HTBLuVA St. Pölten

Higher Institute for Electronics and Technical Informatics

Specialization in Embedded Systems

Voice Assistant Frontend

Author: Artemiy Smirnov
Supervisor: DI Manuel Weigl

St. Pölten, 25.04.2025

Table of Contents

1 Introduction and System Overview 3
1.1 Project Scope 3
1.2 Background and Motivation 3
1.3 Technical Challenges 3
1.4 Choice of Framework 3
1.5 System Architecture 4
1.5.1 Frontend 5
1.5.2 Backend 5
1.6 Qt Framework 5
1.6.1 qmake 5
1.6.2 Signals and Slots 6
1.6.3 Qt Creator and .ui Files 6
2 Implementation and Issues 7
2.1 Spring Boot Prototype 7
2.1.1 Backend implementation 7
2.1.2 HTTP client in Qt 7
2.2 Recording Audio 8
2.2.1 QAudioRecorder 8
2.2.2 FFmpeg 8
2.2.3 Backend recording using CPAL 8
2.3 Communication 8
2.3.1 Socket Architecture 9
2.3.2 Implementation 9
2.4 Layout and Design 9
2.4.1 Window Dimensions 9
2.4.2 Layout Implementation 9
2.5 Settings Menu 10
2.6 Raspberry Pi Integration 12
2.6.1 Operating System Selection 12
3 Results 14
4 Bibliography 15
Bibliography 15

1 Introduction and System Overview

This project implements a frontend application designed primarily for Raspberry Pi hardware, while maintaining compatibility with other Linux/Unix-like systems. Developed as an extension to an existing voice assistant diploma thesis, it focuses on providing a user-friendly interface for audio capture and processing through backend services.

1.1 Project Scope

The project scope encompassed the following objectives:

- Implementation of efficient backend communication, with a target response latency of 1 second
- Development of response playback functionality, offering both textual and audio output modalities
- Integration of multiple input methodologies, including text input, push-to-talk, and voice activation modes

1.2 Background and Motivation

Voice assistants have experienced steady adoption growth, though they were often regarded as supplementary rather than essential tools. This project sought to address this limitation by developing a voice assistant frontend solution that could enhance users' workflows while maintaining minimal intrusiveness. The implementation focused on extending traditional voice assistant capabilities with advanced features such as workspace management and speech-to-text integration.

1.3 Technical Challenges

The development process was expected to encounter several technical challenges, including but not limited to:

- **Voice Activation:** Implementing an effective voice activation system necessitates continuous listening capabilities. This can be achieved either through a wake-word detection mechanism or an advanced language model that can discern commands from regular speech. Both approaches require careful consideration of computational efficiency and responsiveness.
- **Security:** The ability to execute arbitrary commands poses a significant security risk. It is imperative to implement strict access controls and validation mechanisms to prevent unauthorized actions and protect the user from potential threats posed by malicious actors.
- **User Experience:** Ensuring a seamless and intuitive user experience is vital for the adoption of the voice assistant. This includes minimizing response times, providing clear feedback, and designing an accessible and user-friendly interface.
- **Low Latency:** Ensuring efficient communication with the backend is essential to achieve the goal of response delays no longer than 1 second on a Raspberry Pi. While most of the heavy lifting is done by the backend, communication protocols still need to be optimized to ensure low latency.

1.4 Choice of Framework

Several front-end frameworks were considered initially. These include:

- Vue.js[1]

- Godot[2]
- Qt[3]
- SDL[4]
- Low-Level Graphics[5]

Initially, `Vue.js` and `Godot` were considered due to prior experience with these frameworks. However, given that the voice assistant frontend would run on a Raspberry Pi where performance is crucial for ensuring a smooth user experience, these options were ruled out. While low-level frameworks like `SDL` and `Embedded Graphics` were available, their steep learning curve and limited industry adoption made them impractical choices. Ultimately, `Qt` was selected for its robust performance, extensive industry usage, comprehensive documentation, and the fact that it's the framework that is used by my desktop environment (KDE Plasma), so it would have seamless integration with my personal system.

