transform a text sequence into a static vector that encapsulates the semantic essence of the entire sequence. Subsequently, this vector is utilized to categorize the sequence into one of multiple predetermined groups. In the case of intent classification, each vector is associated to a specific *intent* the user expressed in the original sentence. Seq2vec models consolidate information into a singular, comprehensive representation, which is beneficial for tasks requiring the determination of the general meaning or category of a sequence.

2.2.4 Speech Synthesis

Speech Synthesis, sometimes referred to as *Text-to-Speech (TTS)*, is a process that uses ML models and NLP techniques to convert written text into spoken utterances [11]. It is therefore the opposite task of *Speech Recognition*. This method enables a wide range of uses, including aiding visually impaired individuals, providing voice guidance in navigation systems, generating automated customer service replies, and enhancing the naturalness and engagement of interactions in voice assistants and conversational agents.

Advanced Machine Learning models have significantly enhanced voice synthesis systems, improving their naturalness and adaptability across many conditions and languages. In particular, *Deep Neural Networks* and **Transformer-based** models have improved speech creation by creating utterances that closely mimics human intonation and emotion. Yet, they might have drawbacks such as occasional artificial intonation or mispronunciations in intricate language situations.

The Voice Synthesis process is usually split in the following steps to transform text into a spoken waveform:

- *Text Normalization*: it is the conversion of text into a format that can be understood by machines, achieved by expanding abbreviations, numbers, and symbols into their verbal forms;
- *Phonetic Analysis*: it entails transforming standardized text into phonetic or phonemic forms to indicate pronunciation. This stage often involves using phonetic dictionaries or rule-based algorithms to transform text into phonemes;
- *Prosodic analysis*: it examines linguistic features such as intonation, stress, and rhythm in speech production to achieve natural-sounding speech;
- *Waveform Synthesis*: it is the process of creating sound output based on phonetic and prosodic data. This can be achieved through many techniques including concatenative synthesis, which merges recorded voice segments, or parametric synthesis, which uses models to generate speech from scratch.

2.2.5 Text Generation

Text generation, known also as *Natural Language Generation (NLG)*, is an essential aspect of NLP that refers to the automated production of written information by machines [12]. This method serves as the foundation for a wide range of applications, including the automated production of news stories and reports, the development of realistic discourse in conversational agents, the improvement of virtual assistant user experiences, and even support for the process of creative writing.

The **Generative AI**, a type of Artificial Intelligence oriented to producing new data instances that mimic the training data, is fundamental to this discipline. This includes not only text but also images, movies, and other media. Generative AI algorithms acquire patterns, styles, and structures from extensive datasets to create unique outputs that imitate the acquired formats.

Integrating *Natural Language Understanding* into text-generation systems is essential for producing coherent and contextually appropriate documents. These systems are able to comprehend the peculiarities of human language, such as syntax, semantics, and even emotional tones. This results in outputs that are not just grammatically accurate but also engaging and context-aware.

Significant milestones have characterized the development of models in the NLP field for text generation. Initial attempts utilized rule-based systems and basic statistical techniques, which later advanced to complex Machine Learning models such as *Hidden Markov Models* and early *Neural Networks*. Sequence-to-sequence models, **attention** mechanisms, and *Recurrent Neural Networks* have enhanced text generation by providing more sophisticated and contextually sensitive capabilities. A significant change occurred with the introduction of **Large Language Models**, which have transformed the field of text generation, significantly improving the quality, versatility, and applicability of generated text, as described in the next section.

2.3 Large Language Models revolution

A **Language Model (LM)** in computational linguistics is a model that calculates the likelihood of a sequence of words appearing in a language. In Machine Learning, a *Language Model* is a probabilistic model which is aware of the words' statistical distribution in a specific language, thus simulating an understanding of language patterns and giving the impression of being conscious about it. It uses preceding words to anticipate and generate upcoming words, playing a crucial role in activities such as text generation, Speech Recognition, and machine translation [13].

The first examples of Language Models have been the **n-gram models**, pure statistical models which predict the next word of a sentence based on the previous $n-1$ ones. For example, a *bi-gram* model predicts the next word based on the conditional probabilities with respect to the previous one. N-gram models have the big limitation of increasing exponentially their size with respect of the vocabulary and the parameter n. Yet, their restricted comprehension resulted in the advent of neural based models, like **Recurrent Neural Networks (RNNs)**, which provide more extensive contextual understanding and overcome the problem of the *curse of dimensionality*. RNNs were mostly used to solve *sequence-to-sequence* tasks, and became more dominant especially with the advent of **Long Short-Term Memory networks (LSTMs)**, which solved the classic Neural Network problem of the *vanishing gradient* and were also able to get bi-directional context in a sentence. The major problems associated with these models were the complexity in the architecture and the slowness in training, as well as the limits regarding the comprehension of both the right and left context, which were considered separately and just concatenated at the end.

The current revolution came with the *Transformer* Neural Network architecture in the last years, which brought to the advent of **Large Language Models (LLMs)** like GPT and BERT. These new models are extremely powerful in understanding sentences' context and they are trained on huge datasets, vastly outperforming predecessors in the comprehension and the generation of human-like text, finding applications in multiple tasks like content creation, summarization, machine translation and beyond, marking a significant evolution in NLP.

2.3.1 Attention is All You Need

The **Transformer** architecture, presented in the famous 2017 Google paper *"Attention is All You Need"*, revolutionized NLP by introducing the **attention** mechanisms to capture overall dependencies in the text [14]. In the paper the architecture is mainly presented for a Machine Translation task, showing significant advantages in terms of performance and time to train. Furthermore, the *Transformer* allows for parallel computations, which enhance efficiency compared to previous models that processed data sequentially, and better leverages the current GPUs hardware capabilities. The architecture is a massive Neural Network made of two distinct parts: an *encoder* and a *decoder*. The **encoder** converts an input sequence of text into continuous numerical representations, which are then utilized by the **decoder** to produce an output sequence. The *attention* mechanism, used by both modules, enables the model to assess the importance of various words in the sequence with respect to the others, improving comprehension of context.

The overall architecture is summarized in figure 2.1, taken from the original paper. The left part represents the *encoder*, while the right one represents the *decoder*. In simple terms, this is the general process of inference:

1. an input text sentence enters the model as a sequence of **tokens**;
2. the *encoder* produces in parallel a context-aware numerical representation of each token (**embedding**);

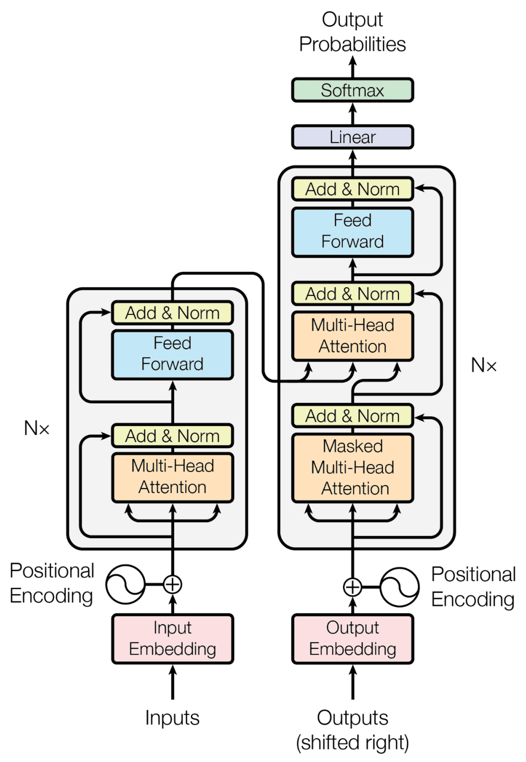


Figure 2.1: The Transformer architecture.

3. the *decoder* takes those embeddings, together with the output sequence generated up to that moment, and generates the **next token** of the sentence.

4. the *decoder* job is repeated, generating one token at a time, until the "*end-of-sentence*" is reached.

Now follows a more detailed overview of the major operations happening in the network [15]. Input sentences are usually processed in batches to both leverage parallelization and reduce the training time, and each input sentence is split into tokens, which are padded to a fixed *sequence length* specific to the architecture. These tokens are then mapped to a numerical representation by means of a pre-computed vocabulary, resulting in the actual *input embeddings* (red block in the figure). Since all the tokens are processed in parallel, in order to preserve the positional information a *positional encoding* is computed, using *sin* and *cos* formulas, and added to the embeddings.

The core of the *encoder* operations is represented by the **multi-head attention** block. In each "head", is computed a distinct **Scaled Dot-Product Attention** of the same input, as shown in the following formula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

Q, K and V are matrices representing three different abstract aspects of the input data, computed by an initial Feed Forward layer network [16]:

- Q is the *Query* matrix, indicating "What we are looking for";
- K is the *Key* matrix, indicating "What we can offer";
- V is the *Value* matrix, indicating "What we actually offer".

d_k is just the main dimension of K, and is used as a scaling factor to prevent numerical operation instability. The resulting *attention* matrices of the multiple *heads* are then concatenated, and after a *layer normalization* and another *Feed Forward* layer processing, they produce a final tensor representation of the input data which is very well context aware and represents the **attention** intensity of each input token with respect to the others. In this *encoder* part, in fact, we talk about a **self-attention** mechanism, since it is computed between two sequences made of the exact same tokens.

The *decoder* part is made of similar foundation blocks. It receives as input the output embeddings generated up to that moment and computes first a **masked multi-head attention**, similarly to what described for the *encoder*. It is called "masked" because an additional mask is applied before computing the *softmax* operation as in 2.1, in order to mask out the next tokens for the attention computation of each token: in this way, we prevent the resulting prediction to depend on future tokens, which would result into "cheating". Next, an additional *attention* sub-layer is adopted, which integrates the result of the encoder and establishes the actual connection between the input language and the output language representations. Indeed, here the Q matrix comes from the *decoder* part, while K and V comes from the *encoder* embeddings, producing as result an attention matrix which very effectively represents the relationships between the input words and the output ones. A final *linear* layer maps these values into actual probabilities for each vocabulary output token, and the most likely is picked up as the new generated token, thus converting the *attention* numerical representations into human comprehensible information.

The described *Transformer* model constitutes the foundation for all the subsequent **Large Language Models**, which further explored the potential of the *attention* mechanisms introducing new elements of innovation. In particular, this architecture paved the way for advanced models like **BERT**, which excels in understanding context in Natural Language Understanding tasks, and **GPT**, known for its generative capabilities.

2.3.2 BERT

The advent of the **BERT** model, which is the acronym for **Bidirectional Encoder Representations from Transformers**, was a significant advancement in the field of Natural Language Processing. This innovative model was introduced in 2018 in the research paper *"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"*, by Jacob Devlin and his team at Google [17]. BERT is based on the *Transformer* architecture: it is constituted by many Transformer **encoder** blocks organized in a stack. This method allows for the execution of numerous NLP tasks and is distinct from previous models that were unidirectional, thereby limiting their understanding of linguistic context. Indeed, BERT's architecture stands out due to its use of **bidirectionality**, which allows the model to have a deeper understanding of linguistic context [18].

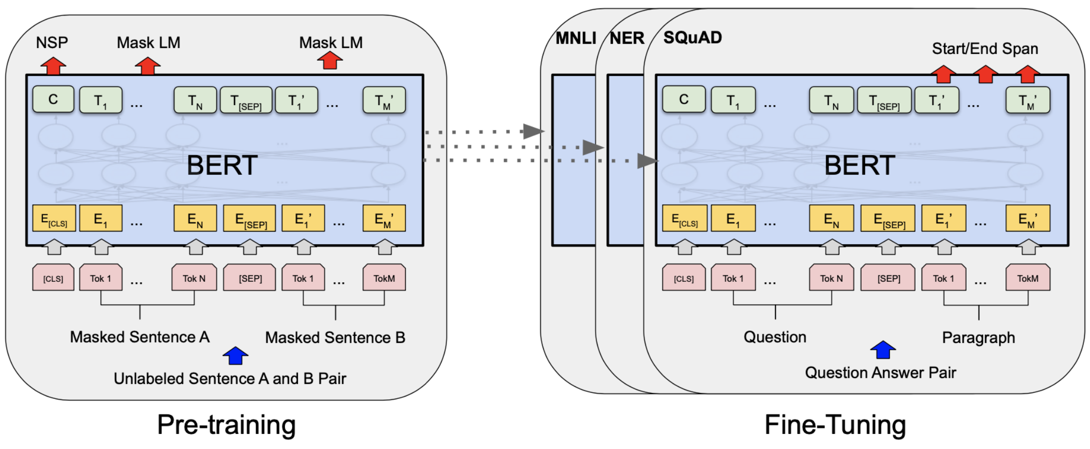


Figure 2.2: BERT training phases: *pre-training* on the left and the *fine-tuning* step on the right (in this example, for a question answering task)

Another important BERT's peculiarity is that its training process is divided in two stages: *pre-training* and *fine-tuning*, as described in figure 2.2. During the **pre-training** phase, the model carries out two simultaneous tasks using unlabeled data: *Next Sentence Prediction (NSP)* and *Masked Language Modeling (MLM)*. BERT learns to anticipate if one sentence logically succeeds another in the NSP problem, gaining a comprehension of sentence relationships. To accomplish this task, each input is made of two sentences which are 50% of the times sequentially taken from the same document, and 50% of the times chosen from different texts. These two sentences are preceded by a special [CLS] token, which stands for "classification", and are divided by another special [SEP] token. The first output token perform thus a binary task, predicting if the two input sentences are consecutive or not. The MLM challenge, instead, requires randomly replacing words in the input text with masks and predicting them using context from the other unmasked words in the sequence. In particular, 15% of the input tokens are selected for this task and each of them:

- is replaced with a [MASK] token with 80% of probability;
- is replaced with a random word with 10% of probability;
- is not replaced at all with 10% of probability.

Each output token corresponding to a mask is in charge of predicting the masked word from the bi-directional linguistic context of the sentence. These two pre-training techniques allow the model to develop a profound comprehension of language syntax and semantics [19]. In the original paper, the model has been trained on a huge unlabeled dataset deriving from a famous book corpora called *BooksCorpus* and the English *Wikipedia* text passages.

Following *pre-training*, which is significantly more computationally expensive, the model undergoes a **fine-tuning** process tailored for specific downstream tasks. To do this, the final fully connected layer is replaced with a tailored one created for the particular task, such *text classification* or *question answering*. While the new layer is learned from scratch, the pre-trained parameters of the rest of the model are only slightly adjusted. This makes the **fine-tuning** process very **efficient** and **fast**, leading to outstanding performance in many tasks. Furthermore, Google publicly released the pre-trained model as *open-source*, making it accessible as a *TensorFlow model* at this GitHub repository (https://github.com/google-research/bert) [20]. They also published it in the major web platforms for Machine Learning models, like *Huggingface* and *Kaggle*, and the main Machine Learning Software libraries started providing and promoting their pre-trained implementation too. This makes very straightforward to create a custom state-of-the-art model leveraging this pre-trained version and fine-tuning it for a specific task, obtaining impressive results in a very short time, and using very few computational resources.

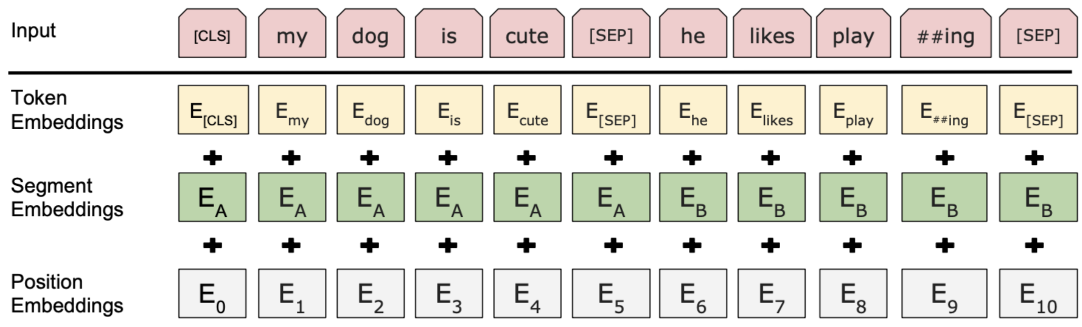


Figure 2.3: BERT input encodings: word embeddings + segment embeddings + position embeddings

After examining BERT's complex training stages, let's focus on its crucial input representation mechanism. BERT is characterized by a specific text **input representation**, as shown in figure 2.3. The model employs the sum of three types of embeddings: *token embeddings*, utilizing *WordPiece* tokenization (introduced in [21]), *segment embeddings*, to distinguish between sentences in the input, and *position embeddings*, indicating the position of each token in the sequence. The *token embeddings* are thus obtained after a first tokenization process for the input sentences, which splits them into words or sub-words using a 30k vocabulary, and classifies unknown pieces with the special [UNK] token; and a second step where these tokens are mapped to integer numbers. BERT's comprehensive input representation technique allows it to effectively handle many NLP tasks that need understanding of either word-level and sentence-level context. The model **output embeddings** are, instead, very meaningful numerical representations of the input tokens, in the form of arrays of continuous values. Those tokens are in fact mapped to a multidimensional space characteristic of the Language Model, where vectors represents very context-aware word representations. This marks an important step forward into the field of *word embeddings*, surpassing well-established techniques like *Word2vec* [22] and *GloVe* [23], which produce just static embeddings without capturing the different nuances of meaning that a word might have in multiple contexts.

Two variants of BERT are featured in the original work: *BERT Base* and *BERT Large*, differing in size and capacity. The **BERT Base** model consists of L=12 Transformer encoder layers, a hidden size of H=768, and A=12 attention heads, for a total of 110 million of parameters. The **BERT Large** model consists of L=24 Transformer encoder layers, a hidden size of H=1024, and A=16 attention heads, comprising 340 million of parameters. The adjustments offer adaptability in terms of computer resources and work complexity. For *hidden size H* is meant the size of the numerical representation array for each input token, that is the output encodings' subspace dimensionality.

The NLP community has developed other models inspired by BERT's architecture to achieve different objectives following the success of the original model. These variations are mainly designed for environments with limited computational resources to be more concise and efficient, like the BERT configurations introduced by Google in 2019 in *"Well-Read Students Learn Better: On the Importance of Pre-training Compact Models"*, like *BERT-Tiny*, *BERT-Mini*, *BERT-Small*, and *BERT-Medium* [24]. Subsequent versions including *ALBERT*, *MobileBERT*, *RoBERTa*, and *DistilBERT* have further expanded the scope of BERT-based models. Each version offers unique advantages suited to specific applications and constraints. For example, **ALBERT** introduces parameter-reduction techniques for improved scaling [25], **MobileBERT** is optimized for mobile environments [26], **RoBERTa** modifies the pre-training procedure for improved performance [27], and **DistilBERT** is a distilled version that retains most of its performance while being significantly smaller [28].

Nowadays, BERT-like models are key elements behind every major *Search Engine* queries, providing state-of-the-art Natural Language Understanding capabilities to better answer to users' needs.

2.3.3 GPT

OpenAI's introduction of the **Generative Pre-trained Transformer (GPT)** models represents a significant advancement in the field of Natural Language Processing and AI in general. The GPT series originated with the initial model, as described in the 2018 paper *"Improving Language Understanding by Generative Pre-Training"* [29]. This model served as the basis for further versions such as GPT-2, GPT-3, up to the current GPT-4. The models have significantly changed the field of NLP by showcasing exceptional skills in producing text that resembles human writing, comprehending context, and executing tasks without needing unique training for each task.

The GPT architecture is derived from the *Transformer* paradigm, similarly to BERT, with a specific emphasis on the **decoder** element. While BERT uses the Transformer *encoder* for bidirectional context comprehension, GPT uses the Transformer *decoder* to produce text. The model consists of stacked layers of decoders that process input tokens sequentially to anticipate the next word in a sentence by taking into account the words that come before it. GPT's sequential processing capacity allows it to produce cohesive and contextually relevant content throughout long periods.

GPT models are trained in two basic stages too: unsupervised *pre-training* and supervised *fine-tuning*. During **pre-training**, the model is trained on a large amount of text data to predict the next word in a phrase based on the words that come before it, without any task-specific training. This step enables the model to develop a comprehensive understanding of language patterns, syntax, and context. The model is then **fine-tuned** for certain purposes by training on a smaller dataset tailored for that task. This method entails the pre-trained model's weights to excel in tasks like translation, question-answering, and text summarization, among others. The *fine-tuning* stage, in addition, becomes essential for customizing the overall linguistic abilities of GPT for specific applications, improving its performance on tasks that demand specialized knowledge or context.

GPT models, similarly to BERT, also use a combination of *token embeddings* and *position embeddings* to encode input text. **Token embeddings** transform words into vector forms to capture semantic details, while **position embeddings** indicate the sequence of words, allowing the model to comprehend sentence structure and flow. GPT can process and generate text that is contextually accurate and structurally cohesive due to this representation.

OpenAI developed more sophisticated iterations of the original model, such as GPT-2 and GPT-3. Each successive model has been distinguished by a growth in scale, encompassing the expansion of the pre-training data size and the sophistication of the model architecture, such as the increase in parameters, layers, and attention heads. **GPT-3** has attracted considerable interest because of its large number of parameters (over 175 billion) and its capability to handle various tasks with minimum task-specific data in a *zero-shot* or *few-shot learning* scenario. Instead, the last two models details, **GPT-3.5** and **GPT-4**, which are at the core of the current ChatGPT software assistant, have not been disclosed by OpenAI, but the size of the latter has been estimated in roughly 1.7 trillion of parameters [30].

GPT models have been utilized in several applications, ranging from generating creative content and providing writing assistance to more intricate activities such as coding support, language translation, and participating in sophisticated human-like discussions. These models are highly versatile and efficient, making them essential in creating AI applications that demand a profound comprehension of language and context.

The GPT series has facilitated continued study in developing more efficient, powerful, and adaptive Language Models as the field progresses. This involves addressing issues including computing efficiency, model interpretability, and ethical problems associated with AI-generated content. But the major problem still characterizing GPT-based models are **hallucinations**, when the model produces text that is either factually inaccurate, incomprehensible, or irrelevant to the input context. This issue arises due to the model's complex design, which does not possess a genuine "understanding" of the content comparable to that of humans. It functions by using patterns acquired from its training data, resulting in cases where it generates material with high confidence but that may be misleading or entirely false.

2.4 Dialogue State Tracking and Management

Dialogue State Tracking (DST) and management are essential elements in the creation and operation of Voice Assistants and conversational agents. These systems attempt to replicate human-like interactions, emphasizing the need of maintaining context and continuity. *Dialogue State Tracking* is the process of comprehending and keeping track of the user's intention along the progression of the conversation. This guarantees that the agent can react suitably and logically throughout an engagement, similar to how a human might in an ongoing conversation. **Task-oriented** conversational agents, for example, created to perform particular activities like booking tickets, making reminders, or giving information, frequently utilize organized tactics to handle conversations and comprehend user intentions.

Ensuring context and continuity in user interactions is crucial for establishing a smooth and authentic conversational flow. For instance, if a user inquires about the current weather and then asks about the weather for the following day, the system should recognize the connection between the two queries without the need for the user to reiterate the initial context. Retaining and linking user inputs across numerous conversation turns significantly improves the user experience. The two major architectures used to accomplish this objective are:

- the *frame-based* architecture;
- and the *dialogue-based* architecture.

The **frame-based architecture**, introduced by the *General Understanding System (GUS)* system, is an early approach in dialogue management, introduced in

1977 in *"GUS, a frame-driven dialog system"* [31]. A **"frame"** in this architecture is a data structure that arranges the information needed for the agent to comprehend and meet a user's request. An instance of this would be seen in a travel booking application, where a *frame* could correspond to the user's intent of booking a new trip, involving sections for the destination, travel dates, number of passengers, and other relevant details. The agent aims to populate these *slots* with the information provided by the user during the discussion.

Within the realm of improving dialogue management systems, the **dialogue-based architecture** represented a notable progression from the frame-based models first established by systems such as GUS. This design utilizes a **Finite State Machine (FSM)** to handle the intricate conversation flow in a dynamic and adaptive way. The FSM method enables a systematic and adaptable approach to managing dialogues, where each state represents a distinct context or stage in the discussion, and transitions are controlled by predetermined rules activated by user inputs or system events. The dialogue-based architecture has been presented in 2002 in McTear's *"Spoken Dialogue Technology: Enabling the Conversational User Interface"* [32]. This design incorporates a complex interaction of different modules that collaborate to analyze user inputs, manage dialogue states, and generate system replies, rather than just switching between states. This advanced design facilitates more organic and effective dialogues between users and conversational agents, enabling voice assistants and other conversational interfaces to be seamlessly incorporated into everyday tasks and interactions. To summarize, the *dialogue-based architecture* prioritizes state management, adaptability, and the integration of NLU and NLG components, marking a significant advancement in creating really conversational user interfaces. More details about this architecture will be presented in section 4.5, for the design part, and in 5.3 for the implementation, since a lot of these concepts have been used for the Voice Assistant created in this work project.

2.5 P2P payments applications

Peer-to-Peer (P2P) payment systems are becoming essential in contemporary financial environments, providing a straightforward method for users to transfer money directly to each other through digital platforms. Popular examples in Italy include *PayPal*, *BancomatPay*, and *Satispay*, each equipped with its own mobile application to enable quick and convenient transactions. The rise in P2P payment adoption is mostly driven by the growing prevalence of smartphones and a societal move towards cashless transactions, motivated by the need for fast, simple, and efficient financial transactions.

However, the rapid expansion of P2P payments presents concerns. **Security** issues are the most important ones because these platforms are easy targets for hackers, which could expose users' financial information. Meeting **regulatory compliance** is another major challenge for these apps due to the intricate network of financial laws and regulations that differ by location and are susceptible to modifications. Furthermore, it is crucial to maintain **user trust**, particularly in consideration of the rising occurrences of fraud and scams on P2P systems. Users' trust in these systems is essential for their ongoing acceptance and usage. Therefore, P2P payment providers need to allocate significant resources to implement strong security measures, clear policies, and efficient customer assistance to handle disputes and answer concerns. Achieving a balance between user ease and security is a delicate task that demands providers to consistently come up with new ideas, making sure that the ease they provide does not compromise user trust or financial security. In the future, P2P payments are expected to advance through the use of blockchain and *Artificial Intelligence* technologies to improve security and UI/UX. For this reason, this work aims to explore and suggest innovative solutions to integrate AI capabilities into these types of mobile applications.

2.5.1 Security and Privacy concerns

Integrating voice assistants into P2P payment platforms adds convenience but also brings significant **security** and **privacy** issues. Ensuring the *privacy* of users' financial information is crucial, as voice assistants have the ability to retrieve and analyze sensitive data such as bank account information and transaction records. Securing this data and preventing illegal access is a crucial task that requires modern encryption techniques and strict access controls.

Furthermore, the dependability of voice-activated services in carrying out transactions with precision and security is a major issue. The system needs to accurately understand user commands that may vary in language and complexity and perform transactions precisely to avoid errors that could result in financial losses or unwanted transfers. Accurately comprehending user intents is another critical factor. Voice recognition technology need to be advanced to accurately interpret the nuances of human speech, such as accents, colloquialisms, and indirect instructions. Incorrectly understanding commands can result in incorrect transactions or violations of privacy, leading to the unintentional disclosure or misuse of personal financial data.

These problems highlight the necessity of strong security measures, sophisticated data encryption, and ongoing enhancements in voice recognition technology to guarantee the secure, confidential, and precise utilization of voice assistants in Peer-to-Peer payment systems.

2.6 iOS development

When developing mobile applications for **iOS** platforms, the selection of the programming language and development frameworks significantly influences the capabilities and performance of the applications, as well as the programmer experience. **Objective-C** was the main language for iOS programming at first, characterized by its dynamic runtime, object-oriented features and syntax inherited from C. However, its verbose syntax and intricate memory management presented difficulties, which led the development of a more modern language, called *Swift* [33]. Apple introduced **Swift** in 2014 to provide a more efficient syntax, *Automatic Reference Counting (ARC)* for simpler memory management, and features such as *optionals* and *generics* to improve safety and performance, making it appealing to contemporary iOS developers [34].

The **iOS Software Development Kit (SDK)** includes a wide range of frameworks and APIs that help in creating iOS applications. For example, the *Core Data* framework is essential for efficient data persistence and management, *Grand Central Dispatch (GCD)* optimizes application performance through concurrent programming, and UIKit components offer pre-designed elements for user interface construction. Other useful frameworks in the context of this Thesis work are *CoreML*, a framework to create, train and easily integrate Machine Learning models into iOS applications; **Speech**, which provides the APIs to perform *Speech Recognition* either via Apple's remote services or on device (from iOS 13); and **AVFoundation**, providing a high-level architecture for working with audio and visual media, like recording, editing, analyzing, and playing back media content.

2.6.1 SwiftUI framework

Developers have the option to use either the programmatic *UIKit* or the declarative *SwiftUI* framework to create the user interface of iOS applications. **UIKit**, an imperative framework, is essential for iOS UI development. Developers must precisely specify the layout and behavior of UI components through code, typically using the Model-View-Controller (MVC) design pattern to organize applications.

Developers have precise control over the application's design and interactivity using this method, although it may become uncomfortable for intricate user interfaces.

SwiftUI, a declarative UI toolkit launched with iOS 13, allows developers to describe UI elements and their interactions in a more intuitive and less error-prone way, marking a significant change in approach. *SwiftUI* simplifies the management of rendering and state transitions by representing components based on their desired states, utilizing the reactive programming model. This streamlines the development process and promotes the use of modern architectural patterns such as *Model-View-ViewModel (MVVM)* and *Combine* for managing data flow and asynchronous activities, leading to more scalable and maintainable application structures [35].

Opting using SwiftUI instead of UIKit for iOS app development offers some notable benefits that address the contemporary developer's requirement for code that is more efficient, readable, and easier to maintain. The primary advantages are:

- **declarative syntax**: SwiftUI employs a *declarative* programming paradigm where developers specify the desired behavior of the UI rather than the step-by-step instructions on how to achieve it. This method simplifies the construction of intricate user interfaces, enhancing code readability and comprehension as contrasted with the *imperative* approach of UIKit;

- **less code**: SwiftUI's declarative approach allows for achieving the same UI with less code compared to UIKit. Reducing the amount of code not only accelerates the development process but also decreases the likelihood of defects and errors, resulting in more stable apps;

- **live preview**: SwiftUI fully integrates with Xcode's *live preview* functionality, enabling developers to view real-time previews of their user interface while coding. The instant feedback loop, along with hot reload features, boosts productivity by removing the necessity to build and run the application after each small modification;

- **advanced animations and graphics**: SwiftUI simplifies the creation of complex animations and custom graphics with less code. Its powerful and intuitive APIs enable developers to add sophisticated visual effects and animations that enhance the user experience.

Chapter 3

Requirements Analysis

The integration of AI-powered voice assistants in mobile applications is an important innovation that improves user interaction and operational efficiency in the continuously evolving mobile app industry. This Chapter focuses on the initial stage of **requirements analysis**, which is crucial for tailoring the voice assistant's features to fit the specific needs of users in a P2P payment application context. This analysis establishes the foundation for creating a Voice Assistant that not only makes financial transactions easier but also prioritizes security and privacy by carefully examining the expectations of the users. **Functional** and **non-functional requirements** are meticulously explored, ensuring the voice assistant's design is not only user-centric but also robust and reliable. In the context of the case study application, described in 1.3, it is relevant to consider that the proposed AI solution is supposed to be an *extension* with respect to the current app functionalities.

This Chapter establishes the foundation for future design and implementation choices, ensuring that they are based on a accurate understanding of consumers' **needs** and preferences for a voice-activated financial assistant.

3.1 User Needs

As users explore the various features of a financial app, the possibility of interacting to a voice assistant becomes more than a simple functionality, but rather a means to satisfy the **need** for a more **user-friendly** and **capturing digital experience**. From a user point of view, in fact, the simplicity of speaking basic voice commands to carry out transactions or access in-app services goes beyond traditional navigation, representing a **more natural** way of interaction, as well as a **more accessible** form of communication for people with reduced haptic capabilities or with less digital expertise. Furthermore, the *ethical* design of this interface is of extreme importance. We require as humans that our machine interactions are conducted in a respectful and transparent manner, reflecting the same trust and integrity we expect in human interactions.

