Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Ingenieurwesen
Facoltà di Ingegneria
Faculty of Engineering

Bachelor in Computer Science

Development of a Voice Assistant to Control Car Seats using advanced AI models

Candidate Elias Binder

Supervisor Prof. Dr. Michael Haller

September 2023

Abstract

Over the last few years, the interaction between passengers and the features of cars has changed drastically. Almost since the beginning of car manufacturing, hardware controls, like physical buttons, have been used as the primary input device. However, technology has evolved strongly, and new types of input devices have been established in the market, like touch-screen devices that gained huge popularity with the introduction of the first iPhone or voice-controlled systems, which arose when Amazon announced their first Echo generation in 2014. These technical achievements have also reached the car industry with a delay. This trend is highly notable at companies like Tesla, which abandoned almost all physical buttons and rely solely on a major touch-screen and haptic sensors in the centre of the car. In addition, all new vehicles these days are delivered with voice control, which can be activated by pressing a button or a wake word to control different features like setting navigation destinations. The development of such a voice assistant to control the car seats will be discussed in this thesis. The intuition is to permit the user to control their car seat using intuitive and natural voice commands without prior knowledge. The whole project will be conducted in collaboration with Tratter S.R.L., a company hired by BMW to re-engineer their car seats. They formed several working groups, where each group is assigned to engineer a different part of the seat. Together with another intern, I participated in the group responsible for the voice control system. The assistant we developed aims to increase safety and make the interaction process more convenient for the operator while driving. Therefore, a new approach to developing such a system is taken, mainly based on cutting-edge natural-language AI models and transcription models, to make the assistant and the whole communication process appear more human-like. The potential impact of this research is significant as the results can be used in the development of any voice assistant and are not only limited to the car industry. Overall, this thesis project discusses the development of a reliable, effective, and user-friendly voice control system for car seats that can be implemented in future BMW models. Furthermore, the results gathered throughout the engineering process can help to enhance already existing voice control systems regardless of the context in which they are used.

References

The GitHub repositories of the finished project can be found in the table below.

Description	**URL**
NodeJS Main Application	`https://github.com/EliasBinder/BMWSeat-VoiceAssistant/settings`
Local Whisper Engine	`https://github.com/EliasBinder/localWhisperAPI/settings`
ESP32 Seat Control Unit	`https://github.com/EliasBinder/ESP32SeatAPI/settings`
Python Wakeword AI	`https://github.com/EliasBinder/BMWSeat-WakewordAI`

Contents

List of Figures

Chapter 1

Motivation

1.1 General history of voice control systems

1.1.1 The Beginnings

In order to reconstruct the history of voice assistants, it is necessary to define the boundaries of the meaning of the words "voice assistant" first. According to the dictionary, a voice assistant is "a computer program that can hold a conversation with somebody and complete particular tasks by responding to instructions or to information that it gathers from that person's digital device"[AS20]. The first program that was released and met this criteria was Radio Rex released in 1922. It was a toy dog that left its dog house when it heard its name "Rex" being said. This was achieved by fine-tuning an electromagnet to the frequency emitted when pronouncing the word "Rex". No computer was used within the system, since modern computers did not exist at that time [She21]. In the following decades speech recognition systems arose that were able to understand more and more words, starting from the digits 0 to 9 in 1952 with the Audrey System developed by Bell Labs. In 1973 the US Department of Defense and the Carnegie Mellon University developed the "Harpy" program that was able to understand over 1000 words (A). This breakthrough invoked the era of exploration, where voice assistants were slowly included in cellphones, like "VAL" developed by BellSouth for IVR telephone systems in 1990. The first voice assistant that was able to recognize natural human speech was the DragonDictate. It launched in 1993 and was limited up to 100 words per minute. However, beginning from the 2000s the area of voice assistants experienced a revolution. In 2003 Microsoft added voice recognition to their Office XP suite, in 2008 Apple and Google launched voice based input and command systems, called Siri and Google assistant respectively. Alexa was then presented in 2014 by Amazon, thereby completing the set of today's most popular voice assistants according to a survey (A) conducted by Microsoft

1.1.2 Modern Voice Assistants

According to a statistic (A) published by Juniper Research the number of digital voice assistants in use worldwide is increasing heavily over the years [Res20]. It demonstrates that the number of digital voice assistants will be doubling from

4.2 billion devices in 2020 to 8.4 billion devices in 2024. As a result, it can be seen that the area of voice assistants appears to be a fast-growing sector in the tech industry. Subsequently, major tech companies, like Apple, Microsoft and Amazon entered the area of voice assistants by developing their own version of an assistant and integrating them into their already famous products.

