



Institute of Communication Acoustics (IKA)
Ruhr-University Bochum

Embedded Multimedia (SS24)
Dr.-Ing. Wolfgang Theimer

Project Report

Pola Voice-Assistant



Gruppe 3

D.S. Viracachá Suarez, C.S. Gomez Mozo, P. Intek,
A.J. Marquez Maldera, M. Brosch, A. Aljeniyat

Bochum, 21.07.2024

Contents

1	Introduction	1
2	Hardware	1
2.1	Raspberry Pi	1
2.2	Remote Computer	2
2.3	Peripherals	2
3	Software Architecture	2
3.1	Consideration of System Features	2
3.1.1	Speech Recognition	4
3.1.2	Wake-Word Detection	4
3.1.3	Graphical User Interface (GUI)	5
3.1.4	System Integration	5
3.2	Inter-Process Communication	5
4	QT Framework	6
4.1	QT Quick vs. QT Widgets	6
4.2	Finite State Machine	7
4.2.1	Unused Approaches	9
4.3	Graphical User Interface	9
5	Wake-Word Detection	10
6	Audio Recording	11
6.1	Unused Approaches	11
7	Speech Recognition	12
7.1	Unused approaches	12
8	Large Language Model	12
9	Text to Speech	13
10	Project Management	14
11	Future Work	15
11.1	Enhanced Use of Ollama API	15
11.2	Usability Improvements	15
11.3	Error Handling Enhancements	15
11.4	Product Improvement	15
12	User Manual	17
13	Installation Script	20

Abstract

The following is a report about the development of a voice assistant. Instead of developing a software that can run on any powerful enough machine, the goal of our project is to create an embedded system on a Raspberry Pi which is a mini-computer. The report focuses on the challenges of coordinating software development and tailoring software to highly limited hardware resources.

1 Introduction

In the context of the module *Embedded Multimedia*, our group was tasked to develop an embedded system. We as a team had the freedom of choosing the feature set of our system, however constraints and requirements have been set at the beginning of the project. The embedded system must fulfill the following criteria:

- Run mainly on a Raspberry Pi mini-computer
- Use multimedia input and output devices
- Include network communication

Our team agreed on the feature set of a typical voice assistant (VA), such as Amazon *Alexa*, Apple *Siri* and Google's VA. To summarize the functionality of a VA, a machine, equipped with a microphone and a speaker, waits for you to say a *wake word* (see 5), then listens to your prompt and understands it using *speech recognition* (see 7). The machine then generates a response to your prompt using a *large language model* (see 8) and finishes the interaction with saying the response over the speaker using *text to speech* (see 9). An additional feature we chose to add is a *graphical user interface* (see 4.3) to improve the user experience.

The development of the individual sub-programs has been split up between the team members, the parts are later united into one overarching software structure. Since the sub-programs need to run sequentially, they are executed by a *finite state machine* (see 4.2).

2 Hardware

2.1 Raspberry Pi

The central node of the VA system is a Raspberry Pi 4B developed by the Raspberry Pi Foundation. It is a mini-computer with a system-on-chip (SoC) architecture which means all processing is done within one central chip. The Raspberry Pi (RPi) is able to run the graphical operating system Raspberry Pi OS which enables local software development. The computational specifications are the following:

- CPU: 1500 MHz, 4 cores
- GPU: 500 MHz, 2 cores
- RAM: 4096 MB LPDDR4
- Operation rate: 4.4 GFLOPS
- Communication: Ethernet, WiFi, Bluetooth

These properties are important constraints for the complexity of our software.

2.2 Remote Computer

A laptop computer was used as an HTTPS server for running Large Language Models. A router connection was established, and communication between laptop and Raspberry Pi occurs over LAN. This allows the use of LLMs that would be too big for the Raspberry Pi to run on its own due to RAM constraints.

The specifications of the laptop server are as follows:

- CPU: Intel® Core™ i7-5500U × 4
- GPU: Intel® HD Graphics 5500 (BDW GT2)
- RAM: 8192 MB
- Model: ASUSTeK COMPUTER INC. X555LB
- OS: Manjaro Linux 6.5.13-7 64-bit

2.3 Peripherals

We use a speaker as our output device and a microphone as our input device. The type of speaker does not affect the reproducibility of the project, whereas the type of microphone does. The microphone of our VA is a *FIFINE K669* condenser mono-microphone, it has a digital output and a USB connection. Furthermore it has a knob for adjusting the output volume level (not the input gain) which is shown to be a useful feature.

3 Software Architecture

3.1 Consideration of System Features

The design of the Pola Voice-Assistant's software architecture is centered around a modular and extensible framework, as illustrated in Figure 1. This approach ensures that the system can efficiently manage various components and processes involved in delivering a seamless user experience. The main components and their interactions and their priority are as follows:

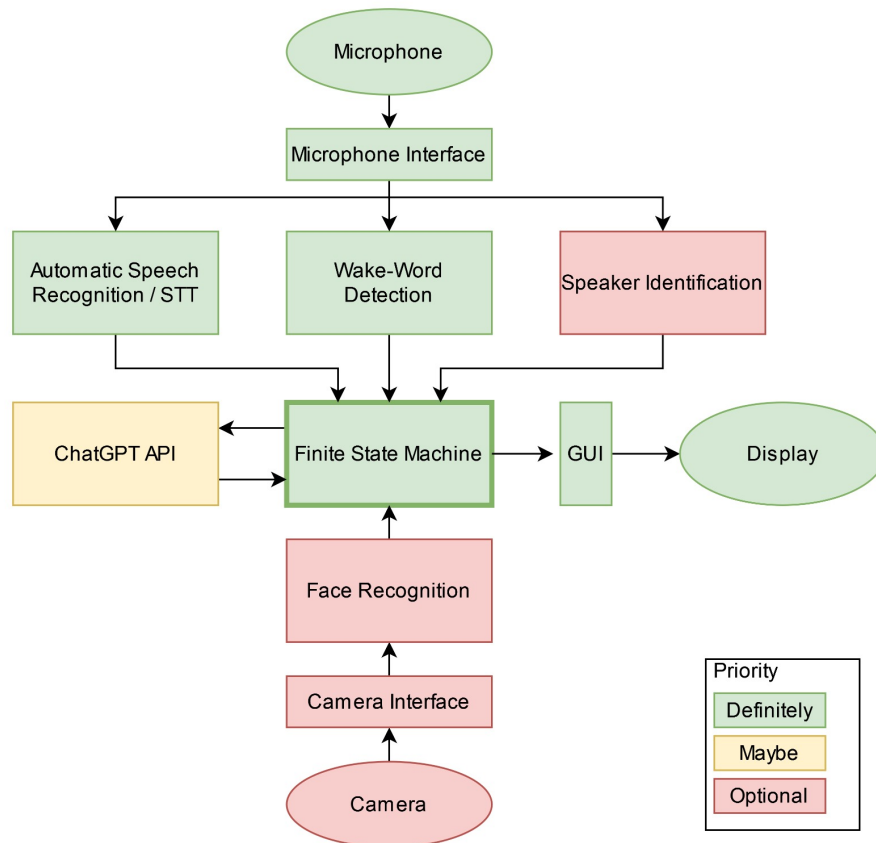


Figure 1: System Architecture Overview

- **Microphone and Microphone interface:**

The system continuously listens for voice input through the microphone. The microphone interface ensures that the audio signals are correctly captured and processed, and the Microphone interface is with multiple system component connected.

- **Wake-Word Detection:**

This module remains active to detect the predefined wake-word ("Hey Pola"). Once the wake-word is detected, the system transitions to a state where it can process further voice commands.

- **Automatic Speech Recognition (ASR) / Speech-to-Text (STT):**

After detecting the wake-word, the ASR module converts spoken language into text. This text is then used as input for further processing.

- **Finite State Machine (FSM):**

The core of the system is managed by a finite state machine that orchestrates the transitions between different states based on events and inputs from other modules. It ensures that the system operates in a logical sequence from wake-word detection to processing and responding to user commands.

- **ChatGPT API:**

The text obtained from the ASR module is sent to the ChatGPT API for generating appropriate responses. The FSM manages the communication with the API and handles the received responses.

- **Graphical User Interface (GUI):**

The GUI provides visual feedback to the user, displaying the current state of the system, the recognized text, and the system's responses. It enhances user interaction by providing intuitive visual cues.

- **Display:**

The output from the GUI is rendered on a display, ensuring that users can see the assistant's responses and other relevant information like the state of the Voice-Assistant.

- **Speaker Identification (Optional):**

For enhanced security and personalized interactions, the system can optionally include speaker identification to recognize different users based on their voice.

- **Camera and Camera Interface (Optional):**

The system architecture allows for the integration of a camera. The camera interface processes video input, which can be used for additional features like face recognition.

- **Face Recognition (Optional):** To further personalize the user experience and improve security, face recognition can be implemented. This module identifies users based on their facial features and can tailor responses or grant access accordingly.

The priority of these features is indicated in Figure 1 by different colors. Green indicates essential features that should definitely be implemented, yellow indicates features that might be implemented, and red indicates optional features that could be added for enhanced functionality.

By organizing the system in this manner, we ensure a flexible and scalable architecture that can be easily extended with new features as needed.

3.1.1 Speech Recognition

Philipp Intek and César Sebastián Gomez Mozo focused on the speech recognition module. They made significant progress in developing and testing the speech-to-text functionality. Their work involved:

- Implementing and optimizing the speech recognition algorithm.
- Conducting tests to ensure accurate transcription of spoken words.
- Planning the integration of this module with other components, particularly how the recognized text would interact with the finite state machine and the ChatGPT API.

3.1.2 Wake-Word Detection

David Santiago Viracachá Suarez and Aldo Jose Marquez Malder on the wake-word detection module. Their tasks included:

- Exploring different strategies for detecting the wake-word "Hey Pola."
- Ensuring that the microphone operates continuously and reliably throughout the system's runtime.
- Integrating the wake-word detection with the microphone interface and ensuring seamless transitions to subsequent states in the finite state machine.

3.1.3 Graphical User Interface (GUI)

Ahmad Aljeniyat and Maurice Brosch were responsible for designing and implementing the GUI. Their work included:

- Sketching the initial design of the GUI, considering the various states of the system.
- Defining how the GUI would visually represent different states and events, such as standby mode, listening mode, and converting speech to text.
- Ensuring the GUI provides clear and intuitive feedback to users, enhancing the overall user experience.

3.1.4 System Integration

The finite state machine, which orchestrates the transitions between different system states, was a collaborative effort. Each module, developed by different team members, was integrated into this overarching structure. This approach ensured that each component could operate independently while contributing to the system's functionality as a whole.

By distributing tasks in this manner, the team could leverage individual strengths and work more efficiently, resulting in a well-coordinated development process.

3.2 Inter-Process Communication

To achieve a simple and reliable forwarding of data between the processes, we use a file-based inter-process communication. This means one program writes its output into a file on the disk and the next program (which needs the output from the previous) reads this file and writes its output to the disk.

The disk in the RPi is a micro-SD card which is a type of Flash-drive. To prevent *Flash-wear-out*, meaning the physical degradation of the drive due to overly repeated writing, we established a *ramdisk* directory. Setting up a ramdisk means telling the computers operating system to treat a block of RAM as if it was part of the disk. This is useful for applications, where the disk is accessed a lot and the files are disposable (since they are lost once the power is off). It also improves the speed at which data can be read and written. We allocated 4MiB of the RAM to be treated as disk for our file-based inter-process communication.