Within the domain of financial transactions, a primary user need lies on safeguarding the **confidentiality** of personal information and ensuring **data protection**. Every voice command implies a level of trust that user's financial integrity is protected with the highest level of accuracy.

These demands and expectations are not generated independently but rather from a manifestation of users' *shared experiences*, conversations within the financial community, and my observations of developing technology trends. They represent a collective agreement on the definition of effective user needs in the era of digital technology.

3.2 Functional Requirements

Based on the user needs introduced in 3.1, a suitable set of **system requirements** has been conceived, in order to identify the specific characteristics of the Voice Assistant, either technical or not. A clear distinction between *functional* and *non-functional* requirements is made: the former being more *technical* and *functional* specifications, while the latter being more *high-level* and *feature-agnostic* properties. For the implementation and integration of the aforementioned Voice Assistant into a P2P Payment app, the following major functional requirements have been identified:

- **Financial Operation Assistance through Voice Commands**: The system will provide support to users in executing financial in-app transactions using voice commands. This includes the functionality of allowing users to transfer funds to their contacts, request money from their peers, and get their financial data, such as checking their balances and transaction history. The voice assistant will analyze the user's spoken instructions and transform them into executable commands within the application;
- **Conversational Interaction for Information Gathering**: The system will engage in a dialogue with the user to obtain all the required information for carrying out in-app actions. This process includes a step-by-step conversation in which the voice assistant requests the user for precise information necessary

for the task, such as the monetary value, recipient details for transactions, and confirmation of the transaction information;

- **Feedback and Confirmation Mechanisms**: The assistant should promptly and unambiguously offer feedback regarding user requests and confirmations prior to carrying out delicate tasks, such as making payments. This guarantees that the user stays well-informed and maintains *control* over the assistant's actions;
- **Effective Error Handling**: The system should possess the ability to proficiently handle errors or misinterpretations in voice commands. The system should provide clear instructions to assist the user in resolving issues or present alternative recommendations, ensuring a seamless and effortless interaction;
- **High-Quality Responses**: The assistant has to offer answers that are not only precise and pertinent but also presented in a straightforward and concise manner. This guarantees that users obtain the necessary information in a comprehensible layout, hence improving the *user experience*;
- **High-Quality Voice**: The assistant has to utilize a high-quality voice, which enhances the user's experience by creating a more *authentic* and *engaging* conversation. The voice must possess clarity, pleasantness, and a suitable level of human-like qualities, hence ensuring comfortable and user-friendly interactions.

3.3 Non-Functional Requirements

Following the same principles introduced for the functional requirements, here a set of *non-functional requirements* is presented, which has been identified to accomplish the user needs mentioned in 3.1:

- **Usability** and **Accessibility**: The voice assistant should be *intuitive*, enabling effortless navigation and operation within the app. This includes *user-friendly* voice commands and responses that suit to individuals with different degrees of technology expertise, thus enhancing accessibility, especially for those with visual or physical disabilities;
- **Effective UI/UX**: The user interface should be designed to promote a smooth and effortless interaction between the user and the voice assistant.

This involves the effective communication of information through a *well-organized* and visually *pleasing* structure, together with providing prompt and tailored responses to user interactions, so ensuring a gratifying user experience;

- **High Accuracy**: The voice assistant must reveal outstanding precision in accurately identifying the user's *speech*, categorizing the user's *intention*, and extracting *referenced things*. This level of precision guarantees that commands are comprehended accurately and that the assistant can precisely recognize and handle the particular specifics of user requests, such as the amounts of transactions or the names of recipients;
- **Efficacy in Operations Execution**: The assistant has to demonstrate efficacy in *operations execution* by efficiently carrying out transactions and effortlessly interacting with the payment app's backend systems, in addition to interpreting user commands. This includes the act of transferring money, verifying available funds, and providing details on an account, all performed with a notable level of *reliability* and precision;
- **Low Latency**: User commands must be quickly attended to by the assistant, ensuring a smooth and rapid flow of interaction. The *minimal delay* in the processing phase is essential for user satisfaction, as it guarantees that users will not encounter substantial interruptions that may decrease the overall effectiveness and *user-friendliness* of the application;
- **Security** and **Privacy**: The system must maintain the utmost levels of *data confidentiality* and *transaction security*. This includes the encryption of data, the secure management of sensitive information such as account details and transaction records, and strong authentication measures to prevent unwanted access;
- **On-Device Processing**: In order to improve **privacy** and **speed**, the voice assistant's main functions, such as recognizing the speech and understanding user intentions, should be carried out directly on the user's device. This reduces the need for external servers and addresses privacy issues;
- **Reusability**: The design of the voice assistant should be *modular*, allowing for effortless integration into various applications or platforms. This enables the assistant to be used in a wider range of situations and sectors, and to be easily expanded to do a wider variety of tasks.

Chapter 4

System Design

In this Chapter an overview of the Voice Assistant design will be presented, describing first the overall structure of the created system, and then going into detail on each of its components. The assistant has been conceived as an **independent software module** containing all the necessary elements to accomplish the functional and non-functional requirements introduced in Chapter 3: from the User Interface to the Machine Learning modules performing Speech Recognition and classification, as well as the entire business logic to manage the dialogue conversation state. *Decoupling* the system implementation from its in-app integration, thus creating a **separate** and **generic iOS codebase**, is fundamental to guarantee the Voice Assistant **flexibility**, making it not only suitable for multiple applications, but also easily testable and maintainable in a more effective way by the programmer.

In each paragraph the focus will be on a specific component of the Voice Assistant module, carefully describing its design phase, the input and the output characterizing its functionalities, and its interactions with the other parts of the system. At the end of the Chapter will be also presented the design choices made for the subsequent integration of the Voice Assistant module into an actual P2P payment application.

4.1 Voice Assistant design

Designing a fully functional Voice Assistant, which seamlessly reproduces human-like conversations and helps the user performing specific articulate operations, is clearly an ambitious task. If we especially consider the mentioned system requirements, expecting to deliver a system working entirely **on-device** to guarantee the maximum level of data confidentiality, privacy and security, it becomes even more challenging. For this reason, in order to preserve the feasibility of the task, some little simplifications have been made in the design of the system, which will be gradually described in the corresponding sections of this document.

Inspired by the conversational agents literature discussed in section 2.1, I decided to restrict my Voice Assistant capabilities to the ones of a **Task-oriented** agent. The assistant will support the user in performing specific in-app operations, guiding them step-by-step to the solution and eventually showing the resulting outcome. The primary goal, in fact, is to effectively satisfy the user's need to accomplish a specific task, like sending some money or checking the status of their bank accounts.

Taking a cue from the Dialogue State Management elements described in 2.4, where two different architectures are presented to create a proper conversational agent, the proposed Voice Assistant has been designed with the same fundamental principles in mind. In particular, key elements from both architectures has been selected, creating a unique framework which is complex enough to effectively manage all the possible conversation aspects in a **state machine**, as in the *dialogue-based* architecture, but at the same time maintaining the simplicity given by pre-defined **frames**, identifying user intents and key entities.

Then following the discussion introduced by the aforementioned *"Spoken Dialogue Technology: Enabling the Conversational User Interface"* [32], the Voice Assistant has been similarly split into multiple components. Each one is dedicated to a very specific phase of the user's speech processing, contributing to transform the received information into a suitable format for the next element, up to the final user who receives back a proper answer to their request. The **schema** 4.1 provides a detailed overview of all the Voice Assistant components and their interactions, but for a different task.

The image has been taken from the mentioned paper work [32], and shows an example of how their system would work in a real conversation in the context of trip reservations. Here the Voice Assistant is shown during the conversation passage in which the user specifies the desired point of departure. The user speech, in the form of an acoustic wave, is processed by the first module, which is in charge of the *Speech Recognition* task. The most likely transcript is selected and then processed by a *Natural Language Understanding* module, in charge of identifying the user's intent and the corresponding mentioned information (point of departure, in this case). Those ones are then used by a *Dialogue State Tracker*, which keeps track of the entire conversation history, and a *dialogue policy* is generated to respond to the user's request. This raw answer is then transformed into an actual textual

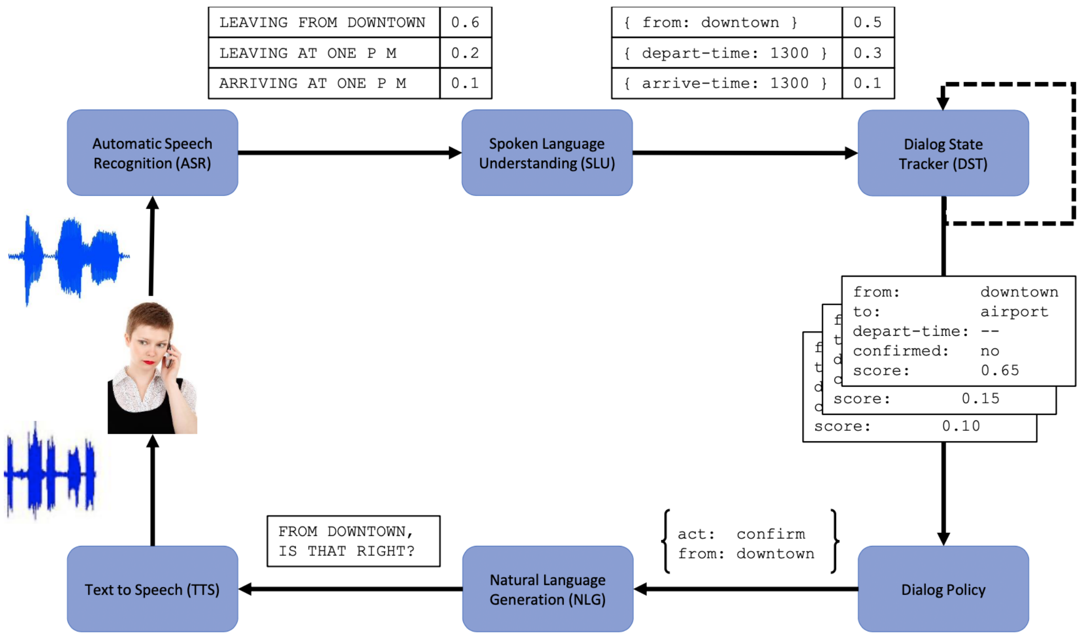

LEAVING FROM DOWNTOWN 0.6
LEAVING AT ONE P M 0.2
ARRIVING AT ONE P M 0.1
{ from: downtown } 0.5
{ depart-time: 1300 } 0.3
{ arrive-time: 1300 } 0.1
Automatic Speech Recognition (ASR)
Spoken Language Understanding (SLU)
Dialog State Tracker (DST)
from: downtown
to: airport
depart-time: --
confirmed: no
score: 0.65
score: 0.15
score: 0.10
act: confirm
from: downtown
FROM DOWNTOWN, IS THAT RIGHT?
Text to Speech (TTS)
Natural Language Generation (NLG)
Dialog Policy

Figure 4.1: Different components of a Voice Assistant interacting to accomplish a user request.

representation by a *Natural Language Generation* module, and finally converted again into audio by a *Speech Synthesizer*, letting the user listen to the answer.

Inspired by the mentioned schema, the following **software components** have been identified for the Voice Assistant proposed in this work:

1. **UI/UX**: in charge of interacting with the user;
2. **Speech Recognizer**: performing the *Speech-to-Text* task;
3. **Text Classifier**: to deal with the *NLU* tasks;
4. **Dialogue State Tracker**: managing conversation state;
5. **Response Generator**: producing the actual response;
6. **Speech Synthesizer**: with *Text-to-Speech* capabilities.

These abstract components are subject to the same logic and the same interactions described above. It's important to highlight that each of these components has been conceived to work **locally** on the user's device, without sending any data over the network, to guarantee the required level of data privacy. The design choices made and their capabilities are be presented in detail in the following paragraphs.

4.2 UI/UX design

The **UI/UX** design for the Voice Assistant prioritizes **simplicity**, **intuitiveness**, and prompt user input. The User Interface is intentionally designed to be basic, with a focus on showing the assistant's responses in a central text box and a big *button* at the bottom for users to *hold* and *record* their questions or commands. This design employs a simple and direct approach to reduce the mental effort required and directs the user's attention onto the process of interacting with the system and receiving feedback.

When a user activates the assistant by pressing the record button, the system promptly starts accepting the input (user's voice) and provides visual or audio signals to indicate that the voice input is being processed. Possible options for this may include incorporating an animated effect surrounding the button, altering the color of the button, or implementing a basic **haptic feedback** mechanism. Feedback is essential since it provides users with the assurance that the system has acknowledged their input and is actively working on generating a response, hence improving the overall User eXperience.

Once the user releases the record button, the system starts analyzing the input and produces a response, which might be just textual or involving also a more complex interaction with the app services, performing actual operations (e.g. perform the transaction). Therefore, there might be a noticeable *delay* from when the user stops talking to when the system returns an actual response. During this waiting time the system is responsible for maintaining an engaging contact with the user. This might be accomplished by including a visual **loading animation** which promptly gives the impression to the user that the system is working.

After the assistant has processed the request, the generated answer is played, but also more feedback is given to ensure continued involvement and effectively manage expectations. This could be a textual message displaying on the screen and that contains the same response given by the assistant, which may come **gradually** and not instantly. The response, in fact, is designed to mimic its human-like generation, as it happens for the spoken response. The objective is to provide the user with regular updates on the state of the system, hence minimizing uncertainty and potential frustration that may arise during periods of waiting.

The described UI/UX design for the Voice Assistant has been centered on the ideas of clarity, efficiency, and constant user engagement. This aims to create a smooth and gratifying connection between the user and the assistant.

4.3 Speech Recognizer design

The **Speech Recognizer** component of the Voice Assistant acts as the primary interface between the user and the system core. It **captures** and **converts** the user's spoken requests into text for further processing. This component utilizes a Machine Learning model to successfully perform the *Speech-to-Text* task, efficiently transforming the user's spoken words into a comprehensible written transcript.

Using the standard tools offered by the **iOS Operative System** for this purpose simplifies the design by eliminating the need to create a tailored model for Speech Recognition. iOS offers indeed a built-in support for SR tasks, which is the same used by *Siri*'s underlying model to assist user's requests for interactions with the Operative System. Utilizing the capabilities of the iOS platform guarantees a smooth integration and functioning of the *Speech Recognizer*, making it an essential and efficient component of the voice assistant's overall design. This strategy not only takes use of the reliability and effectiveness of the platform's built-in capabilities, but also allows you to conform to the system's requirements by prioritizing user privacy and data security. This is achieved by ensuring that all data processing takes place on the device itself, without any external communication, and this is possible thanks to the libraries offered by iOS for this task.

The output of this software component is then a **transcript** of the interpreted user's speech, which can then be processed by the subsequent modules for data processing, in particular by a *text classifier* capable of identifying the key elements of the request.

4.4 Text Classifier design

Crucial to the thesis work, the **Text Classifier** component emerges as the Voice Assistant's main Machine Learning component. This component is responsible for the complex task of *text classification*, which involves analyzing the user's textual input and understanding the underlying *intent* and relevant *entities*. The proposed design includes the development of a **separate ML model** specifically designed for this objective. This represents the core model of the Text Classifier, in charge of reading the user's text, determining the purpose of the text and extracting relevant entities so that the Voice Assistant can respond appropriately.

In order to achieve state-of-the-art performance, the architecture has been designed as an independently implementable module, which utilizes modern Machine Learning tools and frameworks, with a specific focus on utilizing the *BERT* model. The task at hand is well-suited for **BERT** due to its exceptional ability to comprehend **contextual nuances** in text, as described in section 2.3.2. The model, which has been trained on huge amounts of linguistic data, will be subjected to **fine-tuning** to be adapted to the specific requirements of the Voice Assistant. This process will ensure that the model becomes proficient in accurately identifying the different intentions and entities that are pertinent to the *P2P Payments* application's field. After a careful validation procedure, in which the model is fine-tuned with different architectures and hyperparameters' values, the best configuration is selected, according to the results obtained over a proper test dataset.

Embedding the obtained ML module into an iOS application is another crucial element of the design phase. This involves integrating the Machine Learning model into the app, enabling it to operate smoothly within the iOS environment. Thus, the *Text Classifier* becomes an essential module of the Voice Assistant app, collaborating with other parts to receive input from the user and provide relevant responses.

The designed Text Classifier is specifically built to simultaneously perform two main functions:

- **Intent classification**: it includes discerning the user's *goal* from their textual input, regardless of whether it is a request for information, a command, or a query;

- **Entity extraction**: it is the process of *finding* and *isolating* certain relevant pieces of information inside the text, such as people names, amounts, or other pertinent data, which is fundamental for completing the user's request.

Essentially, the Text Classifier is designed as the key component of the Voice Assistant. It has to utilize a **BERT-based Machine Learning model** that has been first independently built and fine-tuned, and then incorporated into the iOS software codebase, with the purpose of effectively manage intent classification and entity extraction. This guarantees that the Voice Assistant can accurately understand user inputs, resulting in a smooth and intelligent interaction experience within the app.

4.4.1 Intent Classification

In order to maintain the feasibility of the task, a **limited set** of possible *intents* has been designed, mostly based on and inspired by the features offered by the case study application, described in 1.3: *check balance*, *check transactions*, *send money*, *request money*, *yes*, *no*, *none*. Those will be the only capabilities actually offered by the Voice Assistant, as well as the only **in-app operations** supported to assist the final user.

The *Text classifier* is then in charge of analyzing the user's transcript and classify it into one of the following instances:

- **Check Balance**: it represents the user's intent to check the current balance of a specific bank account. It must be characterized by *at least one* of the following entities:
  - **Currency**: the currency of the desired bank account;
  - **Bank account**: the name of the desired bank account itself;
- **Check Transactions**: it represents the user's intent to check their made **last transactions**, either as incoming or outgoing ones. This intent might be characterized (*or not*) by the following entities:
  - **Bank account**: the specific bank account name whose transactions the user wants to check;
  - **User name**: the specific user's name involved in the transaction;
- **Send money**: it represents the user's intent to perform a *transaction*, sending some money to another user from a specific bank account. This intent *must* be characterized by the following entities:
  - **Amount**: the specific amount the user wants to send in the transaction;
  - **User name**: the specific *recipient*'s name involved in the transaction;
  - **Bank account**: the specific bank account name used as *source* for the transaction.
- **Request money**: it represents the user's intent to request some money to another user using a specific bank account. This intent *must* be characterized by the following entities:
  - **Amount**: the specific amount the user wants to request;

  - **User name**: the specific *sender*'s name that will be involved in the transaction;
  - **Bank account**: the specific bank account name used as *destination* for the transaction.

- **Yes**: it represents the user's will to *confirm* a certain operation. It is not characterized by specific entities.
- **No**: it represents the user's will to *deny* a certain operation. It is not characterized by specific entities.
- **None**: it represents the absence of any intent by the user. This is particularly useful to still assign a valid '*intent*' when the user just mentions specific entities and nothing else. It might be characterized (*or not*) by any of the previously mentioned entities.

In the context of the *frame-based* architecture described in 2.4, the first four mentioned intents (*check balance*, *check transactions*, *send money*, *request money*) corresponds to specific **frames** for the Voice Assistant, embodying some slots to be filled by specific entity values.

The designed *intents* and the corresponding *entities* are summarized in table 4.1 for better clarity.

Intent	**Entities**	**Frame**
check balance	currency, bank account	yes
check transactions	bank account, user name	yes
send money	amount, user name, bank account	yes
request money	amount, user name, bank account	yes
yes	-	no
no	-	no
none	(any entity)	no

Table 4.1: *Text Classifier* supported intents and corresponding entities.

4.4.2 Entity Extraction

In conjunction with the design of supported *intents*, a limited set of possible **entities** has been defined, to let the Voice Assistant capture the most significant information mentioned by the user. Indeed, the *Text classifier* is responsible not only for the categorization of the user's speech into an actual intent, but also for the *identification* and *extraction* of the relevant **data** regarding the intent, which are fundamental to perform the in-app operations supported by the assistant. For example, if the user express the intent of sending a certain amount of money, it is important to identify this intent, but it is also fundamental to extract the right numerical amount together with its currency, formatting it properly and make this information interpretable by the software.

Below is a list of the *entities* designed for the Text Classifier in order to accomplish the described task, and a brief description of their expected format:

- **Amount**: it includes a **currency**, either as *symbol* or as a *literal* name, and a **sum**, with or without the decimal part, either in a *numerical* or *literal* format
- **Bank**: it is either the *name* of a specific bank, or the indication of the *primary/default* one (each user has defined a primary bank account);
- **Currency**: it is just the information about a currency, either as *symbol* or as a *literal* name
- **User**: it represents a user's name, in the form of *full name*, *first name* or *common name + first name* (e.g. Sister Anne);
- **None**: it represents the absence of any entity (used as a fallback type).

The existence of both currency and amount (which contains a currency too) lies in the decision of simplifying the capturing of those two entities, such that the system will treat an amount as a whole element, instead of a composition of multiple ones. A summary of the mentioned *entities* and their format is also reported in table 4.2 for better reference.

4.5 Dialogue State Tracker design

When building the **Dialogue State Tracker** component for a conversational agent, it is essential to take into account the interactive *roles* of both the user and the assistant during the conversation. Within this particular work, which treats about a *task-oriented agent*, the Voice Assistant has been designed as always initiating discussions by asking questions to the user, who then replies providing additional context and information to the conversation, thereby enabling well-organized and

Entity	**Format**
Amount	currency (symbol or literal) + sum (literal or numerical)
Bank	name or primary/default
Currency	symbol or literal
User	full name, first name or common name + first name
None	(any)

Table 4.2: *Text Classifier* supported entities and corresponding format.

structured interactions. In particular, the proposed architecture follows a **State Machine** model, in which each *state* corresponds to a particular stage in the conversation. This method directs the flow of the conversation based on the user's responses.

A *State Machine* is a well-known computational model in Computer Science that is utilized to define the dynamic behavior of a system, which exists in multiple states [36]. The system experiences a change from one condition to another as a result of external *events*, specifically user interactions in this situation. This paradigm is highly efficient in managing complex interactions in a predictable manner, guaranteeing that the assistant can effectively handle various conversational paths and user objectives.

The *Dialogue State Tracker* here has two main functions: it can either just respond to the user with an **answer** including a query, or in alternative carry out a specific **task**, such as conducting a financial transaction, according to the user's instructions, and provide its outcome at the end. The determination of the output involves analyzing the current state, user input, and the context of the conversation, ensuring the assistant's responses are always relevant and accurate. This component plays a crucial role in providing a smooth and easy-to-use User eXperience. It allows the Voice Assistant to efficiently handle in-app operations and gather relevant information within the P2P payment application.

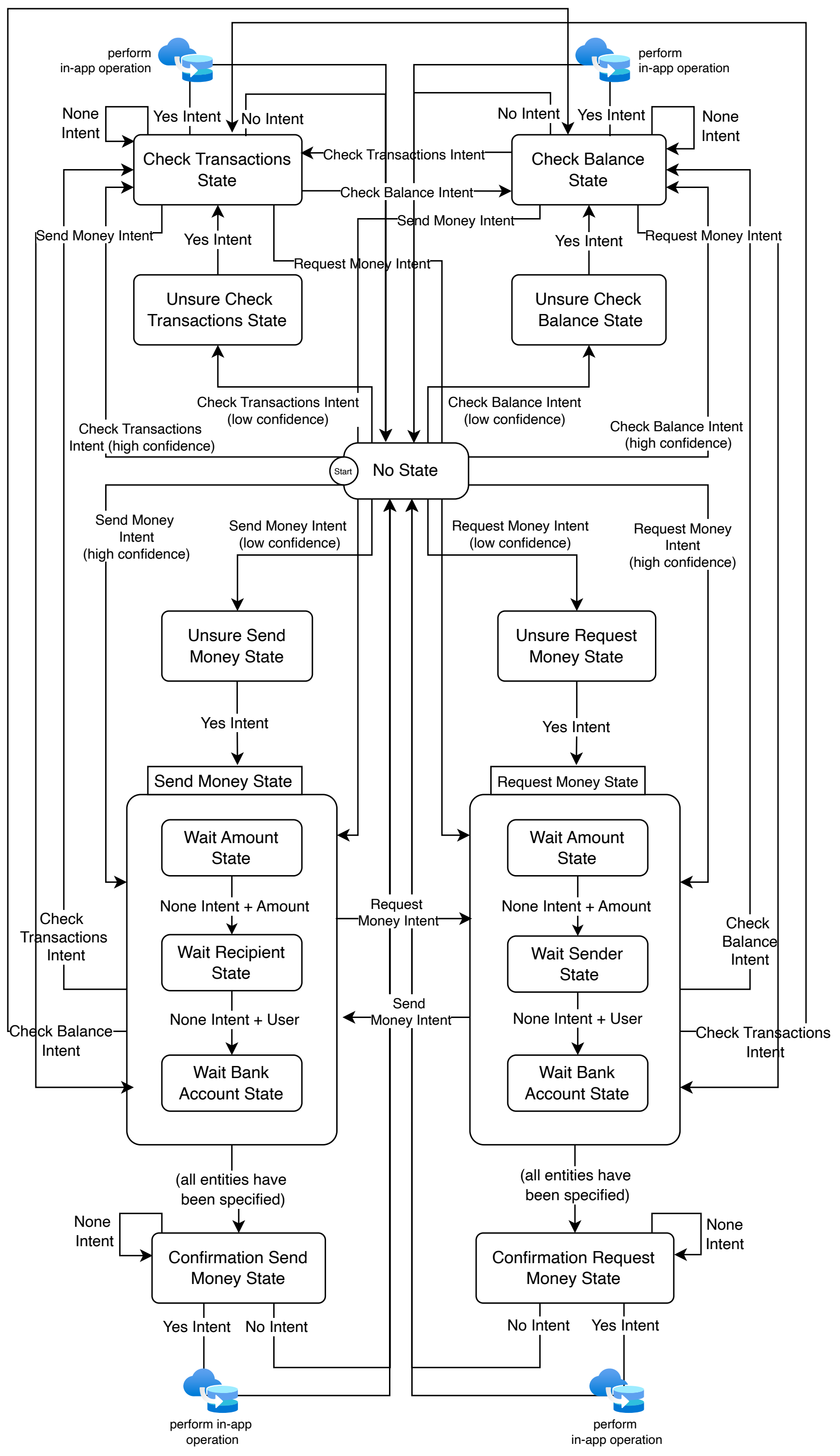


Figure 4.2: *Dialogue State Tracker*: **state machine** simplified diagram.

4.5.1 Main States and Transitions

For the purpose of this specific work, based on the already mentioned considerations on the DST, a finite set of *states* and *transitions* have been defined to design the *State Machine* model and manage all the possible conversation interactions. These *states* specifically reflect the *intents* and the *entities* supported by the *Text Classifier* and defined in 4.4, and in particular they follow the logic of a **frame-based architecture** (see section 2.4), identifying a specific intent context mentioned by the user (e.g. send money) and trying to manage the retrieval of all the possible information related to that *frame* (e.g. amount, recipient, etc.).

The designed *State Machine* is summarized in figure 4.2, where the main *states* and *transitions* are reported in a simplified diagram. The main states are the ones corresponding to a conversation *frame*: **Check Balance State**, **Check Transactions State**, **Send Money State**, and **Request Money State**. The conversation flows to one of these states when the corresponding intent is expressed by the user (e.g. Send Money Intent -> Send Money State). A default state is also considered: the *No State*, which serves as fallback state both at the start of the conversation and after an operation has been successfully requested by the user. During the intermediate phases of the conversation, the State Machine "bounces" between different *sub-states* related to a reference frame (e.g. send money) until all the necessary information are captured in order to perform the specific in-app operation. Each "piece" of information is specified by the user with a *None* intent, expressing the name of the desired entity. To make an example, if the user says "I want to perform a transaction of 3$", the State Machine switch to a *Send Money State*, in particular on the *Wait Recipient State* (since the *amount* information is already filled), and it asks the user for the recipient of that money. The user just expresses the name of the recipient, that will be interpreted as a *None intent* by the Text Classifier, together with the extracted *user entity* (see section 4.4). At this point the State Machine switches towards the *Wait Bank Account State*, waiting for the remaining information to fill, and so on. Once the collected data are enough to accomplish the user's intent, the actual in-app operation is performed (eventually after an additional confirmation stage), and the *No State* is restored.

It is important to mention that some **confirmation strategies** have also been designed, in order to fulfill the requirement expressed in 3.3. In particular, when the intent expressed by the user is identified with *low confidence*, the State Machine has to fall into a temporary *Unsure State*, in which the user has to explicitly specify if that was their intention or not. Furthermore, a *confirmation mechanism* is also provided to perform delicate tasks, like a *transaction* or a *money request*: before executing the job, the State Machine pass by a temporary *Confirmation State*, where the user is provided with a recap of all the information specified so far and an explicit confirmation (*Yes Intent*) or denial (*No Intent*) is requested.

4.6 Response Generator design

The primary objective of this Thesis work is to create a Voice Assistant that can provide realistic responses to user queries. This will be achieved by constructing a component that works just **on-device**, without the need for cloud-based *Generative AI models*. This decision is in accordance with the requirement for **confidentiality**, **protection**, and autonomy from network accessibility, which are essential for a voice assistant incorporated into a P2P payment application.

The **Response Generator** has then been designed as a component utilizing **prepared sentences** and **answer templates**, taking into account these matters. This method simplifies the production of replies and guarantees the reliability and consistency of the system, particularly in the area of financial transactions where accuracy and security are of utmost importance. The templates are carefully designed to address a broad spectrum of user requests, ranging from transaction queries to requests for account information. They are organized in a way that allows them to be automatically filled with pertinent data obtained from the user's interaction context and the application's current status.

An example of a template for confirming a transaction could be as follows: *"The transfer of [amount] to [recipient] has been completed successfully. Is there anything else you would like to do?"*. In this context, the placeholders [amount] and [recipient] are substituted with exact information provided by the user, resulting in a tailored and pertinent response, without requiring complex *Natural Language Generation* algorithms.

The decision to *not* incorporate a more advanced Generative AI model for answer generation was taken after a thorough evaluation of the project's **limitations** in terms of *time* and *resources*. Although these models have the potential to enable more diverse and realistic interactions, they also present major obstacles in terms of computational costs and *integration complexity*, especially for a system designed to function only on a device. This decision highlights the project's practical approach, emphasizing an appropriate balance of functionality and feasibility within the existing limitations. Indeed, the main goal is to provide quick and user-friendly interactions by producing concise responses that are specifically matched to the context of P2P payments.

The mentioned design decisions demonstrates thus a careful balance between desire for sophisticated AI functionalities and the realistic constraints of the project, with the main goal of providing a helpful tool that improves the User eXperience in the iOS payment application.

4.7 Speech Synthesizer design

The **Speech Synthesizer** component also plays an important role in the design of the P2P Payment Voice Assistant, converting the Response Generator's written answer into audible speech. This transformation utilizes the underlying Machine Learning models extensively examined in section 2.2.4 of the literature, which serves as the theoretical basis for *Speech Synthesis*.

However, in order to comply to the design principle of *simplicity* and coherence within the platform, the component leverages the **built-in functionalities** of iOS, particularly those employed by the iOS *VoiceOver* feature. VoiceOver, an assistive technology created to assist individuals with visual disabilities, utilizes the powerful Text-to-Speech tools of iOS to audibly communicate what is displayed on the screen. The *Speech Synthesizer* component achieves thus simplicity and seamless integration inside the iOS environment by utilizing the available internal frameworks.

The component functions in a direct and straightforward manner, accepting the textual responses provided by the *Response Generator* as input and generating **spoken audio** as output. The mentioned design decisions leverages the effectiveness and reliability of iOS's integrated features while also simplifying the development process. This ensures that the component remains lightweight and prioritizes its main goal of providing clear and understandable voice output to users.

4.8 In-App Integration design

It is important to highlight the decision to design the Voice Assistant as a **separate module** that may use specific application data and actions to give responses and carry out specified tasks. The modular architecture guarantees the seamless integration of this software unit into different applications, allowing it to adapt to their specific circumstances and requirements. The assistant can understand and execute app-specific actions by *injecting* customized app data and functionalities within its system, thereby adapting to the specific app environment it is integrated with. By maintaining a clear **separation of concerns**, the Voice Assistant is able to enhance its *flexibility* and *usability*: it focuses on understanding and processing user requests, while leaving the execution of in-app operations to a **dedicated component** within the host application.

