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 4
1.1 Project Scope 4
1.2 Background and Motivation 4
1.3 Technical Challenges 4
1.4 Choice of Framework 4
1.5 Qt Framework 5
1.5.1 qmake 5
1.5.2 Signals and Slots 5
1.5.3 Qt Creator and .ui Files 5
Planning 7
Milestones 7
System Architecture 7
Client-Server Architecture 7
Configuration 8
Microservices 8
2 Implementation and Issues 10
2.1 Spring Boot Prototype 10
2.2 HTTP Client in Qt 10
2.3 Rewriting it in Rust 11
2.4 Recording Audio 12
2.4.1 QAudioRecorder 12
2.4.2 FFmpeg 12
2.4.3 Backend recording using CPAL 12
2.5 Communication 13
2.5.1 Socket Architecture 13
2.5.2 Implementation 13
2.6 Layout and Design 13
2.6.1 Window Dimensions 13
2.6.2 Layout Implementation 13
2.7 Settings Menu 15
2.8 Raspberry Pi Integration 15
2.8.1 Operating System Selection 15
2.9 Protocol Updates 16
2.10 Backend for Frontend 16
2.10.1 Main Window 16
2.10.2 Main Widget 16
2.10.3 Settings Widget 17
2.10.4 Backend Client 17
2.10.5 Audio Player 18
2.10.6 Streaming Buffer 18
2.10.7 main.cpp 19
2.11 UI Redesign and Migration to QML 19
2.12 Design Prototype 20
2.13 Window Dimensions 20

2.13.1 Main Window 20
2.13.2 Settings Menu 20
2.14 QML Design 21
2.14.1 File structure 21
2.14.2 Main Window 22
2.14.3 Record Button 22
2.14.4 Wake Word Toggle 23
2.14.5 Settings Button 24
2.14.6 Log View 24
2.14.7 Settings Menu 25
2.14.8 Radio Buttons 25
2.14.9 Radio Button Group 25
2.14.10 Text Input 27
3 Results 29
3.1 Response Time 29
3.2 User Interface 29
4 Bibliography 31
Bibliography 31

1 Introduction and System Overview

This project aims to implement a frontend application for a voice assistant designed primarily for Raspberry Pi hardware, while maintaining compatibility with other Linux/ Unix-like systems. It is developed in parallel to a diploma thesis which encompasses the backend of the voice assistant, as well as a hardware setup for it.

1.1 Project Scope

The following are the main objectives of the frontend:

- Audio playback of responses from the server
- The ability to start recording using either a button or a wake word
- A user interface for configuration
- Efficient communication between client and server, with a target response time of under 1 second

1.2 Background and Motivation

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

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 solution requires a powerful wake word detection engine that works flawlessly.
- Security: Workspace Management may involve opening applications, in which case prevention of arbitrary code execution is a concern.
- User Experience: This includes minimal response times, clear feedback, and an accessible and visually appealing interface.
- Low Latency: Ensuring a response time of under 1 second requires careful design of communication, ensuring that large data packets are transferred

in streams and audio is played immediately upon availability without having to wait for all the data to arrive.

1.4 Choice of Framework

Several frontend frameworks were considered initially. These include:

- Vue.js[1]
- Godot[2]
- Qt[3]
- SDL[4]

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

- Low-Level Graphics[5]

Initially, `Vue.js` and `Godot` were considered due to prior experience with them. However, given that the voice assistant would run on a Raspberry Pi where performance is crucial for ensuring a smooth user experience,these options were ruled out. While something akin to `SDL` or more low-level graphics was an option, their steep learning curve made them impractical choices. Ultimately, `Qt` was selected for its robust performance, extensive industry adoption, and comprehensive documentation.

1.5 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.

1.5.1 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.

1.5.2 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.

1.5.3 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.

[5]Adafruit TFTLCD library [5]

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.

Planning

Planning is vital to any project, as it establishes a clear path towards success; while one always needs to be creative along the way, a well-established plan helps ensure that the final goal is clear and attainable. Setting clear goals allows the team to allocate resources properly, which includes identifying potential risks, such as exam periods along the way where the team does not have much time to work on the diploma thesis. Nonetheless, remaining flexible is of utmost importance, as unexpected things could happen in life, which may inhibit the team's ability to work for a period of time.

Milestones

Accounting for all previously mentioned points, a plan was established with clear milestones, which would be adjusted along the way to ensure full completion was possible and there were no periods of extended inactivity.

Milestone	**Date**
Backend set up and speech recognition functions	2024-10-04
Functioning software prototype (e.g., weather querying)	2024-11-01
Hardware and software communication functioning	2024-12-13
Interface with LLM	2025-01-31
Final prototype completed	2025-02-28

System Architecture

The project is separated into a frontend and a backend, where the frontend handles user interaction and experience, while the backend processes audio, turning it into the proper assistant output (text, generated audio, or an action on the system).

Client-Server Architecture

In the following flowchart, the frontend is depicted as the client and the backend as the server, communicating via a protocol to be decided based on fit.