4 QT Framework

Since one goal of our voice assistant is usability and accessibility, we chose to develop a Graphical User Interface (GUI), such that the user is able to not only hear the answers to the questions made, but also see the current state of the program, the transcribed voice input and the answers of the program. This enables users to check if the program actually transcribed voice inputs correctly and it also allows them to read the answers of the assistant. This can be useful in cases such as longer or complicated answers that the user might want to re-read or when users are not able to hear the answer due to environment noises or hearing disabilities. Therefore a GUI potentially might help to reach a broader potential user base and make the product more inclusive.

The Graphical User Interface was developed within the open-source Qt Framework. The Qt Framework allows it to design GUIs on many different hardware and software platforms as native applications and therefore native speed and properties.

4.1 QT Quick vs. QT Widgets

When starting the development of the GUI within the Qt Framework, one of the first things we had to do, was to choose between Qt Widget and Qt Quick.

When using Qt Widgets most parts are programmed in C++ or Python and Qt Style Sheets (QSS) can be used aswell, which are pretty similar to Cascading Style Sheets (CSS), for the styling of the widgets.

Its advantages include, that on desktop devices the widgets usually have a native look and that the use of a direct rendering approach can be helpful when developing desktop applications, such as a screen reader, that uses the Windows API to recognize elements such as windows and buttons. With that approach the code does not have to be modified for the elements to be recognized [1]. However designing more complex custom UIs with custom stylings, that might also include animations or effects can be quite difficult in Qt Widgets.

Qt Quick applications on the other hand can also be programmed in C++ and Python, but it provides a declarative scripting language called QML. This allows the developer to program the logic of the program in C++ and Python and the visual design can be written in QML. On top of that the logic can even be written in JavaScript, making the development of GUIs easier for Web Developers, that often do not have as much experience in C++ and Python compared to JavaScript.

While Qt Widgets mostly provides a native look for desktop applications, Qt Quick also allows applications to look native on mobile devices, that run on operating systems such as Android or IOS. On top of that lightweight versions of Qt for Microcontrollers (MCU) are available for Qt Quick. Moreover Qt Quick also simplifies the implementation of touchscreen applications.

We decided to chose Qt Quick over Qt Widgets because we want our software to be as compatible as possible with embedded devices. Also this allows us to reuse the GUI in case we want to develop an app for smartphones that connects to our voice assistant. On top of that when planning the project, we knew that we might want to develop a richer UI with animations and these are easier to implement with QT Quick.

4.2 Finite State Machine

The structure of the project follows the design pattern of a state machine, where each state handle a major task of the system and can transition to another one depending on the events that occur.

This approach allows having a robust flow of events while maintaining a clear separation of concerns between the established software modules, thus making the development process flexible and easily upgradeable.

For the implementation of the state machine, the `QtStateMachine` library was used. Upon execution, a `QStateMachine` object is created and 5 `QState` instances are assigned to it. The use of every state is explained as follows:

- **Wake-Word Detection State:** This is the initial and idle state. It runs the Wake-Word Detection executable until the user says the keyword
- **Recording State:** In this state the audio recording executable captures the audio until a continuous second of silence is detected. Then, it writes the audio into a WAV file
- **Speech-to-Text State:** Here, the transcription executable which uses `FasterWhisper` (See 7) is used to transcribe the contents of the previous recording and writes it into a TXT file
- **Generative pre-trained transformer (GPT) Communication State:** In this state a python script is used to communicate with another machine in the local network that runs the petition read from the TXT through the Ollama API. When the answer of the Large Language Model (LLM) is finished, the answer is then sent back to the Raspberry and saved into a TXT file
- **Text-to-Speech State:** This state runs the Text-to-Speech executable with the TXT answer file as its input, so it gets read out loud to the user

Now, with the function of the states clear, it is necessary to establish an execution logic. For this we make use of the signal and slot mechanism in Qt. In Qt it is possible to emit a signal after a certain event has happened, such as when the value of property changed or when our state machine has entered a new state. These signals can then be connected to slots or signal handlers which can then for example perform some function or change the appearance of some element in the graphical user interface.

Our execution logic is realized by the means of a controller that reads the standard output of the Raspberry every millisecond in order to determine when a process has finished and, subsequently, sends the corresponding signal of the event. The implemented signals are the following:

- **keywordDetected:** Emitted when "kw" is read from the standard output. That is, when the wake-word detection program runs and the sound of saying "Hey Pola" is detected
- **recordingDone:** Emitted when "rec" is read. It signalizes that the WAV file has been successfully created
- **transcriptionDone:** Emitted when "tr" is read. This signal is produced only if the transcription is done correctly (when no error was detected)
- **stateError:** Emitted when "error" is read. This happens during the transcription process when an exception is caught
- **llamaResponse:** Emitted when "ask" is read. It acknowledges that an answer from the Ollama API has been received, and it gets written into a TXT file
- **textSound:** Emitted when "talk" is read. It is written when the Ollama model's answer has been successfully written into a TXT file

Finally, transitions are added to each state, so that the signals can trigger the move from one to another state, thus executing its corresponding task only when the expected outcome occurs. The transitions were assigned as showed in Table 1.

Initial State	Received Signal	End State
Wake-Word Detection	keywordDetected	Recording
Recording	recordingDone	Speech-to-Text
Speech-to-Text	transcriptionDone	GPT Communication
Speech-to-Text	stateError	Speech-to-Text
GPT Communication	llamaResponse	Text-to-Speech
Text-to-Speech	textSound	Wake-Word Detection

Table 1: Transitions of the State Machine

These assignments make the system run in a cycle triggered by the Wake-Word detection. It is important to highlight that the use of the **FasterWhisper** model sometimes leads to unexpected, non-deterministic errors that can be solved simply by running it again. Because of this, it was decided that, instead of starting the interaction from the beginning, the error would just execute the transcription program again, since the audio is already collected in the WAV file. This approach proved to be successful despite its oddity, and also better from the point of view of the user, so it was left as is up to this point. The execution sequence can then be represented by the Fig.2.