This architectural design conforms to the principles of **modularity** and **reusability** in the field of Software Engineering, which enables simpler maintenance, upgrades, and scalability of the software. Additionally, the designed Voice Assistant enables a more secure and efficient implementation by avoiding direct manipulation of app data or operations. Instead, it communicates with a dedicated component that is specifically designed for this purpose. This ensures that operations are carried out within the app's secure and optimized environment. Section 5.9.3 will provide a more in-depth analysis of this design, explaining how custom app data and actions are included into the Voice Assistant and how this design decision affects the overall functioning and performance of the system within the app ecosystem.

4.8.1 App Delegate design

The component designed as a **key intermediary** between the Voice Assistant and the hosting app is the **App Delegate**. It is logically positioned between the *Dialogue State Tracker* and the *Response Generator* in the software architecture. It is specifically responsible for executing the in-app operations as requested by the user. Unlike other components in the assistant design, the *App Delegate* differentiates itself since it is not inherently included as part of the Voice Assistant module. Instead, it functions as an **external unit** that the conversational agent uses to efficiently carry out user-requested tasks. This component is intentionally designed to be **abstract**, requiring the application itself to define the exact implementation details. This allows for a flexible and adaptable integration that is tailored to the unique capabilities and user interactions of the app.

As illustrated in the State Machine diagram shown in figure 4.2, the *App Delegate*'s function is triggered when all the necessary information for a certain task is gathered from the user and verified. This guarantees that activities are exclusively carried out upon a clear user consent, preventing unwanted actions and improving the User eXperience by incorporating an additional level of verification for user commands. This design consideration is crucial, particularly in apps that handle sensitive processes like financial transactions or personal data management, where precision and user permission are of utmost importance.

Essentially, the *App Delegate* component is a crucial design piece that allows the Voice Assistant to expand its capabilities within the app's context, executing activities as instructed by the user. Its abstract design enables adaptable implementation, customized to meet the precise requirements and capabilities of the host application, ensuring that the assistant can efficiently and seamlessly respond to user requests inside the app's environment.

Chapter 5

Voice Assistant Implementation

This Chapter discusses the **software implementation** of the AI-powered Voice Assistant and its **integration** into a P2P payment application, going into details of each technical aspect. In contrast to the previous Chapters, here the implementation phase follows a more chronological order, focusing on the specific steps necessary to make the system operational. This systematic advancement is crucial for comprehending the evolution process from ideation to implementation.

The core aspect of the system involves a set of different **software components**, which have been carefully crafted to operate seamlessly within the iOS ecosystem, as described by the diagram in figure 5.1. Here a brief overview of these components is provided, introducing the implementation details extensively described in the following sections.

- *BERT Text Classifier*: described in 5.1, is the core crucial component that effectively utilizes the ML capabilities of intent classification and named entity extraction, cleverly incorporating **BERT** into the iOS framework by means of a *TensorFlow Lite model*;

- *BERT Preprocessor*: depicted in 5.2 section, is specifically designed for the BERT model, which prepares text input to meet BERT's requirements, hence allowing the classification and the extraction tasks;

- *Dialogue State Tracker*: introduced in 5.3, serves as a multi-functional component, responsible for both monitoring the conversation's state and incorporating

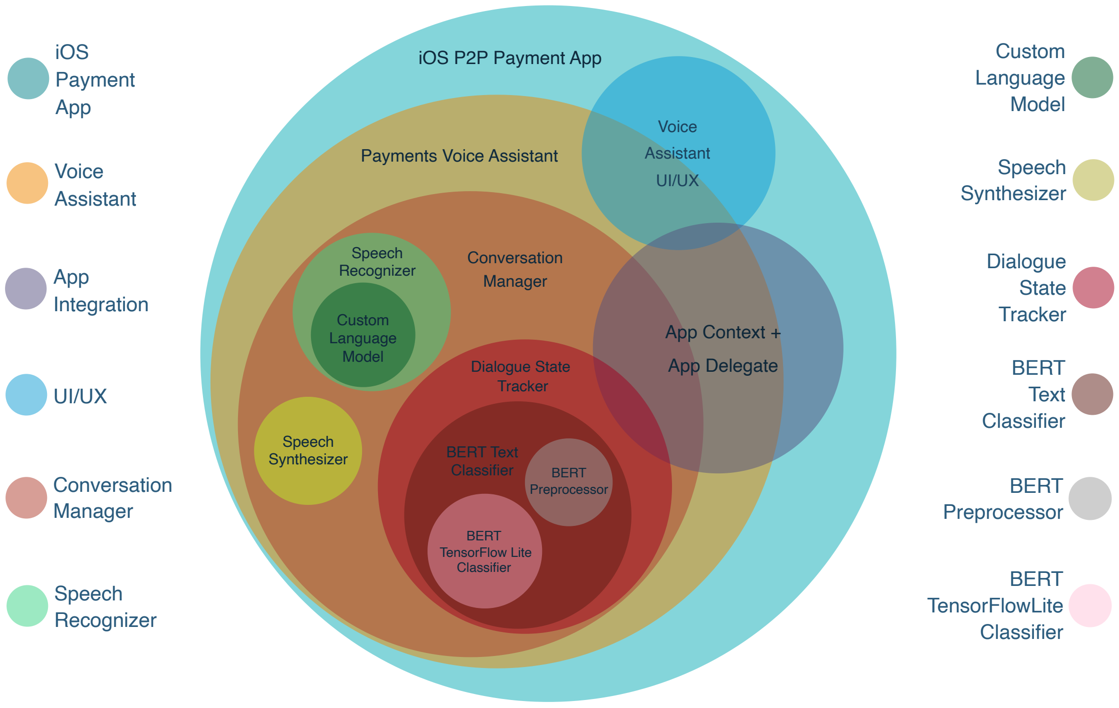


Figure 5.1: Diagram of the Voice Assistant *software system.*

response production capabilities, thus simplifying the system architecture and improving efficiency;

- *Speech Recognizer*: this component effectively manages Speech Recognition by utilizing a *Custom Language Model* to enhance its comprehension and precision, particularly tailored to the unique domain of the application, as portrayed in section 5.4;
- *Speech Synthesizer*: reported instead in 5.5, translates system output into speech, resulting in a user experience that feels authentic and seamless;
- *Conversation Manager*: is responsible for managing a specific conversation with the Voice Assistant, utilizing the DST, the Speech Recognizer, the Speech Synthesizer, and the *App Delegate* to handle inputs, produce meaningful responses, and facilitating in-app operations as well (section 5.6);
- *Payment Voice Assistant* and *UI/UX*: respectively described in 5.7 and 5.8, the former serves as the principal interaction point for the user, encapsulating

the system's functionality and representing the façade of the Voice Assistant; while the latter enhances user interaction with the assistant, making it more intuitive and easily accessible.

The last section (5.9) describes instead the integration process, and an example of actual integration into the case study app introduced in section 1.3.

The entire system's architecture is contained in an independent **Swift framework** module. This decision was mainly due to compatibility problems with the *TensorflowLiteSwift* library, which is not compatible with the *Swift Package Manager (SPM)* at the moment, and would have been a better option in terms of *reusability* and *testability* of the software. This decision highlights the need of compatibility and stability when selecting development frameworks and tools.

A brief overview is needed to clarify the differences between Swift Package Manager and conventional Swift framework modules. The *Swift Package Manager* is a utility for managing the release of Swift code, facilitating the creation of Swift **packages**, their distribution, and the management of dependencies. The standardized framework it provides facilitates package development, adding to the coherence and scalability of the Swift ecosystem. On the other hand, a classic *Swift framework* module corresponds to a **binary** distribution that contains a collection of Swift or Objective-C classes and resources. Basically, the SPM prioritizes high-level package management and dependency resolution, while a traditional Swift framework stresses the encapsulation and distribution of functionalities in binary code. This highlights the contrasting approaches to code management and reuse between the two.

Another important decision made throughout the system's development was the implementation of an interface that is *exclusively* in **English**. The decision was influenced by the utilization of Machine Learning models, particularly the Language Model BERT, which are mainly offered in a **mono-language** version. Furthermore, this decision was motivated by the need to simplify the development process and comply with time and resources limitations.

Overall, the Voice Assistant module, which includes the **User Interface**, **logic**, and **Machine Learning capabilities**, demonstrates careful design and strategic decision-making. This ensures that the system not only fulfills its functional requirements but also aligns with the broader goals of User eXperience and system maintainability.

5.1 BERT Text Classifier

The process of building the **BERT Text Classifier** component within the AI Voice Assistant was the first step, and it was characterized by an organized and thoughtful approach. Due to the complex nature of the *text classification* task and the requirement for a sophisticated comprehension of user intent, it was decided to create a **custom independent model** leveraging state-of-the-art technologies, which would have been later integrated into the iOS codebase. Therefore, *TensorFlow* and *PyTorch* have been the two main alternatives investigated. Indeed, these frameworks constitute the top Python libraries for creating advanced **Machine Learning models**, providing a multitude of tools and capabilities suitable for a variety of AI applications.

The study of the two frameworks was not simply a technical activity, but a conscious choice to ensure that the selected framework would perform effectively in line with the project's overall objectives. The goal was to create a **personalized model** that could accurately *classify user intent* and effectively *extract named entities* from user transcripts. This dual feature was considered crucial for the Voice Assistant to smoothly handle the wide range of customer requests. The decision to construct a customized model came from a desire for **maximum flexibility**. In the fast advancing field of Machine Learning, the capacity to customize models for specific applications is extremely valuable, enabling the refinement and enhancement that pre-made solutions usually lack. This customized design was crucial in guaranteeing that the Voice Assistant would comprehend both the *"what"* behind customer queries and also the *"who"* and *"how much"* at the same time, with equal effectiveness.

After an extensive evaluation, **TensorFlow** was chosen as the preferred framework. Several crucial elements affected this decision. For example, the *original* BERT models, which are fundamental to this project, were initially released in TensorFlow by their authors. This established a strong base to construct upon, guaranteeing compatibility and availability to a broad range of pre-existing resources. Furthermore, the environment of TensorFlow provided the attractive opportunity to make use of **TensorFlow Lite**. This tool aims to optimize the deployment of TensorFlow ML models on **edge devices**, in accordance with the project's goal of developing a responsive and efficient AI Voice Assistant that can function effectively in *mobile environments.*

5.1.1 Dataset generation

Creating an appropriate **dataset** to train the Text Classifier was a major obstacle during the implementation process of the Voice Assistant software. The dataset needed to be carefully designed to efficiently train the model to accurately comprehend and classify user requests. In order to address this issue, a novel method was utilized, leveraging the Natural Language Generation capabilities of *ChatGPT-4* (see 2.3.3) to produce a wide array of **templates** and **entities** linked to the different *intents* the Voice Assistant was expected to manage. This method involved systematically gathering and organizing a diverse range of phrases and sentences that users may use to engage with the assistant, based on the Text Classifier design strategies described in section 4.4, encompassing a wide range of functionalities within the application.

Basically, the generative AI model produced a large set of plausible templates for each designed *intent* (see table 4.1), based on a carefully crafted prompt provided; similarly, for each designed *entity* listed in table 4.2, it generated a lot of possible names and expressions (e.g. first names, bank names, etc.). Later on, **Python scripts** were created to automatically generate sentences by cleverly **merging** the retrieved *templates* and *entities.* By employing this combinatorial approach, it became possible to simulate variations in natural language, resulting in a big **artificial dataset** made of several authentic user requests. Every sentence that was created was carefully **labeled** with the corresponding *intent* and *entities*, which helped the model learn how to classify user intents and extract useful information from the commands. This innovative method was, in fact, highly powerful, allowing to rapidly build an adequate dataset saving time both in data gathering and sample labeling operations.

Intent	**Label**
none	0
check balance	1
check transactions	2
send money	3
request money	4
yes	5
no	6

Table 5.1: Designed *intent labels* for the artificial dataset.

BIO Entity	**Label**
O	0
B-AMOUNT	1
I-AMOUNT	2
B-BANK	3
I-BANK	4
B-CURRENCY	5
I-CURRENCY	6
B-USER	7
I-USER	8

Table 5.2: Designed *BIO entity labels* for the artificial dataset.

Sentence	**Label**	**Intent**
Please display the most recent payments to Lucy Johnson	2	check transactions
Kindly process a transaction of 324 dirhams and one cent to Mila Diaz	3	send money
How much do I have in my bank account right now	1	check balance
Of course I do	5	yes
How do I manage app permissions on my device	0	none
Request to receive some money to my Maybank account from Sophie please	4	request money
I'd have to say no I'm not convinced	6	no

Table 5.3: Example of generated sentences in the artificial dataset, with their associated *intent labels.*

A total of around **30,000 phrases** were generated, each with the corresponding *intent label* and *entity labels*, creating a rich dataset that greatly improved the model's comprehension of user interactions. For the intent, the sentence is treated as a whole, and is provided with a single label. On the other side, for the entities, each sentence is split into multiple *tokens* (which might corresponds to words or sub-words), and each one is provided with a label characterizing the presence (or not) of a specific entity type.

The labels utilized in this process for intents and entities are reported in tables 5.1 and 5.2 respectively.

The labeling process made use of the **Begin-Inside-Outside (BIO)** approach for *Named Entity Recognition* tasks. This scheme played a crucial role in detecting and classifying entities within sentences. In this scheme:

- **"B"** represents the *start* of an entity
- **"I"** represents the *continuation* of an entity;
- **"O"** represents the *absence* of an entity, and is then assigned to tokens that are not part of any entity.

Token	**Label**	**Entity**
kindly	0	O
process	0	O
a	0	O
transaction	0	O
of	0	O
324	1	B-AMOUNT
dir	2	I-AMOUNT
##ham	2	I-AMOUNT
##s	2	I-AMOUNT
and	2	I-AMOUNT
one	2	I-AMOUNT
cent	2	I-AMOUNT
to	0	O
mil	7	B-USER
##a	8	I-USER
diaz	8	I-USER

Table 5.4: Example of a generated sentence in the artificial dataset, with its associated *entity labels*.

For instance, in the command “Send 50 dollars to Alice”, *send money* (3) would be identified as the intent, the words “50 dollars” would be labeled as B-AMOUNT (1) and I-AMOUNT (2) respectively, and the word “Alice” would be classified as B-USER (7). This demonstrates a use case of the model’s capacity to recognize and classify various elements of user requests.

For the sake of completeness, some examples of generated sentences are reported in table 5.3, with their corresponding *intent labels*; while in table 5.4 is reported an example of generated sentence with its *entity labels*. In the latter, a practical example of sentence split into BERT tokens can be seen: some words are taken as a whole, like “transaction”, while others are split into multiple tokens like the plural currency “dirhams”, which is split into the three tokens “dir”, “##ham” and “##s”. This mechanism is in charge of the *BERT preprocessor*, and it will be better described in section 5.2.

5.1.2 Machine Learning Model

During the construction of the **Machine Learning model** for the Text Classifier, the primary goal was to create an advanced model capable of *directly* processing text inputs and producing accurate labels for both *intent recognition* and *entity extraction*. The objective of this ambitious goal was to optimize the processing pipeline by allowing a **single component** to manage the complete complexity of the operation. Nevertheless, this method faced **significant challenges** when adapting the model for use on **mobile devices**. The process of adapting the model to meet the requirements of edge devices, particularly using *CoreML* for iOS devices and *TensorFlow Lite* for compatibility with mobile applications in general, posed unexpected challenges. As it will be better described in 5.1.4, the issues mainly arose from the complex structure and processing operations of the original model involving **strings**, which could not be easily converted into the optimal formats needed for efficient deployment on mobile devices.

Given these challenges, the project’s approach was reconsidered, resulting in the choice to divide the task into **two distinct phases**. This new strategy involved creating a *BERT preprocessor* and a *BERT Text Classifier*, in charge of the two separate sub-tasks. The **BERT Preprocessor** was directly implemented into the Voice Assistant codebase using the Swift language, and its primary function is to transform raw textual inputs into a format that is appropriate for the BERT model, specifically into **BERT encodings**. This process and the role of the preprocessor is better described by the 5.2 subsection. The input text sentences are then transformed into three separate arrays: *’input_type_ids’*, *’input_word_ids’*,

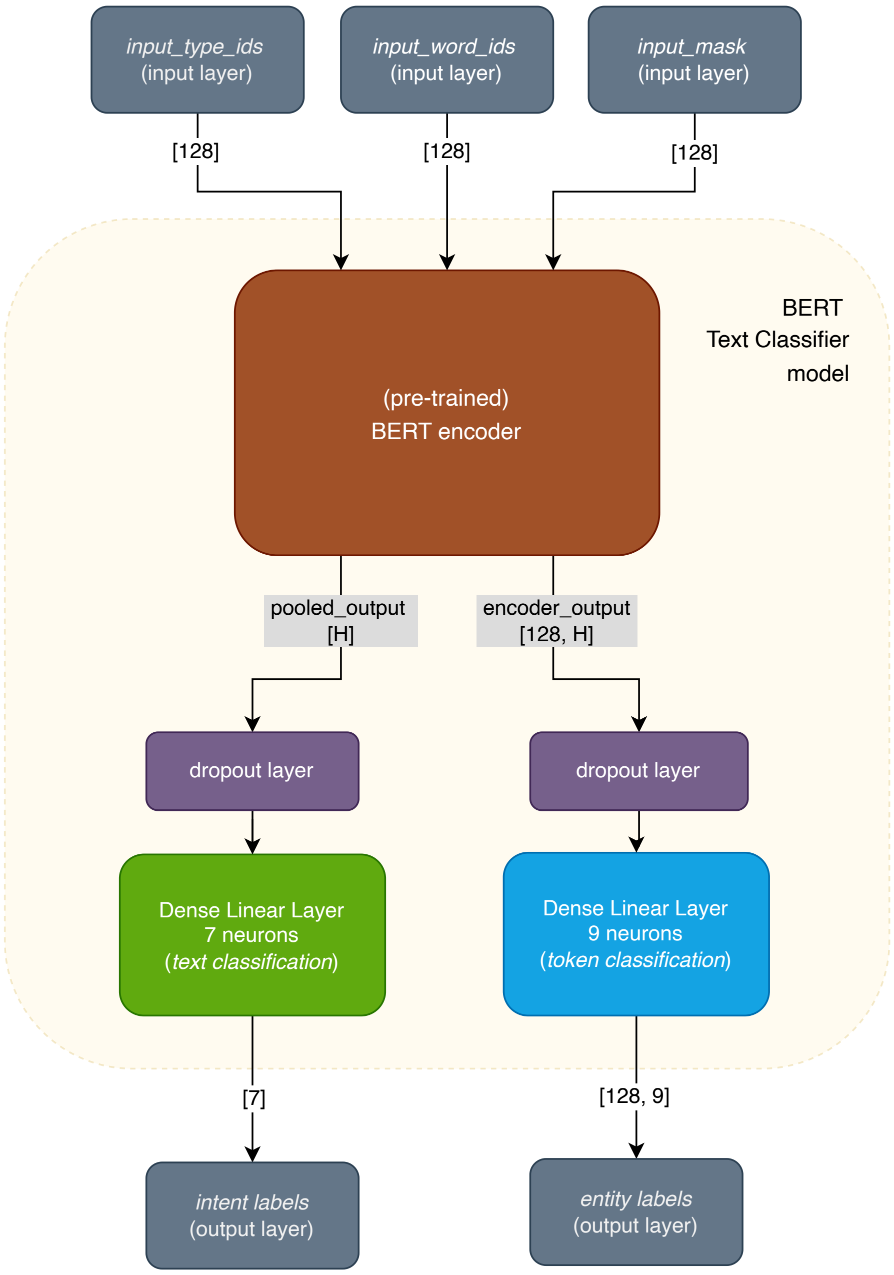

input_type_ids (input layer)
input_word_ids (input layer)
input_mask (input layer)
[128]
[128]
[128]
BERT Text Classifier model
(pre-trained) BERT encoder
pooled_output [H]
encoder_output [128, H]
dropout layer
dropout layer
Dense Linear Layer 7 neurons (text classification)
Dense Linear Layer 9 neurons (token classification)
[7]
[128, 9]
intent labels (output layer)
entity labels (output layer)

Figure 5.2: Diagram of the *Text Classifier* Keras model in TensorFlow.

and *'input_mask'*. Each array has a special purpose in preparing the text for the upcoming inference stage. The second element of the new method involved implementing a **TensorFlow Lite** model that was responsible for carrying out the actual **inference** using the preprocessed BERT encodings. This model was specifically created to identify the user's **intent** behind a given text and extract specific pieces of information (**entities**), accurately labeling the processed text.

The TensorFlow model was implemented in a **Python Notebook** on *Google Colab Pro* due to its greater computing capacity and less restrictions on resource utilization. The improved capabilities of Colab Pro assisted the construction and training of the model, ensuring that the computing demands of the project were satisfied without any resource limits.

The model was built using the **Keras API**, which is well-known for its easy and modular approach to constructing Neural Networks. An overview of the Text Classifier model is provided by the diagram in figure 5.2. Specifically, the architecture was carefully designed to include:

1. a **BERT encoder layer**, obtained from a pre-trained model available on *Kaggle* and accessible by means of *TensorFlow Hub*. This choice was made due to BERT demonstrated efficacy in capturing the intricate contextual relationships inside text;

2. a **dropout layer** (one for each subsequent layer, see next), which was implemented to address the issue of *overfitting* by randomly excluding a fraction of the neurons during training, thus improving the model's capacity to generalize;

3. two distinct fresh **linear (dense) layers** specifically designed for the separate tasks of **recognizing intent** and **extracting entities**. Each linear layer in the model is responsible for transforming the high-dimensional representations generated by the BERT encoder into a space that corresponds to the potential labels for each task. This transformation is achieved using a **softmax activation function**, which provides **probabilities** for each label as the output.

   - The first linear layer has **7 neurons**, as the number of designed intents, and produces an array for each input sentence containing the 7 probabilities associated to the different intents;
   - The second linear layer is made of **9 neurons**, same number as the described BIO entity labels, and produces as output a matrix, having one array of 9 probabilities for each input token.

The *BERT encodings* used as input for the model are of length 128, which is the default *sequence length* for these models, therefore each input sentence is converted into three arrays of 128 tokens. The *BERT encoder* layer generates an output with dimensions of 128 × H, where H denotes the **embedding size** determined by the selected BERT model version. This output provides a comprehensive and concentrated representation of each input token. It utilizes the *attention* mechanism to emphasize the importance of each token in relation to the complete input sequence (see section 2.3.1). In the task of **intent recognition**, the corresponding dense layer uses the *'pooled_output'*, which is a vector representation of the complete input sequence, obtained from the first token ([CLS]) output of the BERT encoder, to predict the overall intent of the input sentence (see 2.3.2 for more details). On the other hand, the process of **extracting entities** involved utilizing the complete 128 × H matrix of *token embeddings*. This allowed the other dense layer to classify each token separately based on the established entity labels.

The design and implementation of this BERT model, which was divided into preprocessing and inference components, played a crucial role in overcoming the first obstacles encountered while adapting the model for mobile deployment. The solution effectively addressed the limitations of mobile device deployment by using a *BERT preprocessor* and a separate *TensorFlow (Lite) model* for inference. This ensured that the Voice Assistant's Machine Learning capabilities were both efficient and suitable for use on mobile devices.

5.1.3 Model training and validation

The **training** and **validation** phase of the *BERT Text Classifier* for the Voice Assistant was a rigorous and detailed process designed specifically to handle the complex requirements of understanding and processing user commands. Using a *BERT Preprocessor* from TF Hub, the dataset described in section 5.1.1 was transformed systematically to match the input requirements of BERT models. As it will be better described in 5.2, this involves tokenization, padding, segmentation, and position encoding of the textual data, resulting in a more consistent representation for training the subsequent models.

Therefore, the **pre-processed dataset** was divided into two parts: **75%** was used for *training*, which included model tuning and *validation*, and the remaining **25%** was used as a *test set* to evaluate the model's ability to generalize to new data that it had not seen during training. This division was crucial in guaranteeing the reliability and performance of the model in practical situations.

Then the Keras model described in 5.1.2 went through a meticulous training process leveraging that training split, differently with respect to the various layers composing the full Neural Network model depicted in figure 5.2. In particular, the trainable model blocks are just the BERT encoder and the two dense linear layers:

- the *BERT encoder* is already a *pre-trained* Language Model, so its weights will go through just a **fine-tuning** step, specific to the current downstream tasks (*intent classification* and *entity extraction*);
- The two *dense layers* are instead **newly created** layers that will be **trained** completely from scratch.

During the validation process, a range of different BERT encoder variants were subjected to extensive testing across various architectural configurations and hyperparameters. The main focus was given to the classic smaller BERT variations [24], published by the authors in the official GitHub repository [20], which vary in terms of the A, H and L parameters, as described in detail in section 2.3.2. In particular, the variants taken into consideration for the validation phase were:

- **BERT mini**, with $L = 4$, $H = 256$, $A = 4$;
- **BERT small**, with $L = 4$, $H = 512$, $A = 8$;
- **BERT medium**, with $L = 8$, $H = 512$, $A = 8$.

These versions were chosen because they could provide a reasonable **compromise** between **resource consumption** and **performance efficiency**, given the primary objective of deploying the model into mobile devices. They are characterized by fewer parameters and, therefore, less computational overhead when compared to their larger counterparts introduced originally, like *BERT base* (L=12, H=768, A=12) and *BERT large* (L=24, H=1024, A=16). The decision to prioritize these variants was based on initial analyses that showed their strong performance for the task, despite the existence of other models like *MobileBERT* or *DistilBERT*. These models were not thoroughly investigated due to time limitations and their minimal performance differences observed in preliminary evaluations. Other powerful BERT-like models like **ALBERT**, instead, have not been deeply studied due to a different encoding strategy adopted with respect to the classical BERT variants, based on *SentencePiece* tokenization instead of *WordPiece*, and requiring thus a completing different preprocessing phase.

The **hyperparameter optimization** technique was crucial, and it continuously allocated a fixed **20%** of the training data for **validation**, allowing for continuous examination and adjustment of performance. The number of training iterations was limited to **3 epochs** in order to reduce the possibility of *overfitting* while yet providing enough exposure to the training data. On the contrary, the hyperparameters that were tested and adjusted iteratively included:

- the **dropout rate**, which was set at 0.1, 0.2, and 0.3 (the same for both dropout layers) to introduce regularization and reduce overfitting by randomly omitting a proportion of neuron activations;
- the **initial Learning Rate (LR)**, with values ranging from 2×10^{-5} to 5×10^{-5}, to determine the appropriate step size for exploring parameter space during gradient descent optimization;
- and the **batch size**, experimented at 16, 32, and 64 to find a balance between computational efficiency and training stability.

An innovative approach was used by include **warm-up steps**, which made up 10% of the initial training strategy. This method incrementally increased the *learning rate* from a starting value close to zero to the desired one, which helped to achieve a more seamless convergence and decreased the chances of experiencing training instabilities. The **"adamw" optimizer**, a modified version of the classic *Adam* optimizer that includes a *weight decay* component, was finally selected for its ability to effectively navigate the intricate parameter landscapes of Deep Learning models. This optimizer improves training efficiency and enhances the generalization of the model by reducing *overfitting* tendency.

The **Sparse Categorical Cross Entropy** *loss function* was used to address the tasks of *intent recognition* and *entity extraction* for both model outputs. This loss function is designed for **multi-class classification** issues. It measures the difference between the projected probability distribution and the true distribution. It is an important tool for providing feedback to adjust the model. And finally, the *performance metric* chosen was the **Sparse Categorical Accuracy**, which provides a clear measurement of the model's accuracy in making predictions, hence making the evaluation process simpler.

The described complex training and validation framework, supported by a careful choice of BERT architectures, precise hyperparameter optimization, and a systematic training methodology, played a crucial role in developing an accurate and efficient BERT-based Text Classifier.

Specifically, the entire validation process across all the different model configurations and hyperparameters followed a systematic **greedy approach**. Each of the 3 BERT variants identified has been tested separately, building and training a Keras model with a default initial configuration. Then, the three mentioned hyperparameters (*dropout value*, *learning rate* and *batch size*) have been varied one at at time, maintaining the ones that previously offered the best model results in terms of *validation loss*, and saving the validation results each time for the current configuration. After an exhaustive variation of the three parameters, the best model configuration has been selected for each BERT variant, that is the one with the lowest validation loss. Table 5.5 synthesizes the **best validation results** obtained for each BERT model variation using the analyzed configurations, and offers also an estimate of the average **training time** spent for each configuration, for the different models.

BERT	**train time**	**learn. rate**	**batch size**	**dropout value**	**validation loss**	**intent accuracy**	**entity accuracy**
mini	~15 min	5e-5	16	0.3	1.414e-3	0.99956	0.99987
small	~35 min	5e-5	16	0.2	8.494e-5	1.00000	0.99999
medium	~60 min	5e-5	16	0.2	3.949e-5	1.00000	1.00000

Table 5.5: Best BERT models' configurations *validation results*.

The **candidate models** for the actual deployment into the Payment Voice Assistant have, therefore:

- **learning rate** $\mathbf{= 5 \times 10^{-5}}$, for all 3 BERT configurations;
- **batch size = 16**, for all 3 BERT configurations;
- **dropout value = 0.2**, for BERT *small* and *medium*, while a value of **0.3** for the *mini* variation.

Additionally, it is noticeable that all the three mentioned configurations already achieve **outstanding results** over the validation set in both *intent recognition* and *entity extraction* tasks, reaching an **accuracy** level very close to perfection (100%). For this reason, the choice of the final model was not only based on the accuracy results, but also by the **memory** and **time resources** employed by the different model variants, especially after their conversion into a suitable format for in-app integration, as described in the next paragraphs.

5.1.4 Lightweight Model conversion

When trying to integrate the *BERT Text Classifier* into iOS applications, two notable frameworks emerged as suitable options for converting the model: *CoreML* and *TensorFlow Lite (TFLite).* **CoreML** is a Machine Learning framework developed by Apple that is specifically tailored for iOS devices. It focuses on improving efficiency and speed for apps that run their ML models directly on the device. On the other hand, **TensorFlow Lite** is a compact solution inside the TF ecosystem, designed to enable the use of Machine Learning models on mobile and Internet of Things (IoT) devices with little consumption of resources.

Although *CoreML* offers great potential and integrates smoothly with iOS development, the conversion procedure of the TensorFlow BERT model faced **unexpected issues**. Surprisingly, even following the official guidelines for converting TF to CoreML [37], it did not overcome these challenges. The problems mostly originated from the sophisticated nature of BERT's operations, including its manipulation of *strings* and complex multidimensional arrays, which caused difficulties in converting them to CoreML's more limited range of operations.

Given these obstacles, **TFLite** was chosen as the alternative route. Nevertheless, this decision was not without of its own difficulties. Like CoreML, TensorFlow Lite faced difficulties in directly converting specific BERT operations, especially those that included complex string manipulations and the model's dependence on *dynamic* array structures.

In the end, a solution was reached by making a strategic choice to **simplify** the model by removing the BERT preprocessing procedures from the TensorFlow model itself, as described in 5.1.2. This method required performing text tokenization and encoding directly within the iOS application, simplifying the model by only incorporating the essential BERT architecture. By simplifying the process, TFLite became a **feasible choice** for converting models.

The conversion of the Text Classifier Keras model into the TensorFlow Lite format begins after the completion of the training phase, once the model is ready for inference. The *TFLiteConverter* object, offered by the Python TFLite framework, is used to easily convert the Keras model into the `.tflite` format. The resulting **compact binary file** contains both the *trained weights* and the essential *low-level instructions* required for performing the model's tasks on edge devices. The TensorFlow Lite file, which has a considerably smaller size, is now ready for the integration into iOS applications. This allows for on-device inference, providing the advantages of **efficient** and private **local processing**. In subsection 5.1.7, more detailed information will be provided on how this model can be integrated into edge devices, and specifically into the iOS environment, with the help of the TFLite SDK.

Leveraging therefore the mentioned approach, the 3 *candidate TF models* identified in 5.1.3 after the validation phase have been converted into TFLite format, in order to analyze the *performance* and the *resource consumption* of their **lightweight versions**. Table 5.6 reports the sizes of the `.tflite` binary files for the 3 converted BERT models.