1.5 System Architecture

The complete voice assistant consists of two parallel projects: this Qt-based frontend and a Rust backend being developed as an ongoing diploma thesis. The frontend serves as the user interface layer, providing efficient access to the backend's voice processing capabilities.

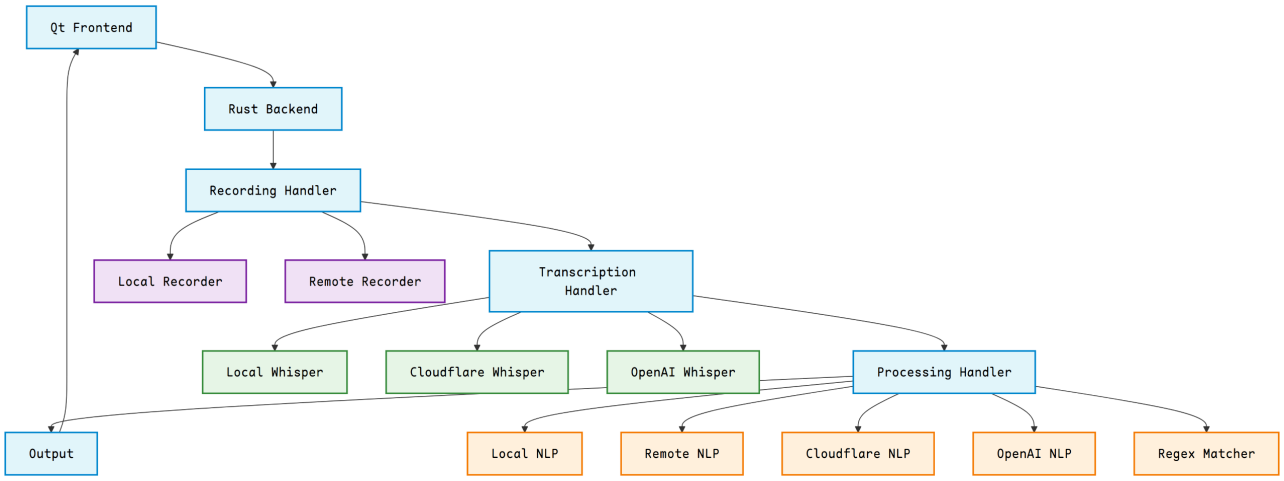

Figure 1: System architecture showing the Qt frontend (current project scope) and the Rust backend components (parallel diploma thesis). The processing pipeline flows from the frontend through various backend stages, each offering multiple implementation options.

[1]Vue.js web frontend framework [1]
[2]Godot game engine [2]
[3]Qt framework [3]
[4]Simple DirectMedia Layer library [4]
[5]Adafruit TFTLCT library [5]

1.5.1 Frontend

The Qt-based frontend layer handles:

- User interface and interaction
- Audio recording control
- Communication with backend
- Result presentation

1.5.2 Backend

The backend system, currently under development, comprises several processing stages:

- **Recording Stage:** Implements both local and remote recording capabilities:
  - Local Recorder for direct hardware access
  - Remote Recorder for distributed setups
- **Transcription Stage:** Provides multiple transcription options:
  - Local Whisper for offline processing
  - Cloudflare Whisper for edge computing
  - OpenAI Whisper for cloud-based processing
- **Processing Stage:** Implements various NLP options:
  - Local NLP for offline processing
  - Remote NLP for distributed processing
  - Cloudflare NLP for edge computing
  - OpenAI NLP for advanced language models
  - Regex Matcher for simple pattern matching
- **Output Stage:** Handles the final processing results

The modular architecture enables flexible deployment configurations. Users can choose between local processing for privacy-conscious applications or cloud-based processing for enhanced capabilities, with the frontend adapting seamlessly to either choice.

1.6 Qt Framework

The Qt framework is a comprehensive toolkit designed for creating cross-platform applications with a focus on performance and user experience. It is particularly well-suited for projects that require a responsive and visually appealing interface, such as the voice assistant frontend described in this documentation. This section provides an overview of the key components and concepts of the Qt framework, including qmake, signals and slots, Qt Creator, and .ui files.