In general, having this structure for the project enables future improvements like making the wake word detection work in a separate thread or having a broader, more specific error handling depending on the current state without sacrificing on stability or robustness of the system.

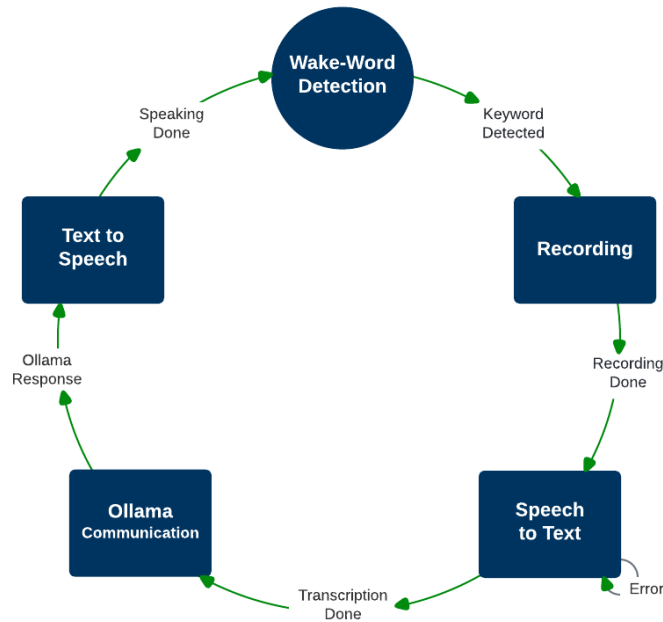


Figure 2: State Machine execution cycle

4.2.1 Unused Approaches

Direct Event Handling without State Machine: One approach considered was direct event handling without using a state machine. This would have meant that each system component implemented its own state transitions and event handling. Although this could have simplified the initial implementation, it was discarded because it would have led to a more complex and less maintainable codebase. The use of a central state machine allowed for clearer separation of responsibilities and easier integration of different modules.

Centralized Error Handling: Another unused approach was centralized error handling for the entire system. This would have involved a single module managing all error states and taking appropriate actions. While this could have provided consistent error handling, it was discarded because it would have reduced the modularity and flexibility of the system. Instead, it was decided that each module would implement its own error handling mechanisms to ensure better isolation and fault tolerance.

By evaluating and discarding these unused approaches, we ensured that the chosen architecture of the Pola Voice-Assistant is both efficient and maintainable while meeting the project's requirements.

4.3 Graphical User Interface

For the graphical user interface we identified the following requirements:

- information on the current state of the program should be displayed
- users should be able to see what is currently expected from them
- a visual representation of the conversation with the Chatbot should be displayed

All visual aspects of the graphical user interface were realized in QML, while the logic is implemented by the Controller that has been described in section 4.2. The GUI interacts with the Controller by implemented signal handlers, that change the appearance of the GUI according to events that are emitted by the controller.

The full design during each state of the program can be seen in section 12.

5 Wake-Word Detection

For a voice assistant, it is expected that it can be controlled through voice at any moment in time. This brings the need of implementing a wake-word detection module that initiates the interaction between the user and the assistant. While discussing the project, the following requirements were agreed for this part of the project:

- When turned on, the system must always have an activated microphone capable of recording human voices
- With the system turned on and in a standby state, the device must signalize that it heard the specified wake-word detection
- The information recorded by the microphone must be stored in a circular buffer, so that it uses memory effectively
- If the interaction with the human has been flagged as finished by the system, the wake-word detection system must restart

And they will be subject to the following delimitations:

- The wake word detection system must detect the keyword before passing microphone access to other systems
- The processing related to the wake word detection system must be performed in an offline fashion
- The recognition of the wake-word must be done in under 1 second

In order to fulfill these, two main means of solution were discussed. First, the implementation of an audio feature extraction algorithm that takes audio samples, calculates its spectrogram using the DFT and checks for the necessary features to detect the wanted keyword. This approach would also require determining those features, potentially by analyzing various samples of the keyword. Second, the use of a machine learning based model that works directly with the time-series signal to identify the keyword. Both approaches were most probably sufficient to satisfy the established requirements, however, it was opted to use the second one, since it was more flexible in case of wanting to add further voice commands through other keywords. Moreover, it also has the potential of being computationally less complex depending on the selected model.

Having this in mind, a search for possibilities to implement this was made, and a library named Porcupine by the company Picovoice stood out, because of their focus on low-spec devices such as the Raspberry Pi, and the ability to easily generate models for new keywords. Finally, this library was included in a Qt Program that starts the microphone as an audio device, records the audio in a circular buffer and detects whether the keyword has been said, successfully satisfying the required demands.

6 Audio Recording

After the wake word is detected, the VA is required to obtain the user prompt. The prompt is provided acoustically and in order to be processed, it needs to be recorded and stored. This task is handled by the `record.exe` program. If executed, the program starts to record the (default) microphone output by storing successive frames into a buffer. This is done using the Qt libraries `<QAudioSource>` and `<QBuffer>`. Every time the input device emits the signal `readyRead`, the data (frame) is added to the buffer which occurs every 50 milliseconds. The chosen sample rate is 16 kHz, because this is the sample rate our automatic speech recognition (see 7) requires.

A different problem of recording the prompt is deciding, when to end it. Setting a fixed recording duration is a bad solution regarding user experience. The recording needs to end, when the user finishes speaking. Luckily, the way the recording works gives us the opportunity to analyze the most recent 50 ms (or 800 samples at 16 kHz) of the input (in practice the frame length can sometimes be 100 ms or 0 ms). We can therefore check, whether the *power* P of a frame is above or below a silence-level-threshold.