1.2 Status Quo

As shown in a survey published by voicebot.ai (A), about half of the population in the United States use the voice assistant integrated into their car. 64 per cent of these users use the voice assistant every month. According to these numbers, the current number of users of this technology in the US is already quite high. However, there is still potential for improvement, which can help to increase the number of users even further.

Current voice assistants, like Google Assistant, work similarly to the voice assistant developed in this thesis. On the device itself, an AI model detects when the user pronounces the wake word, like "OK Google", which then wakes the assistant. After that, a transcription model is used to analyze the words and the tone of the query. If the query is fairly simple, like "Turn on the lights in the bathroom", it gets processed directly on the device. If not, it gets sent to a data centre for advanced analysis. This works great most of the time, however, several improvements can be made. These improvements will be described in detail in the following chapters.

It can be said that by improving car voice assistants and making them more appealing to the customer, they are more likely to be used instead of pressing a button. This brings advantages, such as increased road safety because the driver is less distracted. Instead of searching for a button and pressing it, which requires the driver to take the eyes off the road, a voice assistant allows the driver to remain focused while performing the same action. In addition, voice assistants do not require a high learning curve since the driver can express his desires like talking to a human assistant. This allows for an easier operation of the car. Disabled drivers also gain an advantage from voice assistants since they can continue to operate the car in their custom way and do not need to reach for physical buttons to perform tasks. Lastly, the development of how cars are driven considered as well. In the future, a car will drive autonomously and no longer need a driver. Therefore, having a simple interface, like a voice assistant, to interact with the car makes transportation accessible for everyone since it allows the passengers to request actions without knowing how a car's infotainment or functionality works.

Chapter 2

Project Structure

This project aims to develop a voice assistant for the operator to perform natural-sounding voice command queries. The assistant can move the driver or passenger seat in response to complex queries triggered by pressing a button.

2.1 Comparison with Current Voice Assistants

2.1.1 Innovations of this Project

As described in Chapter 1, current voice assistants mostly use a keyword-parsing technique or even a small domain-related AI model to interpret the user's prompt. This means, given a user prompt, like "Please move my seat a little bit forward.", a traditional voice assistant may scan for the words "seat" and "forward" to determine the action to take. If the user prompts "I cannot reach the pedals" instead, this parsing technique fails because the query neither contains the word "seat" nor any direction to move the seat. Therefore, a large language model (LLM) is used in this project to allow the assistant to parse even complex queries like the one mentioned earlier. This approach is revolutionary in voice assistants since the model already knows how a car is structured and can determine the desired seat position given complex descriptions from the user like the one mentioned earlier.

Likewise, this project has a different approach to the Speech-To-Text model for transcribing user inputs. It takes advantage of Whisper AI, an open-source automatic speech recognition (ASR) model developed by OpenAI, which is competitive with other state-of-the-art ASR models regarding long-form transcription (See Graph B) because it "has been trained on 680,000 hours of multilingual and multitask supervised data collected from the web [Ope22].

However, running these models is a resource-intensive task requiring lots of computational power. For reference, running the "large" model of Whisper takes approximately 10 GB vRAM [Kim22b]. This causes problems since most head units in cars cannot fulfil these requirements. To overcome those limitations without changing the hardware inside the car, which would result in increased production costs, it was decided to outsource these computations into data centers and upload the input data for the models to the data center and download the evaluation of the models. Regarding the ASR model, the input data is an audio recording in wav format and the response would be the

textual representation of what is said in the wav file. For the interpretation model, the input data is a text prompt and the response would be a JSON with specifications about how the seat must be moved.

2.1.2 Benefits of using Large Language Models (LLMs)

Building a large language model involves several complex and time-intensive tasks. The developers must choose an architecture, select a large amount of textual training data, "tokenize" the data and finally fine-tune the model.

Regarding the architecture selection, the developers must choose one that suits the requirements. "A popular architecture for LLMs is called transformer, which uses a technique called self attention to keep track of relationships between words. Transformers analyze pieces of text concurrently, rather than one after the other as was previously the case, and allow machines to develop a sense of context between words" [Shi23].

