HTBLuVA St. Pölten

Higher Institute for Electronics and Technical Informatics

Specializations in Embedded and Wireless Systems

DIPLOMA THESIS

Voice Assistant

A Voice-Controlled System Assistant

Performed in 2024/25 by:

Artemiy Smirnov
Lukas Serloth

Supervisors:

Wolfgang Uriel Kuran
Gerald Gruber

St. Pölten, on 04.04.2025

Submission Statement:
Date:

Supervisors:

Affidavit

The undersigned candidates have chosen to prepare a diploma thesis with the following task description in accordance with the Schulunterrichtsgesetz (School Education Act) § 34 Abs. 3 Z 1 and § 37 Abs. 2 Z 2 [1], in conjunction with the provisions of the Prüfungsordnung BMHS (Examination Regulations for Vocational Middle and Higher Schools), Federal Law Gazette II No. 177/2012 [2], as amended

Voice Assistant
A Voice-Controlled System Assistant

Individual tasks within the overall project:

- **Artemiy Smirnov**: Software Backend
- **Lukas Serloth**: Hardware Development and Assembly

The candidates acknowledge that the diploma thesis must be worked on and completed independently and outside of class time, although class results may be incorporated if appropriately cited as such.

The complete diploma thesis must be submitted digitally and in a printed copy to the supervising teacher no later than **04.04.2025**.

The candidates further acknowledge that cancellation of the diploma thesis is not possible.

Artemiy Smirnov _____

Lukas Serloth _____

Acknowledgments

First of all, we would like to express our sincere gratitude to our supervisor, Wolfgang Uriel Kuran, Dipl.-Ing. We were able to turn to him at any time, and he always took the time to listen. Whether we had questions or encountered challenges, he was always there to support us. The pleasant and respectful collaboration created a productive atmosphere in which we always felt comfortable and motivated. Thanks to his guidance and encouragement, we were able to successfully complete our thesis.
We would also like to thank the faculty and staff of HTL St. Pölten for providing us with an excellent education, valuable resources, and continuous support throughout our studies. Their commitment and expertise played a significant role in shaping our academic journey.
Last but not least, we are deeply grateful to our families and friends. Their patience, understanding, and constant encouragement have been invaluable throughout this entire process. Without their unwavering support, this achievement would not have been possible.

HTL ST. PÖLTEN

DIPLOMA THESIS
DOCUMENTATION

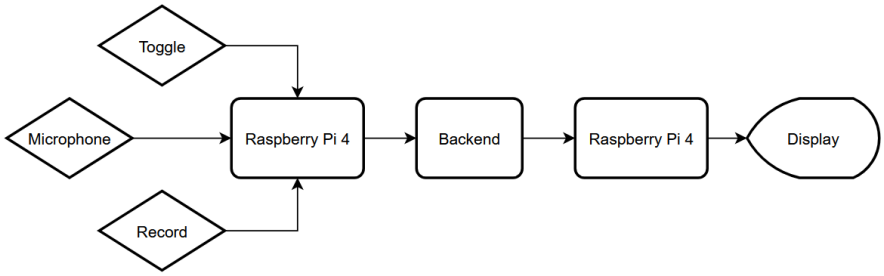
htl bildung mit zukunft

Author(s)	Artemiy Smirnov / Lukas Serloth
Form Academic year	5BHELS / 2024/2025
Topic	Rust Voice Assistant
Cooperation partners	HTL St. Pölten

Task	Create a voice assistant backend and an associated hardware frontend. The frontend should have an on/off button, a switch to enable wake word detection, and a button for explicit recording. The backend should record audio, convert speech into text, parse the text into a structured data format, evaluate the structured format, and return either audio or text depending on what the user configured in the dedicated configuration system.

Realisation	The backend was implemented using the Rust programming language, leveraging its competitive speed while not sacrificing on ergonomics. It was implemented using the microservices architecture, such that each part of the system is its own modular microservice, which can have multiple implementations that the user can pick from. The system features a complete voice assistant setup with physical controls including an on/off button, a background recording toggle, and a recording button. Microphone data is sent to a microcontroller and optionally to a connected computer, ensuring reliable performance. A custom-built case protects the circuit boards and components.

Results	The project meets all requirements. The Raspberry Pi and display are powered with a stable 5V supply from a step-down converter, which accepts either 9V or 7.4V input. This allows the system to be powered via a DC jack or two 18650 Li-Ion batteries. Both the Record and Toggle buttons function as expected. The RGB LED indicates the status of the voice assistant; it lights up with a specific color code when an error occurs and remains off otherwise. The display shows the current output along with a user interface that includes two buttons, which can be interacted with via the touchscreen.

Illustrative graph (incl. explanation)	 When the recording button is pressed, or the wake word is detected, the backend start recording. The backend may be hosted locally on the Raspberry Pi, or elsewhere on the internet. Once processed by the backend, the result is either played back if its audio or displayed if its text.

Accessibility of diploma thesis	The diploma thesis is embargoed for three years. After that, it can be viewed at HTL St. Pölten, Waldstraße 3, 3100 St. Pölten.

Approval (Date / Sign)	Examiner	Dipl.-Ing. W. U. KURAN Head of Department

Contents

1 Introduction

This project aims to deliver a versatile voice assistant implementation, offering users the ability to perform various everyday tasks hands-free and with little effort.

1.1 Background and Motivation

While voice assistants have rapidly gained popularity and their capabilities improve with each new release, many people still associate them with poor speech-to-text performance and frequent requests for repetition. This project aims to address these limitations and develop a user-friendly voice assistant solution, which is production-ready and able to greatly enchance the user's workflow while remaining unobtrusive. Furthermore, it should expand on the feature set of traditional voice assistans, offering workspace management on Unix-like systems.

1.2 Project Goals

- Efficient processing of queries with a goal of minimal response latency, optimally faster than a second.
- Integration into window managers on Unix-like systems, allowing users to control their desktop using voice commands.
- Allowing the user to activate audio recording by either pressing a button, or saying a wake word.
- Extensive configuration, allowing the user to pick and choose every detail of the system.

1.3 Naming

Any modern voice assistant should have a name and a wake word it reacts to; because of the later implementation of Rust in the project, "Ferris" was chosen as the name of the voice assistant, referencing the mascot of the Rust programming language which goes by the same name.

2 Concepts and Theory

It is important to understand the main concepts behind the voice assistant, to be able to properly comprehend the explanations that follow further:

2.1 Voice Assistant

A voice assistant is a software application that enables users to receive assistance and interact with their system using voice commands. using voice commands. More technically, it is a voice-controlled system that utilizes speech recognition and natural language processing to process user input and respond to it, be that by playing audio, displaying text, or executing actions.

2.2 Artificial Intelligence

Artificial Intelligence[1] is a broad subset of computer science with its main focus being on creating systems capable of performing tasks, which would usually require the human mind — or natural intelligence. Such tasks include things like reasoning, making decisions, learning from big datasets and understanding language. Over the years, many approaches to creating artificially intelligent systems have been created, from the simplest rule-based parsers and logical reasoning to more modern techniques featuring learning processes on giant datasets scraped from the internet.

2.3 Machine Learning

Machine Learning[2] is a subset of artificial intelligence that aims to develop algorithms and methods which allow computers to increase their reasoning performance over time without having to reprogram them constantly. There are a lot of machine learning techniques, some more effective than others, but it usually boils down to learning from big datasets and finding patterns in the data which allow the model to make predictions based on those patterns.

[1] Artificial Intelligence [3]
[2] Machine Learning [4]

2.4 Natural Language Processing

Natural Language Processing[3] is a subfield of artificial intelligence and is closely related to the field of computational linguistics[4]. The goal is to automatically understand human language and translate it into a structured form, which can be achieved using various approaches, ranging from simple pattern matching[5] to more advanced systems that utilize machine learning.

2.5 Large Language Model

A large language model[6] is a more advanced type of machine learning system; it is designed to ”understand” natural language at scale, so it is a sort of natural language processing system. They are trained on vast amounts of text, usually obtained by scraping the internet, which enables them to produce coherent, human-like conversation. A large language model is an advanced type of machine learning system designed to understand and generate natural sounding language at scale.

[3]Natural Language Processing [5]
[4]Computational Linguistics [6]
[5]Formal Grammar [7]
[6]Large Language Model [8]

3 Technologies

This chapter provides an overview of the key technologies used in the project. Each technology was chosen for its unique benefits in addressing specific project requirements, from rapid prototyping to high-performance, resource-constrained deployments.

3.1 Programming Languages

This section examines the programming languages employed in the project, outlining their strengths and the context in which they were used.

3.1.1 Kotlin

Kotlin[1] was initially selected for its seamless interoperability with Java. This allowed rapid prototyping by leveraging the extensive Java ecosystem, including existing libraries, frameworks, and development tools. However, after further evaluation and a small Kotlin prototype, it was decided that Kotlin would not be the best choice for this project. The evaluation indicated that as the project scaled with many microservices, Kotlin's performance would likely be insufficient to meet the 1-second response time goal. This led to the consideration and eventual adoption of Rust.

3.1.2 Rust

Rust[2] was adopted for its ability to deliver C-like performance while ensuring memory safety through modern tooling and a robust ownership model. Its compile-time guarantees help avoid common pitfalls such as null pointer dereferences and data races. These features make Rust especially well-suited for managing resource-intensive workloads and scaling microservices on low-end hardware, such as a Raspberry Pi, without sacrificing efficiency or productivity.

[1] Kotlin [9]
[2] Rust [10]

3.2 Frameworks

This section details the frameworks integrated into the project, each chosen to support specific architectural patterns and enhance development efficiency.

3.2.1 Spring

The Spring Framework[3] was chosen for its strong support for building microservices and its deep integration with the Java ecosystem. Built around the Model-View-Controller (MVC) architecture, Spring offers a well-organized structure for developing web applications, which aligns seamlessly with the project's design objectives.

3.2.2 Tokio

Tokio[4] is an asynchronous runtime for Rust that simplifies the development of non-blocking, concurrent applications. Its ability to manage asynchronous I/O operations efficiently makes it ideal for preventing busy waiting, ensuring that the project can handle high concurrency and maintain optimal performance even under heavy load.

3.3 Utilities

These are utilities, such as libraries or other tools that improve quality of life but aren't strictly necessary.

3.3.1 Serde

`serde`[5] is a framework for serializing and deserializing Rust data structures efficiently and generically. It was used to convert json responses to Rust structs for ease of use and safety.

[3] Spring [11]
[4] Tokio [12]
[5] Serde [13]

3.3.2 Reqwest

`reqwest`[6] is a high-level HTTP client library for Rust. It provides an ergonomic interface for making HTTP requests while leveraging the async/await syntax supported by Tokio. The library was chosen for its robust error handling, built-in JSON support through Serde integration, and its simple yet powerful API design.

3.3.3 Tokio-tungstenite

`tokio-tungstenite`[7] is a wrapper that provides Tokio bindings for `tungstenite-rs`, which is a Rust WebSocket library. WebSockets were used extensively in this project, to communicate between backend and frontend as well as for streaming input/output to remote APIs to reduce latency.

3.3.4 CPAL

`CPAL`[8] is a low-level cross-platform audio I/O library for Rust. It provides a pure-Rust interface for audio playback and recording across different operating systems. The library was chosen for its cross-platform compatibility, low overhead, and native integration with Rust's async ecosystem, making it ideal for handling real-time audio processing requirements in the project.

3.3.5 Whisper-rs

`whisper-rs`[9] provides Rust bindings to OpenAI's Whisper model through its C++ implementation. The library enables local speech-to-text transcription without requiring external API calls or cloud services. Integration with the project's Rust codebase is handled through a safe interface to the native C++ code, while maintaining memory safety and proper resource management.

3.3.6 Porcupine

`porcupine`[10] is a wake word detection library built by Picovoice for embedded and IoT applications and is therefore very lightweight and perfect for a Raspberry Pi. It has Rust bindings via a cargo crate and a very simple API allowing for easy integration.