The power can be computed as the mean energy

$$P = \frac{1}{K_{\text{frame}}} \sum_{k'=0}^{K_{\text{frame}}-1} x^2[k - k'], \quad (1)$$

where K_{frame} is the number of samples in a frame (mostly 800) and $x[k]$ is the digital microphone signal at the k -th sample since the beginning of the recording.

The decision to end the recording depends on two parameters, the `silence_level_threshold` P_{th} and the `silence_duration_threshold` t_{th} (between 1 s and 2 s).

Stopping-rule: If uninterrupted silence ($P < P_{\text{th}}$) occurs for a duration of t_{th} , the recording is ended.

However, two exceptions to this rule are established to improve user experience:

- If speech did not start yet, the rule does not apply
- If the interruption is not longer than two frames, it counts as uninterrupted

The first exception allows the user to start speaking whenever they want and the second exceptions prevents environmental sounds from unnecessarily continuing the recording.

The silence-level-threshold is chosen, such that the stop-condition works well in our test environment and it is not adjustable, however, adjusting the volume knob on the microphone changes the threshold, so that it still works in a noisy environment.

The next step is to save the entire recording in a way, that other programs can access it. We write the audio onto the ramdisk (see 3.2) using the WAV format.

6.1 Unused Approaches

The first idea for a stop-condition was similar to the final approach, but using a different formula. We tried to use the envelope of the speech signal, computed using the recursive low-pass filter

$$y[k] = \alpha y[k - 1] + (1 - \alpha)x^2[k], \quad (2)$$

from [2], where $\alpha \approx 0.998$ is a heuristic value for speech sampled at 16 kHz. This envelope was supposed to be used to set up a threshold-based stop-condition. However the stopping sensitivity was extremely sensitive to the value of α and it was not viable for different environmental noise levels. In addition, since we obtain the signal in frames anyway, we do not have to rely on a real time filter and can simply compute the mean energy of each frame, giving us one less parameter to worry about.

7 Speech Recognition

Reading the audio input and transcribing it into text is one vital function in Pola system. Transcription, also known as automatic speech recognition (ASR), is a topic that has been tackled since 1952, some approaches are model based, focused on locate the formants in a power spectrum, using hidden Markov models (HMM) and creating a bank of possible words that the system can handle. Since 1990 with the appearance of autoencoders, Deep Learning (DL) has been a preferred method to fulfill this task. Although it is a really researched topic, there is not a huge variety of open-source ASR software. Whisper, Kaldi, Julius and CMU Sphinx were taken in consideration but the decision of Whisper was made due to its multilingual supported language.

First, implementation of the Whisper model that reads the recorded audio input and returns the transcription was done. Even though the transcription was highly accurate, long processing times were observed (> 10 seconds), so here we meet another hardware limitation. After researching, a recently released model library that could fit our requirements was found. This is a newer version of Whisper called Faster-Whisper. It is an improvement over Whisper, being implemented on a lower level, leading to better performance. Faster-Whisper offers differently sized pre-trained models. They are **tiny** (~ 1 GB), **base** (~ 2 GB), **small** (~ 5 GB), **medium** (~ 13 GB) and **large** (~ 26 GB). Since we require the transcription to run locally on the RPi, we have to consider memory occupancy. The RPi has 4096 MB RAM, leaving only the **tiny** and **base** models as our options. Both run in a satisfactory time of roughly 1s/2s for initialization and 5s/10s for transcription respectively. Since the **base** model has better accuracy, we used it in the final program.

In the scheme of the project, the transcription is performed by the file `transcribe.py` which loads the model, reads the WAV file written by the audio recording step and transcribes it. The final transcription is written to a TXT file on the ramdisk.

7.1 Unused approaches

During the speech recognition development other approaches were also implemented. First implementation was an Streaming Transcription. Here there was no WAV file nor final TXT file created. With this approach we were able to transcribe the input given by the user in real time. Even though it was a good approach because of the fast processing, it was not clear when was the start and the end of the user command. Without this initial and final points there was no clear implementation of the submodule to the Finite State Machine.

8 Large Language Model

LLMs are computational models able to achieve natural language processing. In the last 5 years, LLMs have attracted a huge amount of interest due to its wide area of applications in medicine, education, science, maths and even law. The reason these models play such an important role in these different areas is the role of facilitate communication and self-expression for humans and their interaction with machines. Introducing a LLM into Pola allows the system to process any kind of instruction given by the user. Aside that, the system answer would be able to answer in a more "natural" and easy to comprehend way.

During the conception of the project, Chat-GPT(from OpenAI) and Ollama (from Meta) were taken into account. Initially, a Chat-GPT connection via internet was planned. After some time working on this approach, the decision to swap to Ollama was taken. This due to the option to install the Ollama model in a remote (but still in the same local network) device.

Ollama is a lightweight framework that allows running language models on a local machine. First, the Ollama framework was installed within a Docker container in a laptop. In this laptop the Ollama framework and with it a HTTPS server is created. In this server the Gemma:2B model is set up to run. Gemma is a text-to-text, decoder-only large language model. This model is preferred

for the application due to its relative small size that allows it to be deployable in any laptop.

During the creation phase, the Ollama framework was installed in the same Raspberry, this to create a direct connection within the QT project (state machine) and the LLM submodule. Even though this implementation worked fine and was usable, the response time of the submodule was high in comparison to what would be expected in a voice assistant, therefore, not fit for our project. For this reason, the Ollama framework was decided to run totally in a remote desktop and be connected with the Pola system via Wi-Fi. This wireless communication approach allows the Raspberry to delegate computing resources to another machine and reduce response times of the submodule.