Besides selecting an appropriate architecture, choosing the correct training data for the model is also necessary. It is important to pick sufficient and unbiased training data. Failing to do so may result in "inaccurate representations"[Cha23]. In addition, human intervention is required when training a model. The model may have difficulty understanding data because human "language is nuanced, context-dependent, and often ambiguous"[Cha23]. "Incorporating human expertise reduces this gap"[Cha23]. Lastly, ethical and social considerations should be made to keep the model neutral and avoid offensive outputs. Therefore, insulting, rude and disrespectful training data needs to be filtered by humans before providing it to the model [Cha23].

2.2 Hardware Specification

2.2.1 Microphone

The microphone used in the car is a MiniDSP UMA-8 microphone array mounted behind the rear-view mirror between the driver and the passenger. It consists of 8 microphones mounted in a circle, thus providing outstanding recording quality and several non-standard features. One of these features is the ability to determine the direction of the voice. Because of the technical architecture, it is possible to retrieve the angle from where the sound hits the microphone. Therefore, the microphone sends USB interrupts to the host device following a proprietary protocol that is defined in its manual in a sparse way lacking important information, such as the exact USB endpoint identifier or the Endianness used to transmit the angle [Ltd18]. However, analysing the byte transmission for each microphone's USB endpoint made it possible to reverse-engineer the protocol.

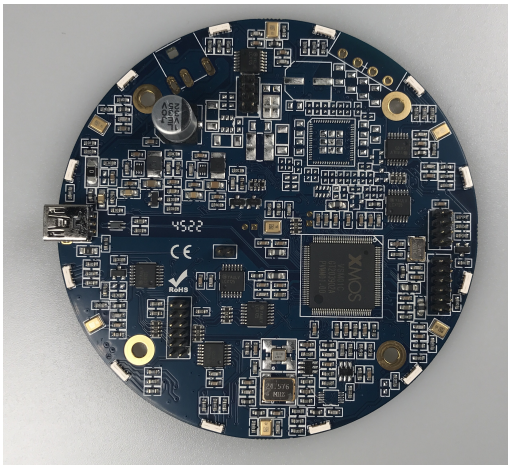

Figure 2.1: Image of a *MiniDSP UMA-8 microphone array*

2.2.2 Computer

Because BMW restricted to grant Tratter access to the already installed head unit of the car, additional computers were installed in the car's trunk. The computers must be operated in a street-legal car, so they must fulfil certain energy usage and safety standards. Therefore, four of the industry computers from the company *Spectra*, named *Spectra PowerBox 300 Advanced 2* have been chosen to be mounted in the car. According to its specification, it offers an Intel Core i5 6300U, 16 Gigabytes of RAM, and an SSD of 256GB [KG]. This hardware configuration provides more than enough resources to perform basic Rest API calls, record audio from the microphone and perform basic computational tasks. However, since no GPU is installed, running AI models locally will result in bad performance. Although the Computer has Windows 10 IoT preinstalled, it has been decided to install pure Debian 12 for stability and performance reasons. In addition, Debian is much more configurable compared to Windows 10 IoT and provides low-level hardware access, which is required to fetch the USB interrupts from the microphone, for example. Overall, four of these industry computers are mounted in the trunk, each having to fulfil one specific purpose. These purposes are

- The Voice Assistant
- The Hypervisor
- An AI system for estimating the height of a driver using a camera
- The Seat Control Unit

. gRPC and Rest APIs are used for communication between the different systems. A large battery pack is also installed in the spare wheel recess to prevent the car's starting battery from being drowned while the engine is not running. This battery pack can either be charged while driving or by plugging a power connector into a power outlet.

2.2.3 Internet Connection

An antenna is installed on the car's roof and uses the new 5G standard to provide high upload and download rates to make a stable and fast internet connection available. This antenna is connected to a router in the back of the car, which supplies all four computers with an internet connection. During development, the router is additionally connected to the intranet of Tratter Engineering to allow for remote computer control within the office building. Furthermore, a hypervisor allows external companies to control their computer from the outside, such as the company *Arrk Engineering* which develops the camera estimation system. However, developers must consider fallback methods since the car may be moved to areas with unstable network connections.