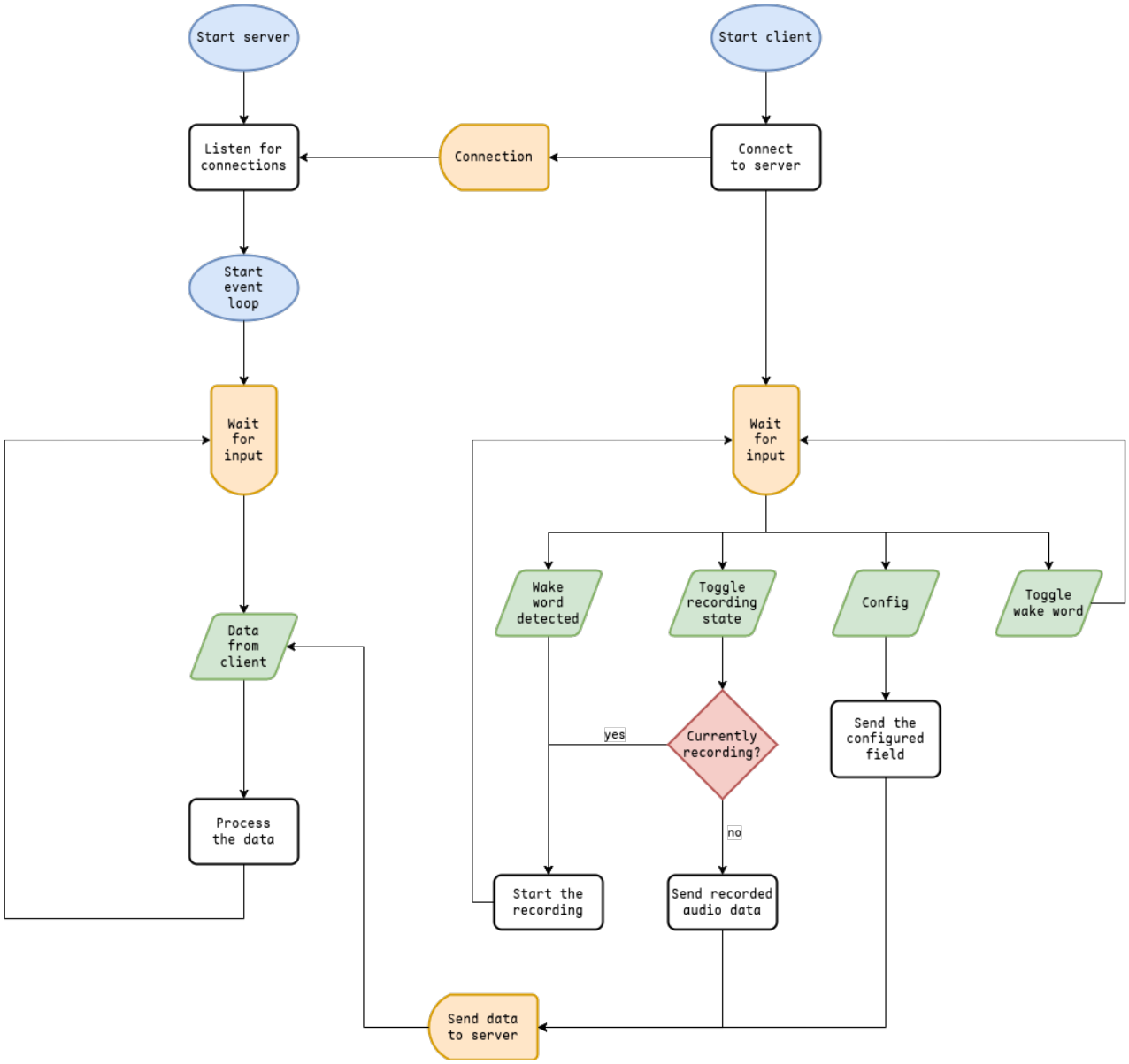

Figure 1: Client-Server Architecture

The system starts with the backend listening for connections, and the client connecting to the backend. Multiple clients may connect simultaneously. Once a client has connected, it records audio (via buttons or wake word + silence detection), sends it to the server, and the server processes it—this is the event loop of the system.

Configuration

The client can send configuration commands selected from a menu and saved in a backend file. TOML[6] would be a great option because it supports simple table-key-value splits for different system parts, while still storing arbitrary key-value pairs.

Microservices

To avoid a monolith, microservices will be used for each part of the system on the backend. The next figure shows how audio and configuration requests flow through these services.

[6]Tom's Obvious, Minimal Language [6]

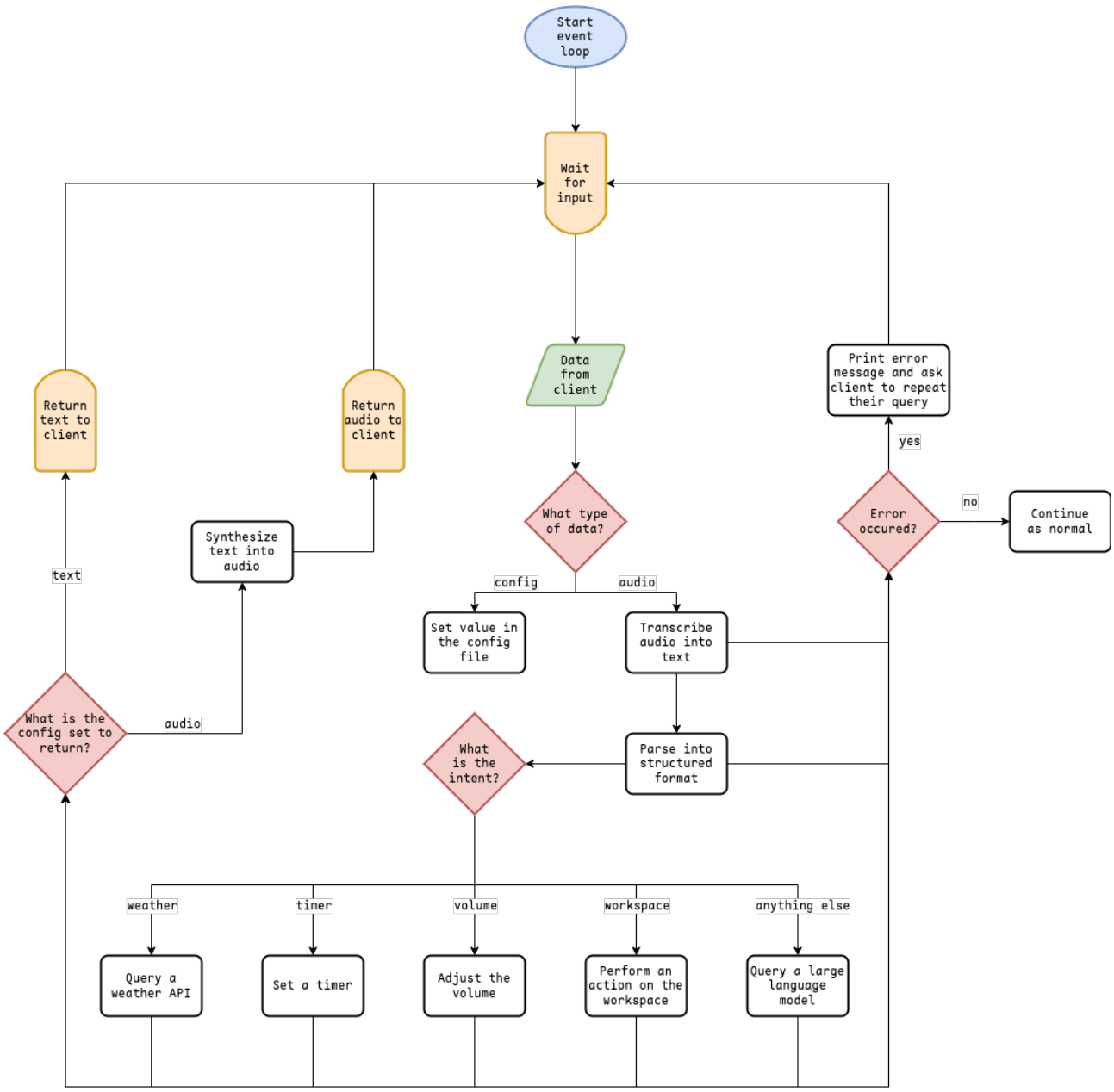

Figure 2: Processing of audio

2 Implementation and Issues

2.1 Spring Boot Prototype

The backend was initially developed using Spring Boot; the prototype featured a REST endpoint, which would await `POST` requests and was accessible at `http://localhost:8080/process`. Shown below is the Spring mapping of this endpoint:

```
@PostMapping("/process")
fun process(@RequestBody content: String): Mono<ResponseEntity<String>> {
  return parsingService.parse(content).map { ResponseEntity.ok(it) }
}
```