BERT candidate	**TFLite model size**
mini	~45 MBs
small	~114 MBs
medium	~165 MBs

Table 5.6: Sizes of the 3 candidate models converted into TFLite binary files.

5.1.5 Model Quantization

During the development of the *BERT Text Classifier* for the iOS Payment App, an important obstacle arose: the sizes of the converted `.tflite` models, as shown in Table 5.6, were **quite large** to be used in a mobile application context. In order to improve effectiveness and versatility in a mobile context, the focus has been placed on utilizing TensorFlow Lite's model quantization methods.

Model quantization is a technique developed to decrease the **size** and enhance the **inference time** of Machine Learning models while maintaining their accuracy at a high level. It accomplishes this by reducing the *precision* of the numbers utilized for representing model parameters, thereby decreasing the size of the model. TensorFlow Lite has multiple quantization techniques, however two main methods has been specifically examined: *Dynamic range quantization* (the default quantization) and *float16 quantization* [38].

- **Dynamic range quantization** implies the conversion of the model parameters, which are originally represented as 32-bit floating-point numbers, into **8-bit integers**. This method greatly decreases the size of the model and the amount of computational resources needed, making it ideal for mobile apps with limited resources. Nevertheless, this approach may sometimes end in a decrease in *accuracy*, potentially impacting the model's overall effectiveness.

- **Float16 Quantization** transforms 32-bit floating-point numbers into **16-bit representations**. Although this method may not achieve the same level of reduction in model size as default quantization, it achieves a more favorable *trade-off* between **size reduction** and **performance preservation**, making it also an attractive option for the application.

In order to implement these quantization techniques on the BERT Text Classifier, specific *flags* have to be configured on the TFLite interpreter while converting the model. This ensured proper processing of quantization and compatibility of the generated quantized models with TensorFlow Lite's runtime environment.

After applying quantization to the Text Classifier Keras model using both the *default* and *float16* approaches, the performance and metrics of the resulting three potential models have been thoroughly assessed. Surprisingly, even though the size of the model was significantly reduced, the quantized models were able to maintain substantially **same levels of performance** compared to their non-quantized counterparts. The predictions remained consistent with no loss in accuracy, highlighting the efficacy of TensorFlow Lite's quantization techniques in optimizing models for mobile deployment. In addition, the **Dynamic range quantization** demonstrated an higher level of model size reduction compared to the *float16* one (respectively of 25% and 50%), and a simultaneous reduction in speed execution on-device which was not significantly worse, given that memory constraints for the target iOS app result significantly more impactful than the inference time. For this reason, the **default quantization** strategy has been selected as the best approach to convert the 3 candidate models. The sizes and the inference times for the mentioned models are reported in Table 5.7, for the sake of completeness.

To summarize, the utilization of model quantization, specifically employing

BERT variant	**TFLite model size**	**quantized model size**	**on-device inference time**
mini	~45 MB	~11 MB	~140 ms
small	**~114 MB**	**~29 MB**	**~320 ms**
medium	~165 MB	~42 MB	~600 ms

Table 5.7: Candidate models sizes and inference times after TensorFlow Lite *quantized conversion.*

traditional and *float16* techniques, has enabled to address the challenge of integrating extensive Machine Learning models in the target Voice Assistant software module. Through a significant reduction in the size of the *BERT Text Classifier*, sophisticated AI capabilities have been successfully integrated into the iOS platform, improving the functionality and the User eXperience of the P2P Payment App.

5.1.6 Model selection and evaluation

Out of the three quantized candidates for the *BERT text classifier*, the **small BERT** encoder variation proved to be the **best choice**. The decision was based on the *significant balance* between **resource efficiency** and **performance**. The *small BERT* encoder variation, despite its little model size, demonstrated outstanding results in terms of *validation loss* and *accuracy* for both intent recognition and Named Entity Recognition tasks, as indicated by Tables 5.5 and 5.7. The most notable aspect of this model is its capacity to consistently achieve high accuracy, with a **100%** rate of success in recognizing intents and an average accuracy of **99.999%** in entity extraction. This is particularly impressive considering its minimal computing requirements.

In addition, the model demonstrated a remarkably **short average on-device inference time**, highlighting its suitability for real-world applications that require prompt reaction. The balance between the size of the model, the speed at which it can make predictions, and its accuracy is an important factor to consider when choosing a model, especially for applications on mobile devices with limited resources. The **small BERT** encoder type performs exceptionally well in this aspect, providing an ideal combination of *efficiency* and *efficacy*.

Table 5.8 provides a comprehensive comparison of the **evaluation results** for the selected model and the other candidates using the *test set* (~7500 sentences).

BERT variant	**test loss**	**intent accuracy**	**entity accuracy**
mini	7.357e-4	1.00000	0.99988
small	**1.106e-4**	**1.00000**	**0.99998**
medium	4.817e-5	1.00000	1.00000

Table 5.8: Best BERT models' configurations *evaluation results*.

The selected model achieved excellent results in terms of accuracy measures still maintaining a competitive inference time, providing more reason for its selection.

The evaluation of the *test set* further confirms that the *small BERT* encoder variation is the most suitable model for the **BERT Text Classifier** in the application. Due to its exceptional performance and its efficient design, this technology is very suitable for incorporating powerful **Natural Language Processing capabilities** into limited-resource environments like mobile devices. The overall configuration parameters, notable performance metrics and TFLite characteristics, for the selected Text Classifier model are summarized in Table 5.9 for reference.

Encoder	BERT small
Dropout value	0.2
Learning rate	5×10^{-5}
Batch size	16
Validation split	0.2
# Epochs	3
Train time	~35 min
Validation loss	8.494×10^{-5}
Val. Intent accuracy	1.00000
Val. Entity accuracy	0.99999
Evaluation loss	1.106×10^{-4}
Eval. Intent accuracy	1.00000
Eval. Entity accuracy	0.99998
Quantized TFLite size	~29 MB
On-device Inference time	~320 ms

Table 5.9: Selected BERT Text Classifier model configuration and performance metrics.

5.1.7 Model integration into iOS

For the creation of the AI-driven Voice Assistant for the Payments app on iOS, *TensorFlow Lite* was used to incorporate a pre-trained *BERT-based text classifier* model. This model was originally built using TensorFlow, and out of the two main choices for model integration on iOS, CoreML and TFLite, the latter was chosen due to its flexibility and effectiveness. The integration process was facilitated by the **TensorFlowLiteSwift SDK**, a powerful framework that provides Swift-based APIs for *loading* TFLite models, *performing inferences*, and *retrieving outputs* in an iOS context. Although *TensorFlowLiteSwift* does not have native support for Swift Package Manager, it has been successfully integrated into the project using **Cocoapods**, a dependency manager that effectively manages external libraries in iOS apps, and for this reason the Voice Assistant module has been implemented as a standard Swift package, as introduced in this Chapter introduction.

In order to use the BERT model in the Voice Assistant module, the quantized `.tflite` model file has been incorporated into the iOS application bundle. Next, the TensorFlowLiteSwift SDK has been employed to create an **Interpreter** object, which is responsible for interacting with the underlying TFLite model. This involved *allocating tensors* for the inputs and outputs of the model, which are essential for carrying out the model's inference process. *Preprocessing* the input text involved transforming user utterances into a format that could be used by the model, using the BERT Preprocessor component that will be described in 5.2, and this step was crucial before making any inferences. This involved the process of tokenizing the text and generating the necessary input tensors such as *'input_type_ids'*, *'input_word_ids'*, and *'input_mask'*, which are the three input arrays of the BERT model, as depicted in Figure 5.2.

Once the TFLite *Interpreter* is successfully initialized, and the input sentence has been properly preprocessed, here are the key steps to perform an inference with the underlying ML model:

1. The input tensors are copied into the *Interpreter* **input buffers** as raw data (in this case they represent all **128-length Float32 arrays**);

2. after that, the actual **inference** is launched, predicting the probabilities for the classification task(s);

3. finally, the raw byte tensors are retrieved from the **output buffers**, as result of the inference phase (in this case, they represent a **7-length Float32 array**

for the intent recognition task, and a **9x128 Float32 matrix** for the Named Entity Recognition).

To provide a smooth application integration, a Swift wrapper class called `BertTFLiteIntentAndEntitiesClassifier` has been created. This class simplifies the process of interacting with the model directly and offers a user-friendly interface for the classification tasks. In addition, a comprehensive component called `BertTextClassifier` was created to encompass the full inference pipeline, which includes the *BERT Preprocessor*, the *TFLite model*, and a *labeling* mechanism. This component is crucial for processing **input string sentences** and generating useful information, particularly in identifying user **intents** and extracting relevant **entities**.

Indeed, after the inference process, the model's output *raw probabilities* are analyzed to obtain precise classification results for both intents and entities. For each task, the labels with the **highest probability** are chosen, therefore just one *intent label* is assigned to the entire input sentence (among the 7 possible), and one *entity BIO label* is assigned to each sentence's token (among the 9 possible). However, only the predictions which exceed a **confidence level** of **0.5** are evaluated, guaranteeing their reliability. When the model's level of certainty drops below this specific threshold, **fallback labels** are given to indicate that there is no clear intent or entity; specifically, the 0 label is assigned in this case for both cases, representing either the *none* intent or the absence of entity. This safe method of assigning labels highlights the dedication to accuracy and reliability in the voice assistant's performance. It guarantees that the system only responds to commands it is extremely confident about, therefore improving the user experience and favoring trust in the Voice Assistant within the app.

5.2 BERT Preprocessor

This section highlights the crucial requirement of building a **BERT Preprocessor** in Swift to efficiently address the aforementioned difficulties related to model conversion in TensorFlow Lite. This method not only eliminates the restrictions faced during the conversion process, but also improves the effectiveness and integration of the *BERT Text Classifier* within the iOS ecosystem. The Text Classifier has been divided into two separate components: the **in-app BERT Preprocessor** and the ML **BERT classifier**, the latter being constructed and trained using TensorFlow. This division revealed to be a strategic solution to overcome the problems encountered.

The incorporation of the *BERT Tokenizer* implementation, which was essential for the Preprocessor, has been based on an TensorFlow Lite project example titled *“BERT QA iOS application”*, available on the official TensorFlow GitHub repository [39]. This implementation, based on the official BERT Python version found on the BERT GitHub repository [20], is crucial for running the tokenizer directly on iOS devices.

The **BERT Tokenizer** follows a methodical procedure to split a sentence into tokens, which can be summarized in the following steps:

1. **Text normalization**: this includes standardizing space characters, changing the text to lowercase (for uncased BERT versions, as the one used in this project), and eliminating accent markers;
2. **Punctuation splitting**: punctuation characters are separated from the rest of the text on both sides;
3. **WordPiece tokenization**: this stage involves utilizing a textual **vocabulary file** that contains tokens (one per line), which are formed by either using complete words or by dividing them into sub-words. When a word is separated into multiple tokens, all the tokens after the first one are prefixed with ’##’.

The English BERT encoder, as the one used in this project, requires the utilization of an English vocabulary file that is the same as the one utilized by the official TensorFlow BERT models. This vocabulary file consists of around 30,000 unique tokens. This tokenizer subsequently converts the resulting string tokens into **integer IDs** using the predetermined vocabulary mapping, where each token corresponds to its integer line index.

BERT relies also on special tokens to function effectively. Among these tokens there are:

- **[PAD]**, which is assigned the value 0 and is used for *padding* to ensure consistent input length (128, in this case);
- **[UNK]**, assigned the value 100, represents *unknown tokens* with respect to the used vocabulary;
- **[CLS]**, assigned the value 101, is placed at the *start* of each sentence;
- **[SEP]**, assigned the value 102, is used to *separate sentences*, particularly in tasks involving multiple sentences, like Question and Answer; or, as in this case, it is used to mark the *end* of a sentence.

A `BertTextClassifier` class has been created to both encapsulate the functionality of the *BERT Tokenizer* and simplify the process of generating *input encodings* needed for the BERT encoder. This class, in fact, not only contains the adapted version of BERT Tokenizer from the mentioned TensorFlow Lite example, but also incorporates the functionality for generating **input BERT encodings**. These encodings consist of 3 arrays of the same length as the encoder *sequence length* hyperparameter:

- **input_word_ids**: represent the **numerical tokens** themselves, with possible padding to reach a length of *sequence_length* (128, in this case);
- **input_type_ids**: indicate the *sentence* to which each token belongs with either 0 or 1, as BERT can handle up to two sentences at a time (in this case they will be all zeros, since always one input sentence is provided);
- **input_mask**: distinguishes tokens derived from actual sentences (1) from those that are *padding* (0).

For greater clarity, a practical example of preprocessing is provided using the sentence 'How much is in my Credit Mutuel account in AED', taken from the artificial dataset described in 5.1.1. Here are the chronological steps performed by the BERT Preprocessor on this input sentence:

1. First, the input string is split into textual tokens using the *BERT Tokenizer*. The result is: [“how”, “much”, “is”, “in”, “my”, “credit”, “mu” , “##tu”, “##el”, “account”, “in”, “ae”, “##d”];
2. Then, just a maximum of 126 (128-2) tokens are kept (to respect the max *sequence length* of 128), and the two special tokens are added at the beginning and at the end of the sentence: [“[CLS]”, “how”, “much”, “is”, “in”, “my”, “credit”, “mu” , “##tu”, “##el”, “account”, “in”, “ae”, “##d”, “[SEP]”];
3. After that, the three **BERT encodings** arrays are created:
   - *input_word_ids*, mapping the tokens to numerical IDs and eventually padding up to 128: [101, 2129, 2172, 2003, 1999, 2026, 4923, 14163, 8525, 2884, 4070, 1999, 29347, 2094, 102, 0, ..., 0];
   - *input_word_ids*, array of just zeros, since all tokens belong to the same sentence: [0, 0, 0, ..., 0];

- *input_mask*, marking all sentence tokens with 1, and the padding ones with 0: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, ..., 0].

The represented complex design and implementation of the BERT Preprocessor and Text Classifier in Swift not only solve the difficulties of converting TensorFlow Lite models but also greatly enhance the usefulness and integration of the Text Classifier on the iOS platform.

5.3 Dialogue State Tracker

The **Dialogue State Tracker (DST)** for the Voice Assistant module has been carefully developed as a **State Machine**, following the design principles outlined in section 4.5. The stateful architecture is essential for effectively handling the constant evolution of user interactions, guaranteeing a consistent and contextually aware flow of communication, and it has been implemented leveraging the *State Pattern*, as it will be better described in 5.3.1.

The core of the DST is a sophisticated **Intent and Entities Extractor**, which is driven by the *BERT Text Classifier*. This advanced component is highly skilled at interpreting the user's spoken input, determining the underlying *intent*, and extracting relevant *entities*. This procedure entails not only comprehending the spoken words but also **interpreting** the user's goals within the context of the given **app-specific information**, such as the user's contacts list and their bank account details. The DST utilizes contextual comprehension to interpret the user's requests in a personalized and appropriate manner.

The Dialogue State Tracker dynamically switches between states by considering the ongoing conversation context, as well as the recognized intent and entities. The dynamic state transition, as depicted in Figure 4.2, is an essential characteristic of the DST, allowing it to adjust and accommodate the changing dynamics of the conversation. The state changes are carefully designed to reflect the logical progression of a discussion, guaranteeing a seamless and intuitive User eXperience.

The Dialogue State Tracker component has also the ability to generate a specific **application response** for each user input sentence, that can be classified into five distinct categories:

- *appError*: returned when a software problem caused unexpected issues, notifying the user of an error occurring within the application;

- *justAnswer*: offers a concise and explicit **written reply** to the user's query;

- *askToChooseContact*: triggered when there are multiple potential *contacts* to choose from, prompting the user to select one (more details can be found in section 5.3.2);
- *askToChooseBankAccount*: this option is similar to the contact selection, but it is used when there are many *bank accounts* to choose from;
- *performInAppOperation*: triggered once all the required information is collected, resulting in the **execution** of an operation within the application by the Voice Assistant;

The responses created by the DST are carefully crafted to incorporate both *textual* information and pertinent *application data*, guaranteeing a smooth and informative experience. This response system with dual characteristics enhances the UX by offering precise and practical information along with the essential data required for app operations, as will be treated in paragraph 5.3.3. Therefore, this method not only improves user engagement but also simplifies the execution of app tasks through voice commands, so making the Voice Assistant an effective tool for navigating and utilizing a P2P payment app.

5.3.1 State Pattern

The **State Pattern**, an important **design pattern** introduced by the "Gang of Four" in their popular book *"Design Patterns: Elements of Reusable Object-Oriented Software"*, is essential in the implementation of the Dialogue State Tracker for the Voice Assistant component. This pattern plays a key role in handling the complex *states* and *transitions* that take place when users interact with the conversational agent.

The *State Pattern* allows an object to easily modify its **behavior** when its internal state changes [based on external inputs], giving the illusion that the object has changed its class [40]. This is particularly important in the context of a Voice Assistant, as the system has to customize its responses to user inputs and its actions based on the current context or status of the conversation. For instance, the assistant's reply to a question concerning transaction history can differ considerably depending on whether the user is now in the process of authorizing a payment or asking about their account balance.

The implementation of the Dialogue State Tracker using the *State Pattern* required the definition of a `DstState` protocol that has methods corresponding to the different **actions** that the user can do during the conversation. Concrete state classes implement this interface to encapsulate the specific Voice Assistant behaviours related to a particular stage of the discussion. These states can include greeting, requesting payment details, completing a transaction, or resolving misunderstandings. The DST states and transitions implemented for this particular solution are the same ones summarized in the design diagram 4.2.

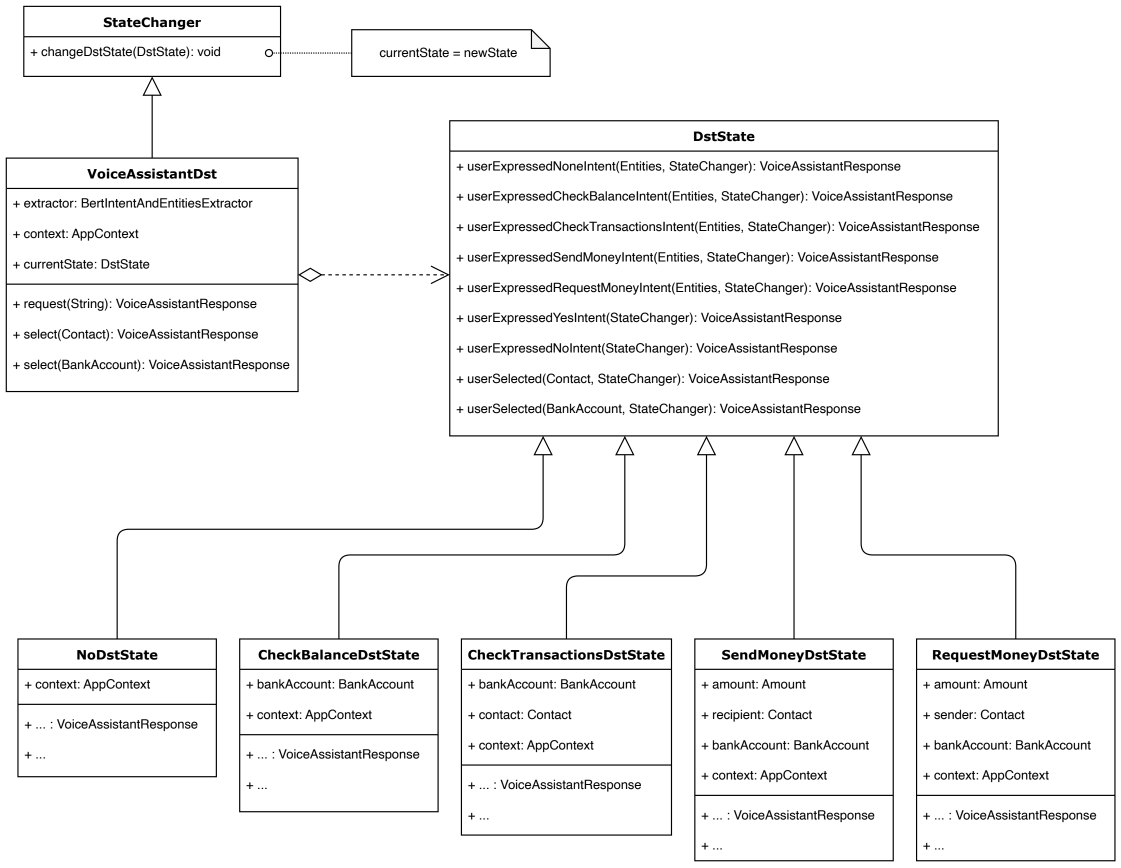

StateChanger
+ changeDstState(DstState): void
currentState = newState
VoiceAssistantDst
+ extractor: BertIntentAndEntitiesExtractor
+ context: AppContext
+ currentState: DstState
+ request(String): VoiceAssistantResponse
+ select(Contact): VoiceAssistantResponse
+ select(BankAccount): VoiceAssistantResponse
DstState
+ userExpressedNoneIntent(Entities, StateChanger): VoiceAssistantResponse
+ userExpressedCheckBalanceIntent(Entities, StateChanger): VoiceAssistantResponse
+ userExpressedCheckTransactionsIntent(Entities, StateChanger): VoiceAssistantResponse
+ userExpressedSendMoneyIntent(Entities, StateChanger): VoiceAssistantResponse
+ userExpressedRequestMoneyIntent(Entities, StateChanger): VoiceAssistantResponse
+ userExpressedYesIntent(StateChanger): VoiceAssistantResponse
+ userExpressedNoIntent(StateChanger): VoiceAssistantResponse
+ userSelected(Contact, StateChanger): VoiceAssistantResponse
+ userSelected(BankAccount, StateChanger): VoiceAssistantResponse
NoDstState
+ context: AppContext
+ ... : VoiceAssistantResponse
+ ...
CheckBalanceDstState
+ bankAccount: BankAccount
+ context: AppContext
+ ... : VoiceAssistantResponse
+ ...
CheckTransactionsDstState
+ bankAccount: BankAccount
+ contact: Contact
+ context: AppContext
+ ... : VoiceAssistantResponse
+ ...
SendMoneyDstState
+ amount: Amount
+ recipient: Contact
+ bankAccount: BankAccount
+ context: AppContext
+ ... : VoiceAssistantResponse
+ ...
RequestMoneyDstState
+ amount: Amount
+ sender: Contact
+ bankAccount: BankAccount
+ context: AppContext
+ ... : VoiceAssistantResponse
+ ...

Figure 5.3: UML class diagram of the *State Pattern* application for the DST.

The class representing the *State Machine* itself, namely the `VoiceAssistantDst` in this case, stores a reference to the **current state** of the Voice Assistant and passes user inputs to the corresponding *state* object. This abstraction not only simplifies the codebase and makes it easier to maintain, but also improves the system's adaptability, enabling simple additions or adjustments of states as the assistant's capabilities evolve. In Figure 5.3 a simple UML class diagram exemplifies the relationships between the *State Machine* class (`VoiceAssistantDst`) and some of the concrete *State* classes, showing the possible *transitions*.

Through the application of the State Pattern, the Dialogue State Tracker effectively manages the course of a discussion, guaranteeing that the Voice Assistant continues to use a meaningful and appropriate language with the user in every situation. This method significantly improves not only the code maintainability but also the UX, by making interactions with the assistant more intuitive, natural, and in line with the user's expectations in the application context.

5.3.2 Entity Matching

This paragraph highlights the importance of the **Named Entities matching** method in the *Dialogue State Tracker* of the Voice Assistant, a crucial mechanism for understanding the intricate relationship between different components. The *Entity Matching* mechanism starts by utilizing the raw entities collected by the *BERT Intent and Entity Extractor*. This technique has been optimized to analyze entities within the user's statements and *reconstructing* relevant textual entities from the tokens provided by the *BERT Text Classifier*. An example that clearly demonstrates this may be observed in the sentence supplied in Table 5.4. In this situation, the method would reconstructs two important entities: an *amount*, represented as "324 dirhams and one cent", and a *user*, identified as "mila diaz".

After the entities are *extracted* and *reconstructed*, the Dialogue State Tracker uses these string entities in the **context** provided by the application. This context contains an array of *user-specific data*, including **bank account** details and **contacts** information, as it will be better described in subsection 5.9.2. The DST carefully extracts pertinent information from the app context, such as identifying a particular contact in the user's phone book, and **formats** it in a suitable manner for application use. For example, in the case of the aforementioned entities, it converts the amount entity into a numerical floating-point representation for transactional purposes, which is "324.01" in this case (currency is omitted here), and select the proper contact from the user's phone book; for instance, in a phone book made of just the three contacts "John Doe", "Lauren Scott" and "Myla Diaz", the latter would be selected.

Some coding strategies are utilized to ensure a seamless congruence between the entities indicated by the user and the contextual information of the program, such as user contacts and bank accounts. These algorithms perform well at extracting numerical data from the user's speech in multiple formats, for example recognizing the placement of currencies, and distinguishing between *numerical* and *literal* representations of values, including the analysis of cents. A feature of this mechanism present in the currency reconstruction, is to take into consideration exclusively the currencies associated to the user's bank accounts, supporting various literal and symbolic representations.

The *Entity Matching* technique utilizes also a simplified yet efficient algorithm for bank accounts and contact names. This algorithm computes the **fraction** of characters that correspond between the user's utterance and the app's contextual information. It selects matches that exceed pre-established **'matching thresholds'** - 0.4 for bank accounts and 0.75 for contact names - obtained after a careful tuning phase.

When the algorithm detects **multiple** possible matches for contacts or bank accounts, it organizes them in descending order according to the "match" metric, and the DST produces an appropriate response, either *"askToChooseContact"* or *"askToChooseBankAccount"*, which prompts the user to clearly select one from the available options (see 5.8 for better details on the UI/UX part). On the other hand, if only one match is detected, the DST will continue as usual, following the natural flow of conversation.

Finally, it is crucial to recognize also the Dialogue State Tracker's smart handling of entities and intents that have **low confidence**, which is determined by a probability threshold of 0.8. During these situations, the DST assumes a *"uncertain" state* and requires further confirmation from the user before making any modifications to its current status. This meticulous technique guarantees a level of verification, especially crucial when the user refers to numerous entities of the same category, such as two distinct amounts or bank accounts, which requires asking clarification or a new query from the user.

5.3.3 Answer Generation

The implementation of the Voice Assistant for the iOS Payment App has merged the **Answer Generation** component into the *Dialogue State Tracker*, which is different from the standard approach of having a separate module as shown in the design diagram (Figure 4.1). The integration was carefully chosen to **simplify** the system's structure and reduce the challenges and limitations usually encountered when deploying individual Machine Learning components for generating answers, especially if they have to run locally on-device, as in this case.

The Dialogue State Tracker, which has been enhanced with *Answer Generation* capabilities, is specifically designed to generate responses by taking into account the current context of the conversation. This is achieved by utilizing a set of **predefined templates** which depends on the current state. The templates are filled with **user-specific information** and **application context**, such as transaction amounts, names, currencies, and other relevant data. This ensures that the responses are both accurate to the app context and customized to the individual user's requirements.

This implementation choice is crucial for multiple reasons. Firstly, it greatly decreases the system's dependency on additional resource-intensive Machine Learning components, including Generative AI models, that are typically used to generate rich textual responses. To ensure smooth integration and optimal performance in mobile apps, the system prioritizes **simplicity** and **efficiency**, while still admitting the limitations of the adopted choice, which might reduce the **naturalness** of the conversational agent answers.

The output produced by the DST, referred to as `VoiceAssistantResponse`, includes not only the generated text but also other related *application data*. This data includes crucial information required for carrying out **in-app tasks**, such as initiating transactions or requesting balance information. This response structure increases the utility of the Voice Assistant, allowing it to work not just as a conversational interface but also as a a **task-oriented agent**, which becomes an essential part of the app's operation. It both guides the user through actions and provides valuable information, enriching the User eXperience.

Essentially, incorporating the Answer Generation feature into the Dialogue State Tracker was an intentional design choice with the goal of maximizing the efficiency and practical usefulness of the Voice Assistant, not only addressing the difficulties caused by complexity and limited resources, but also facilitating smooth interactions inside the app.

5.4 Speech Recognizer

The **Speech Recognizer** component of the Voice Assistant has been implemented using an homonymous class aiming to handle the complex process of converting *speech to text*, thus simplifying the interface with other components in the system illustrated in Figure 5.1. By using the `SFSpeechRecognizer` class and the *AVFoundation* framework to capture sounds from the device's microphone, this class serves as a mediator, abstracting the complexities of the native iOS **Speech** framework.

Prior to starting the Speech Recognition procedure, the application must obtain **user permission** for both *microphone usage* and *Speech Recognition*. This requirement adheres to Apple's guidelines, which require the inclusion of descriptive and meaningful explanations within the "Info.plist" file of the program. The *Info.plist* file in iOS serves as an essential **configuration file**, storing key-value pairs for many app settings, including permissions related to privacy. In order to enable Speech Recognition and microphone access, it is necessary to add particular keys together with user-friendly explanations to tell users about the utilization of their data.

After obtaining authorization, the Speech Recognizer utilizes the *AVFoundation* APIs to enable the microphone on the device, collecting the user's speech as audio input. The `SFSpeechRecognizer` processes the audio and turns the spoken words into text. The conversion process is affected by the given *locale*, which sets the language and dialect that the Speech framework should expect during recognition. The locale option is crucial for precise Speech-to-Text translation since it synchronizes the recognition process with the user's language preferences, which are established in the iOS system settings. Nevertheless, the whole process has some **limitations**, specifically in its language compatibility. At the moment, the system is specifically designed to efficiently identify and understand only **English** sentences and names, and this may not be suitable for a wide range of users with different linguistic backgrounds.

iOS 13 and subsequent versions include **on-device Speech Recognition**, which improves *privacy* and *speed* by processing data directly on the user's device [41]. For this reason, the deployment target of the Voice Assistant module has been set to this version of the Operative System. However, this feature can only work if the user's system settings match the permitted locales. This means that the intended language needs to be included in the system choices for local processing.

iOS 17 introduces even newer APIs for **Custom Language Models**, which enable the customization of the Speech Recognition process to better fit certain application domains [42]. By utilizing these APIs, the Voice Assistant can improve its comprehension of specialized vocabulary and user intents, so significantly improving the accuracy and efficiency of converting speech into written text. Within the implementation of the current Voice Assistant, this innovative feature has also been leveraged by creating a *Custom Language Model* specifically designed for the host application domain. This model includes specialized terminology and widely used terms and expressions in requesting payment transactions. A more detailed explanation of this feature may be found in paragraph 5.4.1. This adjustment guarantees enhanced precision and pertinence in Speech-to-Text for users utilizing iOS 17 or subsequent versions. On devices that have older versions of iOS, the system will use the regular voice recognition capability instead of the specialized Language Model.

Essentially, the *Speech Recognizer* component plays a crucial role in the Voice Assistant by enabling the transformation of spoken words into written text using a combination of the built-in iOS Speech and AVFoundation frameworks. The adopted solution focuses on making the assistant easy to use, ensuring *privacy*, and achieving high *accuracy*. It also emphasizes the ability to enhance innovative capabilities by utilizing a Custom Language Model in the latest iOS versions.

5.4.1 Custom Language Model

In this section, the creation of a **customized Language Model** is explored, specifically designed for the Speech Recognizer component in the iOS environment. Taking advantage of the new features introduced by **iOS 17** in September 2023, the method involves a process of model *fine-tuning* for the Speech Recognizer that is done on-device, an innovative approach made possible by the latest iOS updates [42].

Essentially, this strategy requires providing a dataset that includes statements relevant to the specific field of the application. The purpose of such phrases is to accurately represent common user requests in the host Peer-to-Peer payment application. This will help the *Speech Recognizer* **prioritize** and accurately transcribe these specific types of utterances, improving the performance of the Voice Assistant within the application.

The fine-tuning process occurs through a series of carefully organized steps:

1. First, a **Custom LM data object** is created, including *custom sentences* that represent the common user queries and commands that are expected in the context of the P2P payment application. This object functions as a container for domain-specific language structures and terminology, providing the foundation for later building a more sophisticated and contextually aware Language Model.