1.6.1 qmake

qmake is the build system tool used by Qt to manage the compilation and linking of applications. It simplifies the build process by automatically generating Makefiles based on project files (.pro). A typical .pro file includes information about the source files, headers, and other resources needed for the project. Here is an example of a simple .pro file:

```
TEMPLATE = app
TARGET = myapp
QT += core gui
```

The .pro file specifies that the project is an application (TEMPLATE = app), the target executable is named “myapp”, and it uses the Qt core and GUI modules. Additional elements like source files, headers, and forms can be specified in similar fashion.

1.6.2 Signals and Slots

One of the most powerful features of Qt is its signals and slots mechanism, which facilitates communication between objects. Signals are emitted when a particular event occurs, and slots are functions that respond to these signals. This mechanism allows for a flexible and decoupled design.

The following example demonstrates connecting a button’s click signal to a label’s text update slot:

```
connect(button, &QPushButton::clicked, label, &QLabel::setText);
```

In this example, the QPushButton’s clicked signal is connected to the QLabel’s setText slot. When the button is clicked, the label’s text is updated accordingly.

1.6.3 Qt Creator and .ui Files

Qt Creator is an integrated development environment specifically designed for Qt development. It provides comprehensive tools for code editing, debugging, and UI design. The integrated UI editor allows developers to create and arrange widgets using a visual interface, generating .ui files that represent the UI layout in XML format.

2 Implementation and Issues

2.1 Spring Boot Prototype

2.1.1 Backend implementation

The backend service was initially implemented in Kotlin using Spring Boot[6], a Java-based framework that provides core infrastructure support for web applications. The service exposed a prototype REST endpoint at http://localhost:8080/process, which accepted textual input as a preliminary alternative to audio processing. The endpoint implemented pattern matching to route weather-related queries to a weather service, while other queries were processed through the GPT-4 language model.

2.1.2 HTTP client in Qt

The initial prototype consisted of a minimal user interface implementing three basic components arranged vertically: a QLineEdit widget for text input, a QPushButton for submission, and a QLabel to display the server's response. The button's `clicked()` signal was connected to a slot handling the network communication. When the submit button was clicked, the application sent a POST request to `http://localhost:8080/process` containing the user's input as plain text, and asynchronously processed the server's response through Qt's signal-slot mechanism, updating the label component to display results or errors. This HTTP-based communication was later replaced with TCP sockets to accommodate the backend's transition from REST endpoints, setting the foundation for the eventual audio processing implementation.

[6]The Spring Framework [6]

2.2 Recording Audio

2.2.1 QAudioRecorder

The initial audio recording implementation attempted to utilize Qt's QAudioRecorder class, which provides high-level audio recording functionality, however implementing it proved to be difficult; the QAudioRecorder successfully identified system audio input devices, but starting the recording consistently failed with an "empty resource stream" error. The Qt audio recording example code produced the same errors, which suggests either a bug in QAudioRecorder or, more likely, incorrect audio device configuration in the operating system.

Multiple attempts to modify audio device parameters and recording configurations proved unsuccessful, with persistent resource stream initialization failures preventing establishment of the audio capture pipeline.

2.2.2 FFmpeg

An alternative approach utilizing FFmpeg[7] through a Python helper process was explored. This implementation attempted to leverage FFmpeg's robust audio capture capabilities while maintaining the Qt application's primary control flow. However, this introduction of an external dependency and additional inter-process communication complexity led to an architectural reassessment.

2.2.3 Backend recording using CPAL

CPAL[8] is a multimedia library for Rust which allows for easy audio recording and playback across different operating systems. After the challenges with Qt's audio recording capabilities, CPAL emerged as a straightforward solution for implementing the audio capture backend.

The implementation configures an input stream with standard parameters (16kHz sample rate, mono channel, 16-bit samples) and collects audio samples in a buffer. When a client connects and requests recording, the backend starts capturing audio until receiving a stop signal. The recorded audio data is then immediately passed to the speech recognition pipeline.

Moving the recording functionality to the backend proved to be the right choice. The change resolved the technical issues encountered with client-side recording and simplified the overall architecture by centralizing all audio processing in one place. The frontend now only needs to handle starting and stopping recording, while the backend manages all the complexities of audio capture and processing.