2.3 An Architectural Overview

The voice assistant developed at Tratter S.R.L. is divided into several modules and therefore follows the modular architecture principle. This architectural pattern describes the creation of independent packages that can communicate with each other. Each package is responsible for a small portion of the overall functionality and remains replaceable without affecting other modules [Mal]. This type of architecture has been chosen to facilitate the development experience regarding the division of labour between the two group members and allow for experimenting with different APIs for different purposes, like transcribing user input or interpreting the user's command using large language models. At the beginning of the project, the group members identified a list of required modules, and each member was assigned a specific set of modules to allow for independent working to increase productivity. The results are presented in the following diagram.

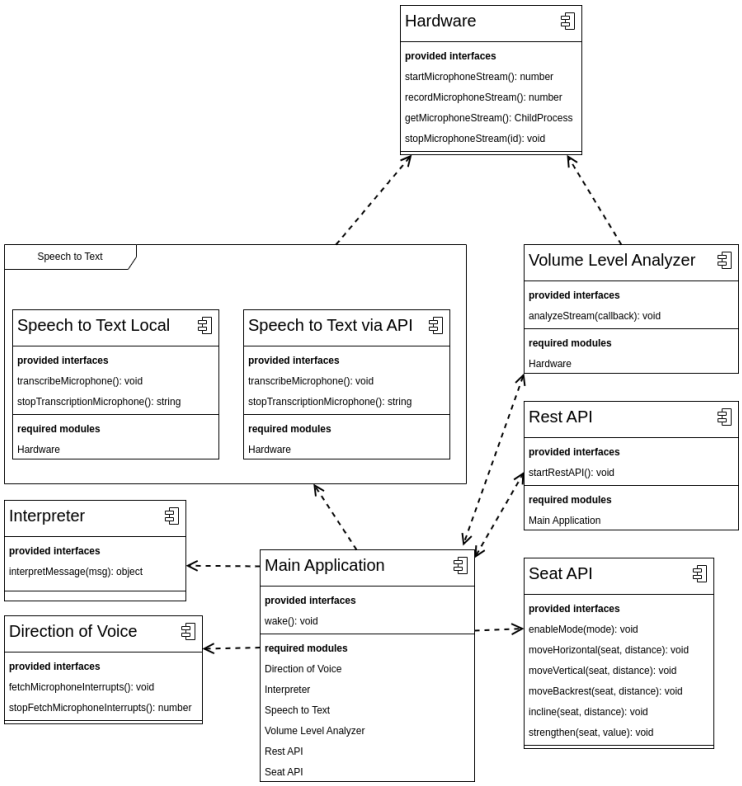

Figure 2.2: List of modules and their dependencies

As described in the diagram, the module "Main Application" deals with the coordination of most of the other modules and therefore is used as the entry point of the whole application (like the "main" method in Java).

Furthermore, it was decided to use TypeScript with NodeJS as the programming language used for the modules described in Figure 2.1. This combination is perfect for usage with external APIs as API calls in TypeScript can be written in a one-liner using 'fetch'. Using NodeJS allows the team to use a wide variety of open-source NPM modules to facilitate the interaction with Hardware, for example.

Later in the development process, it was decided that a wake-word detection model would also be developed to detect when the user says "Hey BMW" to wake the voice assistant. Therefore, a TensorFlow model has been created and trained with sample data to fulfil that task. Python3 is used as the programming language for all self-made AI models of the application. REST API calls realize the communication with the NodeJS main application.

2.4 Cohesion and Coupling of the Modules

As indicated in Figure 2.1, the "Main Application" module represents the central part of the whole Application and connects most of the other modules and ensures the program sequence. To achieve that, it exports a function named

"wake", which wakens the voice assistant to initiate the transcription and interpretation of the user's input. This wake function is called by the REST API module, which provides an endpoint to allow external software to initiate the transcription. This external software can be either the controller of a physical button located in the car or the wake-word detection model detecting the phrase "Hey BMW".

After the wake function has been called, the "Main Application" module calls the "Volume Level Analyzer" module to detect when the user starts and stops talking. As soon as the "Volume Level Analyzer" notices an increased volume level, it invokes the callback provided by the "Main Application", which calls the "Speech to Text" module and the "Direction of Voice" module.