2. Subsequently, the *data object* is subjected to an **export** process, resulting in the creation of a specific **binary file** containing all the necessary information for the actual build of the *Custom LM*.

3. The next stage is **importing** the binary model, which actually launch the *creation* and *compilation* of the Custom Language Model. This stage is crucial as it converts the unprocessed linguistic data into a well-organized model that the iOS *Speech Recognizer* can efficiently employ. After the compilation is

successful, a **Custom LM configuration** object is created, containing all the parameters and characteristics of the newly created Language Model.

4. Finally, the *configuration object* is **incorporated** into the *Speech Recognizer* instance, giving it the capacity to accurately understand and prioritize utterances relevant to the specified domain. This integration is the final stage of the refinement process, enhancing the recognizer's ability to understand and respond to user commands within the P2P payment application context.

A key element of this precise procedure is the two-step process that includes *exporting* and then *importing* the Custom LM data. This split provides a level of **flexibility**, enabling the creation of a stable *LM configuration* at a certain moment, which can then be imported and used at runtime without the need for generating the model in real-time. This method is especially beneficial in situations when the Language Model does not need to be flexibly adjusted to different user settings or application states.

Within the particular context of the Voice Assistant for the P2P payment application described in this Thesis work, instead, the Custom Language Model is generated *in real-time* every time the host app is opened. This dynamic generation relies on the immediate supply of user-specific data, such as bank account information and phone book contacts, which are essential for tailoring the Language Model. The sentences used to construct the LM are obtained from predetermined **templates**, derived from the same artificial dataset used to fine-tune the *BERT Text Classifier* and outlined in section 5.1.1. In Table 5.10 some examples of such sentence templates are reported.

Templates
What are the latest transactions with <name> <surname>
Display my <bank> account balance
Initiate a wire transfer to <name> from <bank> please
I don't think so
Could you assist in receiving 334 dirhams and 32 cents from <name>
I want to send some money from <bank> to <name> <surname>

Table 5.10: Some of the templates used to generate the Speech Recognizer *Custom Language Model.*

The templates are subsequently filled with **personalized user information**, including names and bank account details, resulting in a collection of around **90,000** final sentences. This large set of data serves as the basis for the Custom Language Model, guaranteeing that the Speech Recognizer is carefully adjusted to the user's individual and financial environment, hence improving the effectiveness and significance of voice-activated interactions within the application.

5.5 Speech Synthesizer

The **Speech Synthesizer** is a crucial element in the implementation of the Voice Assistant. This component works as a **simple** yet effective interface, utilizing the built-in iOS `AVSpeechSynthesizer` to simplify the use of the system's **Text-to-Speech** capabilities. Its main function is, in fact, to convert the written responses of the Dialogue State Tracker into audible speech, thereby enabling a user-friendly and easy-to-understand experience, utilizing a voice that is based on the *US-English locale* to articulate the responses.

The Speech Synthesizer's design is intentionally basic, prioritizing reliability and ease of use within the app's ecosystem. This approach not only improves the efficiency of development but also guarantees smooth integration with the overall Voice Assistant software architecture depicted in Figure 5.1. The wrapping of the built-in Speech Synthesizer encapsulates its complexity, offering a high-level interface that simplifies the initiation, voice selection, and playback controls. This allows other components of the Voice Assistant to easily invoke it.

Nevertheless, opting for pure **on-device operation** causes particular limitations on this component. Indeed, it mainly depends on the voices that are present on the user's device, which are also used by the *VoiceOver* feature. Although this guarantees quick compatibility and accessibility, it also implies that the **quality** of the synthetic speech is limited by the predefined system voices. While the voices are clear and understandable, they may not possess the same level of richness and **natural cadence** as more sophisticated cloud-based TTS services. Users who desire a more realistic speech experience have the option to manually download voices of better quality via the iOS settings, although this will result in an explicit user action, besides using up more storage space.

5.6 Conversation Manager

The **Conversation Manager** component serves as the **primary operational interface** for enabling interaction between the user and the Voice Assistant core functionalities in the host P2P payments application, as illustrated in the software diagram 5.1. It utilizes its primary features to manage an **individual conversation** with the user, effectively controlling the flow of communication. It is implemented as a class of the same name offering the essential Application Programming Interfaces for the starting conversation with the user, handling spoken input, and generating auditory responses.

This component is very dependent on the *Dialogue State Tracker* to accurately monitor the context and progression of the conversation. The system interacts with the *Speech Recognizer* to convert spoken user input into written text, and with the *Speech Synthesizer* to convert written responses into spoken words, allowing for a smooth and natural interaction. In addition, the *Conversation Manager* interacts with an **App Delegate**, which is a component provided by the host application, and responsible for carrying out **in-app operations**, such as transactions or queries, based on user requests, as it will better described in subsection 5.9.3.

Its main responsibility is, in fact, to enable in-app operations using this delegate object, particularly when the DST produces a response that necessitates such an action, referred to as *performInAppOperation*. For instance, if the user wants to check their account balance, the *Conversation Manager* will instruct the *App Delegate* to retrieve this information from the backend of the application. In addition, it *enhances* the text answers of the Dialogue State Tracker, such as the success or error messages caused by operations, ensuring that they are clearly presented to the user, both visually and vocally.

Essentially, the *Conversation Manager* plays a crucial role in facilitating user talks with the Voice Assistant, serving as the primary vocal interface for interaction within the P2P payments program. The system's simplified structure guarantees that the user's requests are comprehended, executed, and answered efficiently offering a smooth and user-friendly experience.

5.7 Payments Voice Assistant

This section focuses on the implementation of the **Payments Voice Assistant** component, which is the main component of the system. Its purpose is to enable the interaction between users and the voice-enabled payment operations thanks to the `PaymentsVoiceAssistant` class, which acts as the **primary high-level component** to interact with, at the basis of the assistant module design depicted in Figure 5.1.

When this component is created, it performs the important task of *initializing* the assistant's essential sub-components through **Dependency Injection (DI)**, a systematic methodology which will be better outlined in 5.7.1 and that ensures each component receives the exact dependencies it needs to function efficiently. The sub-components initialized in this process consist of the *BERT Intent and Entities Extractor*, which includes its *BERT Text Classifier*; additionally, there is the *Speech Recognizer*, which may be enhanced with a *Custom Language Model* to improve its comprehension of payment-specific terminologies; lastly, there is the *Speech Synthesizer*, which converts text responses into speech, ensuring a smooth conversational experience.

The *Payments Voice Assistant* offers the capability to dynamically generate a fresh **Conversation Manager** for every user conversation. This is achieved by creating and providing a specific *Dialogue State Tracker* (once again using **Dependency Injection**) that carefully handles the state and context of the ongoing conversation, guaranteeing a coherent and contextually appropriate flow of dialogue, as well as providing all the necessary core sub-components.

The Payments Voice Assistant's design is characterized by its close adherence to the **Singleton Pattern**. This *design pattern*, also described in the famous book *"Design Patterns: Elements of Reusable Object-Oriented Software"* [40], guarantees the creation of a **unique instance** of the `PaymentsVoiceAssistant` class, which is then reused throughout the entire program, eliminating the necessity for redundant initializations. This method not only enhances the user's experience by greatly *lowering* the time it takes to start up after the first setup, but also improves the overall effectiveness of the assistant by reusing its sub-components for future interactions. The *Singleton* mechanism guarantees the existence of only one instance of the assistant in the host application, which ensures a **constant state and behavior** of the assistant across the application. This results in a reliable and consistent user experience.

Overall, the implementation of the *Payments Voice Assistant* component demonstrates strategic design and architecture choices, with the goal of creating a simplified, reliable, and user-centric interface for voice-activated commands, through the implementation of Dependency Injection, dynamic conversation management, and the Singleton Pattern.

5.7.1 Dependency Injection

The **Dependency Injection** is an *architectural principle* that plays a key role in establishing inversion of control in software systems, resulting as the concrete application of the famous *Dependency Inversion* principle. It allows the responsibility of creating dependencies to be transferred to an *external* entity, rather than being handled by the classes that utilize them. This pattern is essential for organizing software systems in a **modular**, **testable**, and **maintainable** manner.

Within the overall framework of this project, *Dependency Injection* is of utmost importance at both the micro (low-level) and macro (high-level) levels. At the **lower level**, the significance of DI is exemplified by specific components like the *Dialogue State Tracker*. This component does not directly create instances of the *BERT Intent and Entity Extractor*, or even the *BERT TFLite model* which uses. Instead, it acquires these elements from an external source. This **separation** guarantees that the DST is not strongly connected to particular implementations of its dependencies, promoting a design that is more convenient to test and expand.

On a **higher level**, the Voice Assistant software module as a whole gains advantages from Dependency Injection too. During **integration**, in fact, the host application provides app-specific objects to the assistant module, such as the **App Delegate** and the **App Context**, as it will be described in section 5.9. This method guarantees that the Voice Assistant can function within any host application without requiring knowledge of the program's complex initialization processes. Additionally, it enhances the **adaptability** of the Voice Assistant to various situations, hence increasing its **reusability** for diverse projects.

The benefits of utilizing *Dependency Injection*, in general, are many and varied. Its main purpose is to observe two of the famous **SOLID design principles** [43] introduced by Robert Martin in 2000 *"Design Principles and Design Patterns"*:

- the *Single Responsibility* principle: it guarantees objects do not have multiple reasons to change, thanks to separation of their concerns, and it is exemplified here by separating the *creation* of objects from their *utilization* [44]. The act of **separating responsibilities** in the system not only increases its *modularity* but also improves its *testability*, as dependencies can be easily simulated or substituted in tests;

- the *Dependency Inversion* principle: it affirms that high-level modules should not depend on low-level modules but both should depend on *abstractions*; and abstractions should not depend on details, but details should depend on abstractions [45]. Indeed, Dependency Injection enhances the level of

abstraction in software design, enabling the development of more adaptable and loosely connected systems. By utilizing abstractions, namely interfaces, instead of specific implementations, the system has a higher level of *adaptability to changes* in business needs or technological stacks.

Additional noteworthy advantages include enhanced **code maintainability**, as the DI framework consolidates the setup of component dependencies, simplifying their management and modification, and enables the use of regulated lifetimes of dependencies, which enhances resource consumption and improves application performance.

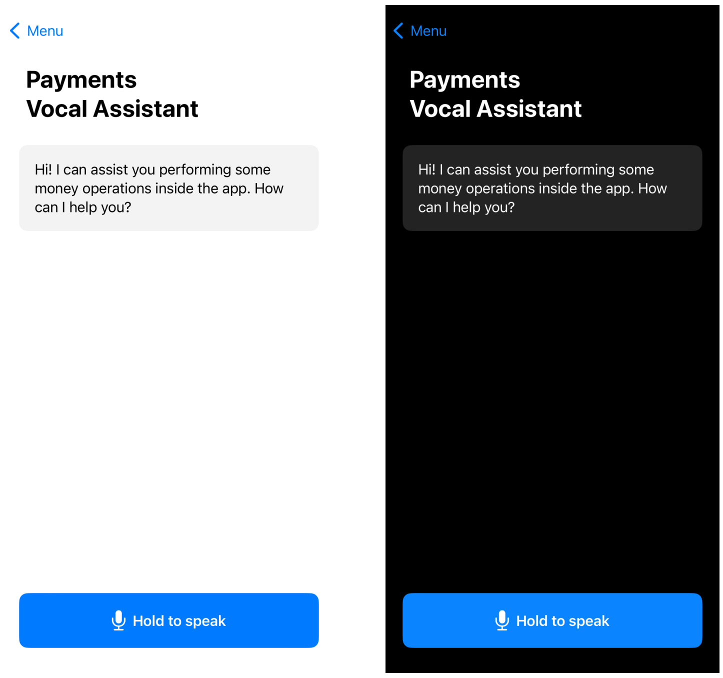
Overall, the implementation of *Dependency Injection* in this project has played a crucial role in establishing a design that is both **robust** and **adaptable**, while also facilitating seamless integration and future improvements. It serves as an important example of the most effective methods in software engineering, making a substantial contribution to the project's success.

5.8 UI/UX

The **SwiftUI** framework, introduced by Apple, has been utilized in the development of the Voice Assistant's **UI/UX**. This modern UI framework, as it has been widely explained in subsection 2.6.1, allows for the *efficient* and *intuitive* design of User Interfaces across all Apple platforms. SwiftUI is widely recognized for its **declarative syntax**, in which developers specify the desired behavior of the User Interface, and the framework handles the underlying implementation. This strategy greatly improves the *speed* of development and decreases the complexity of UI programming.

An important characteristic of this implementation is its capacity to seamlessly switch between **white mode** and **dark mode**, based on the user's **system settings**. This function guarantees a uniform **User eXperience** that corresponds to the user's overall *system preferences*, improving visual comfort in different lighting conditions.

The complete User Interface of the Voice Assistant is contained within a single UI component called `PaymentsVoiceAssistantView`. This component serves as the primary interface for integrating the Voice Assistant module into the payment app, as it will be better described in section 5.9. The Voice Assistant includes all the essential UI logic for controlling the sequence of screens. When the Voice Assistant is first launched in the app, it creates a new instance of the `PaymentsVoiceAssistant`



Hi! I can assist you performing some money operations inside the app. How can I help you?

Hi! I can assist you performing some money operations inside the app. How can I help you?

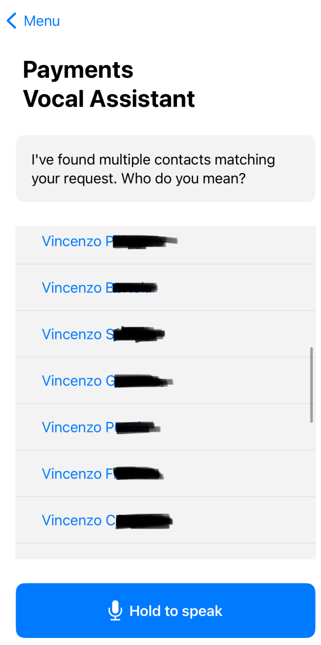
(a) White mode. **(b)** Black mode.

Figure 5.4: Example of the Voice Assistant **start screen**, for both *white mode* and *dark mode*.

object, which plays a crucial role in handling the fundamental logic and functionalities of the Voice Assistant, guaranteeing a smooth integration and interaction flow within the application.

The **User Interface** of the Voice Assistant prioritizes **simplicity** and **effectiveness**, while also complying with **accessibility** requirements to meet the needs of various users. It is organized around a central box that displays the Assistant's responses, as seen in the screenshots of the start screen in Figure 5.4. The content enclosed in this container is displayed *gradually*, one character at a time, creating an immersive and dynamic experience for the user. In addition, *haptic feedback* is employed to improve the user's involvement.

A noticeable characteristic of the interface is the primary button located at the bottom of the screen, which users can push and hold in order to interact with the Voice Assistant. Throughout this interaction, a prominent **microphone animation** is shown (see subfigure 5.5b), indicating the Assistant's listening to the user's voice requests. Visual feedback is essential for establishing a **User eXperience** that seems *natural* and *intuitive*.



Menu

Payments Vocal Assistant

I've found multiple contacts matching your request. Who do you mean?

Vincenzo P[redacted]

Vincenzo B[redacted]

Vincenzo S[redacted]

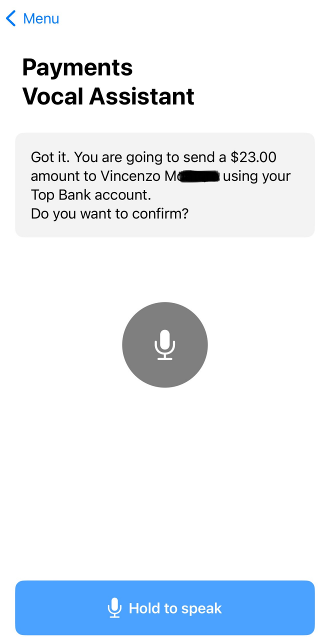
Vincenzo G[redacted]

Vincenzo P[redacted]

Vincenzo F[redacted]

Vincenzo C[redacted]

(a) Choose contact.

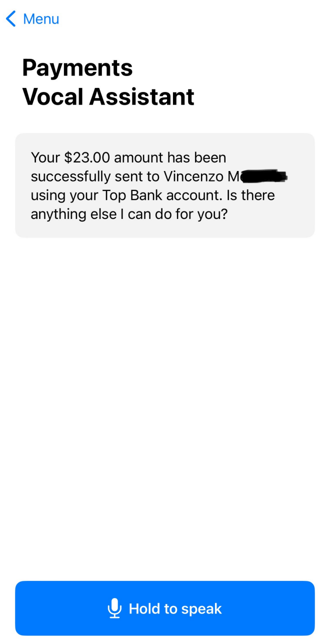


Menu

Payments Vocal Assistant

Got it. You are going to send a $23.00 amount to Vincenzo M[redacted] using your Top Bank account.
Do you want to confirm?

(b) Rec. + send money.

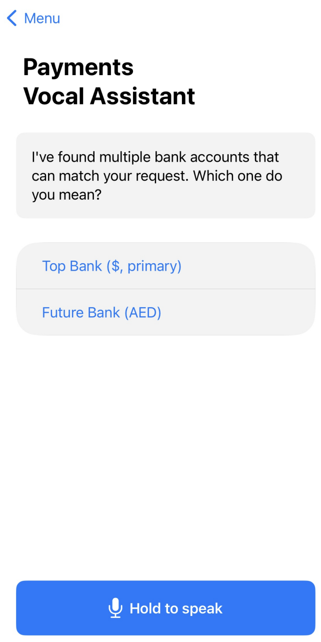


Menu

Payments Vocal Assistant

Your $23.00 amount has been successfully sent to Vincenzo M[redacted] using your Top Bank account. Is there anything else I can do for you?

(c) Payment successful.



Menu

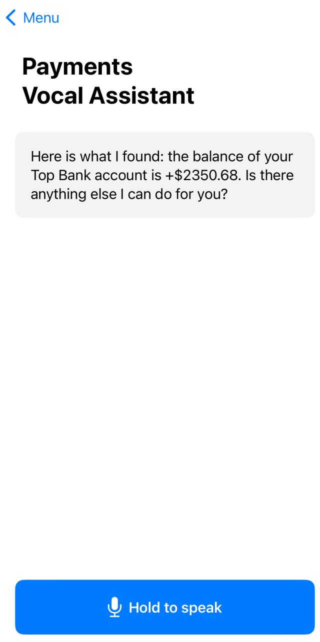
Payments Vocal Assistant

I've found multiple bank accounts that can match your request. Which one do you mean?

Top Bank ($, primary)

Future Bank (AED)

(d) Choose bank account.

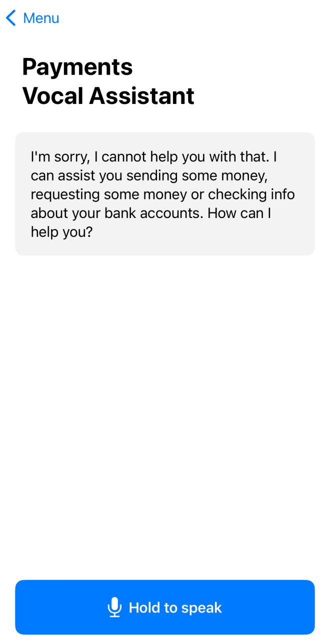


Menu

Payments Vocal Assistant

Here is what I found: the balance of your Top Bank account is +$2350.68. Is there anything else I can do for you?

(e) Check balance.



Menu

Payments Vocal Assistant

I'm sorry, I cannot help you with that. I can assist you sending some money, requesting some money or checking info about your bank accounts. How can I help you?

(f) Unsupported intent.

Figure 5.5: Example of Voice Assistant screens in various use cases.

When the Voice Assistant detects **many possible options** for a user's request, such as *contacts* or *bank accounts*, a *scrollable list* is displayed, enabling the user to make a specific choice. The UI screenshots in subfigures 5.5a and 5.5d illustrate this feature, demonstrating the Assistant's ability to effectively manage disambiguation. All the screens grouped in Figure 5.5 are taken from the test application built specifically for the user evaluation phase, widely described in Chapter 6.

The Assistant is specifically built to handle a wide range of user intentions, including commands that are **not supported** or **not recognized**. In these situations, it provides the user with information regarding the variety of tasks it is capable of executing, directing them towards a constructive interaction, as shown in subfigure 5.5f. The screenshots also demonstrate the process of verifying transaction details before carrying out a payment operation (subfigure 5.5b), as well as displaying successful results (subfigure 5.5c), highlighting the Assistant's role in enabling *secure* and efficient transactions.

The *UI/UX* of the Voice Assistant is designed to be simple and efficient, also allowing clients to customize it to some extent through a *configuration file*. This feature enables **customization** of colors and default messages, ensuring that the Assistant may be personalized to align with the branding and User eXperience objectives of the host application.

5.9 Voice Assistant in-App Integration

In order to incorporate the *Voice Assistant module* into a P2P payment application, it is crucial to note that the module is delivered as a standard **Swift framework**. This framework is essentially a collection of *pre-compiled* code designed for reuse in various Swift applications. Swift frameworks are libraries that contain both executable code and the necessary resources, such as pictures and predefined strings, for the program to work properly. By incorporating this file into a payment application project, developers can leverage its features without the need to manually change the source code. The **integration process** includes the following stages:

1. **Inclusion of the Framework**: The Voice Assistant Swift framework, which is a compiled binary file, needs to be added *anywhere* to the project files of the payment application in Xcode, Apple's Integrated Development Environment (IDE) for macOS.

2. **Embedding the Framework**: To incorporate the framework into the app, it is necessary to mark it as an *"Embedded Framework"* in the project settings

on Xcode. This action guarantees that the framework is included with the application's executable file during the building process, enabling the application to leverage the Voice Assistant's capabilities at runtime.

3. **Minimum iOS Version Requirement**: The Voice Assistant module requires *SwiftUI* and *on-device Speech Recognition* technologies, thus the host payment application must be built for **iOS 13 or newer versions**. SwiftUI, in fact, is a contemporary framework and it requires a minimum deployment target of iOS 13.

4. **Integration of Assistant Features**: In order to incorporate the functionalities of the Voice Assistant into the payment application, it is necessary to import the `PaymentsVoiceAssistant` module, and create an instance of the `PaymentsVoiceAssistantView`. This object contains the whole contents of the assistant module, comprising both the *User Interface* and the *business logic*. Integration necessitates the incorporation of specific elements tailored to the host application into the Voice Assistant View through **Dependency Injection**:

    - **App Context**: an object that includes **user-specific data** that is relevant to the Peer-to-Peer payments domain. This context enables the Voice Assistant to *customize* its features and replies according to the user's information and preferences. Further information regarding this matter will be presented in subparagraph 5.9.2;
    - **App Delegate**: A component utilized by the Voice Assistant to execute **payment operations** on behalf of the user. The delegate serves as a means of communication between the Voice Assistant and the payment processing components of the host application, as detailed in subsection 5.9.3.
    - **Configuration Object** (optional): An object that enables **customization** of the Voice Assistant's appearance to match the branding and UI design preferences of the host application, for example specifying the title of the Voice Assistant screen or the color of its main button.

5. **Presenting the Assistant View**: After instantiating the *Payments Voice Assistant View* with the required context and configuration, it can be displayed within the payment application based on the app's **architectural and navigational design**. The integration method differs based on whether the

host application utilizes SwiftUI or UIKit, Apple's older framework for UI development, whose differences have already been discussed in section 2.6:

- Integrating the Voice Assistant View into apps created with **SwiftUI** is simple because it can be immediately embedded within other SwiftUI views;
- To integrate instead the SwiftUI-based assistant's view into a UIKit `ViewController` for applications utilizing **UIKit**, which follows the Model-View-Controller architectural pattern, an extra step is necessary, as it will be explained in detail in subsection 5.9.1. This step guarantees compatibility between the two user interface frameworks and enables the Voice Assistant to be displayed within a UIKit-based application interface. The same methodology was used to include the assistant into the **case study application** introduced in subsection 1.3, which uses extensively the older framework as iOS architectural technology. A comprehensive explanation of this process will be provided at the end of the section, in 5.9.4.

By adhering to the technological limits and needs described, the Voice Assistant module can be seamlessly integrated into a real P2P payment application, improving the UX with voice-driven features.

5.9.1 SwiftUI compatibility

Integrating the Voice Assistant module into a host payments app, particularly within SwiftUI view hierarchies or a UIKit-based architecture as seen in the case study application, is a **simple** and **straightforward** process. This is mostly given to the simplicity provided by the `UIHostingController`, which is a `UIViewController` introduced by UIKit that simplifies the integration of SwiftUI views into existing UIKit view hierarchies, assuring *compatibility* and making the integration process easier.

Specifically, in order to incorporate a SwiftUI view into a UIKit-based application, developers can easily create a `UIHostingController` and provide the SwiftUI view as an argument. This controller can be used in the same way as any other *View Controller* in the UIKit ecosystem, whether it is presented modally, pushed into a *navigation stack*, or embedded as a child *View Controller*. This approach not only preserves the modularity and ability to be reused of the **SwiftUI** views, but also takes advantage of the rich capabilities and familiarity of **UIKit**. This makes it an excellent strategy for adding new SwiftUI-based features to existing legacy applications.

In the **case study** of the P2P payment application, totally based on a UIKit design architecture, the Voice Assistant module was incorporated using this identical method. By wrapping the `PaymentsVoiceAssistantView` in a `UIHosting-Controller`, the Voice Assistant has been easily integrated into the app, improving the User eXperience without having to modify the existing UIKit-based design.

Although this is now a widely used and efficient approach for integrating SwiftUI views into UIKit apps, it is important to acknowledge that as the new framework progresses and develops, new patterns and methodologies may arise. Currently, this is a very effective and widely accepted solution, because it provides an effective strategy that takes advantage of the benefits of both frameworks.

5.9.2 App Context

When incorporating the Voice Assistant into the payment app, it is essential to provide a specialized **App Context object** that *customizes* the assistant's behavior to match the specific requirements of the application. This context object includes essential user data, such as **contacts** information and **bank account** details, enabling the assistant to carry out customized tasks like financial transactions. In order to simplify this process, in the Voice Assistant module special object types has been created, to handle this data in a way that is *independent* of the specific implementation details of the host application:

- The **"Bank Account"** type, is one of the mentioned Voice Assistant types, and it is particularly relevant. It represents a user's bank account within the assistant's ecosystem by encapsulating important information such as the account ID, name, and *currency* details.

- The **"Currency"** aspect is enhanced with another specialized type that includes both *symbolic* and *literal* representations, guaranteeing a complex comprehension of monetary data.

- Furthermore, a new object for the **"Contact"** has been defined to represent a user's contact within the scope of the payment application. This type is distinguished by a distinct identifier and personal characteristics such as names, which serve to connect the user's personal contacts with those who are *enrolled* in the payment service.

- The last custom object is the **"Transaction"** type, specifically created to incorporate the complex details of payment transactions within the assistant's architecture. The system distinguishes between *incoming* and *ongoing* transactions based on the *sign* of the transaction amount. This type includes characteristics such as the monetary value of the transaction, the individual or entity participating in the transaction (in the form of a *Contact*), and the *Bank Account* linked to the transaction. These features collectively offer a comprehensive overview of the various aspects and details of a transaction.

In order to achieve a smooth integration of the Voice Assistant with the payment application, it is essential to provide an *App Context* object that includes this organized data. This integration also requires ways to *transform* these customized items between the Voice Assistant's internal representation and the specific formats used by the app. This **bidirectional conversion** guarantees that the assistant can seamlessly handle user commands while interacting with the app's features, ensuring a seamless and consistent User eXperience inside the payment application ecosystem.

5.9.3 App Delegate

A *bridge* is required to connect the **voice commands** with the **in-app actions**, in order to integrate the Payments Voice Assistant into the Peer-to-Peer payment application. The **App Delegate** object acts as the concrete representation of this bridge, playing a crucial role in facilitating the Voice Assistant's ability to carry out various in-app operations, including transactions, balance inquiries, and money requests. The development of this object is therefore crucial for the smooth functioning of the Voice Assistant within the application's ecosystem.

The App Delegate object fundamentally implements a certain **protocol**, which is an *interface* that defines the methods used by the Voice Assistant to interact with the application. This protocol includes capabilities such as *initiating transactions*, *retrieving account balances*, and *requesting funds* from other users inside the application. The Delegate object serves as the executor of the orders, converting voice instructions into executable operations.

The implementation phase involves the connection of the Voice Assistant's generic commands with the specific features of the application. When the Voice Assistant is given the command to "send money", the *App Delegate* interprets this as a function call that carries out the transaction within the app, utilizing the already established infrastructure for processing payments.

Moreover, the *App Delegate* plays an essential role in managing the context of operations. It keeps a **mapping** system linking abstract representations of entities, such as a user's *Contact*, to specific application data, like a particular enrolled account. This mapping guarantees that voice commands are interpreted in the specific context of the app, enabling the Voice Assistant to function within the user's personalized app environment, resulting in a customized and efficient experience.

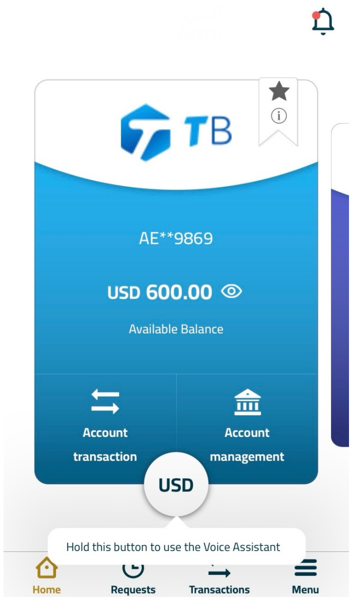
The *App Delegate* typically enables interaction with the application's **backend services**. It interacts with the backend to carry out tasks such as processing transactions and retrieving account details. This ensures that the actions initiated using voice commands are accurately recorded in the user's account and transaction history. This interaction highlights the importance of the Delegate implementing strong **security** mechanisms to protect user data and transactions during the entire process.

Essentially, the **App Delegate** plays a crucial role in integrating the Payments Voice Assistant into the host P2P payment application. The translation layer translates voice-driven commands into precise, safe, and context-aware in-app operations, creating a real and efficient UX for voice-driven financial transactions. The following section, 5.9.4, will provide a more detailed analysis of the practical aspects of this integration. It will explain the technical and operational factors that has been considered in the incorporation of the Voice Assistant into the case study application.

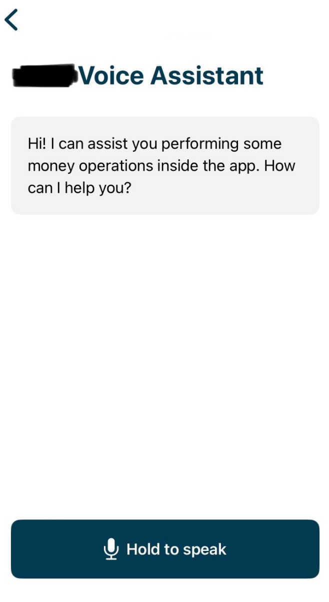
5.9.4 Integration into a real P2P Payment App

In order to assess the effectiveness of the Voice Assistant in a real-world context, it has been seamlessly integrated into the P2P payment application described in section 1.3. The integration was carried out by precisely following the instructions specified in section 5.9. This ensured that the application's underlying *UIKit* architecture and *Coordinator pattern* were utilized to ensure smooth navigation and feature management across different screens of the app. The **Coordinator pattern**, a significant architectural style commonly employed in iOS mobile app development, facilitated the integration by enabling a decoupled approach to managing *app workflows* and *navigation*. This approach allowed for the seamless addition of the Voice Assistant new feature without causing any disruption to the existing app structure.

A specialized **Coordinator** object was specifically created for the Voice Assistant feature, providing a dual function by also serving as the **App Delegate**. By making

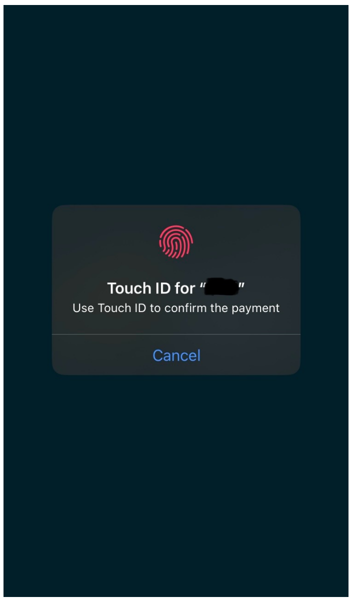


(a) Home CTA.