The LLM submodule is then in charge of reading the transcript file given in the previous state and send the prompt in a HTTPS request to the Ollama server. After this request is done, the submodule is expected to receive an answer which is then saved in a TXT file that can be read by any submodule that needs it.

9 Text to Speech

The simple, yet important feature of text to speech was implemented in order to bring the user closer to the assistant and enhance their experience. It allows people with vision impairment to interact with the device, serving as a much-needed accessibility feature, and also liberates all users from the constant need to focus on the screen.

The Text to Speech module had the following requirements:

- The module must receive a text file and be able to read all contents inside of it
- Contents must be read naturally similar to everyday speech
- There must be no long pauses or breaks in speech





































The Text to Speech behavior was handled by the Text to Speech engine “Speech Dispatcher”. This is a versatile framework that provides an interface between applications and speech synthesizers. However, some complications were encountered, as the engine wouldn’t run on the QT framework. For this reason, a workaround was needed. It was decided to run speech dispatcher through the linux console, as it was completely functional over this method, and it was thus handled as an external application to the program.

The limitations of the console run Speech Dispatcher constrained the number of spoken characters to 75. This severely influenced the Text to Speech module’s design, as it forced the processing of the input text in such a way that it could be completely read out by the framework. This meant separating the input text into strings of around 75 characters, approximating to the nearest space in case the string ended in the middle of a word. Furthermore, a pause would have to be introduced between each string read-out, so as to avoid the VA interrupting itself mid-speech. Since the speech synthesizer is still of a rudimentary level, it reads out every syllable in a monotone way, taking approximately the same time for each line. This guarantees that the read-out time of every text string is constant. It then was determined that the average duration of speech was close to 4.9 seconds, and a corresponding pause was introduced. This allows for a more natural, continuous reading of the text input.

10 Project Management

The development of the Pola Voice-Assistant was dividing among the team members based on their strengths and interests, ensuring an efficient and collaborative approach. Each member focused on specific components, which were later integrated into the overall system managed by the finite state machine.

Here is the outline of the distribution of tasks:

Tasks	Conception	Development	Qt-Integration	Testing
Wake-Word	 	 		
Voice Recording	 	 		
Speech recognition	 	 		
Large Language Model		 		
GUI	 	 		
Finite State Machine	 	 	 	







 Aldo Marquez
 David Viracacha
 Phillip Intek
 Sebastian Gomez
 Maurice Brosch
 Ahmad Aljeniyat

Figure 3: Member distribution along the project

During the project creation 4 main stages were done. Conception, Development, Qt-Integration and Testing. In the Conception stage, the team members are responsible for searching possible solutions that could fit the desired requirements of the submodule. Different approaches are brought into discussion and one solution is decided. In the Development stage each sub group implements the selected solution individually, here each submodule is thought, design and tested as unity, without communications with the other submodules. The integration of all this blocks are done then in the Qt-Integration stage, here the submodule has to be implemented as a Qt project that creates an executable and can be called externally by any programm (in our case, the FSM). In the last stage the required tests are done.

Due to the selected software architecture any error found is then solved within the same submodule without affecting other modules on the system.

In the member distribution there is a division mainly by pairs where every couple is delegated a specific task. After a certain advance of this task, a mixture of the subteams is done. In this mixture one member is moved to another subsystem. This combination is made with the purpose of exchanging knowledge, get a better view of how every module works and bring new ideas to each task.

11 Future Work

In the development of the Pola Voice-Assistant, significant progress was made in creating a functional embedded voice assistant. However, there are numerous opportunities for future improvements and enhancements to make the system more robust, user-friendly, and versatile. The following sections outline key areas for future work:

11.1 Enhanced Use of Ollama API

- **Contextual Conversations:**
Improve the interaction with the Ollama API by leveraging in-conversation context and system-wide prompts. This will enable the voice assistant to maintain context over multiple exchanges, leading to more coherent and relevant responses.
- **API Optimization:**
Refine the API calls to reduce latency and improve response times, ensuring a smoother user experience.

11.2 Usability Improvements

- **Standardized Installation Process:**
Develop a standardized and simplified installation process to enhance the ease of setting up the system. This could involve creating installation scripts or packages that automate the configuration of the software.
- **Voice-Only Setup:**
Implement a setup process that can be completed entirely through voice commands, making it more accessible to users with varying levels of technical expertise.

11.3 Error Handling Enhancements

- **Logging Capabilities:**
Introduce comprehensive logging mechanisms to track system performance and errors. Logs should capture detailed information to facilitate troubleshooting and debugging.
- **Error States and Explanations:**
Develop a dedicated error state within the system. When an error occurs, the system should provide a clear and understandable explanation to the user, possibly with suggestions for resolving the issue.

11.4 Product Improvement

- **End-of-Speech Condition:**
Decide, when to stop the speech recording based on the volume of speech and not the total volume. This could be realized with a filter that filters speech frequencies.
- **Multiple Keywords:**
Enable the system to recognize and respond to different keywords for various functions. For instance, using "Home Assistant" to provide sensor data and other home automation tasks, in addition to the main "Chatbot" function.
- **Speaker Recognition:**
Implement speaker recognition features to enhance security and personalization. This could involve using facial recognition or voice recognition to verify users and tailor responses based on individual profiles.
- **GUI Enhancements:**

- **Microphone Detection:**
Add visual indicators in the GUI to show whether the microphone is active and detected.
- **Input Volume Visualization:**
Implement visual feedback for input volume to help users understand if they are speaking at an appropriate volume.
- **Design Improvements:**
Enhance the visual appeal of the GUI with improved fonts, a more attractive layout, and additional animations to create a more engaging user experience.
- **IoT Integration:**
Expand the functionality to control Internet of Things (IoT) devices such as lights, thermostats, and other smart home devices. This would position Pola as a versatile home assistant capable of managing a variety of connected devices.