The "Speech to Text" module is designed to record a ".wav" file from the user input and send that file to the Whisper model provided by OpenAI to get the transcription. The result gets then returned to the "Main Application" module. During the recording process, the "Direction of Voice" module calculates the average value from the USB interrupts fetched directly from the microphone and returns that value to the "Main Application" module as well.

Next, the "Main Application" module invokes the "Interpreter" module with the transcription and the direction retrieved earlier. Based on that, the "Interpreter" module builds a prompt that includes

- a description of the overall functionality of the voice assistant,
- a hint on whether the driver or the passenger is talking,
- the transcribed user input
- Optionally: The mode to activate or deactivate. The following modes are available:
  - Sleeping
  - Relaxing
  - Driving

and sends it to the GPT-3.5-turbo model, which is also provided by OpenAI. By using a feature of the GPT called "Function Calling", it is possible to obtain a structured response containing

- an identifier of the motor to move
- the distance to move
- the direction of movement

This information is returned to the "Main Application" module.

Finally, the "Main Application" module uses the "Seat API" module to execute the movement of the motor. Therefore, it provides the module with the GPT response obtained earlier. The "Seat API" module then makes a Rest API request to the appropriate seat controller, making the seat execute the desired movement.

Chapter 3

Modules in a Detailed View

In this chapter, a subset of the modules described in chapter 2 (2.3) will be explained in detail, pointing out how they function and what decisions were made during the development process. For any other module not explained in the following sections refer to the thesis "AI-Enabled Voice Control for Next-Generation Car Seats" by Leo Kerschbaumer.

3.1 "Hardware" Module

The "Hardware" Module aims to simplify the interaction between the NodeJS main application and the microphone/speakers. NodeJS itself, however, does not provide direct access to the microphone and the speakers. To overcome that problem, 3rd party software needs to be used.

3.1.1 Microphone

To access microphone data, a child process must be created that calls 3rd party software, like "arecord" on Linux or "rec" on MacOS. Thus, two shell scripts have been prepared for recording audio. The first is for recording audio directly into a .wav file, and the second is for streaming the raw bytes of the microphone to the NodeJS application. Respectively, the first is for creating a .wav file used by the Whisper model for the transcription, and the second is used by the "Volume Level Analyzer" module to compute the volume level.

The NodeJS implementation can be seen in the following code snippet:

```
1  let processes = new Map<string, ChildProcess>
2
3  export const startMicrophoneStream = (name: string) => {
4      processes.set(name, spawn('bash', ['resources/child-processes/
           mic-stream.sh']));
5  }
6
7  export const recordMicrophoneStream = (name: string) => {
8      try {
9          processes.set(name, spawn('bash', ['resources/child-
               processes/mic-recording.sh']));
10     } catch (e){
11         console.log(e);
12     }
```

```
13 }
14
15 export const getMicrophoneStream = (name: string) => {
16     return processes.get(name);
17 }
18
19 export const stopMicrophoneStream = (name: string) => {
20     if (!processes.has(name)) return;
21     const pid = processes.get(name)!.pid;
22     terminate(pid, function (err) {
23         if (err) {
24             console.log('Error:', err);
25         }
26     });
27     processes.delete(name);
28 }
```

To keep track of the various child processes, a "Map" is used. Therefore, each child process can be given a unique name when started. The data stream can be retrieved later, or the child process can be stopped based on that name.

3.1.2 Speakers

For playing audio, the NPM library "play-sound" is used. This library uses a preinstalled audio player, like mplayer, based on the operating system and sends the audio file path to that audio player [shi23]. The procedure used here is the same as for retrieving the microphone input. In this case, a child process is started, calling one of the supported audio players and providing the file path.

3.2 "Speech To Text" Module

The purpose of this module is to transcribe the user's speech to text. Therefore, a .wav file is recorded that contains the user's query. Afterwards, an automatic speech recognition (ASR) model is used to analyze that audio file and a textual representation is returned, which then gets passed to the "Interpreter" module. There exists a wide variety of ASR models. However, in the end, Whisper by OpenAI has been chosen due to its capabilities regarding multi-language support and the ability to understand dialects and malformed sentences. Other options would have been Facebook wav2vec2 and Kaldi Gigaspeech XL.

3.2.1 Comparison of the most popular ASR models

To find the best ASR model for a specific project, developers must compare the statistics of the most popular ASR models and perform tests on how the model handles context-related queries. In the following, the three most popular ASR models, namely *Whisper*, *Facebook wav2vec2 2.0* and *Kaldi Gigaspeech XL* will be compared and the strengths of each model will be pointed out.