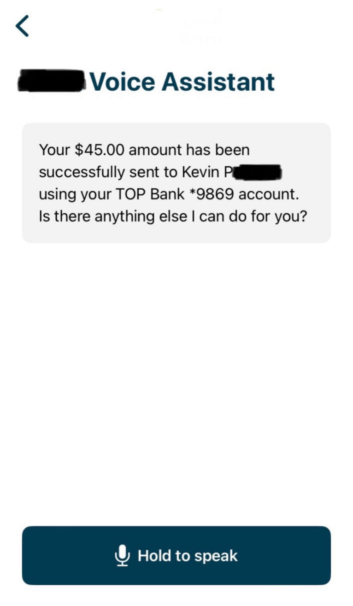

[redacted] Voice Assistant

Hi! I can assist you performing some money operations inside the app. How can I help you?

(b) Conversation start.

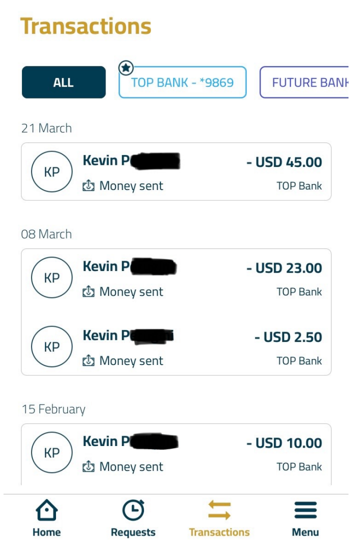

Touch ID for "[redacted]"
Use Touch ID to confirm the payment
Cancel

(c) Biometric auth.


[redacted] Voice Assistant

Your $45.00 amount has been successfully sent to Kevin P[redacted] using your TOP Bank *9869 account. Is there anything else I can do for you?

(d) Send money success.


Transactions

ALL | TOP BANK - *9869 | FUTURE BANK

21 March

KP	Kevin P[redacted] Money sent	- USD 45.00 TOP Bank

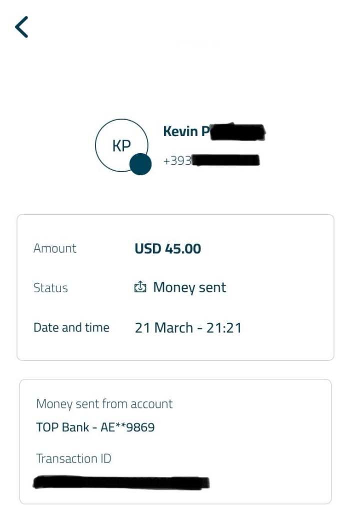
08 March

KP	Kevin P[redacted] Money sent	- USD 23.00 TOP Bank
KP	Kevin P[redacted] Money sent	- USD 2.50 TOP Bank

15 February

KP	Kevin P[redacted] Money sent	- USD 10.00 TOP Bank

(e) Updated transactions.


KP Kevin P[redacted]
+393[redacted]

Amount	USD 45.00
Status	Money sent
Date and time	21 March - 21:21

Money sent from account
TOP Bank - AE**9869

Transaction ID
[redacted]

(f) In-app transaction.

Figure 5.6: Examples of the case study application's screens showcasing the *Voice Assistant integration.*

this strategic choice, the Coordinator was able to start the Voice Assistant feature and manage its whole lifecycle within the app's environment. When activated, this Coordinator creates a `PaymentsVoiceAssistantView`. The view includes important elements such as the user's banking and contacts information, which are stored within the *App Context*. Additionally, a *configuration object* is provided to customize the UI of the Voice Assistant interface, including details like the title and button colors.

The Voice Assistant's interface was created using SwiftUI to take advantage of its declarative syntax and state management capabilities. To seamlessly integrate the `PaymentsVoiceAssistantView` into the UIKit-based app architecture, a `UIHostingController` was used to encapsulate it, as described in 5.9.1.

As the *App Delegate*, the Coordinator also carries out crucial tasks to guarantee safe and effective communication with the app's payment infrastructure, as described in 5.9.3. The system incorporates **biometric authentication** technologies, such as *FaceID* or *TouchID* (depending on the used device model), to ensure the **security** and **privacy** of transactions, emphasizing the commitment to user's data protection.

The **activation** procedure for the Voice Assistant was elegantly incorporated into the main screen of the app. Users can launch the assistant by pressing the **Home button**, which acts as the *Call To Action (CTA)*, for a short length of 0.75 seconds, as shown in subfigure 5.6a. This action triggers the Voice Assistant, as displayed in subfigure 5.6b, which subsequently starts an interactive conversation with the user, providing assistance for different in-app operations. In order to familiarize users with this innovative function, a small **pop-up message** was added on the main page. This message quietly informs users about the existence of the Voice Assistant when the app is launched, as seen in subfigure 5.6a, disappearing after a few seconds.

Figures 5.6d, 5.6e and 5.6f provide additional evidence of the assistant's integration into the app's transactional operations. These screenshots illustrate a practical example of the Voice Assistant during the a $45 payment execution, with the corresponding *biometric authentication* process, a subsequent assistant's response confirming the success of the *send money* operation, and an updated *transactions tab* in the host application displaying the newly executed payment. This demonstrates the assistant's practical effectiveness and smooth integration into the app's ecosystem.

Chapter 6

Testing and Evaluation

Chapter 6 presents an extensive examination of the **testing** and **evaluation** stages for the AI-Powered Voice Assistant discussed in this Thesis work, specifically intended for a Peer-to-Peer payment application. This Chapter is an important point in the development process, because it focuses on evaluating how effective the assistant is in making user interactions more efficient, particularly in helping users access P2P payment services within the host application.

Section 5.1.6 focused on assessing the *TensorFlow BERT classifier* **alone**, however the narration of this Chapter continues by expanding on those preliminary testing steps. The primary objective of this initial phase was to evaluate the effectiveness of the core Machine Learning model only, which is responsible for identifying the intents and extracting relevant entities from user speech, and served as a foundation for conducting a more comprehensive evaluation of the Voice Assistant capabilities.

In section 6.1, the systematic **testing phase** of the Voice Assistant module is explored, covering the *User Interface*, *AI capabilities*, and the complex *core logic* that supports its functioning. This step was carefully planned and executed using a specifically designed **test iOS application**, created for properly assessing the assistant's effectiveness in a wide range of situations. The testing initiative was supported by a varied group of people who willingly participated as **tester users**, offering important opinions on the assistant's practical usefulness and User eXperience.

In section 6.2, the Chapter turns its attention to the careful assessment of the **results** obtained during the testing phase. The review was mostly based on feedback obtained from users using a standardized **online form**, in which they provided their feedback and opinions of the Voice Assistant. This data collection was crucial in comprehending the assistant's impact on *user engagement*, its *effectiveness* in facilitating Peer-to-Peer financial transactions, and its overall *acceptance* within the user base.

In synthesis, the Chapter attempts to provide an extensive evaluation of the AI-Powered Voice Assistant's performance by employing an approach that includes extensive testing and careful examination. The evaluation assesses the assistant's capacity to fulfill the user's requirements outlined in chapter 3, with a specific focus on improving the **accessibility** and user-friendliness of P2P payment services within the application environment.

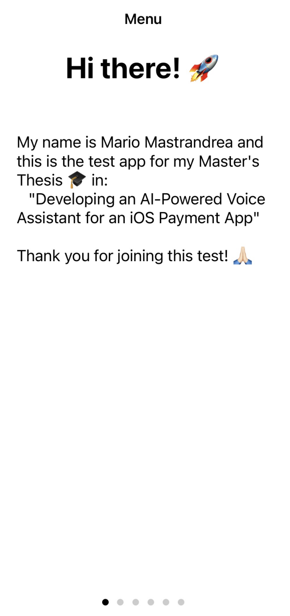
6.1 Voice Assistant Test Application

The creation of a dedicated iOS application for testing played a crucial role in assessing the performance of the Voice Assistant in the P2P payment field. The choice of having a specific **test application** was fully driven by the necessity of having a *complete* software target to test the Voice Assistant module, especially to test the UI/UX effectiveness. Therefore, it was not possible to evaluate the module's complete set of capabilities in isolation.

The test app was specifically created to replicate the basic functions of a Peer-to-Peer payment application. It primarily focuses on voice-activated features, including the ability to send money, request money, check an account balance, and examine the most recent transactions, which are the main capabilities offered by the Voice Assistant proposed in this project work.

This test application is provided with **introductory screens**, as depicted in Figure 6.1. The purpose of these screens is to familiarize users with the app's objectives within the scope of this Thesis project and the Voice Assistant testing phase (subfigure 6.1a). They also provide a concise description of the **domain context** in which the Voice Assistant functions, i.e. a P2P payment application where the user possesses bank accounts in multiple currencies, as depicted in the screenshot shown in subfigure 6.1b. In order to create a more realistic experience, the application incorporates a simulation of two fake bank accounts: one at "Top Bank" in US dollars ($) and another at "Future Bank" in dirhams (AED). This allows users to transfer or request money to/from their phone contacts as if they were actual users of the application's services. Indeed, when the Voice Assistant feature is activated, the program requests permission to access the user's contacts, which is necessary for the assistant to work correctly.

Subfigure 6.1c displays the features supplied by the Voice Assistant, while

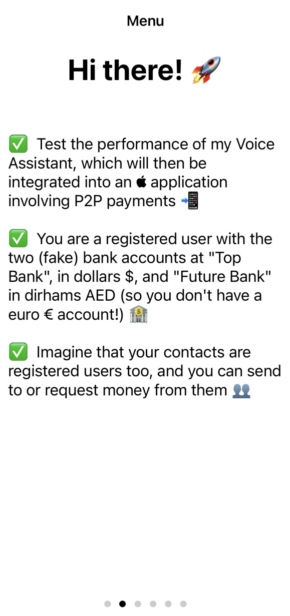

Menu

Hi there! 🚀

My name is Mario Mastrandrea and this is the test app for my Master's Thesis 🎓 in:
"Developing an AI-Powered Voice Assistant for an iOS Payment App"

Thank you for joining this test! 🙏

(a) Intro.


Menu

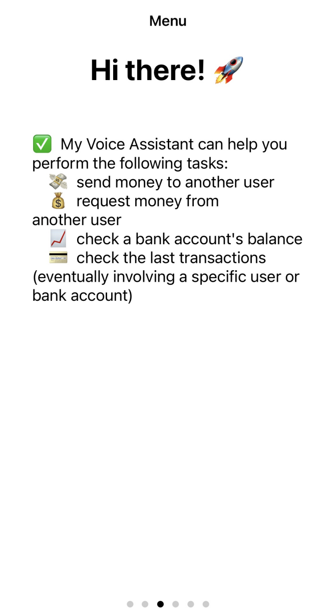
Hi there! 🚀

✅ Test the performance of my Voice Assistant, which will then be integrated into an application involving P2P payments 📲

✅ You are a registered user with the two (fake) bank accounts at "Top Bank", in dollars $, and "Future Bank" in dirhams AED (so you don't have a euro € account!) 🏦

✅ Imagine that your contacts are registered users too, and you can send to or request money from them 👥

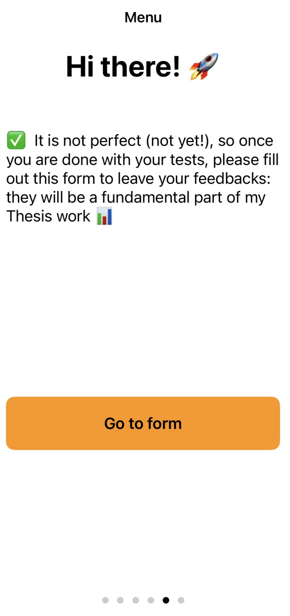
(b) App context.


Menu

Hi there! 🚀

✅ My Voice Assistant can help you perform the following tasks:
💸 send money to another user
💰 request money from another user
📈 check a bank account's balance
💳 check the last transactions (eventually involving a specific user or bank account)

(c) Assistant capabilities.

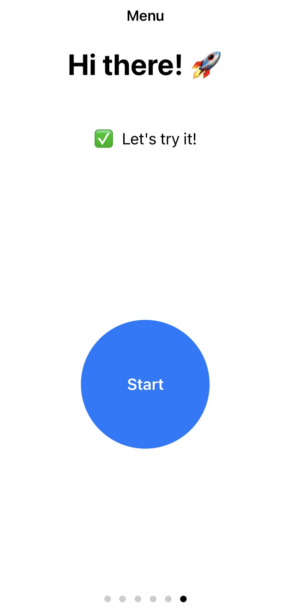

Menu

Hi there! 🚀

✅ It is not perfect (not yet!), so once you are done with your tests, please fill out this form to leave your feedbacks: they will be a fundamental part of my Thesis work 📊

Go to form

(d) Evaluation form link.

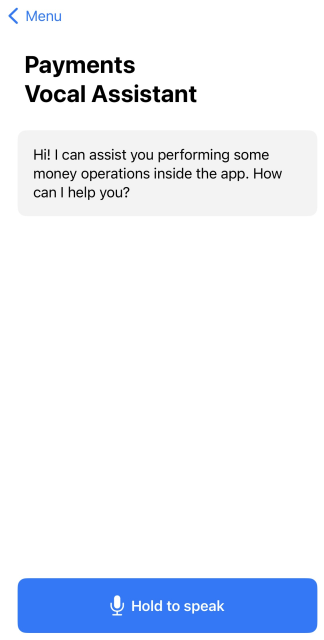

Menu

Hi there! 🚀

✅ Let's try it!

Start

(e) Start assistant.


< Menu

Payments
Vocal Assistant

Hi! I can assist you performing some money operations inside the app. How can I help you?

Hold to speak

(f) Conversation start.

Figure 6.1: Main screens of the *Test iOS Application*.

subfigure 6.1d provides a link to the **online form** that should be utilized to provide feedback on the Voice Assistant. Finally, Figure 6.1f depicts the Voice Assistant initiating a discussion with the user, displaying the identical features and screens as seen in Figure 5.5.

The User Interface of the test application was implemented using SwiftUI, which made it easier to incorporate the Voice Assistant module. This aligns with the methodology described in section 5.9 for the assistant module integration. In order to replicate the behaviors of a host payment application, a simulated *App Context* and an *App Delegate* mock were created. Such objects provided fake bank account information and the list of user's contacts, while also imitating the actual in-app operations.

To acknowledge the significance of user feedback during the testing phase, the test application was made accessible on *TestFlight*, a platform that facilitates apps' public release and convenient accessibility for various kinds of users, as elaborated in subsection 6.1.1. This method not only made it easier to *spread* the test application, but also made the process of gathering important user feedback simpler. This feedback is fundamental for analyzing and improving the functionality and User eXperience of the Voice Assistant.

6.1.1 Test iOS App Release on TestFlight

The **deployment** of the iOS application on Apple's **TestFlight platform** for the purpose of testing was a fundamental stage in the *evaluation process*. This stage required an extensive **configuration** of the *App Store Connect* platform, starting with the creation of an *Apple Developer Account*. An **Apple Developer Account** is important for anyone who intend to publish applications on the *App Store* or distribute them for testing purposes [46]. It grants access to developer materials and serves as a portal for submitting apps to Apple's platforms.

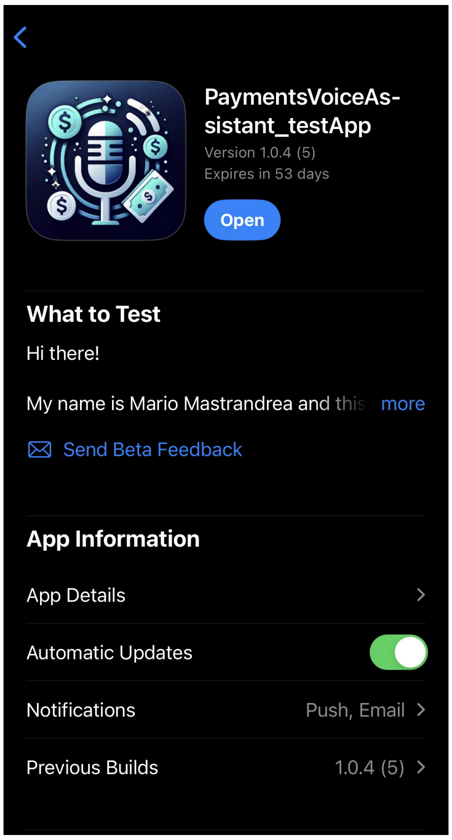
In order to prepare the application for TestFlight distribution, a number of elements were carefully configured. At first, a special **Certificate** has been created, which functions as a digital signature to verify the app's source and *integrity*. This *Certificate* is essential for verifying the *authenticity* of the app's creator and guaranteeing that it has not been altered.

Subsequently, an **App Identifier** was generated, which is a distinctive string that serves as a means of distinguishing the app from others available on the App Store. The *Identifier* is essential for features such as Push Notifications and app-specific preferences, but they are not included in the test application being discussed.

In addition, a **Provisioning Profile** was established to link the app's *Identifier* with the development devices. This *Profile* allows the app to operate exclusively on **authorized devices** and is crucial for testing the app prior to its public distribution.

After completing these preparatory steps, the application was submitted to TestFlight, thereby enabling external testers to access it by means of ad-hoc invitations. The group of testers, consisting of **50** friends who volunteered for the testing phase, offered valuable feedback on the app's voice assistant capabilities. Their thoughts were crucial in identifying the advantages and disadvantages in the development process, and they suggested changes for any potential future work on the project, as it will be better discussed in subsection 6.2.2.

Two figures are associated to this phase. Figure 6.2 displays a screenshot of the iOS app's **information** on **TestFlight**, providing details about the app's version, build number, and description specifically for testers. Figure 6.3 exhibits the **app**


PaymentsVoiceAssistant_testApp
Version 1.0.4 (5)
Expires in 53 days
Open

What to Test
Hi there!
My name is Mario Mastrandrea and this more
Send Beta Feedback

App Information
App Details
Automatic Updates
Notifications Push, Email
Previous Builds 1.0.4 (5)

Figure 6.2: TestFlight app details.

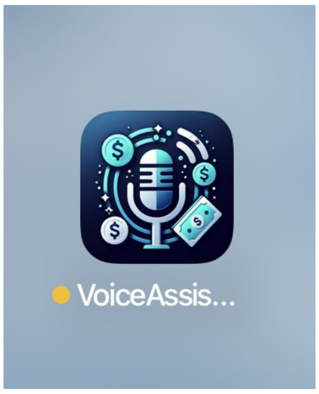


Figure 6.3: Test app icon.

icon as it appears on an iOS device, serving as a visual indicator for testers to readily identify and open the application.

The testing phase on TestFlight served as both a technical necessity and an opportunity to collect user feedback. This data can be used to create future improvements on the AI Voice Assistant, resulting in a more seamless and functional app experience for all future users.

6.2 Voice Assistant Evaluation

The **evaluation phase** of the Voice Assistant was carefully designed to measure the **effectiveness** and **user satisfaction** of the built solution. An essential element of this phase involved developing an online **Google form** to get comprehensive feedback from users who interacted with the Voice Assistant integrated within the test application. This form played a crucial role in collecting valuable information on the users' experiences, their levels of **satisfaction**, and any **difficulties** they had when using the Voice Assistant. The form consisted of a wide variety of questions, such as rating scales to measure user satisfaction and open-ended questions to gather specific feedback and suggestions for enhancement.

The *aggregation* of data via the Google form offered a systematic method for understanding the effectiveness of the Voice Assistant from a user-oriented perspective. Participants were requested to provide feedback on their interactions with the assistant inside the test app, involving a set of specified tasks designed to assess the functionality, accuracy, and user-friendliness of the Voice Assistant.

The evaluation phase is based entirely on user feedback regarding the usage of the given **test iOS application**. The final integration of the Voice Assistant into the *case study P2P payment app* (detailed in section 5.9.4) was limited to a few manual testing. Because of **confidentiality** concerns, in fact, it was not possible to carry out an extensive assessment of the Voice Assistant in the real payment application. Nevertheless, this manual testing played an important role in guaranteeing the smooth functioning of the Voice Assistant within the app's environment, validating its ability to improve user interactions in a safe and effective way.

The remaining sections will provide a more in-depth analysis of the evaluation process. Subsection 6.2.1 will provide a detailed description of the **evaluation form**'s structure, including the logic behind the selected questions. Subsequently, subsection 6.2.2 will provide a comprehensive **examination** of the findings derived from the form, providing valuable observations on the users' experiences, levels of satisfaction, and the overall efficacy of the Voice Assistant in supporting user interactions inside the P2P payment application.

6.2.1 Evaluation Form

A carefully designed **Google form** was utilized to collect extensive user feedback during the evaluation of the Voice Assistant. The form was methodically structured into **distinct sections**, each designed to analyze various aspects of the user's interaction with the Voice Assistant, thereby offering an exhaustive evaluation of its *usability*, *functionality*, and overall *user satisfaction*.

1. The first part of the form focused on **Usability and Interaction**. It included questions that examined how easy it was to start and interact with the Voice Assistant, how well it understood voice commands and queries, how natural the interactions felt, and how quickly the Assistant responded. This part was crucial in evaluating the inherent *naturalness* of the Voice Assistant and its effectiveness in enabling user commands.

2. Following that, the survey focused on the **Functionality and Effectiveness** of the Assistant, with tailored questions aimed at evaluating its ability in performing specific tasks such as checking balances, reviewing recent transactions, sending money, and more complex operations like comprehending specified amounts and names. This portion was crucial in comprehending the Assistant's operating *efficiency* and its *capacity* to seamlessly manage financial transactions.

3. The **Reliability and Error Handling** section went deeper into the Assistant's performance, specifically examining the frequency of *misunderstandings* or providing *inaccurate* information, as well as the system's ability to *recover* from errors or assist users in resolving them. These aspects are crucial for maintaining user trust and facilitating natural interactions.

4. The form also had a **Security and Privacy** section, with questions that evaluated the clarity of instructions and answers regarding *transaction authorizations*, as well as users' trust in the *security* and *privacy* of their financial transactions data when utilizing the Voice Assistant. This section highlighted the importance of applying strong security and privacy measures in financial applications for maintaining user trust.

5. Finally, the **User eXperience and Satisfaction** part assessed the overall user perception towards the Voice Assistant, including factors such as the *pleasantness* of the Assistant's voice, the overall *satisfaction* with the experience, and the probability of using the Voice Assistant for real future transactions. This section aimed to evaluate the general impact of the Voice Assistant on the User eXperience and its potential for future usage.

The form mainly consisted of **Likert scale** items [47], which let participants to rate their experiences on a scale ranging from 1 to 10. Additionally, there were **open-ended questions** that allowed users to submit specific complaints or suggestions for improvement. The decision to use a 1 to 10 scale instead of a more limited range was intentional in order to include a wider variety of user answers. This allows for a more detailed review of the Voice Assistant's effectiveness across several parameters.

6.2.2 Evaluation Results

The analysis of the Google form responses gathered for the *Voice Assistant evaluation* reveals that the system demonstrates various degrees of performance in different areas. A total of 50 replies offer a full assessment of its **effectiveness** and highlight areas that require **improvement**. The full report with the form results has been attached in **Appendix A**, for the sake of completeness.

- The **Usability and Interaction** capabilities of the Voice Assistant were credited for their good feedback. Approximately 54% of users considered the activation and interaction procedure to be uncomplicated and *easy* to understand, rating it between 9 and 10, as shown in Figure 6.4. Another 32% of users expressed a level of satisfaction with the user-friendliness, rating it between 7 and 8. Nevertheless, comprehending voice commands and queries with precision proved to be a more difficult task, as the majority of testers only achieved *satisfactory* results, with success rates ranging from 6 to 8, accounting for 64% of the responses. Additionally, 20% of the testers reported *accuracy concerns*, with success ratings ranging from 4 to 5. This highlights the necessity of improving the system's voice recognition capabilities in order to more effectively handle user commands. The users' perception of the assistant's interactions being *natural* and the *speed* of its responses were moderately positive, as 66% of users ranked the natural interaction flow between 7 and 9 on a scale of 1 to 10. However, a substantial 36% of users expected slower response times, indicating that the system's processing speed may be enhanced to provide a more seamless User eXperience. Users experienced challenges in communicating their requests due to the system's limited ability to reliably identify certain words or phrases, particularly those that include **non-English terms**. Additionally, the system's sequential approach to processing queries could be restrictive. These observations emphasize the need for the Voice Assistant to embrace a more adaptable and contextually aware processing strategy in order to better comprehend user intentions.

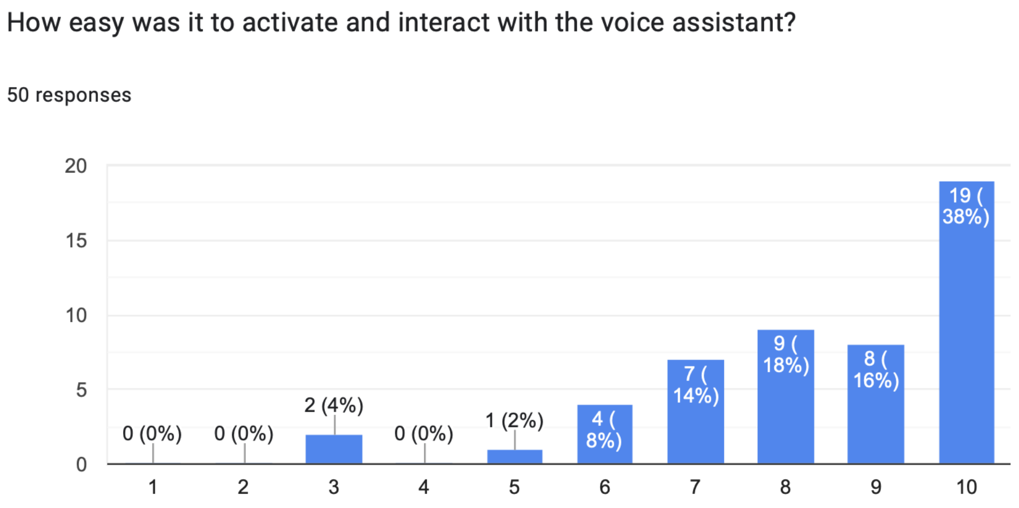


Figure 6.4: Results for a question on **usability and interaction**.

- Regarding the **Functionality and Effectiveness**, the assessment of the Voice Assistant's performance in tasks such as *checking balances*, accessing history of *transactions*, handling *money transfers*, and processing *money requests* produced **excellent results**. As illustrated in Figure 6.5, users expressed high levels of satisfaction, particularly with high ratings (8-10) ranging from 55% to 75% in all four tasks. Nevertheless, there were occasions of *misinterpretation*, particularly in identifying certain bank or contact names and handling intricate inquiries, suggesting the need for enhancement in the system's comprehension of context and ability to recognize entities, particularly for names that are **not** in English.
- In the **Reliability and Error Handling** section, the accuracy of the Voice Assistant in comprehending requests exhibited a diverse distribution over the

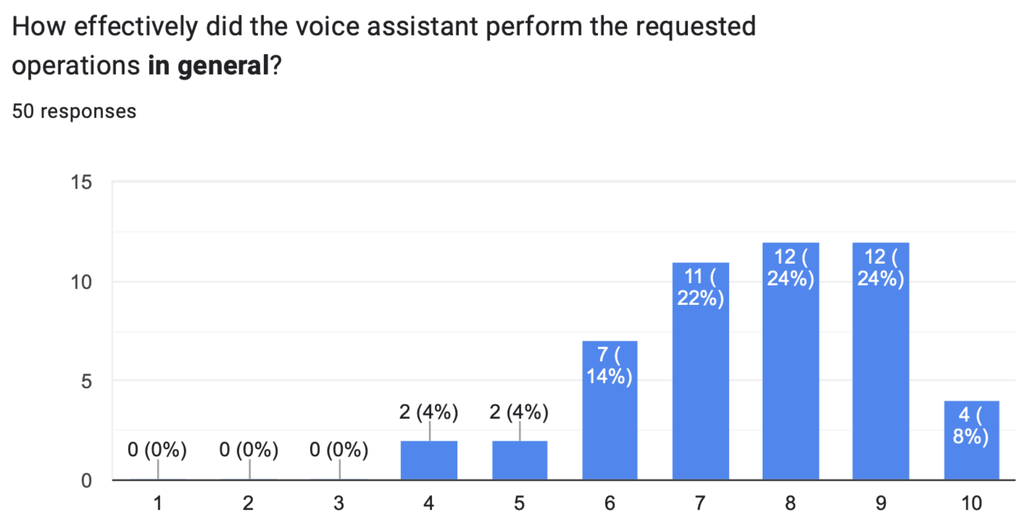


Figure 6.5: Results for a question on **functionality and effectiveness**.

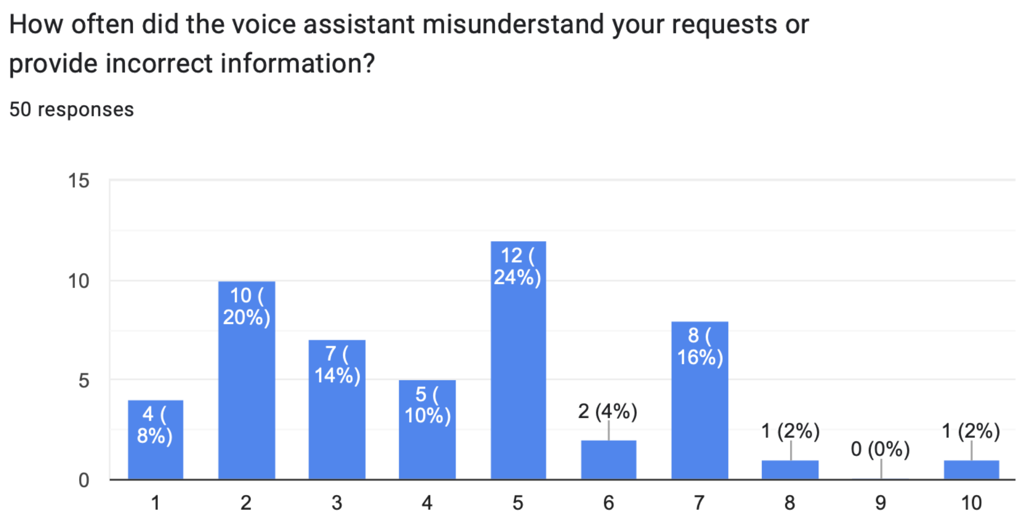


Figure 6.6: Results for a question on **reliability and error handling**.

range of satisfaction (Figure 6.6). The system's capacity to address errors and guide users towards the correct solutions was well-received, as indicated by 52% of respondents expressing high satisfaction rates (8-10). This suggests that the system possesses a strong basis for addressing misunderstandings and offering useful feedback, although there is still potential for further enhancements.

- The system's ability to ensure **Security and Privacy** in transactions was **highly appreciated**, as seen by 74% of users expressing confidence in the system's management of their financial activities, rating it between 7 and 10, as depicted in Figure 6.7.

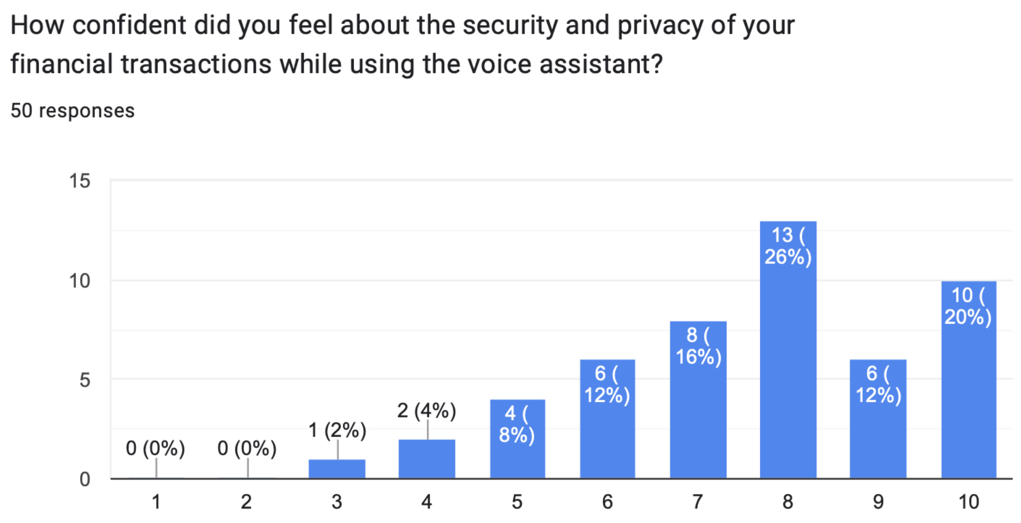


Figure 6.7: Results for a question on **security and privacy**.