2.3 Communication

TCP[9] sockets handle the communication between frontend and backend components. While HTTP was sufficient for the text-only prototype, the future switch to real-time voice processing to facilitate voice activation, as well as the goal to get response times under 1

[7]Fast Forward Moving Picture Experts Group [7]
[8]Cross-Platform Audio Library [8]
[9]Transmission Control Protocol [9]

second required faster data transfer. TCP sockets offer increased speed compared to HTTP, as well as ease of use, especially in Qt, as shown later.

2.3.1 Socket Architecture

The backend establishes a local TCP socket bound to port 8080, acting as a server for incoming client connections. At this point, the implementation was limited to single-client operation, though work was underway to implement asynchronous client handling.

The communication protocol implemented a simple command-response pattern, where the frontend sent commands and received either processing results or error messages. Two primary commands were supported:

- `START_RECORDING`: Initiates audio capture
- `STOP_RECORDING`: Terminates capture and processes the audio

2.3.2 Implementation

The frontend client implementation used `QTcpSocket` as well as signals and slots for asynchronous communication, allowing you to potentially configure settings while waiting for the request to finish, although the goal still was to return a response in under 1 second. TCP communication using `QTcpSocket` is very easy, with two main commands:

```
if (socket->state() == QTcpSocket::ConnectedState)
    socket->write("message");
```

The backend implementation maintains recording state and provides error handling for both recording, transcription, as well as processing operations. Response messages are transmitted back to the frontend for user feedback or error display.

2.4 Layout and Design

2.4.1 Window Dimensions

Initial development utilized a window size of 1920×1080 pixels, anticipating full-screen operation on HD displays for Raspberry Pi deployment. However, this proved suboptimal for development and unnecessarily large for the application's purposes. After analyzing similar applications, particularly the JetBrains Toolbox[10], a more compact 400×600 pixel window was implemented. The final dimensions proved largely inconsequential due to the implementation of proper scaling layouts, which ensure appropriate rendering across different display configurations.

2.4.2 Layout Implementation

Qt applications require specific layout management for proper scaling behavior. The framework provides several layout options including QVBoxLayout, QHBoxLayout, QGridLayout, and QFormLayout. The implementation opted for QVBoxLayout as the primary layout manager, which arranges child elements vertically within the application window. This choice facilitated a natural top-to-bottom organization of interface components.

[10] JetBrains Toolbox [10]

2.5 Settings Menu

A settings menu was implemented to facilitate user configuration. Navigation between the main interface and settings was accomplished through a hamburger button and back arrow positioned in the top left corner.

The interface implementation produced two primary views, as illustrated in Figure Figure 2 and Figure Figure 3.

Two notable aspects of the implementation were:

- The interface appearance was determined by the selected Qt theme, with Breeze Dark utilized in the development environment
- The settings interface was designed with a vertical spacer to emulate planned future expansion of configuration options