Facebook wav2vec2 2.0

wav2vec2 is an open-source ASR model published by Facebook with several benefits and downsides, which can be seen below:

Benefits	**Downsides**
Unique Architecture	Limited ASR Support
Self-Supervised Training	Narrow Training Domain
Latent Vector Mapping	Resource-Intensive
Contrastive Training	Spelling Errors
Versatility	Inference Challenges
	GPU Memory Constraints

Figure 3.1: Benefits and Downsides of the Facebook wav2vec2 model based on [Sea22]

This model is suitable for audio data tasks since it uses a *featurization frontend* that efficiently processes audio waveforms. Furthermore, it does not require labelled / pre-transcribed audio data for training. Besides the featurization frontend, it also uses Latent Vector Mapping, an effective way to encode audio information. In addition, it can understand and distinguish audio features by identifying masked values in encoder outputs. Masked values are stored in addition to a data structure, which holds information about which values are correct and which are not. However, the base model published by Facebook itself cannot perform transcription tasks. To achieve that, the model must be extended by adding a "head" and training it on labelled speech data. An open-source model called wav2vec2-large-960h already exists, where these steps have already been performed. It can be found here: `https://huggingface.co/facebook/wav2vec2-large-960h`. Another downside is the narrow training domain. Given that the model has been trained using

clean, read speech from audiobooks, it has difficulties understanding noisy, conversational audio. One of the most important things that need to be considered when using this model is the computational power required to run it, given that it uses a large-capacity transformer encoder stack. Due to its large input sequence lengths and capacity, running inference on GPUs can be problematic and is likely to require a certain workaround to make the model perform well. On top of that, the model is likely to produce spelling errors because it operates with a character vocabulary. Furthermore, it is difficult to process long-form audio data. However, using pre-processing and batch techniques would increase those boundaries.

Kaldi Gigaspeech XL

Gigaspeech XL is part of the open-source Kaldi framework. Its pros and cons will be presented in the following table:

Benefits	**Downsides**
Accuracy	Complexity Usability
Availability of Gigaspeech Dataset	Pre-processing and Post-processing Requirements
Versatility	Inference Speed
	Lack of Complete Model Availability

Figure 3.2: Benefits and Downsides of the Gigaspeech XL model based on [Sea22]

Gigaspeech XL is reported to be one of the most accurate pipeline models available. However, this holds only for context related to the domains included in the Gigaspeech Dataset, which is publicly available and comprises 10,000 hours of labelled conversational English speech. Despite being a pipeline model, Gigaspeech XL can be adapted for various ASR tasks and domains. The major downside of this model is its complex usage. The Kaldi framework is considered academic and hard to use for inexperienced users. In addition, users must perform extensive audio pre-processing, like transcoding, chunking, and staging, before the model can be fed with the audio. On top of that, the Gigaspeech XL model performs poorly compared to other ASR options, thus making it not suitable for real-time applications like this project. Additionally, Kaldi does not provide the model as a ready-to-use package. Developers must integrate essential components, such as an ivector embedder and an RNN language model, to achieve transcription functionality.

Whisper

The last open-source model that will be analyzed is called Whisper. Its benefits and downsides are listed in the table below:

Benefits	**Downsides**
Multilingual Training	Lack of Label Verification
Architecture Simplicity	Unknown Training Data Characteristics
Multiple Model Sizes	Speed Variation between models
Audio Pre-processing	Generative Model Limitations
Large Training Corpus	Model Size
Multitasking Capability	
Punctuation and Capitalization	
Segment-Level Timestamps	

Figure 3.3: Benefits and Downsides of the Whisper model based on [Sea22]