[6] Reqwest [14]
[7] tokio-tungstenite [15]
[8] CPAL [16]
[9] Whisper-rs [17]
[10] Porcupine [18]

3.4 Programs

Figure 3.1: Logo Altium Designer

Altium Designer is a powerful electronic design automation tool used for creating schematics and PCB layouts. It provides an integrated environment for designing, simulating, and documenting electronic circuits. Both the schematic and the PCB were developed using Altium Designer.

3.4.1 Autodesk Fusion

Figure 3.2: Logo Autodesk Fusion

Autodesk Fusion 360 is an integrated CAD/CAM tool that brings together design, engineering, and manufacturing on one platform. For the diploma, the enclosure was designed using Fusion.

4 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.

4.1 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	04.10.2024
Functioning software prototype (e.g., weather querying)	01.11.2024
Components and PCB ordered	15.11.2024
Hardware and software communication functioning	13.12.2024
PCBs completed	10.01.2025
Case completed	31.01.2025
Interface with LLM	31.01.2025
Final prototype completed	28.02.2025

Table 4.1: Project Milestones

4.2 System Architecture

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

4.2.1 Client-Server Architecture

In the following flowchart, the frontend is depicted as the client and the backend as the server, comunicating via a protocol that is yet to be decided depending on which ends up fitting the implementation best.

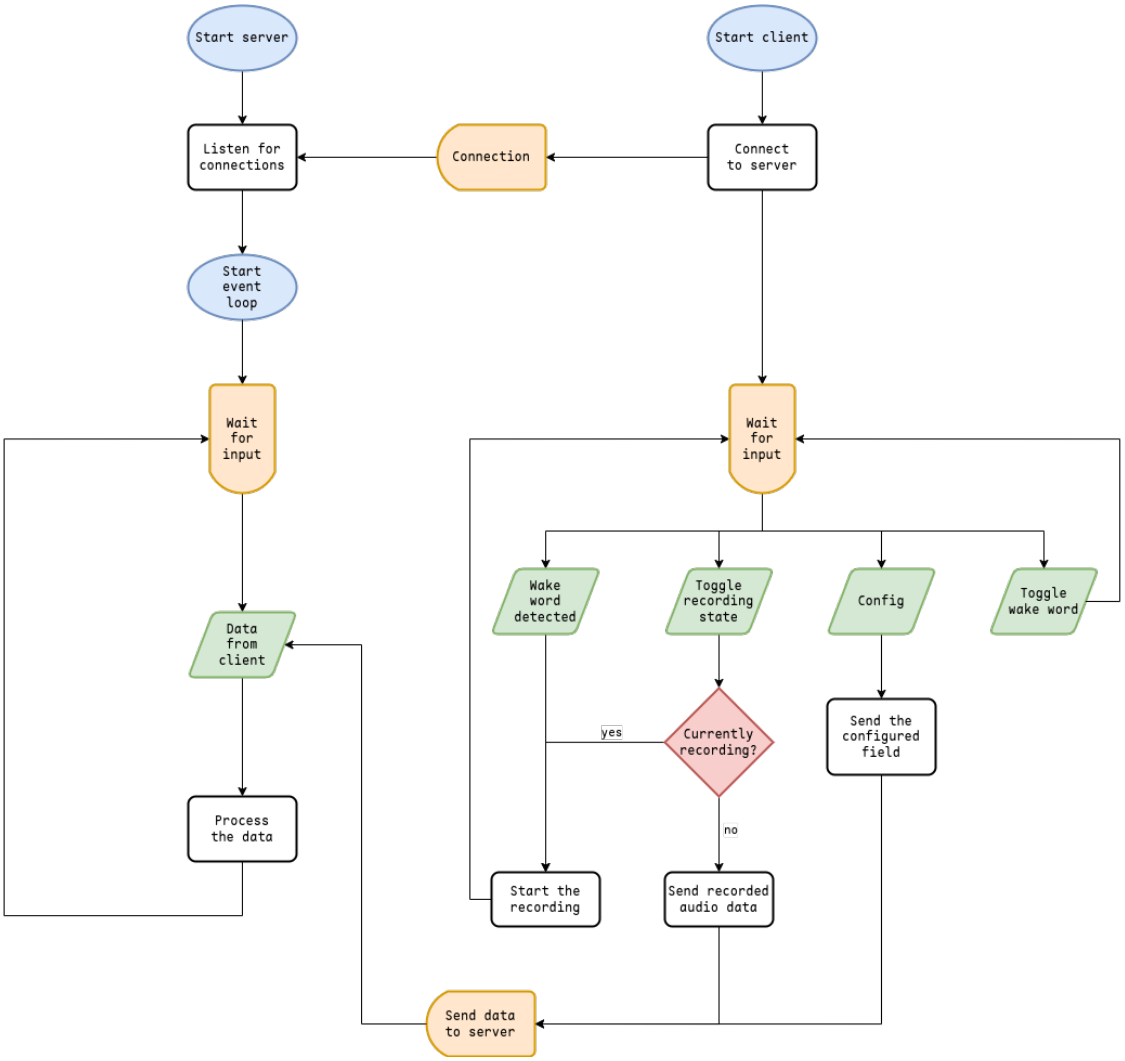


Figure 4.1: Client-Server Architecture

The system starts with the backend listening for connections, and the client connecting to the backend. The word connections is plural here, because the backend is intended to be able to handle multiple clients simultaneously; having multiple configurations running at the same time would be quite a challenge to implement, and it is not planned, especially because multiple clients would usually

just mean multiple smart home nodes, which run on the same or similar hardware and do not need varying configurations. Once a client has connected, it should handle recording audio by itself, be that using buttons or a wake word and silence detection; the client would send the audio to the server once finished, and the server would process it — this is the event loop of the system.

4.2.2 Configuration

The client can send configuration commands, which will be selected from a menu on the client interface and saved into a configuration file on the backend. TOML[1] would be a great option because it allows for a simple table-key-value structure, allowing it to split the configuration for different parts of the system (such as recording, transcription, or parsing) while still being able to store key value pairs like most other configuration formats.

4.2.3 Frontend

There does not need to be a frontend for the system to function, besides for recording and playing audio; technically, all that is needed for a frontend is an audio recorder/player and/or a text display and a client that can connect to the backend. For testing and debugging purposes, and for the completeness of the project, a reference implementation will be created that is part of a school project and outside of the scope of this thesis. It will be written in Qt/C++ and designed to support every feature that the backend offers, including manually recording, using a wake word and silence detection, configuring the system, playing back audio as well as displaying text. The source code of this frontend will be hosted on GitHub[2].

4.2.4 Language and Framework

The backend will be written using Kotlin and the Spring Framework, utilizing the extensive feature set, documentation, and support the framework and the language both have. Spring Boot is a Java-based framework that allows developers to build huge stand-alone applications using any language that compiles to the JVM, which are mainly Java, Kotlin, Groovy and Scala; however only Java and Kotlin are supported and maintained out of the box. Much of its configuration is handled automatically, living by a concept called ”convention over configuration” where the user only needs to do minimal configuration to get a working

[1] Tom’s obvious minimal language [19]
[2] Voice Assistant reference frontend repository [20]

application. Out of the supported JVM languages, Kotlin was chosen for its modern approach to the JVM. It is statically typed, null safe by default, can be both functional and object-oriented and has lots of ergonomic syntax sugar.

4.2.5 Microservices

A major issue in many large applications is the monolithic structure of the software, where everything is tightly-coupled and changing individual bits of the program often requires extensive and breaking changes in the event loop and other parts of the program which should be compeletly unrelated. This is where the microservices architecture[3] comes into play; this type of architecture organizes the program into many loosely-coupled parts which can each have various different implementations to allow the user to choose between them.
The following flowchart shows how this structure is planned to be implemented; the event loop mentioned in the previous flowchart 4.1 is implemented here and can be inserted in place of the previous event loop, which only contained a brief ”process the data” function.

[3]Microservices Architecture [21]

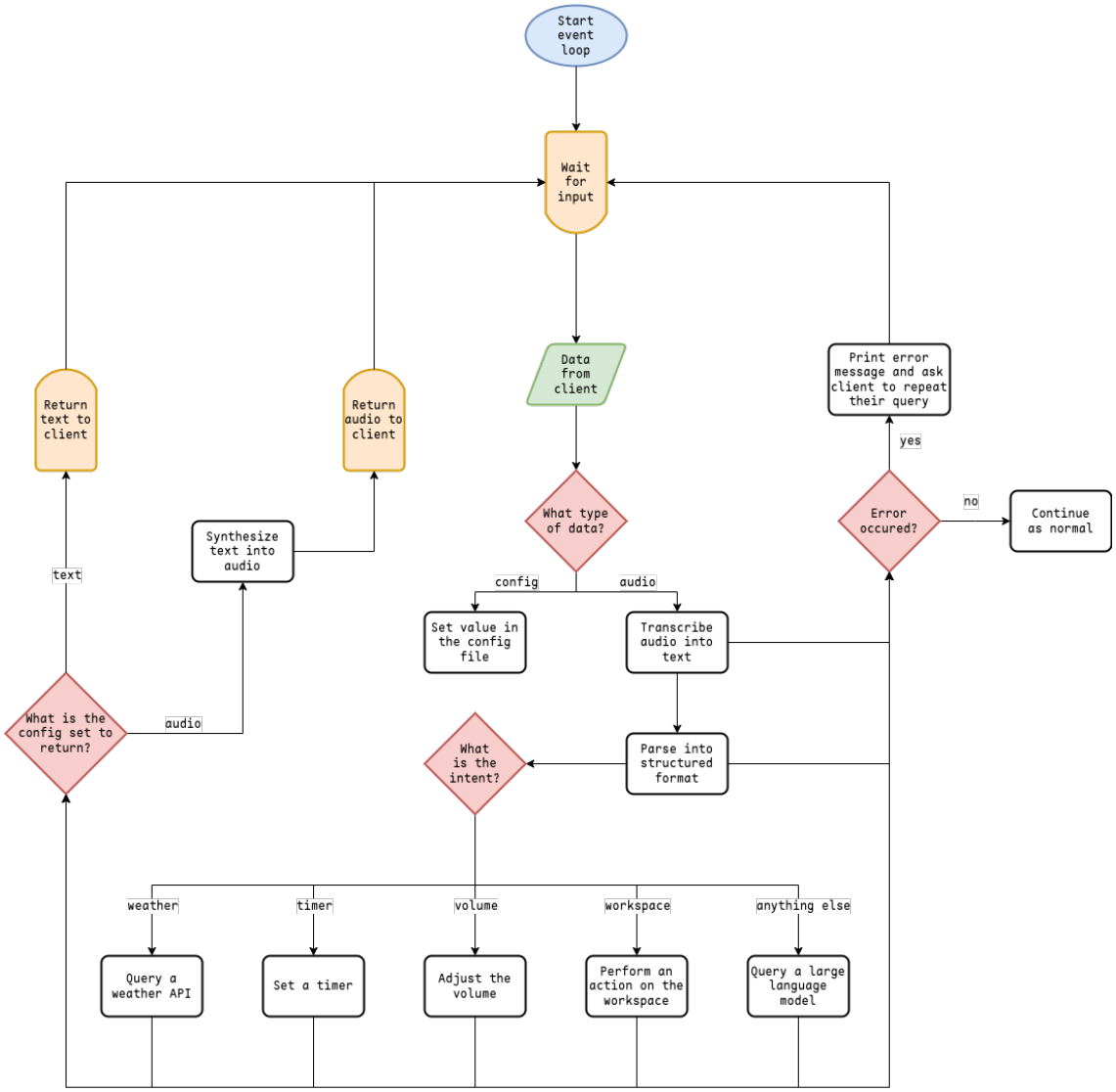


Figure 4.2: Processing of audio

The process starts by receiving data from the client, where this can be either audio or a configuration. Depending on the type of data, different things are done:
If it is a configuration, then the configuration file is updated with the new value that was requested by the client.
If the data is audio, then it is processed as input; audio data is first transcribed into text using a speech-to-text microservice. After trascription, the text is parsed using the parsing microservice, which can either be a pattern matching parser or a machine learning parser, both of which perform natural language processing on the input text to transform it into a structured format that a computer can un-

derstand and use in the next step. Going further, the structured format contains an intent, which, with a relatively high likelihood, is the action, which the user wants the system to perform. The plan is to implement the most important actions which a voice assistant should have (Querying the weather, setting a timer, moving windows on the computer, adjusting the volume, and asking an LLM), and, if there is enough time remaining, implement more actions or allow the user to implement their own, using something akin to a YAML[4] configuration, allowing them to define which system commands should be executed, how the voice assistant should respond or which APIs should be called when the action is performed. After parsing everything into a simple structured action, the action will be put through a pattern matching statement, such as a switch-case or an if-else chain, where the appropriate microservice will be invoked, depending on which intent the user most likely had. Once the invoked microservice is done processing, its output is sent back to the client, either as text or as audio, depending on what response type the user has configured.