It would accept an HTTP `POST` request with a string in its body, which was the input sent by the user. This was the first implementation, and it did not support audio yet, in order to keep it simple and iterate quickly. A pattern-matching parser was implemented using Kotlin's `when` statement, which is similar to a switch–case in other languages. Below is a demonstrative, simplified version of this parser:

```
when {
  "units" in input -> unitsService.getUnitsInfo()
  "weather" in input -> {
    geocodingService.getLocation(input).map { (town, lat, lon) ->
      weatherService.getForecast(lat, lon)
    }
  }
  else -> openAIService.getCompletion(input).map {
    it.choices.first().message.content!!
  }
}
```

It had three services implemented, those being:

- Units Service: Returns the units that are in use (metric or imperial)
- Weather Service: Queries the OpenWeatherMap API using coordinates from the OpenStreetMap Nominatim API
- Open AI Service: Queries an LLM from OpenAI if the parser does not find the pattern of any other service

2.2 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.

2.3 Rewriting it in Rust

Because Kotlin runs on the Java Virtual Machine[7], it does not offer particularly good performance. Since the plan was to run the voice assistant on a Raspberry Pi, a decision was made early in the development of the project to abandon the current backend and rewrite it in Rust — a significantly more performant alternative that does not compromise on ergonomics. Rust compiles to native machine code and is therefore nearly as fast as C or C++, only with slow compilation times and some overhead introduced by the borrow checker — both of which can be disregarded when compared to the limitations of the JVM.

Rust takes an innovative approach to systems programming by combining low-level control with modern language features that do not compromise on ergonomics. It uses errors as values via the `Result` enum:

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

And by utilizing its powerful pattern-matching capabilities, this becomes an excellent way to avoid almost all runtime errors in a simple and elegant fashion.

Rust features a powerful type system with algebraic data types. Its structs (product types) are similar to classes in object-oriented languages, while its enums (sum types) enable expressive pattern matching. Structs and enums in Rust can hold multiple values — including named fields — far beyond primitive types. For example:

```
enum Foo {
  Bar(f64, f64, f64),
  Baz(String, String, String),
  Qux {
    quux: f64,
    quuz: f64,
  },
}

struct Bar {
  Baz: f64,
  Qux: (f64, String, i128),
}
```

These data types shine in `match` and `if let` statements for powerful, ergonomic pattern matching.

Interfaces are ubiquitous in many languages — Rust replaces them with traits. Traits let you specify that a struct implements certain behavior, and you can write `impl Trait for Struct { … }` to wire everything together. This approach decouples implementation from usage more flexibly than traditional interfaces.

Another elevator pitch for Rust is its focus on memory safety via the borrow checker. Ownership and borrowing rules ensure that variables have a single owner, can be

[7]Java Virtual Machine [7]

immutably borrowed many times or mutably borrowed once, and that all borrows respect lifetimes. At first the borrow checker can feel restrictive, but it ultimately guarantees memory safety and prevents data races (unless you opt into `unsafe` blocks). Low-level operations — like writing to a hardware register — still require `unsafe`, but the vast majority of code remains completely safe.

2.4 Recording Audio

2.4.1 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.

2.4.2 FFmpeg

An alternative approach utilizing FFmpeg[8] 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.

2.4.3 Backend recording using CPAL

CPAL[9] 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 it receives 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.

[8]Fast Forward Moving Picture Experts Group [8]

[9]Cross-Platform Audio Library [9]

2.5 Communication

TCP[10] sockets handle the communication between frontend and backend components. While HTTP was sufficient for the text-only prototype, the future implementation of audio output streaming and the goal to get response times under 1 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.