First of all, Whisper has been trained with multilingual training data which means it can perform well in various languages. Furthermore, Whisper is extremely efficient when transcribing, thanks to its simple model architecture. It consists of 2D CNNs and a symmetric transformer encoder/decoder stack. This simplicity can lead to faster training and inference times. In addition, OpenAI published various sizes of the model each trained with a different size of sample data which allows developers to choose a suitable version for their project without wasting computing power. In general, whisper can be seen as the most developer-friendly model presented in this thesis, as it already contains features like audio pre-processing, which are not available in other models, such as Gigaspeech XL. For the largest Whisper model, OpenAI used a massive dataset of 680k hours of speech data for training, surpassing the scale and diversity of other ASR models like Gigaspeech XL and wav2vec2. This extensive training data can contribute to better performance. Whisper is also the only model that is not limited to transcribing audio data. It is also possible to perform tasks like language detection, voice activity detection, ASR, and translation, using a single output head. The model also adds punctuation and capitalization naturally to its output and does not suffer from spelling errors, thereby improving transcript readability. Furthermore, Whisper provides segment-level timestamps as part of its output, which can be valuable for tasks requiring alignment between audio and transcriptions. As these benefits may sound very convincing, Whisper also suffers from some drawbacks. The training data is weakly supervised, meaning humans have not verified the labels. This may result in inaccuracies in the final output. Another problem is that no information about the source and domain characteristics of the training data was published making it challenging to assess its performance in specific contexts. For controlling a car seat, however, Whisper works just fine. Comparing the smallest Whisper model and the largest, it can be noticed that the smallest model is approximately up to 30x faster than the largest. As a generative encoder/decoder model, Whisper can suffer from issues like repetitive words. This may affect output quality in some cases. Lastly, all Whisper models are relatively large in terms of the number of parameters used. This results in a higher demand for computational resources.

Model Selection

In summary, the Gigaspeech XL model is highly accurate but comes with usability challenges. In contrast, the wav2vec2 model offers unique architectural advantages but may require additional effort to optimize for specific tasks and domains. Whisper has several benefits, such as multilingual support, but it also comes with unknown training data quality and computing power challenges, especially for larger model sizes.

The following diagram shows the Word-Error-Rate of the models listed above:

Overall WER

Whisper Normalization

Dataset	Kaldi	wav2vec 2.0	Whisper
Conversational AI	64.2	36.3	19.9
Phone call	69.9	31.0	16.6
Meeting	44.0	27.4	13.9
Earnings Call	65.8	28.1	9.7
Video	47.6	23.3	8.9

Simple Normalization

Dataset	Kaldi	wav2vec 2.0	Whisper
Conversational AI	63.4	34.5	26.5
Phone call	70.0	30.9	20.8
Meeting	45.8	28.7	16.1
Earnings Call	69.1	35.3	11.2
Video	47.5	23.5	11.2

Figure 3.4: Comparison of the Word-Error-Rate between the models *wav2vec2 2.0*, *Gigaspeech XL* and *Whister*. Lower values are better. Image taken from [Sea22]

The table indicates that Whisper outperforms both other ASR models in every scenario. As a result, the developers decided to use Whisper in the final project, given that the effort required for the implementation is also fairly small, as described in the following sub-section.

3.2.2 Implementation

As indicated in the modules diagram (2.3), this project has two implementations regarding the usage of Whisper. One uses the official OpenAI API, and the other runs completely offline. Both implementations use the large-v2 model, which is the largest model provided by Whisper. Furthermore, it is the successor of the large-v1 model, and only these two models offer multilingual support [Kim22b]. In the following, both implementations will be discussed, pointing out their benefits and downsides.

Using OpenAI API

When using the OpenAI API, the programmer must upload an audio file to a data centre used by OpenAI using an HTTP POST request in the following format:

```
curl --location 'https://api.openai.com/v1/audio/transcriptions' \
--header 'Authorization: Bearer YOUR_API_KEY' \
--form 'model="whisper-1"' \
--form 'file=@"/path/to/transcription.wav"'
```

The API then returns a textual representation of what was said in that audio file as a JSON:

```
{
    "text": "move my seat all the way to the front."
}
```

Using Whisper in this specific way discloses several benefits and downsides, described in the table below.

Benefits	**Downsides**
There is no need for powerful hardware in the car	Requires a good internet connection
Easy to implement	Paid by minutes to transcribe/translate
	Data is shared with 3rd party data-center
	An OpenAI developer account with available credits is required

Figure 3.5: Benefits and Downsides of Using the OpenAI API for Whisper

To summarize, using this implementation saves the car's production costs because no powerful hardware is required. However, the voice assistant would only function with a fast internet connection. Furthermore, this solution costs 6 cents per 10 minutes of audio transcribing. As a result, the car manufacturer has to estimate how much and when the voice assistant will be used most of the time. Based on the outcome, it has to be decided which one of the two implementations to choose.