[4]YAML Ain't Markup Languagež122 [22]

4.3 Hardware Overview

The voice assistant is built to run on a Raspberry Pi 4, chosen for its small form factor and flexibility. While not especially powerful by modern standards, the Raspberry Pi 4 offers enough performance to handle voice input and process the user's request using remote APIs. Nevertheless, it is still possible and perfectly acceptable to run the voice assistant on any other Unix-like system, whether that's a work computer or a more powerful single-board computer made to run machine learning models locally.
The system is designed to work either with a regular power supply or a rechargeable battery; adding a battery is crucial to keep the system active during power outages.

4.3.1 User Interface and Hardware Components

The printed circuit board includes an RGB LED to signal the current state of the system, be that lighting up green when recording, or red when an error occurs. The user can start a recording in three different ways:

- Press record button on the touchscreen
- Press the dedicated physical recording button
- Say the wake word (if enabled)

There is also a button — both on the touchscreen and a physical button — that allows the user to toggle wake word detection.

4.3.2 Display and Visual Feedback

A 10.1-inch LCD display should be connected via HDMI to the Raspberry Pi — this display allows text output and touchscreen input, especially for configuring settings.

5 Implementation of the Software Backend

5.1 Communication

Communication between client and server is important to get right; Initially, HTTP was used with plain text messages, under the assumption that the frontend would record audio. The frontend should record audio and send it to an HTTP endpoint, where it can be further processed — although the first implementation of the backend had an HTTP post endpoint expecting plain text, to ensure that the team can iterate quickly and to keep it simple at first.

5.2 Spring Boot Prototype

The backend was initially developed using Spring Boot, as previously described in 4.2.4. 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

5.3 Rewriting it in Rust

Because Kotlin runs on the Java Virtual Machine[1], 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 Java Virtual Machine.
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 using the `Result` enum:

[1] Java Virtual Machine [23]

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

And by utilizing its powerful pattern-matching capabilities, this becomes an awesome way to avoid almost all runtime errors in a simple and elegant way.
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 are more powerful than in other programming languages, as they can hold multiple values, and even named fields, not just singular primitive types. Using this concept, the following is possible:

```
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 can then be used in `match` statements, as well as the `if let` statement for really powerful and fun-to-use pattern matching.
Interfaces are a very common thing in programming languages nowadays — Rust does not have interfaces, it has traits: they are very similar to interfaces in the sense that they allow specifying that a certain struct implements a certain feature, or multiple certain features and allows for the a trait to be a function parameter, not a specific implementation. However, traits differ with scoping and the defition of which traits are implemented for a certain struct: to implement an interface for a certain class, one has to specify that the class implements that interface, however, to implement a trait, one can just write `impl Trait for Struct` and then import the trait wherever it needs to be used.
Another elevator pitch of Rust is its focus on memory safety; Rust has a system called the `Borrow Checker`, which does what you think it does: It checks borrows — Rust has a concept called `Ownership and Borrowing`, where a variable always

has an owner and can either be immutably borrowed any number of times or mutably borrowed once at the same time. Borrowing a value means holding a reference to it, but not owning it, hence not being able to pass it as an owned value to another function without first cloning it. Another feature of the borrow checker are lifetimes: Any variable is borrowed only until its lifetime expires and any value a variable holds must live at least as long as it, otherwise a compile-time error will be thrown.
At first, the borrow checker may seem excessively limiting; however, once properly understood, it provides an almost perfect guarantee of memory safety and prevents data races (except when the unsafe keyword is explicitly used). Developers sometimes must bypass these safeguards for operations that cannot be done safely — such as writing to a specific register on a microcontroller to control an LED or set up an interrupt service routine.

5.4 Audio Recording

The original plan for audio recording was to require any frontend implementations to record audio and send raw audio data to the backend. Attempts were made to implement this in the reference frontend using Qt/C++, but these efforts were unsuccessful — implementing an audio recorder in Qt requires extensive knowledge of the underlying operating system and audio drivers, and the documentation was insufficient to cover every edge case. This led to the idea that audio recording could be a microservice like any other, but it would have to be synchronous, since audio recording is inherently a blocking process — you can not do anything while recording and you cannot record twice at the same time using the same device. However, one could argue that if multiple clients — for example, a collection of smart home nodes, each with their own microphone — wish to use the audio-recording microservice remotely, then an asynchronous implementation would be required. The issue was that a microservice uses a Rust trait as its specification, and this trait can not support synchronous and asynchronous implementations simultaneously. A decision had to be made: Is it better to wrap the local recording implementation into an asynchronous API, or make the asynchronous implementation blocking. Ultimately, while blocking the asynchronous remote recorder would be much easier, it would also heavily slow down a system with multiple clients connected, so the decision was made to turn the API of the local recorder asynchronous.

The following flowchart demonstrates the new system design, with audio recording and wake word detection on the backend:

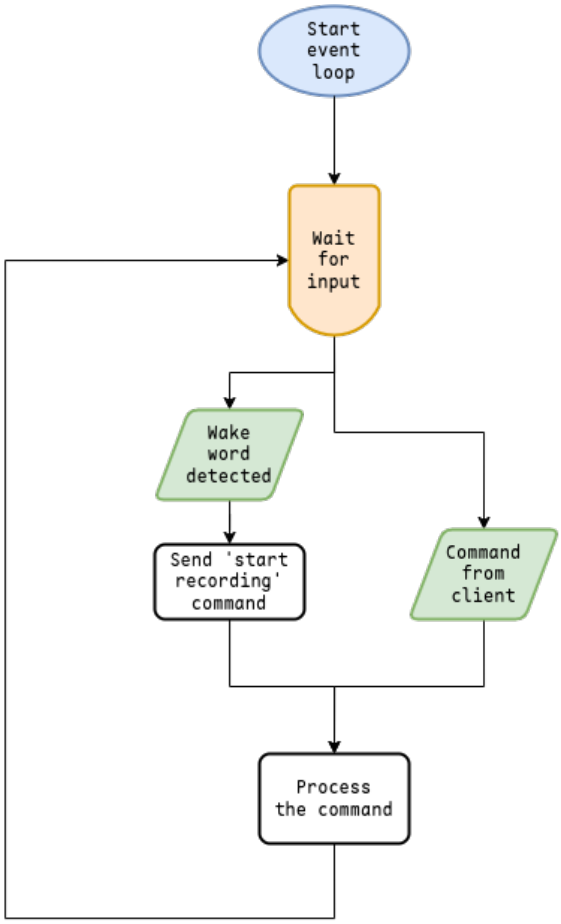


Figure 5.1: New audio system

With the new system, the recording happens on the backend, as does the wake word detection — to achieve this, a recording microservice was created, using the following trait:

```
pub trait RecordingService {
    async fn start(&self) -> Result<()>;
    async fn stop(&self) -> Result<Bytes>;
}
```

5.4.1 Protocol Updates

With the new audio recording redesign, rapid full-duplex communication was required, making HTTP no longer a viable option. While it is technically possible to achieve full-duplex communication using techniques such as Server-Sent Events or WebSockets over HTTP, switching to raw TCP proved to be much simpler and more effective. Raw TCP communication allows both the server and client to send messages simultaneously — for example, the server could send a response while the client is sending a request to start a new recording.

5.4.2 Local Recorder Implementation

CPAL[2] is a Rust library that provides easy access to the audio interface of the system, allowing one to record audio from Rust code. Using Hound[3], the output of CPAL could be encoded into a WAV file to be sent to the transcription microservice. However, integrating CPAL's inherently synchronous and non-thread safe `Stream` into an asynchronous API proved to be challenging, as Rust does not allow synchronous code to run inside asynchronous context, due to concerns for memory safety and avoiding data racing.

5.4.3 Temporary Unsafe Implementation

CPAL's `Stream` is not thread-safe because it does not implement the `Send` and `Sync` traits. A very intuitive but improper way to circumvent the constraints is to simply implement an unsafe wrapped for `Send` and `Sync`; this was done as an interim stopgap measure to allow work to be done on other parts of the project while deliberating about how to implement a proper asynchronous wrapper. This inherently violates the compiler's thread-safety guarantees, but is acceptable as long as there is only one instance of the local recorder (so only one client).

```
pub struct UnsafeSendSync<T>(pub T);
unsafe impl<T> Sync for UnsafeSendSync<T> {}
unsafe impl<T> Send for UnsafeSendSync<T> {}
```

[2]Cross-Platform Audio Library [16]
[3]Hound [24]

5.4.4 Isolation of Synchronous Recording

The day after implementing the `UnsafeSendSync` wrapper for CPAL's `Stream`, it was decided to simply isolate the `Stream` into its own dedicated synchronous thread, separate from the async interface, and communicate using a shared buffer, which could safely be sent across threads./ This buffer was a 10-minute long ring buffer, which means that after 10 minutes of recording, it would overwrite the first samples of the recording. 10 minutes was chosen as the cut-off because it seems unrealistic that someone would talk for more than 10 minutes straight to a voice assistant. Access to this buffer is synchronized using a mutex, and an atomic counter that keeps track of the total number of samples that have been recorder thus far. Errors that are thrown inside the synchronous thread are sent to the asynchronous interface via a Tokio multi-producer single-consumer channel and propagated upstream.

5.4.5 Wake Word Detection

A wake word, typically the name of the voice assistant (for example, "Hey Siri" or "Okay Google"), allows hands-free activation—making the assistant significantly more convenient to use. This project utilizes the Porcupine[4] library for wake word detection, and "Ferris" as the wake word, which is the name of the voice assistant, as mentioned in 1.3. The detection process is implemented in the following code:

```
fn process_audio_data(
    data: &[f32],
    frame_buffer: &mut Vec<i16>,
    porcupine: &Porcupine,
    wake_word_enabled: &Arc<AtomicBool>,
    is_recording: &Arc<AtomicBool>,
) {
```

This function accepts raw audio data and maintains a frame buffer. Two atomic booleans — which are thread-safe wrappers for booleans — namely `wake_word_enabled` and `is_recrding` are passed to this function.

```
for &sample in data {
    let sample_i16 = (sample * 32767.0) as i16;
    frame_buffer.push(sample_i16);
}
```

[4]Porcupine [18]

Audio is stored in floating point samples, which are an easy format to use for WAV files. These samples need to be converted to 16-bit integers, because Porcupine expects 16-bit, pulse code modulated signals. Each converted sample is added to the frame buffer for processing.

```
if frame_buffer.len() >= porcupine.frame_length() as usize {
    if let Ok(keyword_index) = porcupine.process(frame_buffer) {
```

When the buffer has reached a sufficient size, it is processed by Porcupine; the process method returns a keyword index which can then be used to determine what the outcome was.