By addressing these areas, the Pola Voice-Assistant can become a more powerful, user-friendly, and adaptable tool, capable of meeting the diverse needs of its users and keeping pace with advancements in technology.

12 User Manual

Once the Pola Voice-Assistant application is started, you will see the user interface shown in fig. 4. From here, you will only need your voice to operate the program.

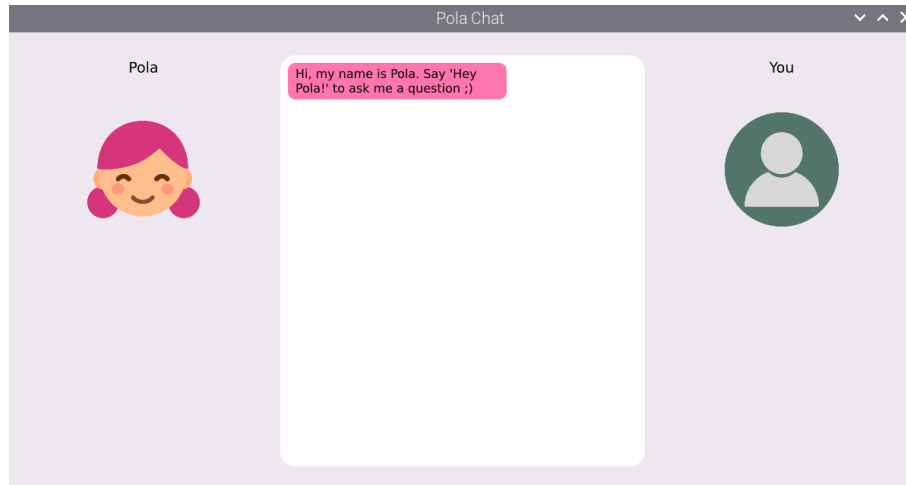


Figure 4: GUI in the Wake Word Detection state

The GUI is designed in a chat-format, widely known from private messenger applications or chat-bots. As typical, your contribution to the chat appears from the right and your chat partners contribution from the left. You are immediately invited to start the conversation with the worlds **"Hey Pola!"**.

Once you said the wake word, two predetermined messages will appear for clarity reasons. As seen in fig. 5, the message "Hey Pola!" is added on your side and the message "How can I help you?" is added on the assistant side.

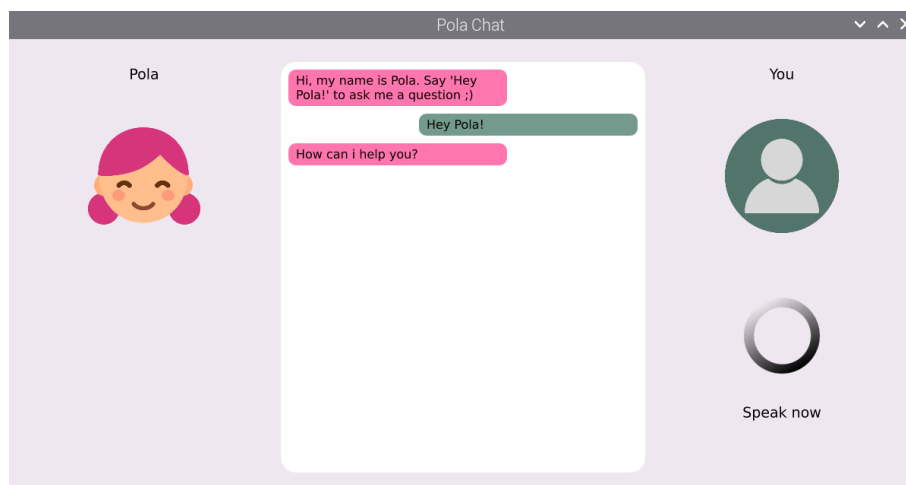


Figure 5: GUI in the Audio Recording state

A spinning circle appears on the user side. It has the caption **"Speak now"**, so you are expected to say something to Pola. Once you are done speaking, the program will detect your silence and end the recording, indicated by the circle disappearing.

The circle asking you to speak disappears and now you have to wait until the program understood what you said. As seen in fig. 6, this is indicated by a spinning circle appearing on the assistant side with the caption **"Wait for transcription"**.

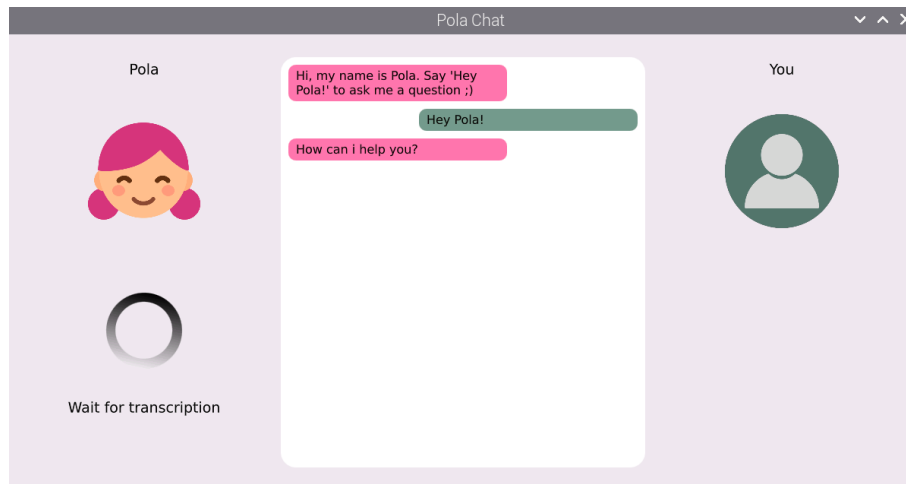


Figure 6: GUI in the Speech Recognition state

The program will show your transcribed prompt as a message and begin to generate a response, seen in fig. 7. The caption of the spinning circle will change to **"Wait for response"**.