2.5.1 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

2.5.2 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");
```

Similarly, `socket->readAll()` can be used to read data.

2.6 Layout and Design

2.6.1 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[11], 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.

2.6.2 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

[10] Transmission Control Protocol [10]

[11] JetBrains Toolbox [11]

window. This choice facilitated a natural top-to-bottom organization of interface components.

2.7 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 3 and Figure 4.

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

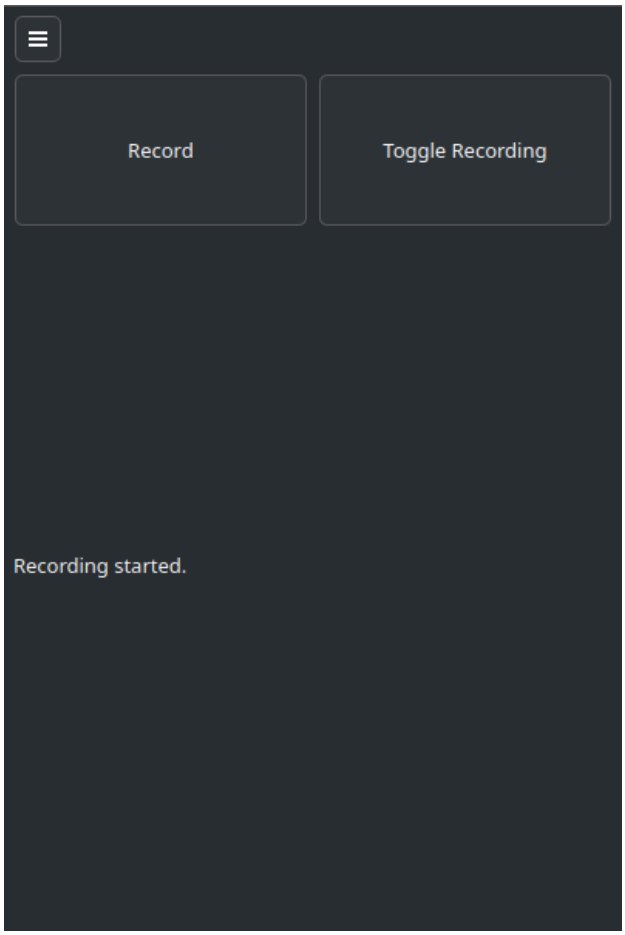

Figure 3: Main Window

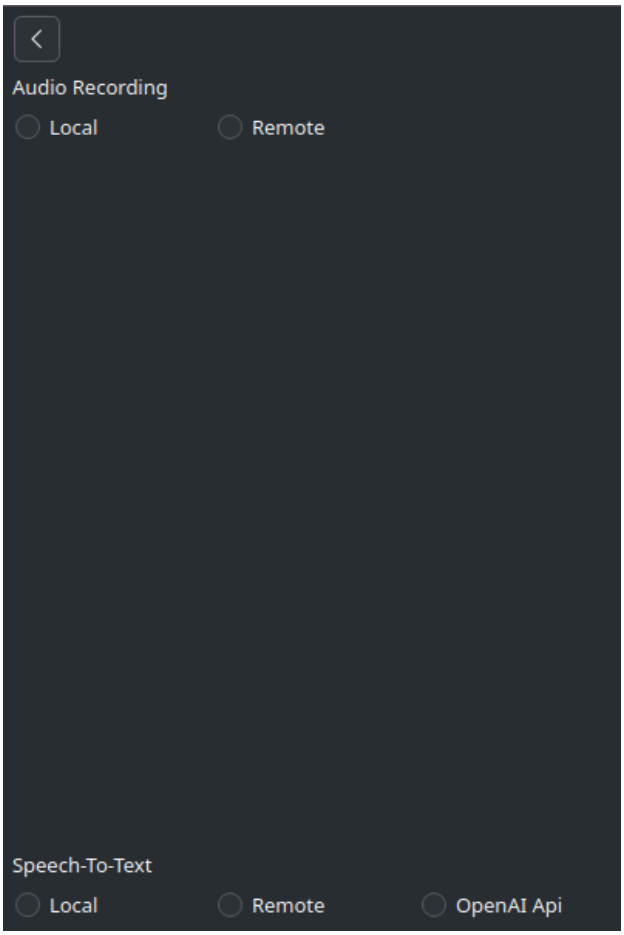

Figure 4: Settings Window

2.8 Raspberry Pi Integration

2.8.1 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.

2.9 Protocol Updates

Raw TCP made it difficult to separate text from binary data, requiring too much extra logic. To simplify things, the setup was changed to use WebSockets over TCP. WebSockets include built-in message types, which makes it easier for both sides to tell whether data is text, binary, or a control message. Qt uses `QWebSocket` for WebSocket communication and it is almost a drop-in replacement for `QTcpSocket`, with only minor changes necessary.

2.10 Backend for Frontend

To manage UI components and to communicate with the server, Qt usually requires a C++ backend; this backend is not the actual backend of the application, it is the backend of the frontend, also called `Backend for Frontend`, or `BFF`. The BFF manages UI logic and communication with the server; the file structure of the BFF was as follows:

```
include/
├── audioplayer.h
├── backendclient.h
├── mainwidget.h
├── mainwindow.h
├── settingswidget.h
└── streamingbuffer.h

src/
├── audioplayer.cpp
├── backendclient.cpp
├── mainwidget.cpp
├── mainwindow.cpp
├── settingswidget.cpp
└── streamingbuffer.cpp

gui/
├── mainwidget.ui
├── mainwindow.ui
└── settingswidget.ui

main.cpp
```

2.10.1 Main Window

The `mainwindow.h` and `mainwindow.cpp` files contain logic for instantiating the app and for switching between the main widget and the settings widget.

2.10.2 Main Widget

The `mainwidget.h` and `mainwidget.cpp` files provide the logic implementation for the main widget that the user sees when they first open the app; it contains the record button, the toggle wake word button, and the text output field; here, functions of the backend client are called to communicate to the server when the record button and the toggle wake word buttons are pressed.

2.10.3 Settings Widget

The `settingswidget.h` and `settingswidget.cpp` files provide the logic for the settings widget, calling functions from the backend client to configure settings on the server when the user modifies a setting.

2.10.4 Backend Client

The `backendclient.h` and `backendclient.cpp` files provide the logic for the heart of the application: Communication between this frontend and the server; it defines multiple functions, signals and slots, as seen below:

```
#pragma once
#include <QObject>
#include <QWebSocket>

class BackendClient : public QObject
{
  Q_OBJECT

public:
  explicit BackendClient(const QString &host, quint16 port, QObject *parent =
nullptr);
  void cancel();
  void config(const QString &element);
  void startRecording();
  void stopRecording();

signals:
  void binaryMessageReceived(const QByteArray &message);
  void textMessageReceived(const QString &message);

private slots:
  void onConnected();
  void onBinaryMessageReceived(const QByteArray &messge);
  void onTextMessageReceived(const QString &message);
  void onBytesWritten(qint64 bytes);

private:
  void sendMessage(const QString &message);

  QWebSocket *socket;
};
```

Some of these are self-explanatory: `startRecording()` and `stopRecording()` simply send messages to the server via `sendMessage()` to either start or stop the recording. The `cancel()` function cancels the recording, without processing it, however, no button was ever implemented for it, so it is effectively dead code.

The signals `binaryMessageReceived()`, and `textMessageReceived()` were connected to equivalent signals in `QWebSocket` and to their corresponding slots, `onBinaryMessageReceived()`, and `onTextMessageReceived()`.

These trigger when the server sends a message, be that text or binary:

- If the message is text, it is printed to the text view in the main widget, for the user to see.
- If the message is binary, it is assumed to be WAV audio (as that is the only binary message that the server can send) and passed on to `AudioPlayer`.

2.10.5 Audio Player

The `audioplayer.h` and `audioplayer.cpp` are the core of audio playback. If audio is selected (instead of text), then the backend client will pass on a WAV `StreamingBuffer` to the audio player, and the audio player will then play what is available inside the `StreamingBuffer`. Audio is added by the backend client using the `appendAudioData()` function:

```
bool AudioPlayer::appendAudioData(const QByteArray &audioData)
{
  if (audioData.isEmpty()) {
    qWarning() << "Received empty audio data!";
    return false;
  }
  m_streamBuffer->appendData(audioData);
  play();
  return true;
}
```

2.10.6 Streaming Buffer

As there is no default Qt implementation for a buffer that supports arbitrary data appending while being read from with correct position marking, a new buffer had to be implemented: `StreamingBuffer` in `streamingbuffer.h` and `streamingbuffer.cpp`; it defines the following member variables:

```
  mutable QMutex m_mutex;
  QByteArray m_data;
  qint64 m_readPos;
```

A `QMutex` for concurrent access of the underlying data, a `QByteArray` for storing the data, and a `qint64` for the current position that is being read from.

The `StreamingBuffer` allows appending of data at an arbitrary point, by locking the mutex and appending the data and emitting `readyRead()` after.