```
if keyword_index >= 0 && wake_word_enabled.load(Ordering::Relaxed) {
    is_recording.store(true, Ordering::Relaxed);
}
frame_buffer.clear();
```

A non-negative keyword index means that a wake word was successfully detected. If wake word detection is enabled, the system transitions to recording mode. The frame buffer is cleared afterward to prepare for the next batch. This cycle continues, allowing continuous monitoring for wake word occurrences.

Below is a flowchart overview showcasing the overall audio recording system, including wake word detection:

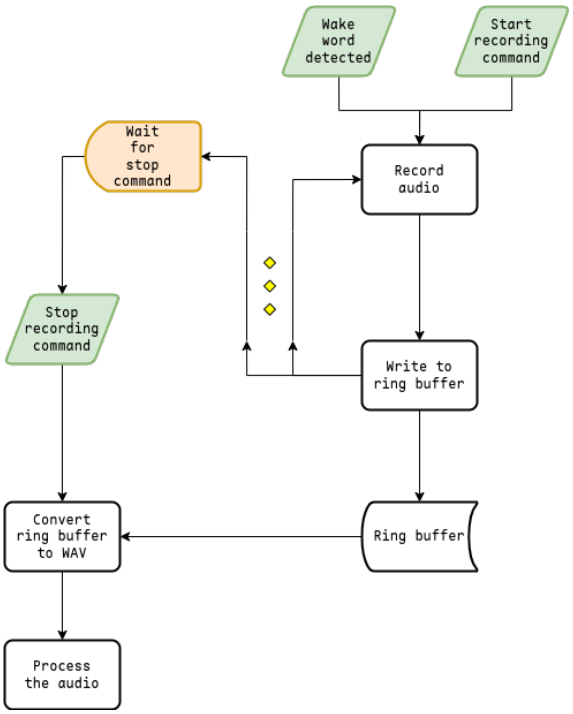


Figure 5.2: Audio recording system and wake word detection

Recording starts when the wake word is detected or the `start recording` command is received. Any recorded audio data is written into a ring buffer, while the `stop recording` command is awaited in parallel. When the recording is stopped, the captured audio data is extracted from the ring buffer, converted into WAV format, and sent to be transcribed by speech-to-text.

5.5 Protocol Updates

Raw TCP made it difficult to separate text from binary data, often requiring extra logic or a custom protocol. 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.

5.6 Speech-to-Text Transcription

5.6.1 Audio Preprocessing

Raw audio is often filled with noise and inconsistent volume levels. Preprocessing techniques — like spectral subtraction — can help isolate vocal frequencies and reduce background noise. Volume is then normalized to a consistent value. The cleaned up audio is then transformed into numerical features[5] using methods such as the Mel Frequency Cepstrum[6] or spectrograms.

5.6.2 Neural Network Analysis

The preprocessed features are then input into neural networks trained on thousands of hours of voice data, which map acoustic signals to text. There are two main types of neural networks used for audio processing, those being recurrent neural networks[7], particularly long short-term memory[8] variants, which process audio sequentially, retaining context over time, as well as transformers, using self-attention[9], which in turn model relationships across the entire sequence in parallel. Hybrid models can combine convolutional layers for phoneme recognition with transformers for better and broader interpretation.

5.6.3 Decoding Ambiguity

Neural network outputs are inherently non-deterministic, which introduces some amount of ambiguity in the transcribed text. Decoding algorithms help resolve some uncertainty, integrating statistical language models that leverage context to select the most likely transcription. When dealing with variable-length inputs and timing discrepancies, connectionist temporal classification[10] is applied to collapse repeated predictions and blank sections, which provides a preliminary aligned sequence. This output is then further refined by attention-based models, which focus on ambiguous segments to provide a more accurate and coherent end result.

[5] Feature in machine learning [25]
[6] Mel Frequency Cepstral [26]
[7] Recurrent Neural Network [27]
[8] Long short-term memory [28]
[9] Self-attention [29]
[10] Connectionist Temporal Classification [30]

5.6.4 Rust Trait

The transcription service is defined via a Rust trait:

```
pub trait TranscriptionService: Send + Sync {
    async fn transcribe(&self, audio: &Bytes) -> Result<String>;
}
```

5.6.5 OpenAI Whisper

OpenAI Whisper is an open-source implementation of automatic speech recognition. The following is a description by OpenAI[11].
"""Whisper is an automatic speech recognition (ASR) system trained on 680,000 hours of multilingual and multitask supervised data collected from the web. We show that the use of such a large and diverse dataset leads to improved robustness to accents, background noise and technical language. Moreover, it enables transcription in multiple languages, as well as translation from those languages into English. We are open-sourcing models and inference code to serve as a foundation for building useful applications and for further research on robust speech processing."""
Integration in Rust uses the whisper-rs library:

```
pub struct LocalWhisperClient {
    context: Arc<WhisperContext>,
}

impl LocalWhisperClient {
    pub fn new(model: impl Into<String>, use_gpu: bool) -> Result<Self> {
        let mut params = WhisperContextParameters::default();
        params.use_gpu = use_gpu;
        let context = Arc::new(
            WhisperContext::new_with_params(&model.into(), params)?
        );
        Ok(Self { context })
    }
}
```

This code instantiates a new `LocalWhisperClient`, using all default parameters besides enabling GPU processing if the user has configured it to be enabled. The GPU is much faster than the CPU when running machine learning models, hence why it can be used.

[11] OpenAI Whisper [31]

5.7 Parsing

Parsing is the process of analyzing text input and converting it into a structured format representing its meaning, or the intent that the user most likely had — in order to determine which actions should be performed. This can be accomplished using natural language processing, be that a simple rule-based parser or a more advanced natural language understanding (NLU) machine learning system. Rule-based parsing involves matching specific patterns in the input — using techniques such as regular expressions or Rust's `match` statement — to extract structured data. In contrast, NLU-based parsing leverages machine learning models, which can be described to actually "understand" the underlying natural language, hence being able to provide much better results when it comes to intent parsing.

5.7.1 Action

An `Action` encapsulates the parsed result, including intent, entities, and the original input text:

```
pub struct Action {
    pub intent: Intent,
    pub entities: Vec<Entity>,
    pub text: String,
}
```

An IntentKind is the actual intent that the parser thinks the user wants to perform, while the confidence is the probability of this intent being correct. Since a rule-based parser is a deterministic pattern matching function, it does not have a confidence score, hence why `confidence` is wrapped in an `Option`.

```
pub struct Intent {
    pub name: IntentKind,
    pub confidence: Option<f32>,
}

pub enum IntentKind {
    LlmQuery,
    SetTimer,
    WeatherQuery,
    DecreaseVolume,
    IncreaseVolume,
    SetVolume,
    CloseWindow,
```

```
    MaximizeWindow,
    MinimizeWindow,
    SwitchWorkspace,
    ShowDesktop,
    Other(String),
}
```

5.7.2 Pattern Matching

A rule-based parser was implemented using Rust's `match` statement. It operates very similar to the previosly shown Kotlin `when` statement, detecting words and patterns in the input. Below is a simplified demonstration of this `match` statement.

```
match input_lower.as_str() {
    x if x.contains("close") => CloseWindow

    x if x.contains("minimize") => MinimizeWindow

    x if x.contains("maximize") => MaximizeWindow

    x if x.contains("timer") || x.contains("alarm") => SetTimer

    x if x.contains("switch") && (x.contains("workspace")
    || x.contains("desktop")) => SwitchWorkspace

    x if x.contains("volume") && x.contains("increase") => IncreaseVolume

    x if x.contains("volume") && x.contains("decrease") => DecreaseVolume

    x if x.contains("volume") && (x.contains("set")
    || x.contains("adjust")) => SetVolume

    x if x.contains("weather") || x.contains("whether")
    || x.contains("heather") => WeatherQuery
    _ => LlmQuery
}
```

5.7.3 NLU-Based Parser

An NLU-based solution was implemented using Rasa[12] for more adaptable parsing. Rasa offers dialogue handling, however, for this project, only the NLU component is used, and dialogue handling may be considered in the future. Text inputs are tokenized and undergo feature extraction before being processed by the DIETClassifier, a transformer model that performs both intent classification and entity extraction.
Entities are extracted using the BIO tagging scheme[13], enhanced with spaCy's `SpacyEntityExtractor`. spaCy models perform named entity recognition using statistical techniques trained on corpora such as OntoNotes. The final output includes the identified intent with a confidence score and a list of labeled entities. This output is then transformed into an `Action` object for downstream use.

5.8 Geocoding

The geocoding service translates text-based location names into geographic coordinates. It creates `GeocodeResponse` objects, which can later be used for weather queries. Each request requires latitude, longitude, and an API key, with optional parameters for `units`, `exclude`, and `lang`. An implementation of the geocoding service was a Nominatim[14] client; Nominatim is the internal search engine used by OpenStreetMap to convert addresses into coordinates, and they provide a public API for it, which is available for free and works quite well, so there was no need for another implementation.

5.9 Weather

Querying the weather is a typical feature of voice assistants, so it was implemented here as well; the people at OpenWeatherMap are generous enough to provide 1,000 free calls to their weather API every day, so there was no need to set up a payment system or create another implementation of the weather service. The weather service is one of the simplest in the project — it queries the OpenWeatherMap API and deserializes the response into Rust structs using the serde library.

[12] Rasa [32]
[13] Inside-outside-beginning tagging [33]
[14] Nominatim [34]

5.10 Timers

Timers are among the first features associated with a voice assistant, making proper implementation essential. Timer functionality is achieved by spawning a new Tokio task that waits for the specified duration; upon expiration, a desktop notification is sent to indicate that the timer has finished.

```
pub trait TimerService: Send + Sync {
    async fn set(&self, duration: Duration, description: String) -> Result<String>;
}
```

Each timer requires a `Duration` and a description, which specifies the output message when the timer completed.

5.11 Raspberry Pi Setup

The voice assistant was designed to be deployed on a Raspberry Pi, whether it's a dedicated node in a smart home or just a helpful assistant on someone's desk. Given the Raspberry Pi's very limited hardware resources, particular care had to be taken to reduce its local computational load. Where possible, services were designed with remote execution in mind, to allow them to run on low-end hardware. Alpine Linux[15] was chosen as the operating system for its minimalism, fast boot times, and low memory usage.

5.11.1 Installation Procedure

To install the system, an aarch64 Raspberry Pi image was downloaded from the Alpine Linux website and then flashed onto a microSD card using `dd`:

```
sudo dd if=alpine-rpi-*.img of=/dev/sdX bs=4M status=progress
```

The built-in `setup-alpine` script was executed to configure basic settings on the system. The script guided configuration of essential system settings including: keyboard layout, hostname, timezone, network configuration, and partitioning. KDE Plasma was then installed to provide a full desktop environment:

```
apk add plasma-desktop sddm xdg-utils
rc-update add sddm
```

This choice was deliberate, since the reference frontend is written in Qt, which is well integrated into KDE Plasma.

[15] Alpine Linux [35]

Connecting the RGB LED and physical buttons was done using the RPPAL[16] Rust library — if a recording was active the LED would light up green and if an error occured it would light up red. This could be configured further, but is not neccessary because errors are reported via text or audio either way.

[16]Raspberry Pi Peripheral Access Library [36]

6 Implementation of the Hardware Frontend

6.1 Schematic v1.0

A schematic was designed in Altium to serve as an initial prototype and to set up the project.

6.1.1 GPIO Header

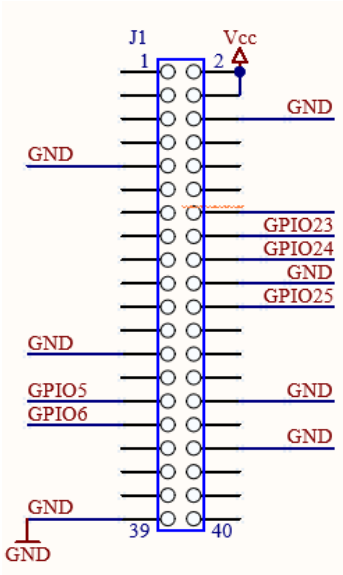


Figure 6.1: 40-pin header designed for the Raspberry Pi 4

This connector is a 40-pin header that can be plugged directly onto the GPIO pins of the Raspberry Pi 4. The design allows the connector to be attached directly to the GPIO header of the Raspberry Pi, saving space and eliminating the need for extra cables. It is also very practical, as the board can be easily removed from the Raspberry Pi 4 at any time. The Raspberry Pi 4 is powered with 5V. This input voltage is then regulated evenly later on.

6.1.2 LED Control Circuit

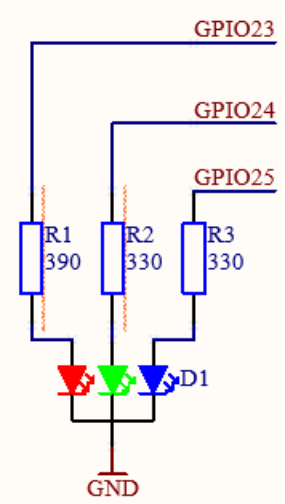


Figure 6.2: LED control circuit connected to GPIO pins

In this circuit, an RGB LED is connected to the Raspberry Pi. This LED shows the current status of the voice assistant. For example, when the recording is active, the LED should light up green, and when an error occurs, it should light up red — Each color of the LED is connected to a GPIO pin through its own resistor. These resistors make sure that not too much current flows into the Raspberry Pi or the LED. The red, green, and blue LEDs are connected to GPIO23, GPIO24, and GPIO25. A common cathode RGB LED is used, so it is also connected to ground.

6.1.3 Push-Button Circuit

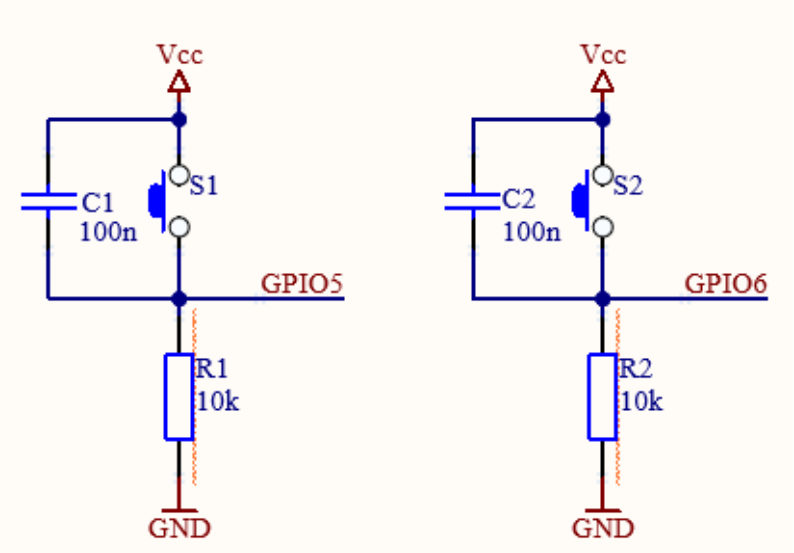


Figure 6.3: Push-button circuits for Record and Toggle functionality

The circuit has two buttons connected to the Raspberry Pi 4. Each button is connected to a specific GPIO pin and controls a function of the voice assistant. Button S1 serves as the Record button. When pressed, it starts the recording; pressing it again stops the recording. Button S2 is the wake word toggle button, pressing it enables or disables wake word detection. Both buttons have a pull-down resistor, which ensures that the pin stays low when the button is not pressed. In addition, each button has a capacitor to prevent bouncing, which could otherwise cause multiple signals from a single press. The buttons are directly connected to the 40-pin header of the Raspberry Pi. GPIO5 is connected to S1. GPIO6 is connected to S2

6.2 Raspberry Pi 4

A Raspberry Pi 4[1] with 2GB RAM was chosen. It features a quad-core processor running at up to 1.5GHz and has 2GB of memory. It includes various ports such as USB ports and an Ethernet port. Additionally, the Raspberry Pi has two Micro-HDMI ports, so the display is connected to one of these ports. A microphone is connected via the AUX port for voice recordings. The Raspberry Pi 4 operates at 5V and can draw up to 3A under heavy load. These values represent the absolute maximum ratings.

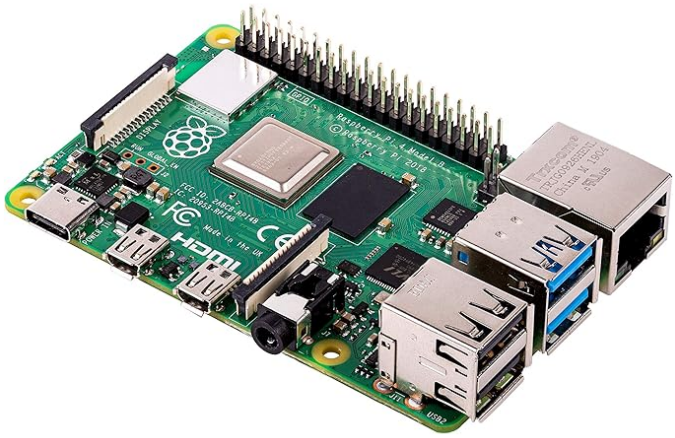


Figure 6.4: Raspberry Pi 4

[1] Raspberry Pi 4 [37]

6.3 Case v1.0

The Raspberry Pi 4 was modeled at a 1:1 scale to ensure precise fitting and accurate positioning of components.

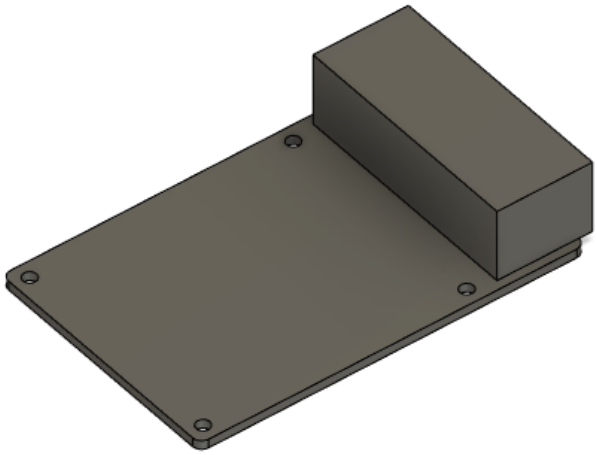


Figure 6.5: 3D Model of the Raspberry Pi 4

6.3.1 Design

The Raspberry Pi consists of two main sections: a flat base plate with mounting holes and a raised rectangular area that represents the USB and Ethernet ports of the Raspberry Pi. The base plate forms the foundation of the Raspberry Pi. The basic structure is a flat, rectangular surface with four screw holes. At the end of the plate, there is a rectangular elevation. This section is intended for the USB and Ethernet ports of the Raspberry Pi and serves as a placeholder for the four USB ports and one Gigabit Ethernet port.

6.3.2 Custom Enclosure

The Raspberry Pi was used as a reference for the measurements and the positioning of the parts, but it was not included in the printed model. The enclosure was built to be stable and easy to use.

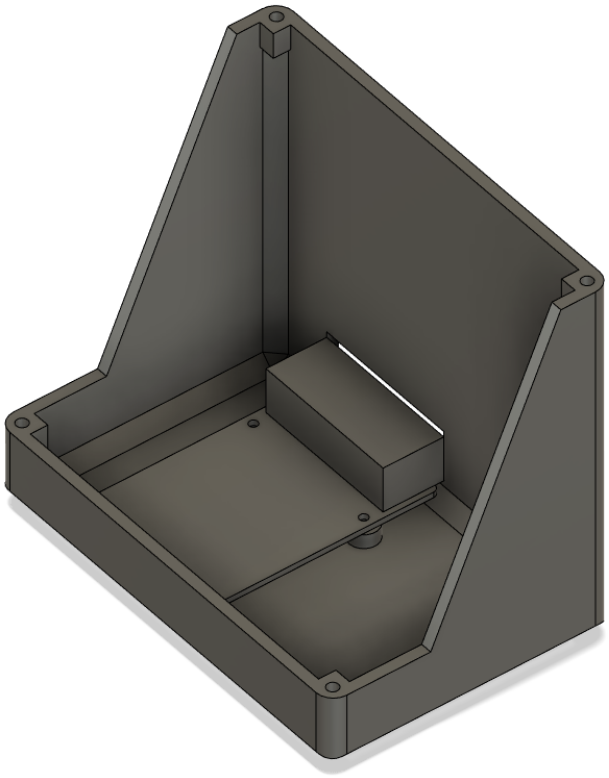


Figure 6.6: 3D Model of the custom enclosure

It is 13cm high, 10cm wide, and 14.8cm long. This size provides enough space for the Raspberry Pi and the circuit board without making the enclosure look too big. The bottom of the enclosure is a flat, rectangular plate. It serves as the base on which the Raspberry Pi is mounted. There is a slightly raised area on the surface where the Raspberry Pi is screwed in. A rectangular extrusion is included for the USB and Ethernet ports, helping to plan accurate cutouts later. In the center, there is an empty space where the circuit board will be placed. The enclosure has four walls. The two side walls are slightly angled backward so that the enclosure stands at a slant, making it easier to view the screen later. On the back side, there is a rectangular cutout for the Raspberry Pi's USB and HDMI ports. There is also a round cutout for the DC power jack. At each of the four corners of the base, there are round mounting points for screws. These are fitted with threaded

inserts to make it easier to screw and unscrew them later. The cover of the case is an important part of the whole setup. It protects the electronics inside and also has important functions. The cover can be firmly attached to the case, but it can also be easily removed. There is a large opening in the cover for a display, and additional holes for buttons and an LED.

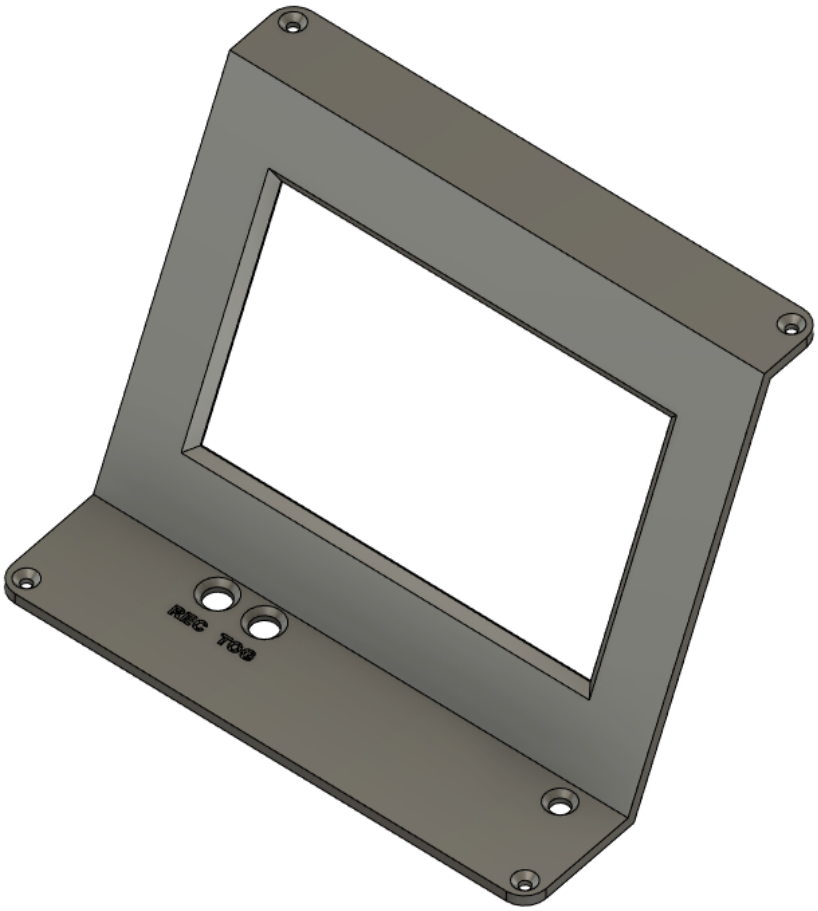


Figure 6.7: 3D Model of the enclosure cover

6.3.3 Cutouts

At the top of the cover is a large rectangular cutout for the ISR display. It is 11 cm long and 7.5 cm wide, so the display fits perfectly. The edges are rounded to prevent damage to the display.tioning of this window ensures the display remains centered within the user's field of view. Below the display, there are two round holes for buttons. One is the record button, which starts a recording. The other is

the wake word toggle button. The holes are large enough for the buttons to fit in properly and be pressed easily. To the right of the buttons, there is a small round hole for an RGB LED. This LED shows the current status of the system — for example, whether it is recording, idle, or if an error occured. The LED is placed in a spot where it is clearly visible but does not get in the way.

6.3.4 Mounting

The cover is designed to be securely attached to the enclosure using four screws. There is a hole for a screw in each corner of the cover, making it easy to screw the cover in place. Threaded inserts are built into the screw holes to ensure the screws hold more firmly.

6.4 Complete case

The image below shows the complete enclosure. It features a slanted front face with a large rectangular cutout that provides space for a display. Below this cutout are two small circular openings, intended for the toggle button and the record button. The abbreviations for the buttons are also engraved, serving both for orientation and enhancing the appearance. The enclosure has mounting holes at all four corners on both the top and bottom sides, allowing the cover to be securely attached. The overall design has a rounded style. The edges and corners have been smoothed, both for safety and for aesthetic reasons.

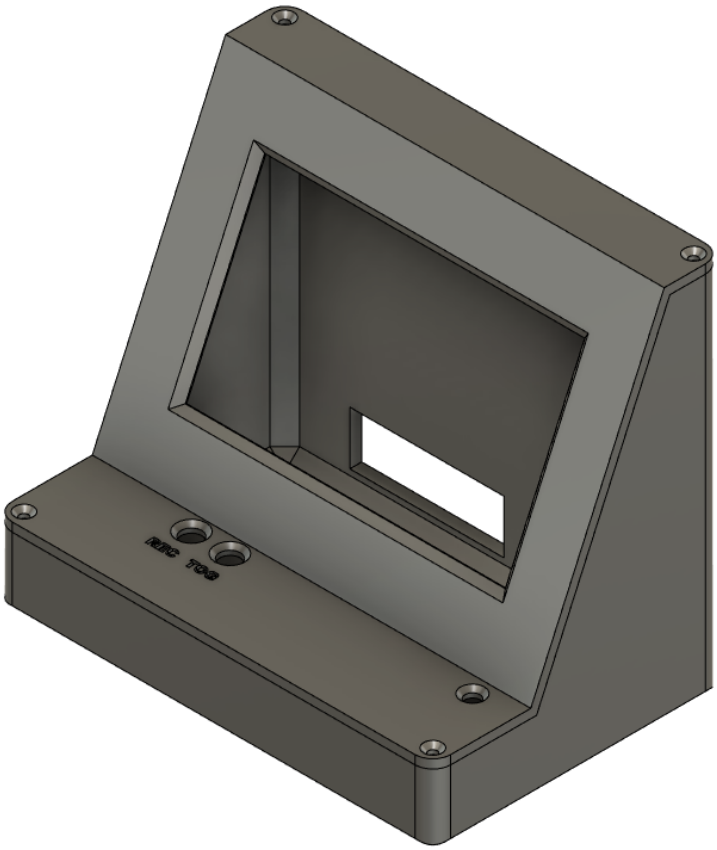


Figure 6.8: 3D Model of the entire case

6.5 Display

The 10.1-inch IPS touchscreen display[2] was developed for the Raspberry Pi 4 and other IoT devices with HDMI ports. It has a resolution of 1024x 600 pixels and displays clear images. This allows various outputs and buttons on the screen to be shown more precisely.

Figure 6.9: HAMTYSAN Raspberry Pi Display