- Finally, from the **UX** perspective, the *pleasantness* of the voice (58%, rated 8-10) and the **overall Satisfaction** of users with the Voice Assistant were also rather **high**, indicating a positive User eXperience. Approximately **60%** of the participants expressed a **high level of satisfaction** with the whole experience, rating it between 8 and 10, as shown in Figure 6.8. Additionally, over *half* of the respondents indicated a willingness to use the Voice Assistant for future financial transactions in an actual application, with a rating between 7 and 10.

Proposed additions for the future include the addition of functionalities such as manual backtracking, enhanced integration with other user interface components, more advanced confirmation prompts, incorporating a **more authentic voice**, and greater recognition of **non-English names**. These enhancements have the potential to greatly boost user engagement and the general efficiency of the assistant.

In general, the Voice Assistant achieved satisfactory performance in enhancing **usability** and **User eXperience**, with notable strengths in *security*, *privacy*,

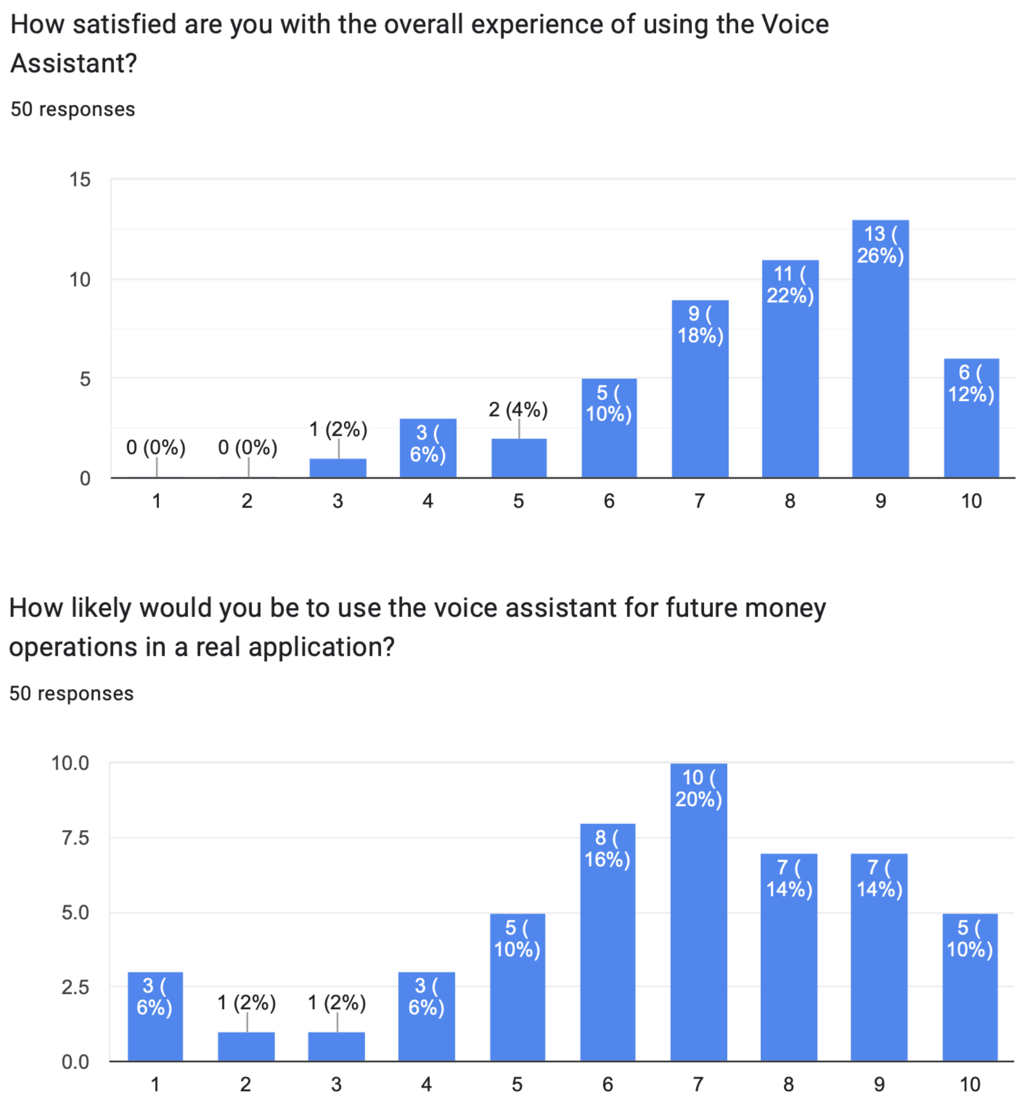


Figure 6.8: Results for questions on **user experience and satisfaction**.

and the **effectiveness** of interactions. Nevertheless, there are many aspects that might be enhanced, particularly in improving the accuracy of entities recognition, the speed of response, the naturalness of the voice, and the system's flexibility in handling complex or sequential requests. It is essential to address these limitations in order to enhance the capabilities of the Voice Assistant and ensure that it properly fulfills the needs of users.

Chapter 7

Conclusion and Future Work

In the final Chapter of this Thesis, a comprehensive analysis of the journey is conducted, methodologies, results, and insights gained from the innovative project of an AI-Powered Voice Assistant in a Peer-to-Peer payment application. This project was not simply a theoretical exercise but an exploration into the field of R&D, representing the *starting point* of what could potentially become a real innovative product.

The discussion begins by examining the **methodology** used, which is based on the earlier sections of the Thesis, specifically section 1.2. This reflection signifies more than just looking back; it involves an in-depth analysis of the *decisions*, *processes*, and novel *approaches* used to incorporate advanced AI technology into a mobile application.

One of the primary goals of this chapter is to analyze the **outcomes** that were acquired, namely the ones emphasized in section 6.2. The deployment of this Voice Assistant represented not only a technological milestone, but also showcased the concrete advantages that AI can offer in financial transactions, by improving *User eXperience*, *accessibility*, and *operational efficiency.* These findings serve as evidence of the voice assistant's ability to simplify financial transactions, while keeping the same level of *privacy* and *security* in the process, confirming the hypothesis established at the beginning of this research.

It is crucial to recognize that this project was initiated as a **Research and Development** activity, rather than a finished product. This differentiation is crucial as it emphasizes the *investigative* character of the study and its inherent potential for improvement and advancement. Although still in the experimental phase, the project produced **remarkable outcomes**, demonstrating the effectiveness of the AI Voice Assistant in a practical context of a payment application.

This achievement sets the foundation for future initiatives that might transform this first model into a complete and powerful product that has the potential to transform the procedures by which financial transactions are carried out.

In this Chapter, the detailed **adopted process** will be examined in section 7.1, carefully evaluating the design, development, and integration of the voice assistant in the P2P payment application, and an interpretation of the achieved results will be presented in the context of the literature topics treated in the project. This analysis will not only emphasize the accomplishments but also illuminate the *difficulties* faced, providing significant perspectives for future research and growth.

Section 7.2 will provide a thorough examination of possible **enhancements** to the Voice Assistant, based on user feedback and the knowledge gained during the project. This section will provide an outline for improving the assistant's functionality, usability, and integration, assuring its development from an initial prototype to a robust and flexible solution.

Finally, in section 7.3, an overview of the primary **findings** of the project will be provided, including the fundamental ideas, accomplishments, and contributions of this research to the fields of Artificial Intelligence, mobile applications, and financial technology. This summary is not only a careful evaluation of the journey done, but also serves as an inspiration for future studies in the field of AI-driven financial solutions.

7.1 Discussion

The project of creating an AI-Powered Voice Assistant for a P2P payment app on iOS devices involved exploring and developing various aspects of **innovation**, **integration issues**, and the complex nature of **user engagement** with emerging technologies in the financial sector. The goal of the project was to completely transform the way users interact with the system by incorporating a **voice-activated interface**, thus providing a more natural way of interaction with the app financial services. This would improve *accessibility* and maintain *privacy* by processing data **locally**. Nevertheless, the journey was characterized by a sequence of interesting experiments, mistakes, and discoveries.

- The central focus of the project was the creation and refinement of a **Machine Learning model** specifically designed for accurately identifying and comprehending spoken words (*entities*) and *intentions* within the application's framework. The objective was to create an exceptional and **complete model** that could easily carry out the complete inference process. This commitment, although innovative, faced significant obstacles when confronted with the inherent restrictions of the technologies used, specifically for the integration into iOS devices, for their computational capabilities, and for the used **ML libraries limitations** on state-of-the-art models as the *Large Language Models*. The desire for a comprehensive model had to be adjusted to a more *modular* and resource-conscious approach, highlighting the careful balance between ambition and practicality in technological progress.

- During the initial phase, which focused on **studying** advanced technologies such as Large Language Models and specifically **BERT** models, it became clear that there was an unintentional **bias** towards these models, which prevailed over the potential benefits of other models and methodologies in the development of voice assistants. This omission demonstrates the great impact of the specialization in technological research and development, highlighting the need for a more comprehensive and inclusive examination of the technology landscape. The lessons from this design process emphasized the significance of being flexible and prepared to changes in response to evolving discoveries, requirements and constraints.

- A crucial element of the project involved comprehending and matching with the **requirements of the user**. It became clear that doing **more specific surveys** and actively involving **different categories of users** might have greatly enhanced the design process, allowing for the customization of the voice assistant's functions to better meet the needs of a broader and more varied group of users, particularly individuals with haptic or visual impairments or limited technological proficiency. This reflection highlights the potential for future improvements, emphasizing the importance of inclusive design and the significant influence of technology in making financial services accessible to everyone.

- The process of integrating the designed ML solution into the iOS environment revealed the complex relationship between **software advancements** and **hardware constraints**. The impracticality of a single, all-encompassing model required a specific change in approach to utilize separate software components and prioritize processing on the device, in order to guarantee the feasibility of the solution. This stage of the project exemplified the iterative process of technical advancement, in which each obstacle serves as an opportunity for troubleshooting and innovation.

The knowledge obtained from the project goes beyond the immediate range of voice assistants in financial applications. It aligns with the wider conversation on the **incorporation of AI in mobile apps**, emphasizing the crucial significance of designing with the user in mind, offering a more natural *human-machine interaction*, exploring various technology options, and establishing a mutually beneficial connection between software capabilities and hardware limitations. The project establishes a foundation for future investigations in AI-driven applications, promoting a well-balanced strategy that aligns user requirements, technology capabilities, and device environments.

To summarize, the process of creating an AI-driven voice assistant for an iOS P2P payment application provided **valuable findings** that go beyond the specifics of the project. It highlighted the complex relationship between *User eXperience*, *technological advancement*, and the *practical constraints*, demonstrating opportunities for future projects in the field of AI-powered mobile applications.

7.2 Possible improvements

After assessing the results and gaining practical experience during the development of the Voice Assistant, various opportunities for **improvement** have been identified. These potential enhancements arise not only from the quantitative and qualitative **feedback** collected through the *evaluation form* (Section 6.2.2) but also from **personal observations** and **practical knowledge** obtained during the project. The objective of these upgrades is to further improve the *User eXperience*, enhance the *functionality* of the Voice Assistant, and assure its *flexibility* to a diverse range of user needs and situations. The following are crucial areas indicated for future progress:

- **Multilingual Support**: enabling multilingual support for the Voice Assistant would greatly improve its *accessibility* and expand its user base. This task encompasses not only the comprehension and the translation of instructions and replies but also the adjustment of cultural and linguistic aspects to guarantee seamless communication across different languages. It necessitates the utilization of either *multi-lingual* Language Models or multiple models tailored for different languages.

- **Improved Named Entity Recognition**: the precision of identifying entities inside a specified application domain, especially for *non-English names* and specialized terms like bank account names, can be significantly enhanced.

Utilizing more sophisticated matching algorithms or integrating contextual learning could reduce misunderstandings and improve the assistant's reliability.

- **Better Amount Identification**: enhancing the technique for accurately detecting transaction amounts, especially for particular terms, is of crucial importance. By implementing advanced parsing algorithms and enhancing contextual comprehension, it is possible to minimize errors in transactions processing.
- **Enhanced Context-Aware Conversation Flow**: improving Conversation State Management could enhance the **intuitiveness** and **naturalness** of the conversational experience. The Voice Assistant can provide more relevant responses and accurately predict user requests by comprehending the context and maintaining the conversation's flow.
- **Separation of Response Generation**: By decoupling the *Response Generation* component from the *Dialogue State Tracker*, an higher *flexibility* in creating responses might be reached. This modular approach would enable the seamless incorporation of several languages and enable the generation of more dynamic and contextually suitable replies.
- **UI/UX Innovation**: The exploration and experimentation with various kinds of User Interface and interaction models, such as the *tap-to-talk* feature for the assistant's main button (instead of holding it), might have the potential to improve the User eXperience. Enhancing the interaction model improves the **accessibility** and **user-friendliness** of the Voice Assistant.
- **Distinct answer Management**: Differentiating between the *visual representation* of an answer and its *spoken pronunciation* might help resolve problems associated with the quality of Speech Synthesis. Adapting the pronunciation to align with the desired tone and context can enhance the **authenticity** and **captivation** of interactions.
- **Expansion of Supported Intents**: Expanding the range of supported *intents* and, as a result, extending the supported *conversational states* in the Dialogue State Management, will greatly increase the capabilities of the Voice Assistant. By including a broader range of user instructions and queries, the assistant is able to offer more extensive help and **usefulness**.

By implementing these upgrades, this experimental solution can be transformed into a **fully-functional** and **sophisticated** AI Voice Assistant. However, it is important to choose a well-balanced process that takes into account both the technological feasibility and the potential influence on the UX. Continuous user feedback and iterative development will play a crucial role in determining the order of importance of these features to meet the changing needs of the users.

7.3 Conclusions

In conclusion, the creation of an AI-Powered Voice Assistant for a Peer-to-Peer payment app for iOS platforms is a significant step in **incorporating AI technology**, specifically *Natural Language Processing*, **into mobile apps** in the financial services sector. This Thesis has effectively demonstrated the practical utilization of advanced ML models, such as **BERT**, for text classification and unique Dialogue State Tracking methods. These techniques enable the seamless execution of complicated user interactions simply using voice commands. Integrating such an assistant into a real P2P payment application not only improves the *User eXperience* by allowing for intuitive, hands-free operation, but also gives priority to user *privacy* by **processing data on the device alone**.

Nevertheless, this research also revealed limitations that are inherent in existing AI technologies and their use in mobile environments. Notable issues include overcoming the limitations of Large Language Models integration into mobile devices, ensuring the security and privacy of sensitive financial transactions, and obtaining high accuracy in *Speech Recognition* and *Natural Language Understanding* in a variety of linguistic circumstances.

In the future, there are many opportunities for further research and advancement in this particular subject. First, delving into more sophisticated NLP models and methodologies could address the complex comprehension of user intent and conversational context. Furthermore, the incorporation of multimodal Artificial Intelligence, which combines auditory, visual, and textual inputs, presents a promising opportunity for developing software solutions that are more *functional* and *user-friendly*. Lastly, it is essential to prioritize also the continuous advancement of **AI ethics** and governance principles in order to effectively deal with societal concerns regarding privacy, security, and inclusion in AI-driven applications.

By overcoming these limitations and pursuing new research opportunities, the incorporation of AI technologies into mobile applications, particularly in the financial industry, will advance to the next level. This progress will provide more advanced, secure, and user-focused solutions that respond to the various requirements of the global user base.

Acronyms

AI	Artificial Intelligence
ALBERT	A Lite BERT
API	Application Programming Interface
ARC	Automatic Reference Counting
BERT	Bidirectional Encoder Representations from Transformers
BIO	Begin-Inside-Outside
CS	Computer Science
CTA	Call To Action
CUI	Conversational User Interface
DI	Dependency Injection
DistilBERT	Distilled version of BERT
DL	Deep Learning
DNN	Deep Neural Network
DSM	Dialogue State Management
DST	Dialogue State Tracker
DST	Dialogue State Tracking
EDDA	Electronic Direct Debit Authorization

EU	European Union
FinTech	Financial Technology
FSM	Finite State Machine
GCD	Grand Central Dispatch
GPT	Generative Pre-trained Transformer
GPU	Graphical Processing Unit
GUS	General Understanding System
HMM	Hidden Markov Model
IDE	Integrated Development Environment
IoT	Internet of Things
IT	Information Technology
LLM	Large Language Model
LM	Language Model
LR	Learning Rate
LSTM	Long Short-Term Memory network
MB	Megabyte
ML	Machine Learning
MLM	Masked Language Modeling
ms	millisecond
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
NER	Named Entity Recognition

NLG	Natural Language Generation
NLP	Natural Language Processing
NLU	Natural Language Understanding
NN	Neural Network
NSP	Next Sentence Prediction
OS	Operative System
P2P	Peer-to-Peer
PC	Personal Computer
R&D	Research and Development
RNN	Recurrent Neural Network
RoBERTa	Robustly Optimized BERT pretraining Approach
seq2seq	sequence-to-sequence
seq2vec	sequence-to-vector
SDK	Software Development Kit
SPM	Swift Package Manager
STT	Speech-to-Text
SR	Speech Recognition
SS	Speech Synthesis
TF	TensorFlow
TFLite	TensorFlow Lite
TTS	Text-to-Speech
UI	User Interface
UX	User eXperience

VCS	Version Control System
VUI	Voice User Interface

Glossary

Cocoapods Dependency manager that manages external libraries in iOS applications

CoreML iOS framework to create, train and easily integrate Machine Learning models into iOS applications

FaceID Biometric authentication method used in Apple devices based on user face recognition

Git Major software used as distributed Version Control System to keep track and manage file changes in a Software Development process

GitHub Cloud platform using Git and hosting remote repositories

Google Colab Cloud service offered by Google providing a web-based environment to run Jupyter notebooks

iOS Apple iPhone and iPad Operative System

macOS Apple MacBooks' Operative System

Swift Modern programming language used to create iOS applications

SwiftUI Apple's Swift framework to create iOS User Interfaces using a declarative paradigm

UI/UX The combination of User Interface and User eXperience areas

TestFlight Official online platform offered by Apple to distribute and test iOS applications with final users

TouchID Biometric authentication method used in Apple devices based on user fingerprint recognition

UIKit iOS framework to create User Interface using a programmatic approach

UML Short for Unified Modeling Language, is a standardized modeling language with different sets of diagrams to document artifacts of a software systems

Xcode Official Integrated Development Environment (IDE) software used to create iOS applications

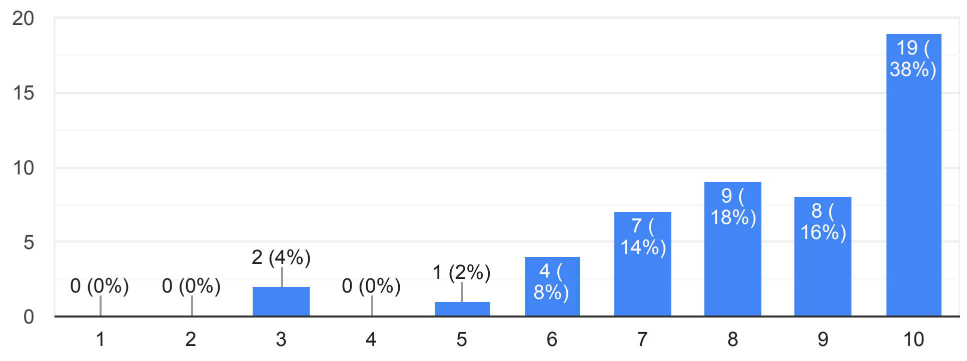
Voice Assistant Evaluation Results

50 responses

Usability and Interaction

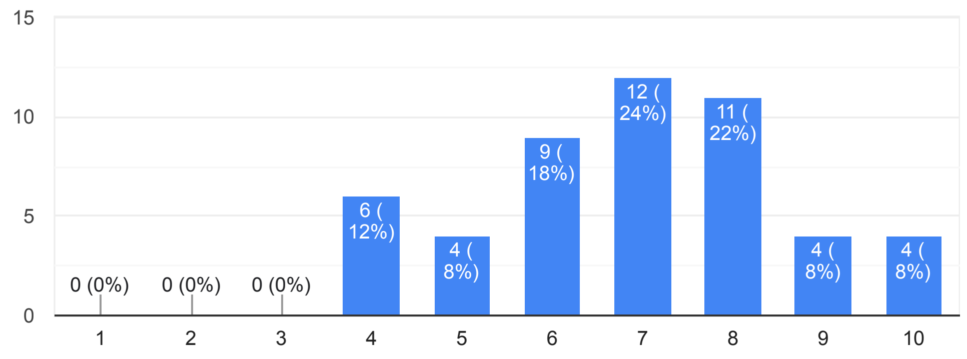
How easy was it to activate and interact with the voice assistant?

50 responses



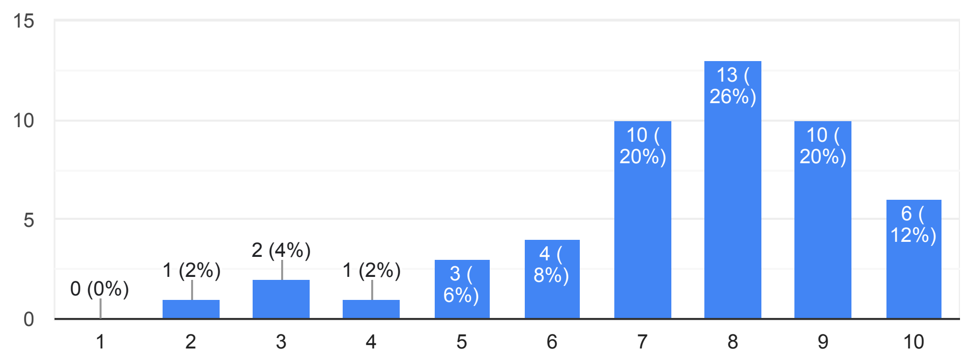
Was the voice assistant able to accurately understand your voice commands and queries?

50 responses



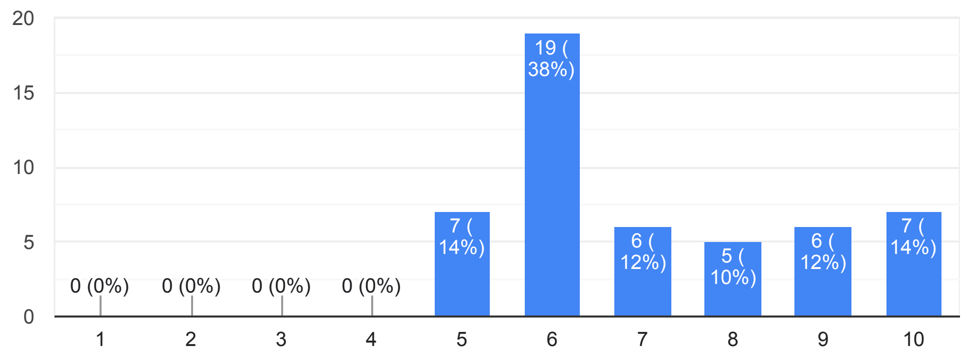
How would you rate the naturalness of the interactions with the voice assistant?

50 responses



How would you rate the speed of the voice assistant's responses and actions? (if you like it select 5-6)

50 responses



Did you experience any difficulties in communicating your requests to the voice assistant? If so, please specify.

27 responses

It is difficult to recover from a wrong interpretation of the voice

The assistant doesn’t recognise easily the word ‘transaction’, sometimes.

Ciao Mario, secondo me l unico problema è relativo al fatto che procede sequenzialmente sulle domande. Non puoi quindi chiedere tutto in un’unica frase, l assistente comunque ti forzerà a procedeere per steps: prima l amount poi il contatto e poi il bank account. Sarebbe meglio studiare un modo per estrarre tutti i token da una frase e se sono tutti presenti non chiedere niente, se no se c’è qualcosa che manca, chiedere solo quello che manca.

È capitato che confondesse l’operazione di ricezione denaro con l’invio: dopo avermi chiesto quanti soldi ricevere e la fase di selezione del contatto, mi ha chiesto di confermare l’operazione di invio denaro piuttosto che ricezione

Sometimes catch just one piece of information when you say everything at once

The Assistant had problems understanding the names of the banks.

The voice assistant couldn't understand semantic meaning, e.g. "Send all of my money"

Yes she didn’t understand the bank name and I had trouble requesting money instead of sending money

It didn’t understand “one dollar” and it had some difficulties with names

Un paio di volte non è riuscito a capire quale banca utilizzare e a chi inviare i soldi non trovando quindi i contatti nella rubrica

When I used synonyms of “transactions” the assistant didn’t understand and helped me. I would probably expand the vocabulary so I wouldn’t be tied to the exact terms.

I think it does not understand synonyms or related sentences that do not include exactly the words that are pronounced by the assistant

I had two issues : I couldn’t check my bank account (I tried different ways to say it but it didn’t work) and the voice assistant didn’t understand long or complicated contacts like « Grandmum Lulu » for example

The assistant shows some difficulties in understanding my request when it contains additional, irrelevant information (such as "I want to send x money to y for the private lessons received"), and when the name of the person in the request (the recipient or sender of the money) consists of only one word.

Yes, for instance when the assistant asks me from which bank I want to perform the action, in any case, even if before I asked to request money from someone, the assistant changes my request and only assumes that I want to send money.

The assistant had trouble understanding my request when I used a currency that it did not know. It did not tell me to use another currency but kept on asking to repeat my request.

It didn't understand Italian names

It seems not to recognize the singular (one dollar: ko, one dollars: ok).

Yes, when she asked if there was anything she could help me with further. I said no, but she said she could not help me with that

I asked to check all information about both bank account but it did not work

it does not reject my requests it only works in affirmation

Occasionally it would not quite get all the details of my commands, but that could also be due to the low volume of my voice compared to the surrounding noises.

The voice assistant has some difficulties understanding the names of the person I want to send money to.

Si, l'assistente non riconosceva alcuni contatti e inoltre dava diversi problemi a riconoscere la banca selezionata

it didn't really understand the names to which to send the money

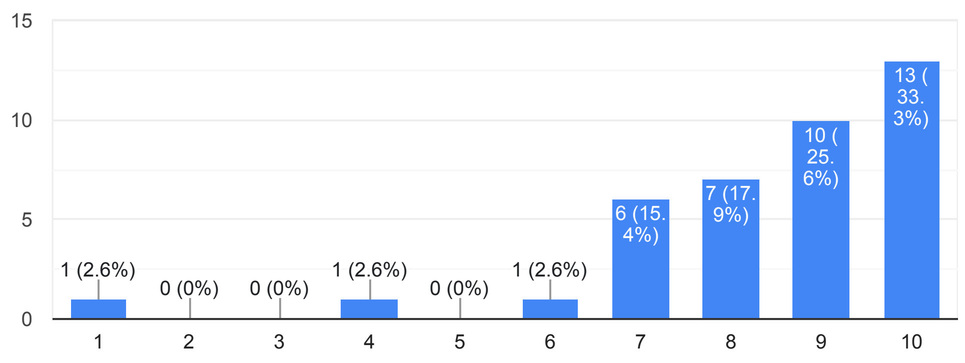
Yes, they didn’t understand the amount i wanted to send and kept looping

It did not understand me sometimes, and said so, but it only once misunderstood (as in thinking I was saying something I was not)

Functionality and Effectiveness

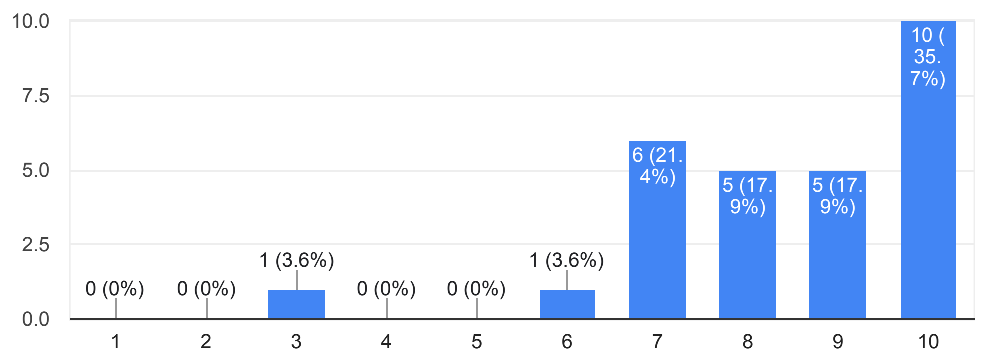
How effectively did the voice assistant assist you **checking your balance**? (if you have not tried it, skip it)

39 responses



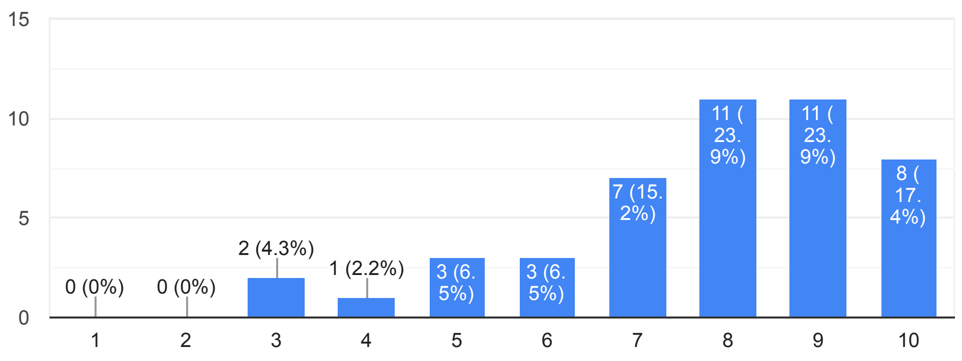
How effectively did the voice assistant assist you **checking your last transactions**? (if you have not tried it, skip it)

28 responses



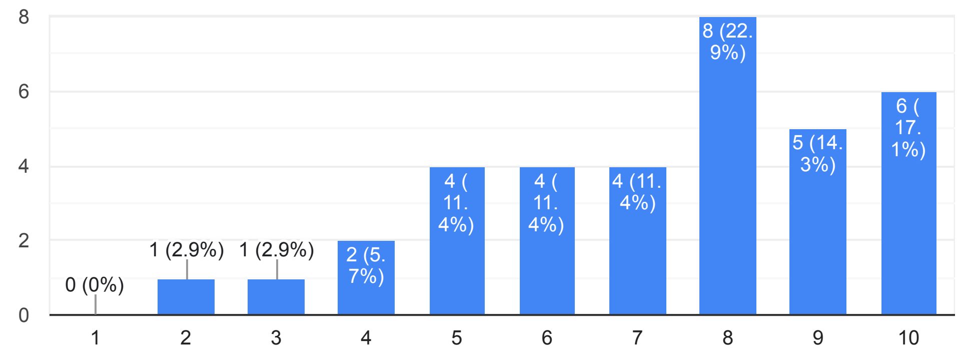
How effectively did the voice assistant assist you **sending some money**? (if you have not tried it, skip it)

46 responses



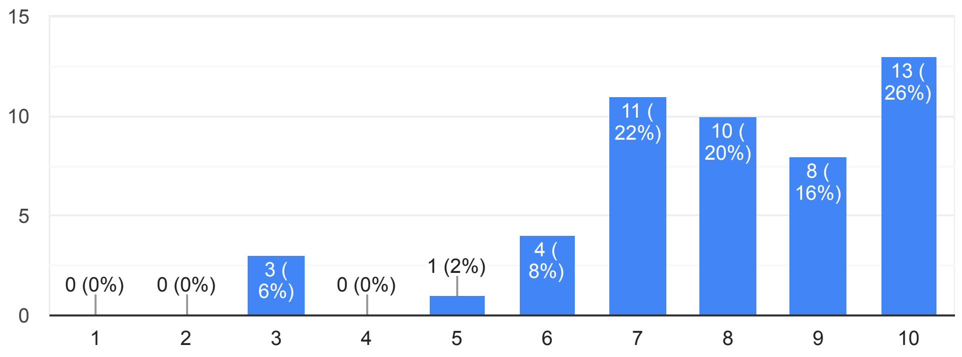
How effectively did the voice assistant assist you **requesting some money**? (if you have not tried it, skip it)

35 responses



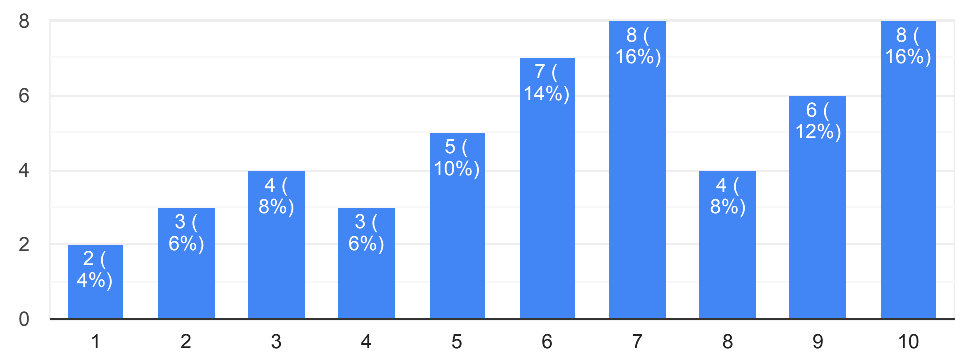
How effectively did the voice assistant understand an **amount** you specified?