```
void StreamingBuffer::appendData(const QByteArray &data)
{
  qDebug() << "Appending data of length" << data.length();
  QMutexLocker locker(&m_mutex);
  m_data.append(data);
  emit readyRead();
}
```

Reading is done via `readData()`, which locks the mutex, copies the audio data to a buffer and updates the current read index:

```
qint64 StreamingBuffer::readData(char *data, qint64 maxSize)
{
  QMutexLocker locker(&m_mutex);
  if (m_readPos >= m_data.size())
    return 0;
```

```
  qint64 bytesToRead = qMin(maxSize, static_cast<qint64>(m_data.size() -
m_readPos));
  memcpy(data, m_data.constData() + m_readPos, bytesToRead);
  m_readPos += bytesToRead;
  return bytesToRead;
}
```

Interestingly, a Qt buffer (`QIODevice`) implementation requires a `bool atEnd()` function to be implemented, but since the only way to know if more data is coming is through a marker at the end, and since that would require more unnecessary overhead, the `atEnd()` function simply always returns `false`.

```
bool StreamingBuffer::atEnd() const
{
    // Even if temporarily no data is available, more data might be appended later.
    // Return false to indicate the stream is not ended.
    return false;
}
```

2.10.7 main.cpp

`main.cpp` initializes the application, the main window, the window size, and executes the application:

```
#include <QApplication>
#include <src/core/mainwindow.h>

int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  MainWindow w;
  w.setMinimumSize(400, 600);
  w.show();
  return a.exec();
}
```

2.11 UI Redesign and Migration to QML

Up until now, the user interface was built using the `Design` window in Qt Creator, a visual editor for the XML-based `.ui` file format used with Qt Widgets. After setting up the main window and a basic version of the settings window, it became clear that creating a visually appealing interface would be challenging, mainly due to the outdated and limited nature of the `.ui` designer. Building something that takes minutes in QML would take hours using `.ui` files, so switching to QML seemed like the better option. QML offers a more modern approach, with a CSS-like syntax and support for inline JavaScript, making it easier to write and maintain programmatically. The backend, meanwhile, stayed largely the same and continued to use C++.

To support the migration, the project structure was reorganized. Instead of the previous `src/core` and `src/gui` directories, a new layout was introduced with separate `backend` and `qml` directories. The existing `.ui` files were moved into the `qml` directory to serve as references during the transition. An initial QML version of each `.ui` file was created, focused solely on replicating functionality. At this stage, no specific attention was given to design or styling— the priority was to ensure that the interface behaved as expected using

QML. Once the basic structure was in place, each view was gradually refactored into smaller, reusable components. This made the codebase easier to navigate and maintain, and laid the groundwork for later visual improvements.

2.12 Design Prototype

Experience with previous projects showed that designing a UI without any reference material is inefficient, especially when working with an unfamiliar language. This often leads to creating a UI, realizing it doesn't look right, then discarding it, resulting in wasted time on elements that won't be part of the final design.

2.13 Window Dimensions

At this point, a display had been purchased that would end up being used for the frontend, and its dimensions were `1024x600`, so from now on, all windows will match that size.

2.13.1 Main Window

To avoid this, the layout of the main window was initially prototyped in Figma, a design tool known for its ease of use and effectiveness in creating UI references. Figma's clean and simple interface makes it easy to create a layout quickly, with the option to gradually refine it by adding details such as colors, shadows, and animations.

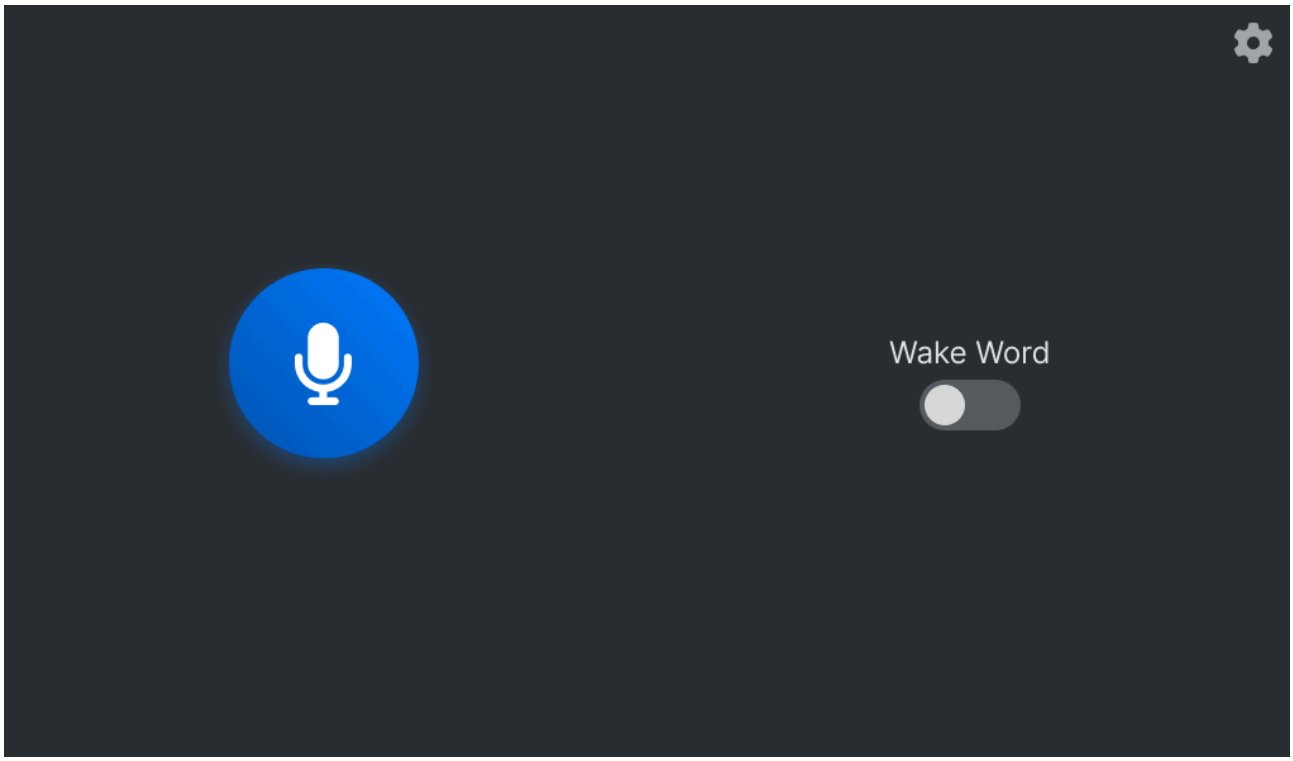

Figure 5: Final Figma design

The UI is clean and intuitive, featuring a record button, a toggle switch for the wake word, and a gear icon for the settings menu.

2.13.2 Settings Menu

A decision was made not to redesign the settings menu in Figma; the original design from the `.ui` file was already visually appealing enough, such that with minimal QML styling, it would be production-ready. This is an important point to consider when "predesigning" UI

with Figma: The settings window is so simple that making a reference design would be of no use and would only waste time.

This decision proved to be correct, showing that not every aspect requires extensive planning. Sometimes, the implementation is simple enough to just proceed with it without doing extra work.

2.14 QML Design

Qt Quick is a modern cross-platform UI framework which uses Qt Meta-object Language (QML)[12] for UI design. QML is a declarative language with syntax similar to CSS, and it supports inline JavaScript for handling logic and interaction. Below is a small example snippet:

```
Rectangle {
  width: 100
  height: 100
  color: mouseArea.pressed ? "orange" : "lightgray"

  MouseArea {
    id: mouseArea
    anchors.fill: parent
    onClicked: {
      console.log("Rectangle clicked at:", mouse.x, mouse.y);
    }
  }
}
```

This snippet demonstrates how QML is used to build user interfaces in a declarative way. Instead of writing code that manually creates and updates UI elements, the structure and behavior are described directly in terms of what the interface should look like and how it should respond to interaction.

A Rectangle is defined with a fixed size and a color property that updates automatically based on the state of the MouseArea. When the mouse is pressed, the color changes to orange; otherwise, it remains light gray. The logic for user interaction is embedded directly using inline JavaScript inside the MouseArea, which handles click events and logs the mouse position.

2.14.1 File structure

QML works great with a modular structure, as it allows you to define your own components and work with them just as you would with built-in components. It was clear that a modular component structure would be the most beneficial, especially with the settings menu having lots of repeated components.

The following organized structure was created:

[12]Qt Meta-object Language [12]

```
qml/
├── components/
│   ├── ActionButton.qml
│   ├── RecordButton.qml
│   ├── SettingsButton.qml
│   ├── SettingsRadioButton.qml
│   ├── SettingsRadioButtonGroup.qml
│   ├── SettingsTextInput.qml
│   └── WakeWordToggle.qml
├── Main.qml
└── Settings.qml
```

2.14.2 Main Window

The main window is composed of multiple subcomponents:

- Settings button (gear icon)
- Recording button (blue circle with a microphone icon)
- Wake word toggle switch
- Log view (Text area at the bottom of the screen)

They are implemented using their respective subcomponents and sometimes a wrapper container for easier positioning.

2.14.3 Record Button

The record button is implemented using a QML Button with a `FontAwesome` microphone icon and a Rectangle background, with a radius of `width / 2`, effectively turning it into a circle:

```
Button {
  contentItem: Text {
    text: "\uf130"
    font.family: "FontAwesome"
  }
  background: Rectangle {
    radius: width / 2
  }
}
```

The color is a vertical linear gradient between two shades of blue:

```
gradient: Gradient {
    GradientStop { position: 0.0; color: "#007bff" }
    GradientStop { position: 1.0; color: "#0056b3" }
}
```

Additionally, there is a small drop shadow on the bottom of the button to emphasize it more, it is created using a layer on the background rectangle:

```
layer.enabled: true
layer.effect: DropShadow {
    horizontalOffset: 0
    verticalOffset: 6
    radius: 15
    samples: 15
    color: Qt.rgba(0, 123, 255, 0.3)
}
```

Connecting the button to the backend is made really simple in QML, using an `onClicked` event it can simply call the backend `startRecording()` and `stopRecording()` functions:

```
onClicked: {
    recording = !recording
    if(recording) backend.startRecording()
    else backend.stopRecording()
}
```

For these functions to be invokable from QML, they need to be annotated with `Q_INVOKABLE` on the backend:

```
Q_INVOKABLE void startRecording();
Q_INVOKABLE void stopRecording();
```

2.14.4 Wake Word Toggle

The wake word toggle is built with a Qt Quick Controls Switch for the on/off behavior. The track and the knob are each drawn as a Rectangle, with radii of `width / 2`. The knob's x position is bound so that when checked it sits 4 px from the right edge, and when unchecked it sits 4 px from the left.

```
Switch {
  id: wakeWordToggle
  indicator: Rectangle {
    radius: width / 2

    Rectangle {
      radius: width / 2
      x: wakeWordToggle.checked ? parent.width - width - 4 : 4
    }
  }
}
```

Both the knob and the track feature black drop shadows. The QML snippet below defines the knob's drop shadow, while the track uses a similar shadow with larger parameters for a more diffuse effect:

```
layer.enabled: true
layer.effect: DropShadow {
    horizontalOffset: 0
    verticalOffset: 2
    radius: 4
    samples: 8
    color: Qt.rgba(0, 0, 0, 0.3)
}
```

The switch is connected to the backend using the following event trigger:

```
onToggled: {
  wakeWordControl.enabled = checked
  backend.setConfig("recording.wake_word_enabled=" + (checked ? "true" : "false"))
}
```

Similarly to the record button, the `setConfig()` function in the backend needs to be marked with `Q_INVOKABLE`:

```
Q_INVOKABLE void setConfig(const QString &element);
```

To ensure the wake word toggle is set to the correct state when connecting to the server, the following function was created to update the state once the server sends the current configuration:

```
function updateEnabled(configLine) {
    if (configLine.startsWith("recording.wake_word_enabled=")) {
        let value = configLine.split("=")[1]
        wakeWordControl.enabled = (value === "true")
        wakeWordToggle.checked = wakeWordControl.enabled
    }
}
```

The `configUpdateReceived` signal from the backend was connected to this function:

```
Component.onCompleted: {
    backend.configUpdateReceived.connect(updateEnabled)
}
```

2.14.5 Settings Button

The settings button is a simple QML `Button` using the `FontAwesome` gear icon:

```
Button {
  contentItem: Text {
      text: "\uf013"
      font.family: "FontAwesome"
  }
}
```

Inside the instantiation of the settings button in the main window, the `onClicked` signal was connected to visibility toggles:

```
onClicked: {
    settingsScreen.visible = true
    mainScreen.visible = false
}
```

2.14.6 Log View

To display error messages from the backend, a TextArea inside a ScrollView was added to the main window. It is hidden by default to avoid taking up space when there is no content. A standalone component wasn't created for this, as the component would be quite small, and the advantages of modularity wouldn't justify the extra complexity in this case.