The screen has a 16:9 format and a capacitive touchscreen that supports up to five simultaneous touches. This makes operation very accurate and easy. No additional drivers are needed. The connection to the Raspberry Pi is made using an HDMI cable for the video signal and a USB Micro-B cable for the touch function.

[2]Raspberry ISR Display [38]

6.6 Microphone

The chosen microphone was an AGPTEK lavalier microphone[3], a small clip-on microphone. It captures sound from all directions and delivers clear and precise recordings.

Figure 6.10: AGPTEK Lavalier Microphone

The microphone is very sensitive and can capture good sound even in noisy environments. The sound recorded by the microphone is transmitted to the device through an aux-to-USB-A adapter. Once the input reaches the Raspberry Pi, it is sent to the backend. There, the software processes the audio.

6.7 Schematic v2.0

6.7.1 Power Supply Circuit

This circuit was designed to provide a stable 5V power supply, either through a DC input or a 9V battery. The voltage regulation was handled by the LM2576 voltage regulator, which steps down the input voltage to 5V. Later, however, the 9V battery was replaced with two 18650 lithium-ion batteries to improve energy efficiency, increase capacity, and reduce the need for frequent battery replacements.

[3] AGPTEK Microphone [39]

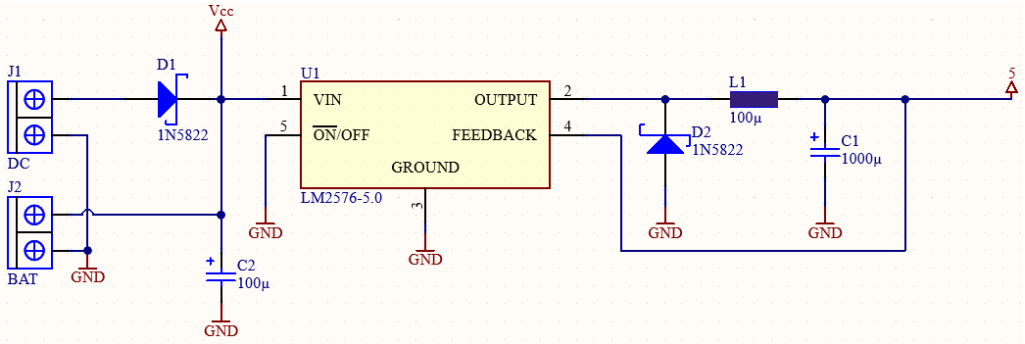


Figure 6.11: Power supply circuit diagram

6.7.2 18650 Li-Ion battery

18650 lithium-ion batteries are small, rechargeable cells with a typical voltage of around 3.7V. When two of them are connected in series, the total voltage is about 7.4V. A Battery Management System (BMS) is added to ensure safe operation of the battery and to prevent issues such as overcharging or deep discharge. In this circuit, the LM2576 voltage regulator converts the 7.4V into a stable 5V output.

6.7.3 LM2576 step-down voltage regulator

The LM2576[4] series of monolithic integrated circuits provide all the active functions for a step-down (buck) switching regulator. Fixed versions are available with a 3.3V, 5V, or 12V fixed output. Adjustable versions have an output voltage range from 1.23V to 37V. Both versions are capable of driving a 3A load with excellent line and load regulation. These regulators are simple to use because they require a minimum number of external components and include internal frequency compensation and a fixed-frequency oscillator. The LM2576 series offers a high efficiency replacement for popular three-terminal adjustable linear regulators. It substantially reduces the size of the heat sink, and in many cases no heat sink is required.

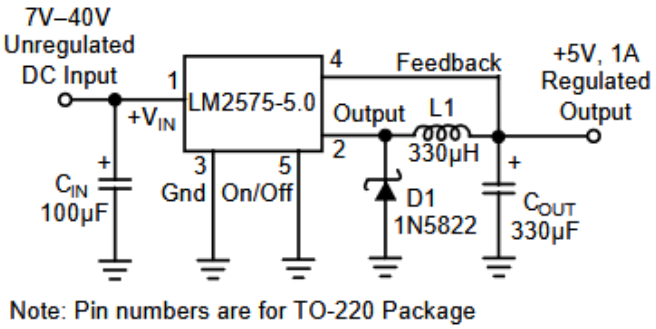


Figure 6.12: Typical Application Diagram

[4]Typical Application [40]

6.7.4 Pinout

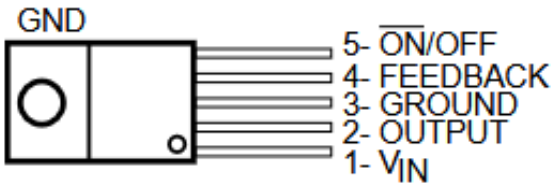


Figure 6.13: LM2576 Pinout

6.7.5 Circuit

Power Input

The circuit has two options for power supply: a DC jack connector J1 for an external power source, and a battery connector J2 for two lithium-ion 18650 batteries connected in series. The Battery Management System (BMS) is mounted directly on the lithium-ion batteries. To prevent current from flowing between the two sources, a Zener diode is used. Capacitor C1 helps stabilize the input voltage.

Voltage Regulation

The LM2576[5] voltage regulator converts the input voltage into a stable 5-volt output. The VIN pin receives the input voltage. Pin 3 is connected to ground. Pin 5 is the ON/OFF control pin, which is connected to ground to enable the regulator. Pin 2 provides the regulated 5 volts.

Output

To make the 5-volt output as smooth as possible, it is stabilized with an inductor. The Schottky diode D3 prevents current from flowing back into the circuit. Capacitor C2 further reduces voltage fluctuations.

[5] Pinout LM2576 [41]

6.8 Battery-powered supply

6.8.1 2S Li-Ion Battery Pack BMS

This Battery Management System[6] is specifically designed for two 18650 lithium-ion battery cells connected in series. This results in a typical voltage of about 7.4V. The BMS includes a special IC that constantly monitors important values such as voltage, current, and temperature. This ensures safe operation and extends the battery's lifespan.

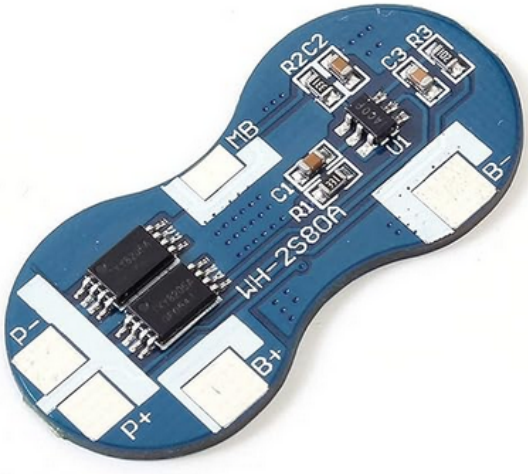


Figure 6.14: DollaTek 3Pcs 2S Li-Ion BMS

The BMS performs many protection functions. It checks each cell individually to make sure the voltage does not get too high or too low. If the current becomes too high or too low, the BMS shuts off the battery to prevent damage or overheating. The temperature is also monitored: if it gets too hot, the BMS disconnects the circuit to prevent dangerous situations like a fire. The system also protects the battery from overcharging by stopping the charging process once a certain maximum voltage is reached. In case of deep discharge, it stops discharging to prevent damage to the cells. If a short circuit or sudden high current is detected, the BMS immediately disconnects the battery to protect it. In a 2S configuration, it's important that both cells are charged evenly. The BMS includes a balancer that equalizes the voltage between the cells so both charge to the same level. This improves performance and increases the battery pack's lifespan. An important part of the BMS are the MOSFETs. These act as electronic switches and control the flow of current between the battery and the device or charger. Under normal conditions, the MOSFETs are turned on and allow current to pass. But if the BMS

[6] DollaTek 2S Li-Ion BMS [42]

detects an error—such as overcharging or a short circuit—it switches the MOSFETs off. This breaks the circuit and protects the battery. Once everything is back to normal, the BMS turns the connection back on.

6.9 PCB Design

6.9.1 Top-Layer

On the top layer of the PCB, there are two push buttons. The left button is the 'Record' button, and the right button is the 'Toggle' button. They have been placed so that they end up in the center of the enclosure, providing sufficient space to press them comfortably. The enclosure was then designed with two recesses that allow the buttons to protrude through, ensuring easy access for the user.

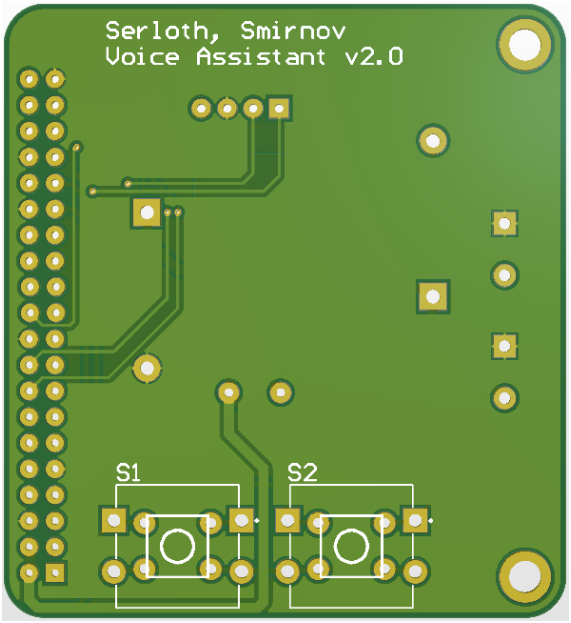


Figure 6.15: Top-Layer

6.9.2 Bottom-Layer

All the important components for power supply and other functions are placed on the bottom side of the circuit board. The LM2576 is a voltage regulator that takes a higher input voltage and converts it into a lower, stable output voltage. It is placed near the power input connectors so that the power paths stay short.

The large 20x20 header connects the board directly to the Raspberry Pi 4, providing a strong mechanical and electrical connection. There are two small 2-pin headers for power input: one for connecting a Li-Ion battery, and one for a DC jack (power adapter). This allows the system to be powered by either a battery or an external power supply. There is also a 4-pin header for connecting an RGB LED. The LED is mounted at the top of the enclosure and connected via a cable, while the control electronics stay on the bottom side of the board. Two diodes are included in the power paths to control the flow of current from the battery or the DC jack to the voltage regulator. They also protect against reverse polarity or backflow of current. An inductor works together with the LM2576 to convert the voltage. It temporarily stores energy and is sized based on the needed output power. Capacitors are used to stabilize the voltage at both the input and output of the voltage regulator. The capacitor shown in the circuit diagram is especially important for the output side. It must have sufficient capacitance and an appropriate voltage rating. Resistors R1, R2, and R3 serve as current-limiting resistors for the LED. Resistors R4 and R5 are used as pull-down resistors. Placing these components on the underside has the advantage of freeing up space on the top side for mechanical parts. Splitting the layout across two layers also allows for a more compact design.

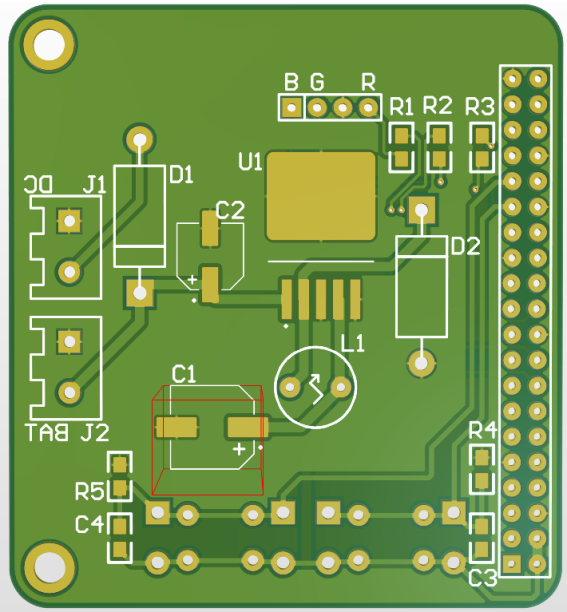


Figure 6.16: Bottom-Layer

6.10 Case v2.0