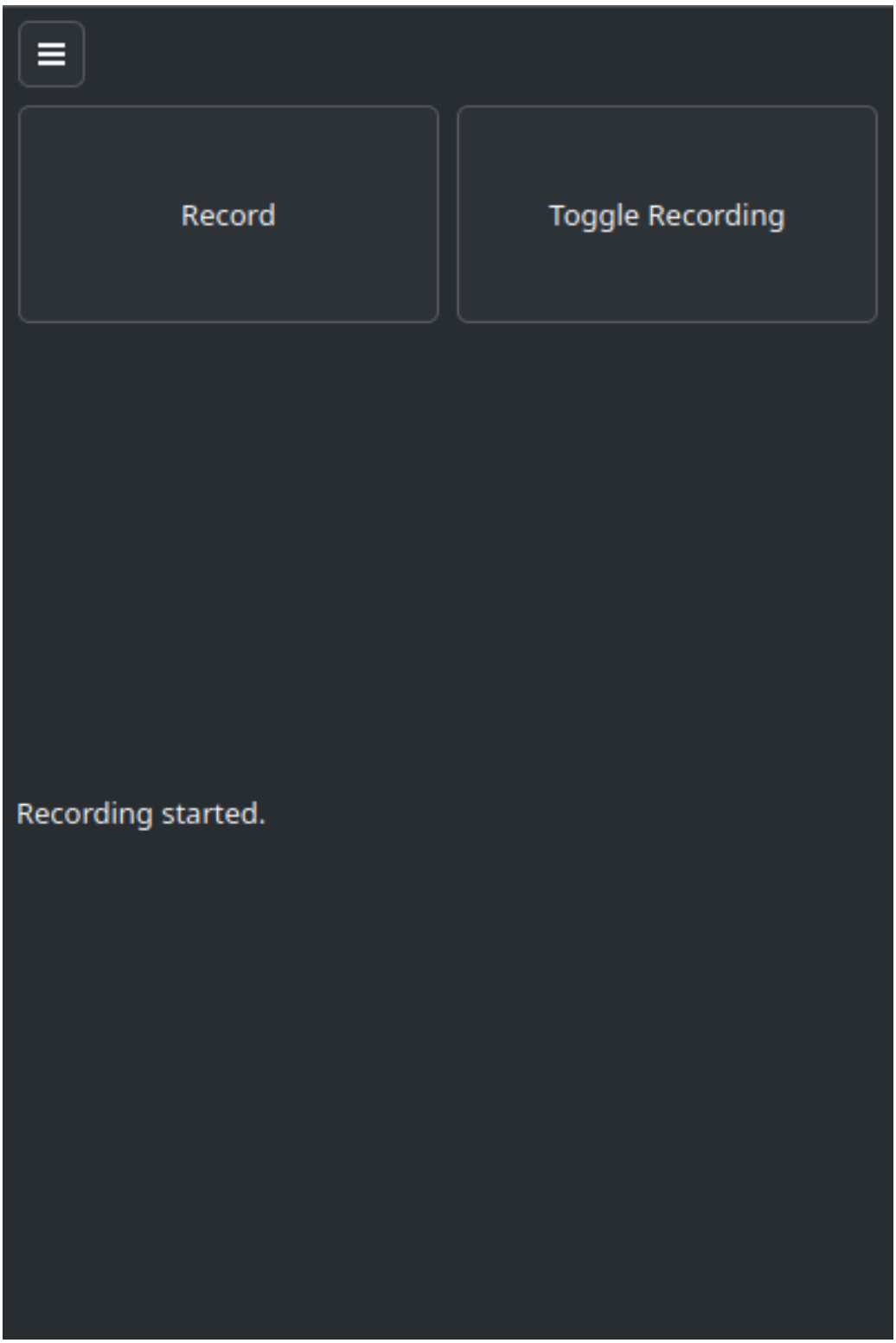

Figure 2: Main Window of the Qt application

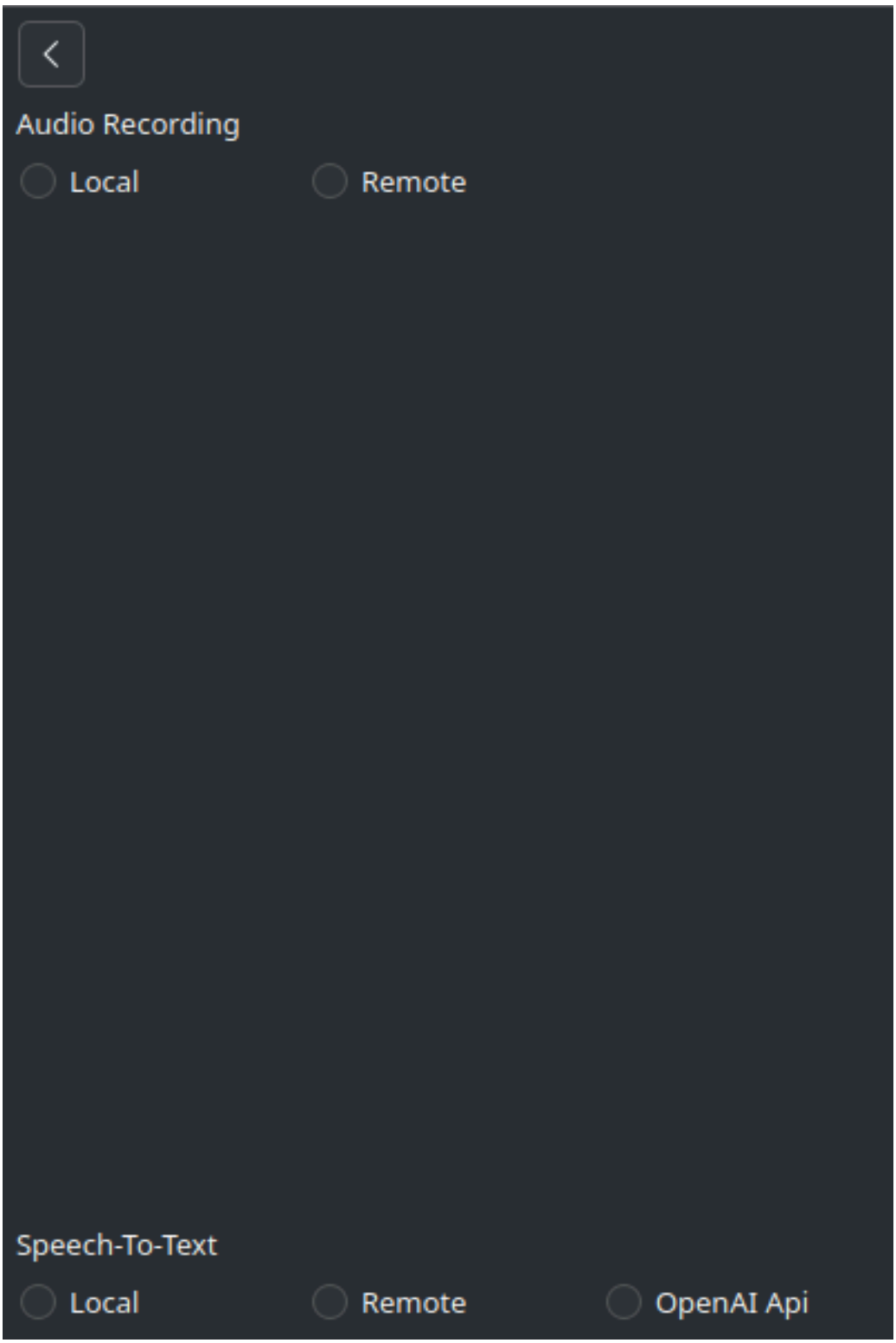

Figure 3: Settings Window of the Qt application

2.6 Raspberry Pi Integration

2.6.1 Operating System Selection

Alpine Linux proved to be the best choice for running the system on Raspberry Pi hardware. Its minimal design and stripped-down core components made it ideal for our embedded application. By using musl libc and BusyBox instead of traditional tools, Alpine runs with much lower RAM and storage overhead than standard Linux distributions. This efficiency was critical since real-time audio processing and language models already tax the

Pi's resources. The simple apk package manager made installing dependencies straightforward, while keeping the system lean.

3 Results

4 Bibliography

Bibliography

[1] E. You, “Vue.js.” [Online]. Available: https://vuejs.org/

[2] Juan Linietsky Ariel Manzur and the Godot community, “Godot Engine.” [Online]. Available: https://godotengine.org/

[3] Q. Group, “Qt Framework.” [Online]. Available: https://www.qt.io/

[4] S. Lantinga, “Simple DirectMedia Layer.” [Online]. Available: https://www.libsdl.org/

[5] Adafruit, “Adafruit TFTLCD library.” [Online]. Available: https://github.com/adafruit/TFTLCD-Library

[6] VMWare, “The Spring Framework.” [Online]. Available: https://spring.io/

[7] Fabrice Bellard and the FFmpeg team, “FFmpeg.” [Online]. Available: https://ffmpeg.org/

[8] Pierre Krieger and the RustAudio community, “Cross-Platform Audio Library.” [Online]. Available: https://crates.io/crates/cpal

[9] W. contributors, “Transmission Control Protocol.” [Online]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[10] JetBrains, “JetBrains Toolbox.” [Online]. Available: https://www.jetbrains.com/toolbox-app/