```
ScrollView {
    id: outputScrollView
    visible: false

    TextArea {
        id: output
        readOnly: true

        background: Rectangle {
            color: "#1a1d21"
            opacity: 0.8
        }
    }
}
```

The C++ backend emits a `textMessageReceived` signal, which was connected in QML to the function below. This function updates the text in the log view and makes it visible.

```
function onTextMessageReceived(message) {
    output.text += message + "\n"
    outputScrollView.visible = true
}
```

2.14.7 Settings Menu

The settings menu needed to have multiple categories each with:

- Multiple Radio Buttons for implementation selection
- Potentially some text inputs for things like base URL or language model selection

2.14.8 Radio Buttons

Radio Buttons themselves are fairly straightforward, they use a QML `Radio Button`, with the outer circle and the selection dot defined as `Rectangle` components with radii of `width / 2`

```
RadioButton {
  id: radioButton

  indicator: Rectangle {
    radius: width / 2

    Rectangle {
      radius: width / 2
      visible: radioButton.checked
    }
  }
}
```

2.14.9 Radio Button Group

Radio buttons by themselves are useless, they need to be part of a group to allow the user to select between multiple different options. Radio button groups are implemented using a `ButtonGroup` component and an event trigger that sends the selected configuration to the server immediately after the button is pressed, removing the need to press `Apply`:

```
ButtonGroup {
  id: buttonGroup
  onCheckedButtonChanged: {
    if (checkedButton && !root.updatingFromServer) {
        root.selectedValue = checkedButton.configValue
        backend.setConfig(root.configKey + "=" + checkedButton.configValue)
    }
  }
}
```

The group also has a text label to describe which configuration this group modifies, it is placed above the radio buttons. Instantiating a radio button group requires configuring the following property variables:

```
property string title: "Settings Group"
property var options: [
  { text: "Option 1", value: "option1" },
  { text: "Option 2", value: "option2" }
]
property string configKey: "setting.key"
property string defaultValue: options.length > 0 ? options[0].value : ""
```

For example, below is the text-to-speech configuration:

```
SettingsRadioButtonGroup {
  title: "Text-to-speech"
  configKey: "synthesis.implementation"
  options: [
    { text: "Local", value: "piper" },
    { text: "ElevenLabs API", value: "elevenlabs" }
  ]
  defaultValue: "piper"
}
```

The `configKey` property is defined by the server, and is formatted as `table.key` from the TOML configuration file that the server uses. Not all fields from the server configuration have been included in the settings menu, such as `recording.porcupine_sensitivity` or the entire weather configuration, as to avoid cluttering the ui — configuring the porcupine wake word detection sensitivity is unnecessary, as the default just works, and putting the weather configuration in the settings menu would not be productive, as the only reason someone would edit the weather config is if they are self-hosting the weather config, and in that case, they will be capable of editing the config and other users will not be confused by the weather API base url being a configurable field. If the user still wants to tinker with config fields that were not included in the settings menu, they can configure them manually in `$XDG_CONFIG_HOME/.config/voice/config.toml`. Below is the default configuration file that is copied to the above path on the first run of the server:

```
[geocoding]
base_url = "https://nominatim.openstreetmap.org/"
user_agent = "eagely's Voice Assistant/1.0"
implementation = "nominatim"

[llm]
deepseek_base_url = "https://api.deepseek.com/"
ollama_base_url = "http://localhost:11434/"
deepseek_model = "deepseek-chat"
ollama_model = "deepseek-r1:7b"
implementation = "deepseek"

[parsing]
rasa_base_url = "http://localhost:5005/"
implementation = "patternmatch"

[recording]
device_name = "pipewire"
implementation = "local"
remote_url = "ws://localhost:5555/"
porcupine_sensitivity = 1
wake_word_enabled = true
wake_word = "ferris.ppn"

[response]
response_kind = "audio"

[server]
host = "127.0.0.1"
port = 8080

[transcription]
local_model_path = "base.bin"
local_use_gpu = true
deepgram_base_url = "https://api.deepgram.com/v1/"
implementation = "deepgram"

[synthesis]
elevenlabs_base_url = "wss://api.elevenlabs.io/"
elevenlabs_model_id = "eleven_multilingual_v2"
elevenlabs_voice_id = "21m00Tcm4TlvDq8ikWAM"
piper_base_url = "http://localhost:5000/"
piper_voice = "en_US-ljspeech-high.onnx"
implementation = "elevenlabs"

[weather]
base_url = "https://api.openweathermap.org/data/3.0/onecall/"
implementation = "openweathermap"
```

2.14.10 Text Input

As visible in the default configuration above, some fields clearly require textual configuration and not just radio button selection. A text input field was created using QML `TextInput` and the following event trigger:

```
onTextEdited: {
    backend.setConfig(root.configKey + "=" + text)
}
```

Notably, the event is called `onTextEdited`, and it only triggers when the user actively types something in, not just when the text is programmatically changed — which would trigger the `onTextChanged` event.

This distinction is important because the server echoes any configuration updates to all connected clients, and each client's settings menu gets updated accordingly. If `onTextChanged` were used instead, it could result in an infinite loop:

- The client sends an update to the server.
- The server receives and broadcasts it back to all clients.
- The client sees the text change and sends it again, repeating the cycle.

While this loop might not occur if the echoed configuration is identical to what the client already has, issues arise if the server modifies the value — for example, by trimming whitespace or removing a trailing `/` from a URL. In such cases, the text input would detect a change, triggering another send, leading to the feedback loop.

3 Results

All in all, the project was a success, every feature that was initially planned is implemented.

3.1 Response Time

Concerning the low response times: For the frontend, there is nothing more left to optimize to achieve the 1 second goal. Currently, responses take between `1.8 seconds` and `multiple minutes`, depending on which configuration is used; running large machine learning models locally on a Raspberry Pi is a terrible idea and causes severe delays, which is why every part of the system that requires a machine learning model offers either a cloud-based alternative, or a simpler algorithm instead. If ran with a powerful network connection and on powerful hardware, the system can theoretically achieve the goal response time of under `1 second`.

3.2 User Interface

The final UI is shown below:

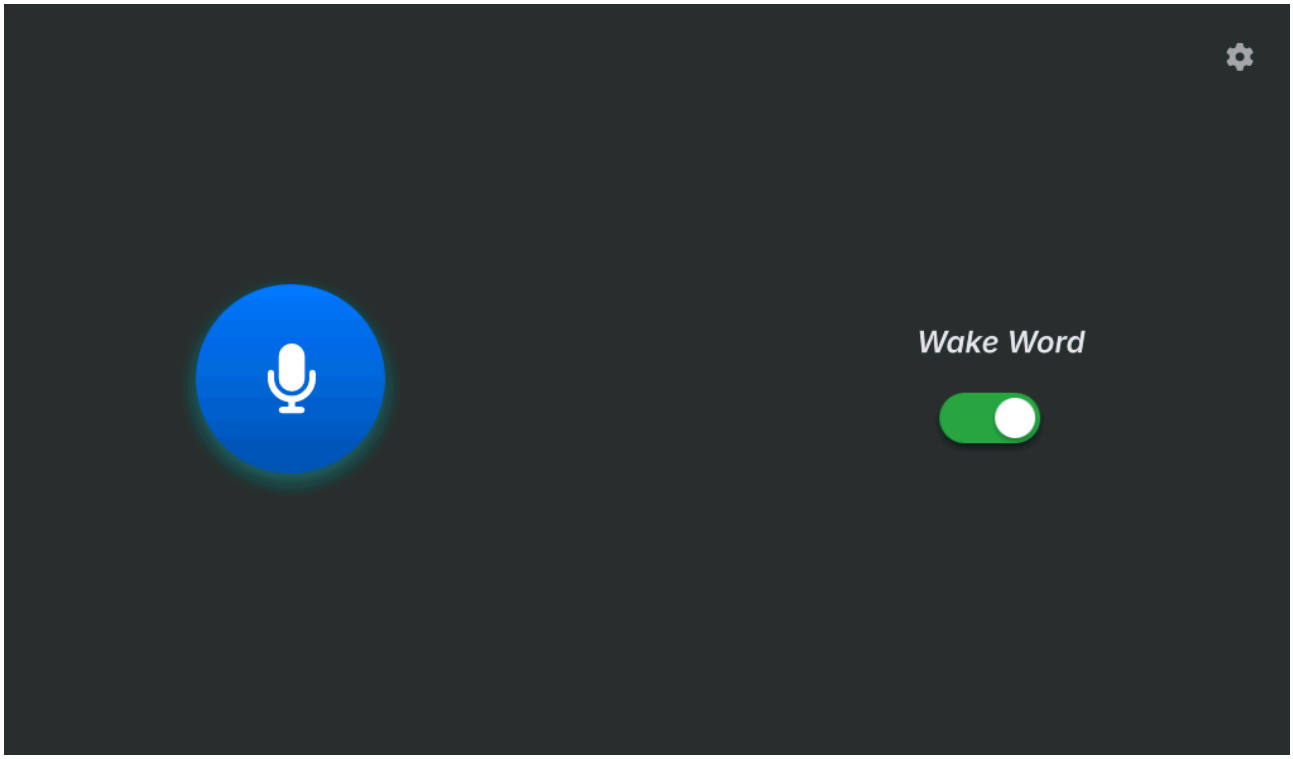

Figure 6: Final Main UI

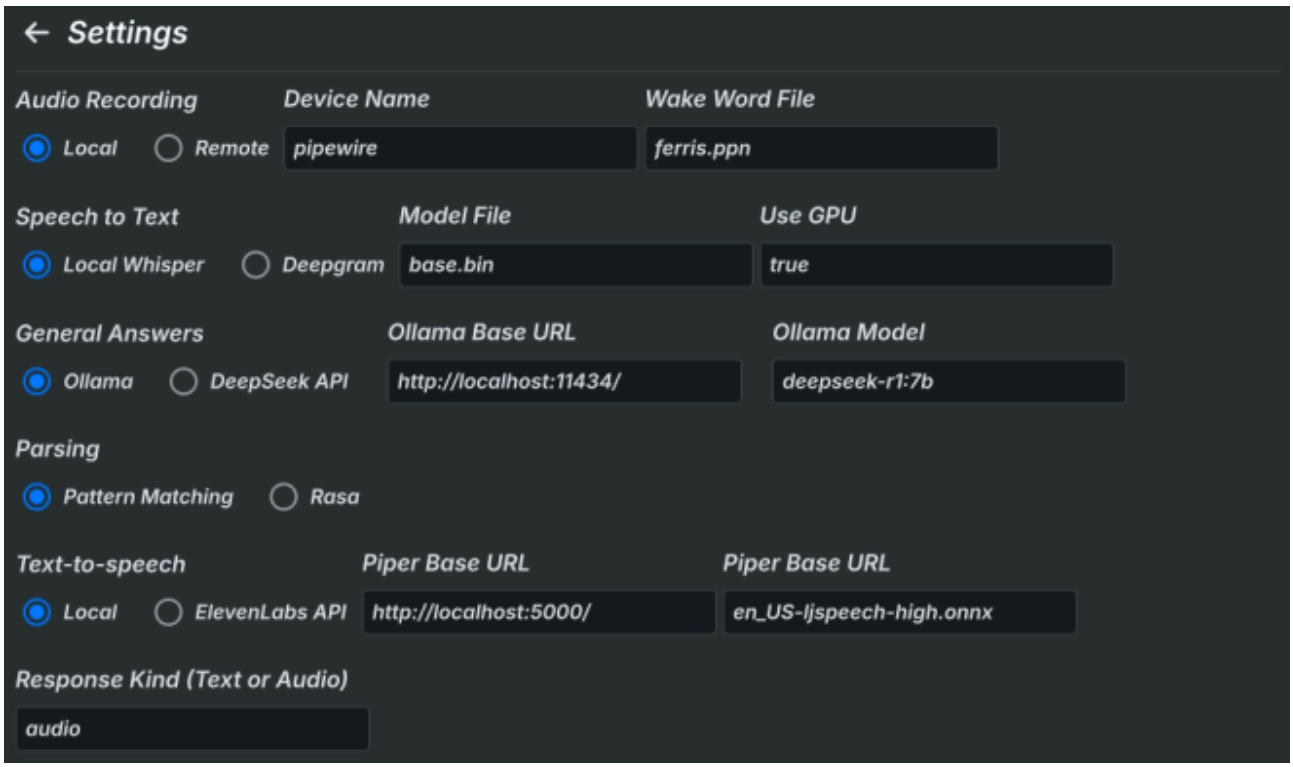

Figure 7: Final Settings UI

4 Bibliography

Bibliography

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

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

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

[6] Tom Preston-Werner, “TOML: Tom's obvious minimal language.” [Online]. Available: https://toml.io/

[7] Wikipedia, “Java Virtual Machine.” [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine

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

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

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

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

[12] Qt Group, “Qt Meta-object Language.” [Online]. Available: https://doc.qt.io/qt-6/qmlreference.html