Because the initial measurements were inaccurate, the display did not fit properly into the housing. The measurements were then revised so that the opening is large enough for the screen and the screw holes precisely align with those of the display. Additionally, the buttons have been reworked. Since their positions were adjusted in PCB version 2.0, the corresponding button openings in the housing were moved further inward. As a result, the buttons can now be easily pressed through the newly adjusted openings.

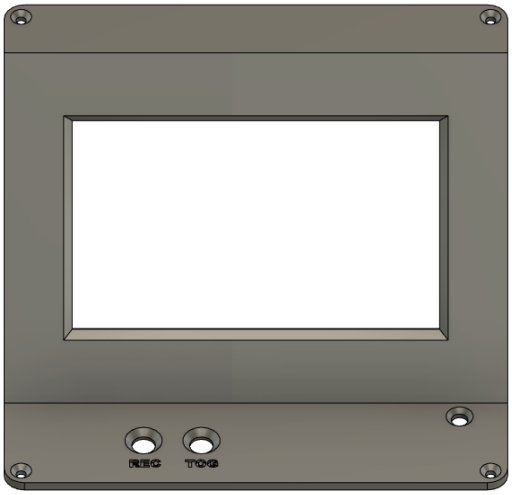


Figure 6.17: Case v2.0 front

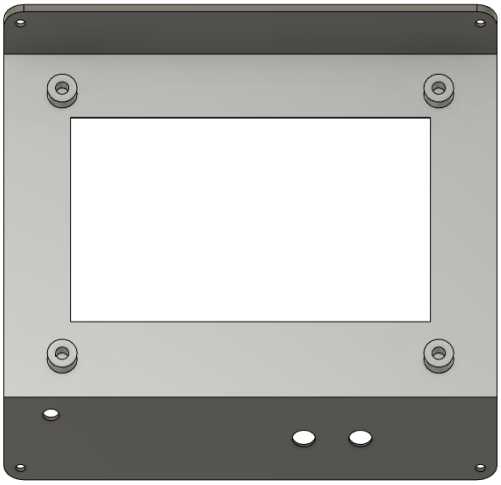


Figure 6.18: Case v2.0 back

7 Diploma Seminars

7.1 First Diploma Seminar

Title: Voice Assistant

Graduates: Lukas Serloth, Artemiy Smirnov

Secretary: Lukas Serloth

Class: 5BHELS

Supervisor: Dipl.-Ing. Wolfgang Kuran

Location: HTBLuVA St. Pölten / Room W111

Date: 17.10.2024

Time: 12:10 to 12:45

7.1.1 Meeting content

- Current Status and Progress
- Structure of the diploma thesis
- Review of various diploma theses

7.1.2 Summary

Current Status and Progress: The current status was discussed along with the next steps to be taken. The software has already been programmed to display the weather from any desired location. For the hardware, work on the schematic has already begun.

Next steps: The remaining software modules will be programmed and tested. The schematic will be completed so that the final PCB can be ordered soon. Additionally, the structure of a diploma thesis was discussed, and various examples were reviewed.

7.2 Second Diploma Seminar

Title: Voice Assistant

Graduates: Lukas Serloth, Artemiy Smirnov

Secretary: Artemiy Smirnov

Class: 5BHELS

Supervisor: Dipl.-Ing. Wolfgang Kuran

Location: HTBLuVA St. Pölten / Room W111

Date: 04.12.2024

Time: 16:40 to 17:10

7.2.1 Meeting content

- Current Status and Progress
- Rewriting of the Software in Rust
- Further steps for programming the Raspberry Pi 4
- Completion of the case

7.2.2 Summary

Current Status and Progress:
The schematic has been completed, and the first version of the PCB has been ordered. Currently, the first version of the enclosure is being 3D designed.

Next steps:
The PCB should be assembled and properly connected to the Raspberry Pi 4 and the screen. The enclosure should also be completed. Initial test attempts using audio should be conducted and tested.
The diploma thesis is expected to be completed between the end of February and mid-March. If the project progresses according to plan, additional projects will be assigned.

7.3 Third Diploma Seminar

Title: Voice Assistant

Graduates: Lukas Serloth, Artemiy Smirnov

Secretary: Lukas Serloth

Class: 5BHELS

Supervisor: Dipl.-Ing. Wolfgang Kuran

Location: HTBLuVA St. Pölten / Room W111

Date: 03.03.2025

Time: 12:10 to 12:45

7.3.1 Meeting content

- Current Status and Progress
- Further steps for programming the Raspberry Pi 4
- Completion of the case v2.0
- Discussions about on PCB Assembly

7.3.2 Summary

Current Status and Progress:
The schematic v2.0 and PCB v2.0 have been completed and have already arrived. Additionally, new calculations must be made for the dimensions of the case, as the display currently does not fit. The software is fully operational.

Next steps:
The PCB will be assembled, and the project will undergo final testing. Additionally, the enclosure will be redesigned. For this, new measurements for the display are required, as well as adjustments to the button holes, since their positions have changed in PCB v2.0.

8 Results

The result of the project ended up being a fully operational voice assistant with a hardware frontend and a Rust backend. The hardware frontend contains a 3D-printed case covering a Raspberry Pi and a 10.1-inch HDMI display; the Raspberry Pi is powered by either a DC jack or a battery, depending on the users preference — this is realized using a custom printed circuit board and a battery management system. The custom PCB connects to the Raspberry Pi's GPIO pins and provides a more accessible alternative to the displays touchscreen, that being a recording button, which when pressed, can start or stop the recording of a query, as well as a wake word switch — this allows the user to enable or disable wake word detection (and therefore hands-free usage) using a button. There is also an RGB led on the PCB, which shows different color codes when errors occur, albeit any errors that occur during processing will be made known to the user via the voice assistant output.
The software backend is implemented in the Rust programming language, picked due to performance and developer ergonomics. It is built using a microservice architecture, where each part of the program is its own separate microservice trait, allowing multiple implementations for each microservice and therefore, using the robust configuration system, the ability to configure each step of the way from an audio input to an output — be that configuring a microservice to run locally or remotely, or configuring parameterns, such as if local machine learning models should be run on the GPU or on the CPU.
The following flowchart demonstrates the architecture, with the backend depicted as the server and the frontend depicted as the client:

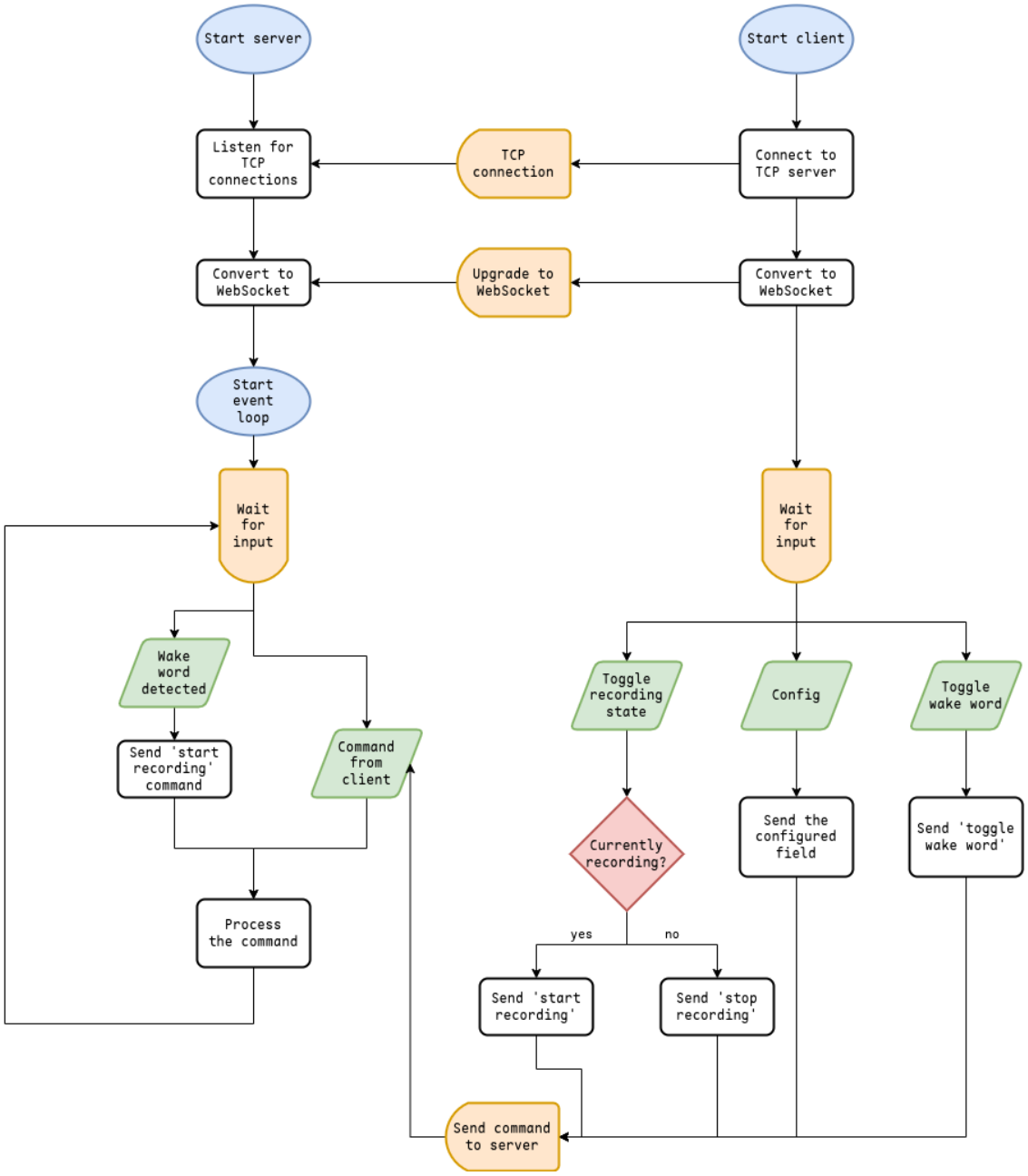


Figure 8.1: Client-Server Architecture

The process starts with the server listening for TCP connections, and a client connecting to the server, immediately after which, the connection is upgraded to WebSocket. In the main event loop, the server waits for input from the client, while the client waits for input to the user; the user can start the recording by

pressing the record button, after which the client checks if there is a recording currently running:

- If there is no recording active, the client sends a `start recording` command to the server
- If there is a recording active, the client sends a `stop recording` command to the server

another — arguably much simpler — way for the user to start the recording is to say the wake word, which the server listens for using the audio recording microservice. The user can also configure a setting on the client (in the reference frontend application implementation this is done via the settings menu), which sends a `config table.key=value` command to the server, where table, key, and value are some setting that the user configured. Finally, the user can also toggle wake word detection, which sends a `toggle wake word` command to the server and either enables or disables wake word detection.

8.1 Final Discussion

The project can be considered complete — all required features have been successfully implemented. Future enhancements could focus on expanding the feature set and improving the user experience when interacting with the frontend. Moreover the audio recording system should be overhauled, possibly by moving it back to the frontend; Timers are also not perfect yet, as they do not have a ringtone. Additional speech-to-text models may also be explored to improve accuracy and response time.
As a note to self for the future: More extensive testing is a generally a good idea — testing manually wastes a lot of precious time and hunting bugs when automated tests could have discovered them automatically should not be necessary. Usability studies should be conducted to better understand how the system performs in real-world scenarios, while interacting with real users.

9 Economic Viability

9.1 Calculations

Product	Costs
PCB v1.0	19.95€
Case v1.0	1€
Raspberry Pi 4	68.10€
IPS Display	66.54€
AGPTEK Microphone	13.10€
Case v2.0	1€
LM2576	10.21€
BMS 2S Li-Ion Battery Pack	6.04€
18650 battery holder case	10.07€
DC socket	10.07€
PCB v2.0	28.95€
Total	**235.53€**

These are the expenses incurred for the diploma thesis. All costs were related exclusively to the hardware components. The materials were purchased and financed privately.

9.2 Profitability analysis

Product	Costs
Case v1.0	1€
Raspberry Pi 4	30€
IPS Display	30€
Microphone	1€
Case v2.0	1€
LM2576	2€
BMS 2S Li-Ion Battery Pack	2€
DC socket	2€
PCB v2.0	1€
Total	**70€**

For mass production, PCB v1.0 costs can be disregarded, as only PCB v2.0 will be utilized. The price of PCB v2.0 is €4 per batch of 5 pieces, with additional shipping costs. When ordering 100 pieces, shipping costs are estimated at €5 per batch of 5 units.