50 responses



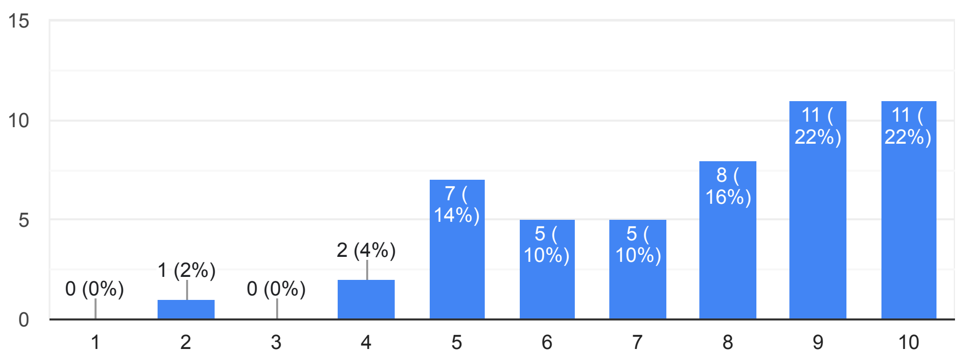
How effectively did the voice assistant understand a **name** you specified?

50 responses



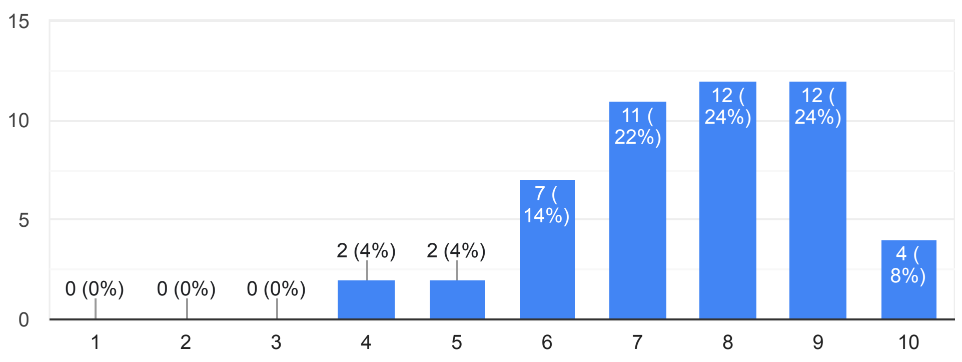
How effectively did the voice assistant understand a **bank account** you specified?

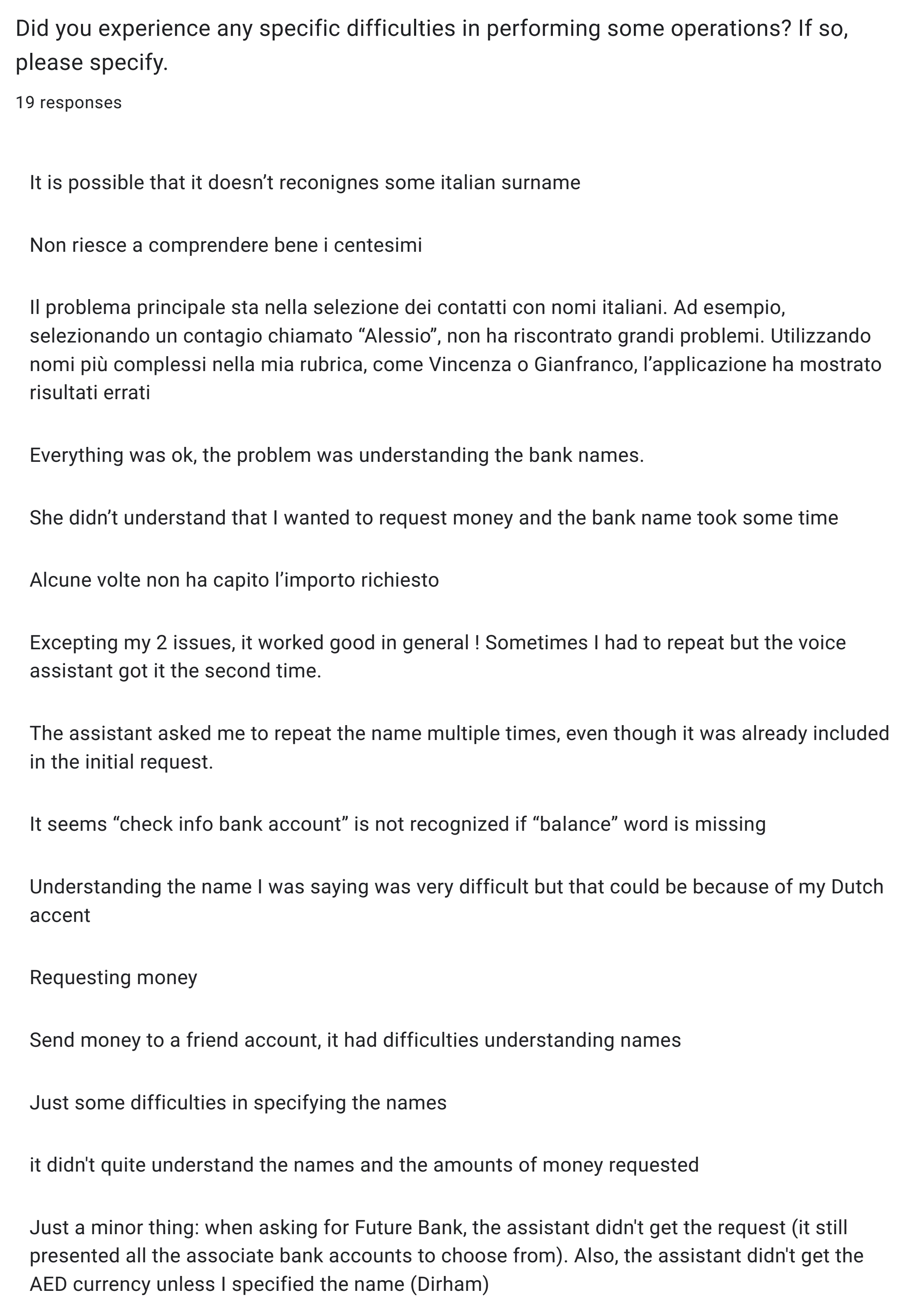
50 responses



How effectively did the voice assistant perform the requested operations **in general**?

50 responses





Did you experience any specific difficulties in performing some operations? If so, please specify.

19 responses

It is possible that it doesn’t reconignes some italian surname

Non riesce a comprendere bene i centesimi

Il problema principale sta nella selezione dei contatti con nomi italiani. Ad esempio, selezionando un contagio chiamato “Alessio”, non ha riscontrato grandi problemi. Utilizzando nomi più complessi nella mia rubrica, come Vincenza o Gianfranco, l’applicazione ha mostrato risultati errati

Everything was ok, the problem was understanding the bank names.

She didn’t understand that I wanted to request money and the bank name took some time

Alcune volte non ha capito l’importo richiesto

Excepting my 2 issues, it worked good in general ! Sometimes I had to repeat but the voice assistant got it the second time.

The assistant asked me to repeat the name multiple times, even though it was already included in the initial request.

It seems “check info bank account” is not recognized if “balance” word is missing

Understanding the name I was saying was very difficult but that could be because of my Dutch accent

Requesting money

Send money to a friend account, it had difficulties understanding names

Just some difficulties in specifying the names

it didn't quite understand the names and the amounts of money requested

Just a minor thing: when asking for Future Bank, the assistant didn't get the request (it still presented all the associate bank accounts to choose from). Also, the assistant didn't get the AED currency unless I specified the name (Dirham)

The assistant didn't understand currency

It didn't exit when ask " Is there anything else I can do for you?" and the answer is no

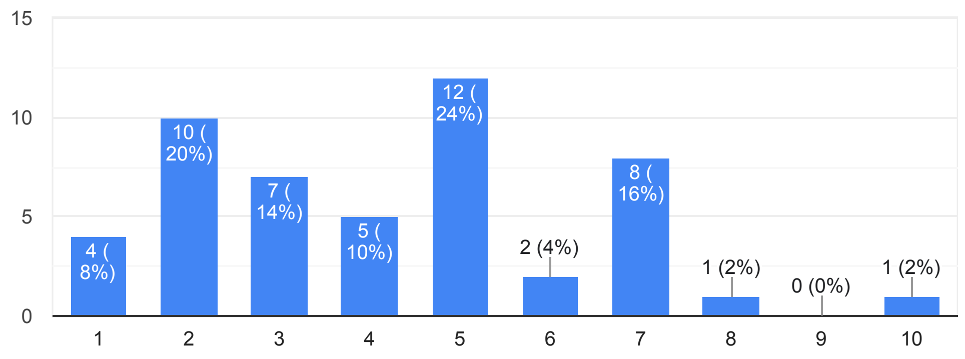
Most of the difficulty came from not understanding some names from my contacts, specifically the swedish names. Tried to pronouce them with an english accent which sometimes helped. It did get most of the names with similar pronounciation in swedish and english

I encountered some difficulties asking to send money to a contact. In particular if I say "send $100 to Mario Mastrandrea using Top Bank Account" it never worked. I tried multiple times and I always received some misunderstanding from the vocal assistant

Reliability and Error Handling

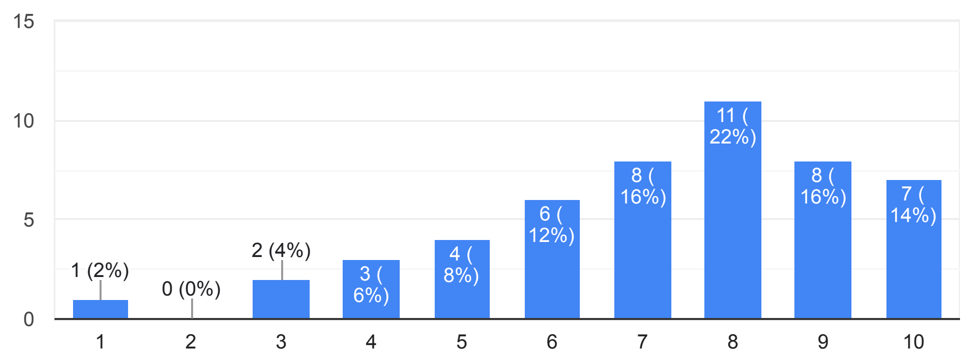
How often did the voice assistant misunderstand your requests or provide incorrect information?

50 responses



Were you satisfied with the voice assistant's ability to recover from errors or guide you on how to correct them?

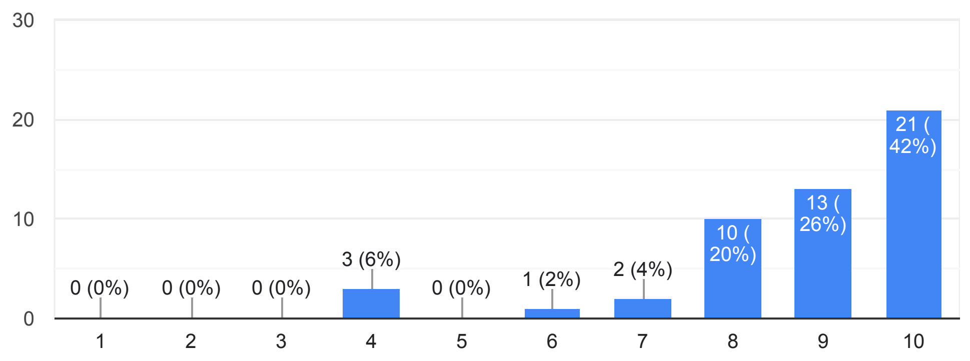
50 responses



Security and Privacy

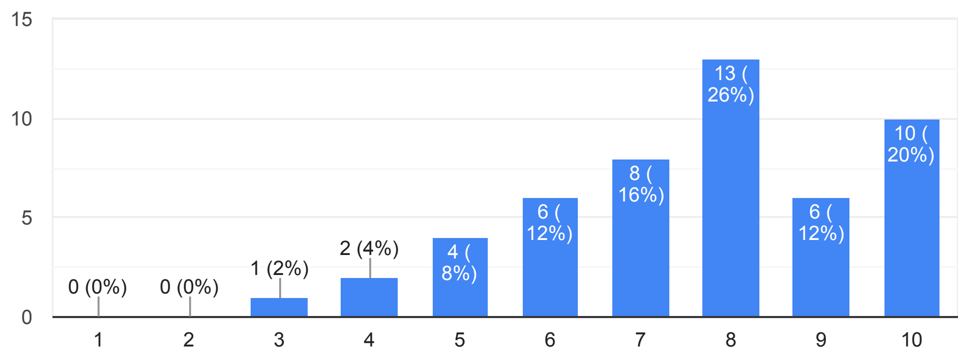
Were the voice assistant's prompts and responses clear in terms of any authorization or confirmation needed for transactions?

50 responses



How confident did you feel about the security and privacy of your financial transactions while using the voice assistant?

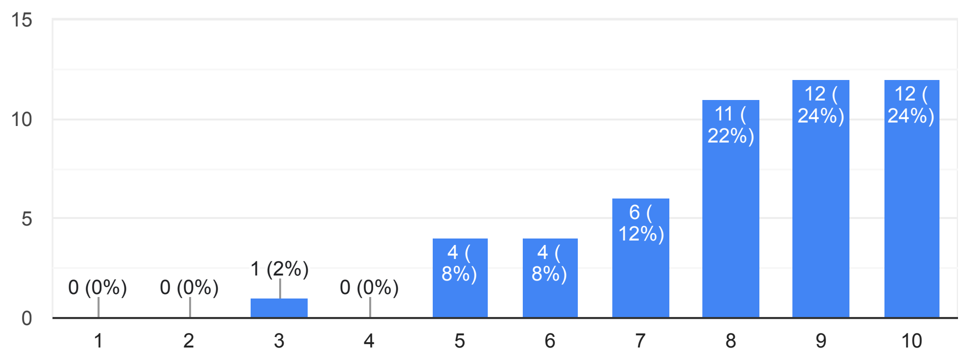
50 responses



User Experience and Satisfaction

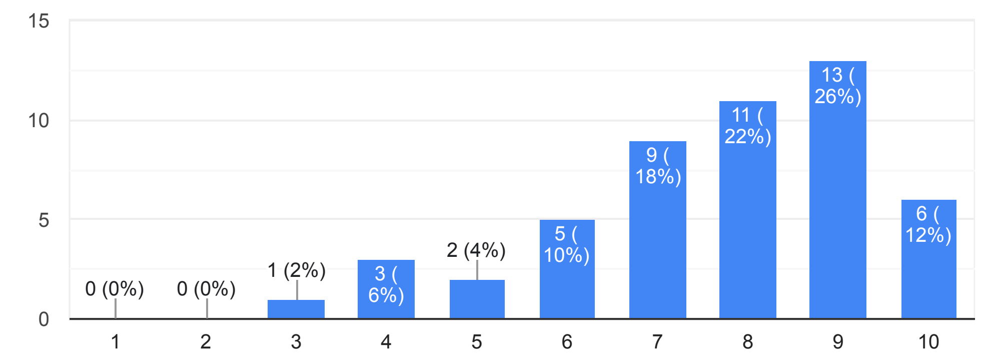
Was the voice assistant's voice pleasant and easy to understand?

50 responses



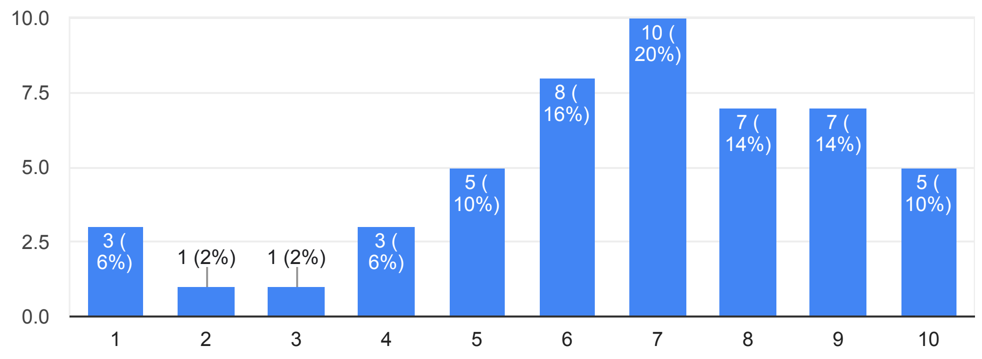
How satisfied are you with the overall experience of using the Voice Assistant?

50 responses



How likely would you be to use the voice assistant for future money operations in a real application?

50 responses



What features or capabilities would you like to see added to the voice assistant in the future?

20 responses

I would like to see a back option to go to the previous question manually

More interaction with other bottons

Nel momento in cui ha compreso l'utente con il quale scambiare denaro, mi aspetterei una richiesta di conferma dell'utente selezionato vocalmente e/o graficamente, e non la selezione dell'account bancario

Paying pending requests, set predefined recurrent payments with some "IDs" so that you don't need to explain every time how much you want to send/request to/from who.

It would be great if it was able to pay an order on a website

When I made a mistake of giving the wrong bank name (future bank) when I was sending in USD. It corrected me and told me the options according to currency. It would be good to have the same suggestions for the currency (this can also helps understand how to say the name of the currency in case people use a different name), also maybe for contacts, some suggestions based on recent sent, or similar names.

La possibilità di pianificare le transazioni. Usare contemporaneamente entrambi i conti per una transazione e quindi suddividere l'importo. Conoscere più accuratamente i movimenti dei conti (es. Quanto ho speso il 5gennaio?)

Features for online purchases

Use any bank account and convert in the desired currency

Maybe the possibility to redo an operation could be great, or just send a reminder in case of asking for money

- display the history of transactions in the last day/week/months
- transfer money form a bank account to an other

Maybe a bit more of a menu where you can choose easier

Split a expense between multiple people

Option to choose speech or not

To show whether this month the expenses were higher or lower than the previous month

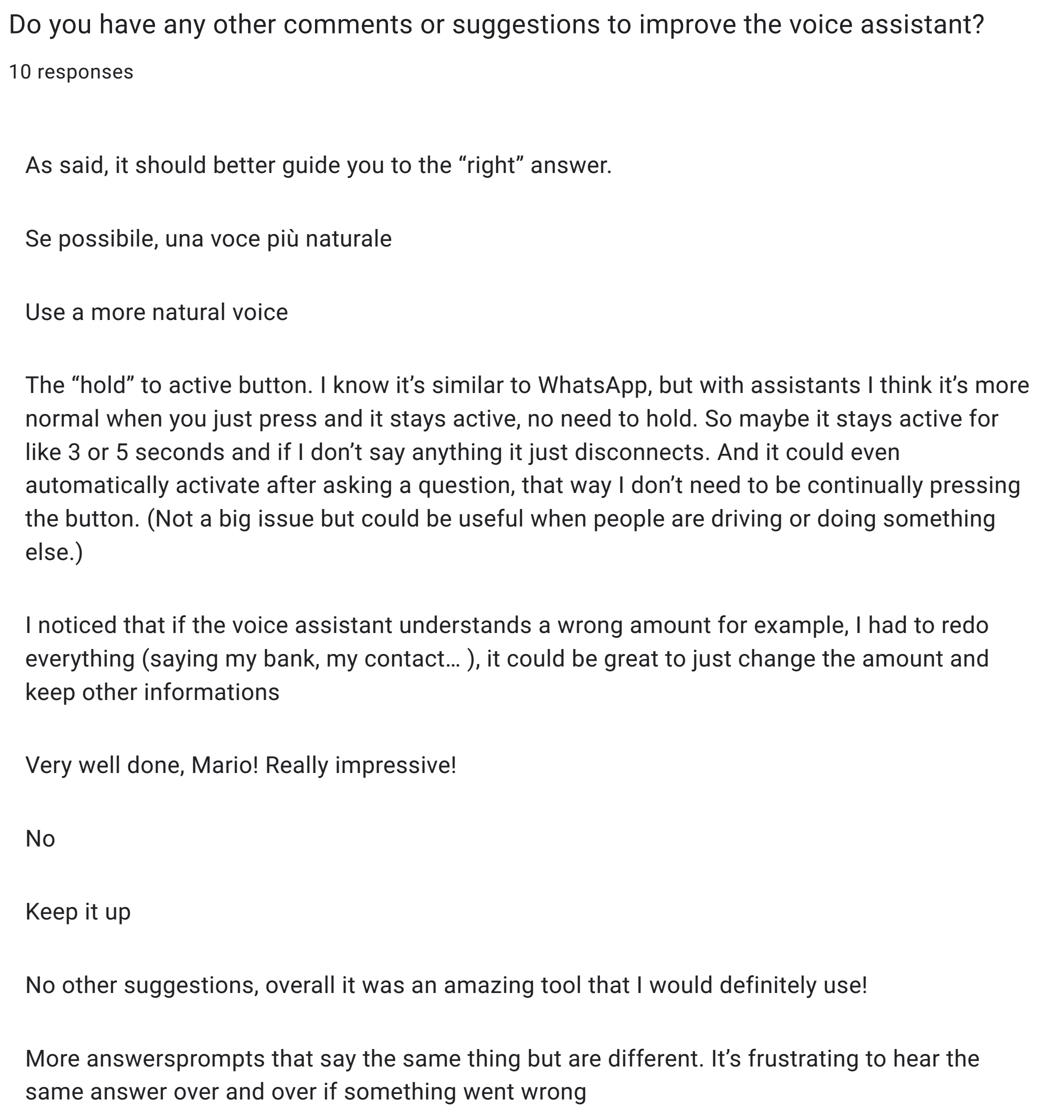
Voice recognition for safety purposes

I would like for the assistant to also support other currencies (e.g. Euro) and to be able to specify the amount in a smoother way (e.g. without pronouncing its sign)

Possibly calculations or splitting transactions

Understanding non-english names better

Instead of integrate the vocal assistant in the app, would be better to integrate the corresponding functionalities into Siri, avoiding the user to open the app and making faster the operations(e.g. Satispay)



Do you have any other comments or suggestions to improve the voice assistant?

10 responses

As said, it should better guide you to the “right” answer.

Se possibile, una voce più naturale

Use a more natural voice

The “hold” to active button. I know it’s similar to WhatsApp, but with assistants I think it’s more normal when you just press and it stays active, no need to hold. So maybe it stays active for like 3 or 5 seconds and if I don’t say anything it just disconnects. And it could even automatically activate after asking a question, that way I don’t need to be continually pressing the button. (Not a big issue but could be useful when people are driving or doing something else.)

I noticed that if the voice assistant understands a wrong amount for example, I had to redo everything (saying my bank, my contact…), it could be great to just change the amount and keep other informations

Very well done, Mario! Really impressive!

No

Keep it up

No other suggestions, overall it was an amazing tool that I would definitely use!

More answersprompts that say the same thing but are different. It’s frustrating to hear the same answer over and over if something went wrong

Bibliography

[1] 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 p. 8).

[2] Alan M. Turing. «Computing Machinery and Intelligence». In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/lix.236.433. URL: https://doi.org/10.1093/mind/lix.236.433 (cit. on p. 8).

[3] IBM. *(1961) Shoebox - IBM Archives (78-013).* URL: https://mediacenter.ibm.com/media/(1961)+Shoebox+-+IBM+Archives+(78-013)/0_4m2ynnkk (visited on 02/26/2024) (cit. on p. 8).

[4] Bruce T. Lowerre. «The Harpy speech recognition system». PhD thesis. Carnegie Mellon University, Pennsylvania, Apr. 1976 (cit. on p. 8).

[5] Apple. *Siri - Apple.* URL: https://www.apple.com/siri/ (visited on 02/26/2024) (cit. on p. 8).

[6] Amazon. *Amazon Alexa Voice AI.* URL: https://developer.amazon.com/en-US/alexa (visited on 02/26/2024) (cit. on p. 8).

[7] Google. *Google Assistant, your own personal Google.* URL: https://assistant.google.com (visited on 02/26/2024) (cit. on p. 8).

[8] Wikipedia. *Natural Language Processing - Wikipedia.* URL: https://en.wikipedia.org/wiki/Natural_language_processing (visited on 02/27/2024) (cit. on p. 9).

[9] CodeEmporium - YouTube. *Natural Language Processing | CodeEmporium - YT playlist.* URL: https://www.youtube.com/watch?v=LIRwZDEMn2o&list=PLTl9hO2Oobd_bzXUpzKMKA3liq2kj6LfE&pp=iAQB (visited on 02/27/2024) (cit. on p. 10).

[10] Wikipedia. *Speech recognition - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Speech_recognition` (visited on 02/27/2024) (cit. on p. 10).

[11] Wikipedia. *Speech synthesis - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Speech_synthesis` (visited on 02/27/2024) (cit. on p. 11).

[12] Wikipedia. *Natural language generation - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Natural_language_generation` (visited on 02/27/2024) (cit. on p. 12).

[13] Wikipedia. *Language model - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Language_model` (visited on 03/01/2024) (cit. on p. 13).

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is All you Need». In: *Advances in Neural Information Processing Systems.* Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Red Hook, NY, USA: Curran Associates, Inc., 2017, pp. 6000–6010. DOI: `10.5555/3295222.3295349`. URL: `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf` (cit. on p. 14).

[15] Jay Alammar. *The Illustrated Transformer.* URL: `https://jalammar.github.io/illustrated-transformer/` (visited on 03/02/2024) (cit. on p. 15).

[16] CodeEmporium. *Self Attention in Transformer Neural Networks (with code) - YouTube video.* URL: `https://www.youtube.com/watch?v=QCJQG4DuHT0&t=416s` (visited on 03/02/2024) (cit. on p. 16).

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: `10.18653/v1/N19-1423`. URL: `https://aclanthology.org/N19-1423` (cit. on p. 17).

[18] CodeEmporium. *BERT neural network - YouTube video.* URL: `https://www.youtube.com/watch?v=xI0HHN5XKDo` (visited on 03/03/2024) (cit. on p. 17).

[19] Jay Alammar. *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning).* URL: `https://jalammar.github.io/illustrated-bert/` (visited on 03/03/2024) (cit. on p. 18).

[20] Google. *google-research/bert: TensorFlow code and pre-trained models for BERT*. URL: `https://github.com/google-research/bert` (visited on 03/15/2024) (cit. on pp. 18, 58, 68).

[21] Yonghui Wu et al. «Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation». In: (Sept. 2016). arXiv: `1609.08144 [cs.CL]` (cit. on p. 19).

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: `1301.3781 [cs.CL]` (cit. on p. 19).

[23] Jeffrey Pennington, Richard Socher, and Christopher Manning. «GloVe: Global Vectors for Word Representation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: `10.3115/v1/D14-1162`. URL: `https://aclanthology.org/D14-1162` (cit. on p. 19).

[24] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *Well-Read Students Learn Better: On the Importance of Pre-training Compact Models*. 2019. arXiv: `1908.08962 [cs.CL]` (cit. on pp. 20, 58).

[25] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: `1909.11942 [cs.CL]` (cit. on p. 20).

[26] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. *MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices*. 2020. arXiv: `2004.02984 [cs.CL]` (cit. on p. 20).

[27] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2020. URL: `https://openreview.net/forum?id=SyxS0T4tvS` (cit. on p. 20).

[28] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: `1910.01108 [cs.CL]` (cit. on p. 20).

[29] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. «Improving Language Understanding by Generative Pre-Training». In: 2018. URL: `https://api.semanticscholar.org/CorpusID:49313245` (cit. on p. 20).

[30] Matthias Bastian. *GPT-4 has more than a trillion parameters - Report*. URL: `https://the-decoder.com/gpt-4-has-a-trillion-parameters/` (visited on 03/04/2024) (cit. on p. 21).

[31] Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. «GUS, a frame-driven dialog system». In: *Artificial Intelligence* 8.2 (1977), pp. 155–173. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/0004-3702(77)90018-2`. URL: `https://www.sciencedirect.com/science/article/pii/0004370277900182` (cit. on p. 23).

[32] Michael F. McTear. «Spoken dialogue technology: enabling the conversational user interface». In: *ACM Comput. Surv.* 34.1 (Mar. 2002), pp. 90–169. ISSN: 0360-0300. DOI: `10.1145/505282.505285`. URL: `https://doi.org/10.1145/505282.505285` (cit. on pp. 23, 32).

[33] Wikipedia. *iOS - Wikipedia*. URL: `https://en.wikipedia.org/wiki/IOS#Development` (visited on 03/05/2024) (cit. on p. 25).

[34] Apple. *Swift - Apple Developer*. URL: `https://developer.apple.com/swift/` (visited on 03/05/2024) (cit. on p. 25).

[35] Apple. *SwiftUI Overview - Xcode - Apple Developer*. URL: `https://developer.apple.com/xcode/swiftui/` (visited on 03/05/2024) (cit. on p. 26).

[36] Wikipedia. *Finite-state machine - Wikipedia*. URL: `https://en.wikipedia.org/wiki/Finite-state_machine` (visited on 03/10/2024) (cit. on p. 40).

[37] Apple. *Converting TensorFlow 2 BERT Transformer Models - Guide to CoreML Tools*. (Visited on 03/15/2024) (cit. on p. 61).

[38] Google. *Model optimization | TensorFlow Lite*. URL: `https://www.tensorflow.org/lite/performance/model_optimization` (visited on 03/15/2024) (cit. on p. 62).

[39] Google. *TensorFlow Lite - BERT QA iOS Example Application*. URL: `https://github.com/tensorflow/examples/tree/master/lite/examples/bert_qa/ios` (visited on 03/16/2024) (cit. on p. 68).

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: `http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1` (cit. on pp. 71, 81).

[41] Anupam Chugh. *iOS On-Device Speech Recognition | Medium blog.* URL: `https://betterprogramming.pub/ios-speech-recognition-on-device-e9a54a4468b5` (visited on 03/17/2024) (cit. on p. 76).

[42] Apple. *Customize on-device speech recognition - WWDC23 - Videos - Apple Developer.* URL: `https://developer.apple.com/videos/play/wwdc2023/10101/` (visited on 03/17/2024) (cit. on pp. 76, 77).

[43] Wikipedia. *SOLID - Wikipedia.* URL: `https://en.wikipedia.org/wiki/SOLID` (visited on 03/18/2024) (cit. on p. 82).

[44] Robert C. Martin. *Design Principles and Design Patterns.* Object Mentor, 2000 (cit. on p. 82).

[45] Wikipedia. *Dependency inversion principle - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Dependency_inversion_principle` (visited on 03/18/2024) (cit. on p. 82).

[46] Apple. *Apple Developer.* URL: `https://developer.apple.com` (visited on 03/22/2024) (cit. on p. 97).

[47] Wikipedia. *Likert scale - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Likert_scale` (visited on 03/22/2024) (cit. on p. 101).