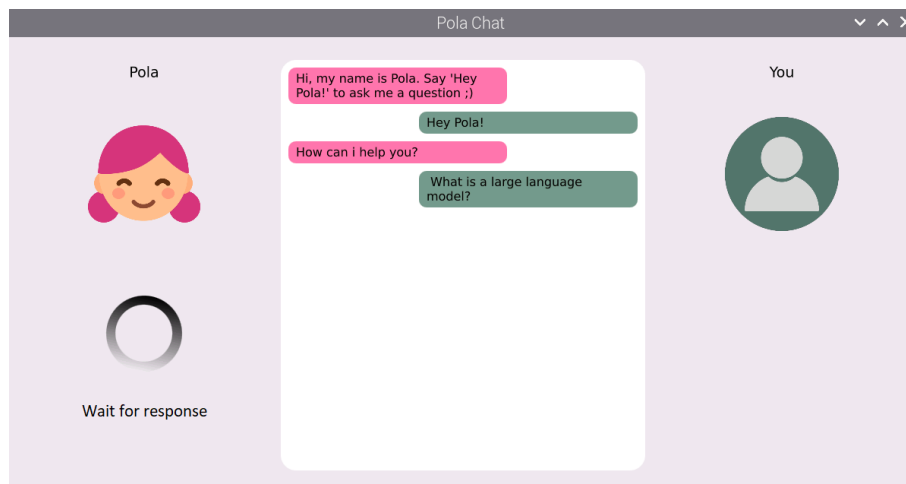


Figure 7: GUI in the Response Generation state

When the response is generated, it will appear as a message. In addition, Pola will read her message out loud. Fig. 8 shows the final chat stage.

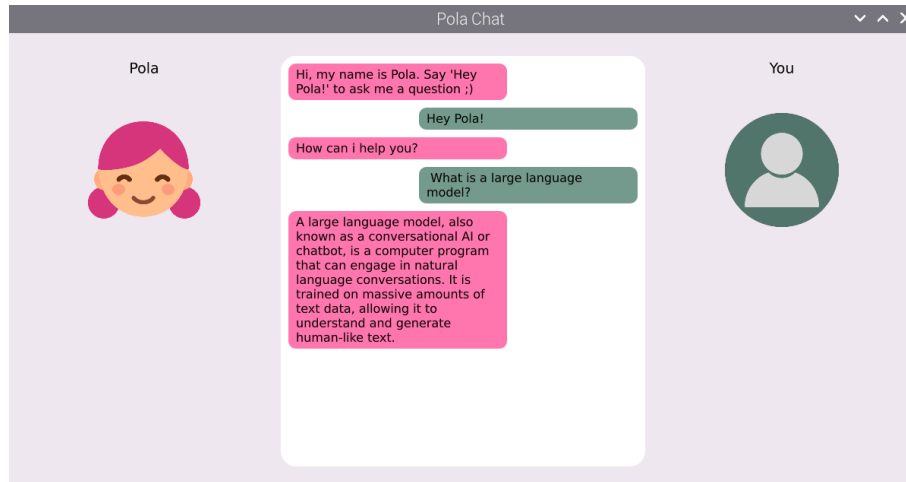


Figure 8: GUI again in the Wake Word Detection state

From here, you can start the process all over again by saying "Hey Pola!".

13 Installation Script

The following script can be used to install and set up the Hey Pola project on a Raspberry Pi:

```
#!/bin/bash

echo "Welcome to the Hey Pola installer"

# Requirements
echo "Installing non-Qt system packages (speech-dispatcher)"
sudo apt-get install speech-dispatcher
sudo apt-get install speech-dispatcher-audio-plugins

echo "Installing Python requirements (whisper, faster-whisper)"
sudo pip install whisper faster-whisper --break-system-packages

echo "Cloning git into /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/ ..."
cd /home/pi
mkdir .HeyPolaProject
cd .HeyPolaProject
git clone https://github.com/dviracachas/Embedded-Multimedia-2024.git

echo "Copying porcupine shared library for Raspberry Pi 4b"
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/resources
sudo cp libpv_porcupine.so /usr/local/lib

# Compilation of the Qt Projects
echo "Compiling source code ..."
echo "HeyPola Wake-Word-Detection"
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/HeyPola
qmake HeyPola.pro
make

echo "fw_Record"
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/fw_Record
qmake fw_Record.pro
make

echo "qt_llama"
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/qt_llama
qmake qt_llama.pro
make

echo "TextToSpeech"
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/TextToSpeech
qmake TextToSpeech.pro
make

echo "HeyPolaStateMachine"
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/HeyPolaStateMachine
qmake HeyPolaStateMachine.pro
make
```

```
cd /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/resources
touch AccessKey.txt

echo "Installation done :)"

# Warnings
echo -e "\e[1;43mPlease remember to put your Porcupine Access Key
      in /home/pi/.HeyPolaProject/Embedded-Multimedia-2024/resources/
      AccessKey.txt to make sure everything works correctly!\e[0m"
echo -e "\e[1;43mAlso edit the IP Address in /home/pi/.
      HeyPolaProject/Embedded-Multimedia-2024/qt_llama/AskPola.py to
      the IP running the Ollama API in your local network\e[0m"
```

References

- [1] Łukasz Kosiński. *QML vs Qt Widgets – detailed comparison*. URL: <https://scythe-studio.com/en/blog/qml-vs-qt-widgets-detailed-comparison>.
- [2] Lawrence R. Rabiner and Ronald W. Schafer. *Digital processing of speech signals*. eng. Englewood Cliffs (N.J.) : Prentice-Hall, 1978. ISBN: 0132136031. URL: <http://lib.ugent.be/catalog/rug01:000016539>.