To analyse the profitability of the finished product, the cost per unit was calculated, taking into account both the development and production costs.
To calculate the development costs, the total working hours excluding time spent writing the documentation were multiplied by the average hourly wage:

$$265\,\text{h} \times 70€/\text{h} = 18550\,€$$

The total hardware cost of one production unit equals 70 €. The estimated time to manufacture 100 units is about 10 hours, hence an effective time spent per unit of 6 minutes. The manufacturing cost is calculated based on the work required to assemble one unit and the typical man-hour rate for a minimum wage employee hired to assemble the product:

$$10\,\text{h} \times 10€/\text{h} = 100€$$

$$100€ \;/\; 100 = 1€$$

The total cost of one production unit amounts to 71 €. A reasonable price is 139.99 €, which grants a profit margin of 97.2%. This price could later be modified depending on market response.
With this profit, the Break-Even-Point is calculated as follows:

$$\frac{18550€}{68.99€} \approx 269 \text{ units}$$

Thus, 269 units must be sold to break even with the development costs.
To achieve a 30% return on investment, the calculation is as follows:

$$\frac{18550€ \times 1.3}{68.99€} \approx 350 \text{ units}$$

Therefore, approximately 350 units must be sold to reach a 30% ROI.

10 Time Tracking

10.1 Artemiy Smirnov

2024		
Week	**Task Description**	**Hours**
36	Preparation	12
37	Weather request implementation	17
38	OpenAI API request implementation	10
40	Hardware parts organization	5
42	Documentation	8
44	Natural language processing setup	10
46	User configuration implementation	17
48	Documentation	8
49	Documentation	5

2025		
Week	**Task Description**	**Hours**
4	Documentation	11
5	Rust microservices implementation	23
6	Refactoring and Rasa client implementation	19
6	Documentation	9
7	Refactoring	6
7	Documentation	5
8	Config and text-to-speech implementation	18
8	Documentation	11
9	Timer implementation	4
9	Documentation	7
10	Documentation	6
11	DeepSeek and Deepgram clients	16
12	ElevenLabs client, volume service, and small improvements	15
13	Wake word detection	12
14	Documentation and final touches	22

Total	**276**

10.2 Lukas Serloth

2024		
Week	**Task Description**	**Hours**
36	Preparation	10
37	Preparation	12
38	Schematic v1.0 considered and designed	6
39	PCB v1.0 considered and designed	9
40	Documentation	11
48	Order components	6
43	Case considered and designed	20
44	Case considered and designed	18
45	Assemble PCB v1.0	3
46	Documentation	14

2025		
Week	**Task Description**	**Hours**
3	Schematic v2.0 considered and designed	9
4	Order components for Schematic v2.0	4
5	PCB v2.0 considered and designed	14
6	Case redesigned	9
7	Assemble PCB v2.0	4
9	Documentation	13
11	Documentation	6
12	Documentation	8
13	Documentation	15

Total	**191**

List of Figures

Bibliography

[1] *Bundesgesetz über die Ordnung von Unterricht und Erziehung in den im Schulorganisationsgesetz geregelten Schulen*. 1986. URL: https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=10009600.

[2] *Prüfungsordnung BMHS, Bildungsanstalten*. 2012. URL: https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=20007845.

[3] Wikipedia. *Artificial Intelligence*. 2025. URL: https://en.wikipedia.org/wiki/Artificial_intelligence.

[4] Wikipedia. *Machine Learning*. 2025. URL: https://en.wikipedia.org/wiki/Machine_learning.

[5] Wikipedia. *Natural Language Processing*. 2025. URL: https://en.wikipedia.org/wiki/Natural_language_processing.

[6] Wikipedia. *Computational Linguistics*. 2025. URL: https://en.wikipedia.org/wiki/Computational_linguistics.

[7] Wikipedia. *Formal Grammar*. 2025. URL: https://en.wikipedia.org/wiki/Pattern_matching.

[8] Wikipedia. *Large Language Model*. 2025. URL: https://en.wikipedia.org/wiki/Large_language_model.

[9] JetBrains. *The Kotlin Programming Language*. 2025. URL: https://kotlinlang.org.

[10] Rust Foundation. *The Rust Programming Language*. 2025. URL: https://www.rust-lang.org.

[11] VMware. *The Spring Framework: A comprehensive framework for enterprise Java applications*. 2024. URL: https://spring.io.

[12] Tokio Project. *Tokio: An asynchronous runtime for the Rust programming language*. 2025. URL: https://tokio.rs.

[13] Erick Tryzelaar and Serde Contributors. *Serde: A framework for serializing and deserializing Rust data structures*. 2025. URL: https://github.com/serde-rs/serde.

[14] Sean McArthur. *Reqwest: An ergonomic HTTP client for Rust*. 2025. URL: https://github.com/seanmonstar/reqwest.

[15] snapview. *Tokio-tungstenite: A Rust WebSocket library for the Tokio framework*. 2025. URL: https://github.com/snapview/tokio-tungstenite.

[16] Pierre Krieger and the RustAudio community. *Cross-Platform Audio Library*. 2025. URL: https://crates.io/crates/cpal.

[17] tazz4843. *Whisper-rs: Rust bindings for OpenAI's Whisper speech recognition model*. 2025. URL: https://github.com/tazz4843/whisper-rs.

[18] Picovoice. *Porcupine: A high-accuracy wake word engine for mobile and IoT devices*. 2025. URL: https://picovoice.ai.

[19] Tom Preston-Werner. *TOML: Tom's obvious minimal language*. 2025. URL: https://toml.io.

[20] Artemiy Smirnov. *Reference frontend for the voice assistant*. 2025. URL: https://github.com/eagely/voice-frontend.

[21] Wikipedia. *Microservices Architecture*. 2025. URL: https://en.wikipedia.org/wiki/Microservices.

[22] Ingy döt Net Clark Evans and Oren Ben-Kiki. *YAML Ain't Markup Language™*. 2025. URL: https://yaml.org/.

[23] Wikipedia. *Java Virtual Machine*. 2025. URL: https://en.wikipedia.org/wiki/Java_virtual_machine.

[24] Ruud van Asseldonk. *Hound: A WAV encoding and decoding library in Rust*. 2025. URL: https://crates.io/crates/hound.

[25] *Feature in machine learning*. 2025. URL: https://en.wikipedia.org/wiki/Feature_(machine_learning).

[26] Wikipedia. *Mel-frequency cepstrum*. 2025. URL: https://en.wikipedia.org/wiki/Mel-frequency_cepstrum.

[27] Wikipedia. *Recurrent neural network*. 2025. URL: https://en.wikipedia.org/wiki/Recurrent_neural_network.

[28] Wikipedia. *Long short-term memory*. 2025. URL: https://en.wikipedia.org/wiki/Long_short-term_memory.

[29] GeeksForGeeks. *Self-attention*. 2024. URL: https://www.geeksforgeeks.org/self-attention-in-nlp.

[30] Wikipedia. *Connectionist temporal classification*. 2025. URL: https://en.wikipedia.org/wiki/Connectionist_temporal_classification.

[31] OpenAI. *Whisper: OpenAI's Speech Recognition Model*. 2025. URL: https://openai.com/index/whisper.

[32] Rasa Technologies. *Rasa: Open-source machine learning framework for automated text and voice-based conversations*. 2025. URL: https://rasa.com.

[33] Wikipedia. *Inside-outside-beginning tagging*. 2025. URL: https://en.wikipedia.org/wiki/Inside%E2%80%93outside%E2%80%93beginning_(tagging).

[34] OpenStreetMap. *Nominatim: Geocoding address search engine*. 2025. URL: https://nominatim.openstreetmap.org.

[35] Alpine Linux development team. *Alpine Linux: A Security-Oriented, Lightweight Linux Distribution*. 2025. URL: https://alpinelinux.org.

[36] Rene van der Meer. *Raspberry Pi Peripheral Access Library*. 2025. URL: https://github.com/golemparts/rppal.

[37] Raspberry. *Raspberry Pi 4*. 2019. URL: https://www.berrybase.at/raspberry-pi-4-computer-modell-b-2gb-ram.

[38] HAMTYSAN. *HAMTYSAN Raspberry Pi ISR Display*. 2025. URL: https://amzn.eu/d/f81Nwft.

[39] AGPTEK. *AGPTEK Lavalier Microphone*. 2025. URL: https://www.amazon.de/AGPTEK-Omnidirectional-Kondensator-Transformation-Videokonferenz/.

[40] LM2576. *Step down Converter LM2576*. 2025. URL: https://www.ti.com/lit/ds/symlink/lm2576.pdf.

[41] LM2576. *Pinout Step down Converter LM2576*. 2025. URL: https://www.ti.com/lit/ds/symlink/lm2576.pdf.

[42] DollaTek. *2s Li-Ion BMS*. 2025. URL: https://5.imimg.com/data5/ZP/IL/AG/SELLER-752145/7-4v-2s-protection-circuit.pdf.

Note: All bibliography entry URLs that have the year 2025 set were accessible and correct as of April 1st, 2025. All entries with the year 2024 set were accessible and correct as of December 31st, 2024, but there is no real worry of them being taken down within the next few years, save for some niche Rust libraries.