Given that this project uses TypeScript with NodeJS as its primary programming language, the implementation is fairly simple. There is not even the need to write the POST request by hand because OpenAI published a client library to interact with their APIs. A client library provides callable functions to the developer to facilitate the interaction with remote services. Under the hood, the client library creates and sends the POST request to the data centre while also dealing with possible errors that may occur, like an invalid API key.

First, the developer has to initialise the library to use the NodeJS client library for OpenAI. This can be accomplished in the following way:

```
import {Configuration, OpenAIApi} from "openai";

class OpenAI {
    public openai: any;
```

```
5       constructor() {
6           const api_config:Configuration = new Configuration({
7               apiKey: 'YOUR_API_KEY',
8           });
9
10          this.openai = new OpenAIApi(api_config);
11      }
12  }
13
14  export default new OpenAI();
```

The apiKey mentioned on line 7 can be obtained by creating an OpenAI developer account. A "Singleton" pattern is applied to make one instance of the class OpenAI globally available.

This instance is then used to invoke the transcription on the recorded .wav file as shown in the following code snippet:

```
1   let stopTranscription = (): Promise<string> => {
2       return new Promise((resolve, reject) => {
3           resolve('')
4       })
5   }
6
7   //Function to start transcription process
8   export function transcribeMicrophone(){
9       //Start recording a .wav file from the microphone stream using
            the "hardware" module
10      recordMicrophoneStream('transcription');
11
12      //Redefine stop function to stop recording and invoke the API
            call to Whisper at the end of the transcription
13      stopTranscription = async () => {
14          //Stop the recording process of the .wav file
15          stopMicrophoneStream('transcription');
16
17          //Transcribe the recorded file with the Whisper API
18          let resp = {data: {text: ''}};
19          try {
20              resp = await openAI.openai.createTranscription(
21                  fs.createReadStream('resources/transcription.wav'),
22                  "whisper-1"
23              );
24          }catch (e){
25              console.error('whisper error: ' + e)
26          }
27
28          //Return the result
29          return resp.data.text;
30      };
31
32  }
33
34  //Function to stop transcription process
35  export async function stopTranscriptionMicrophone(){
36      const result = await stopTranscription();
37      stopTranscription = (): Promise<string> => {
38          return new Promise((resolve, reject) => {
39              resolve('')
40          })
41      }
42      return result;
43  }
```

This file makes two functions globally available, namely "transcribeMicrophone" and "stopTranscriptionMicrophone". As soon as the "Volume Level Analyzer" module recognizes an increased volume level, it triggers the "transcribeMicrophone" function, and when the volume goes down under a certain threshold, the "stopTranscriptionMicrophone" function is called.

- **transcribeMicrophone**: This function uses the "Hardware" module to start the recording of a .wav file. In addition, it resets the private "stopTranscription" function to stop the recording and do the Whisper API call using the recorded .wav file. The result of this API call will be returned when calling the "stopTranscription" function the next time.
- **stopTranscriptionMicrophone**: This function calls the private "stopTranscription" function. If no transcription has been started beforehand, the "stopTranscription" method will return an empty string. However, in case a recording is running, the "transcribeMicrophone" has changed the "stopTranscription" method to the procedure described under the "transcribeMicrophone" explanation. In addition, this function resets the "stopTranscription" function to return an empty string in any case to complete the circle.

Running the model locally

In case sufficient processing power is available, running the model locally lets the developers and users benefit even more, as shown in the following table:

Benefits	**Downsides**
No internet connection required	Powerful hardware must be used
Free of charge regardless of the amount of data to transcribe	Implementation requires Python skills as well
No data is shared with 3rd parties	It is difficult to use the Whisper model with other programming languages than Python
No OpenAI developer account is required	
More fine-tuning possibilities since the model is open-source	

Figure 3.6: Benefits and Downsides of Running the Whisper model locally

Since no API calls to external systems are required in this implementation, it is obvious that this implementation is far more stable and robust than the implementation with the Whisper API because the dependency on 3rd party services has been minimized.

The Whisper Model uses the "PyTorch" machine learning framework, so Python must be used to interact with the model. This required the developers to specify an interface between the NodeJS main application and the Whisper model running in Python. It was agreed on the following approach:

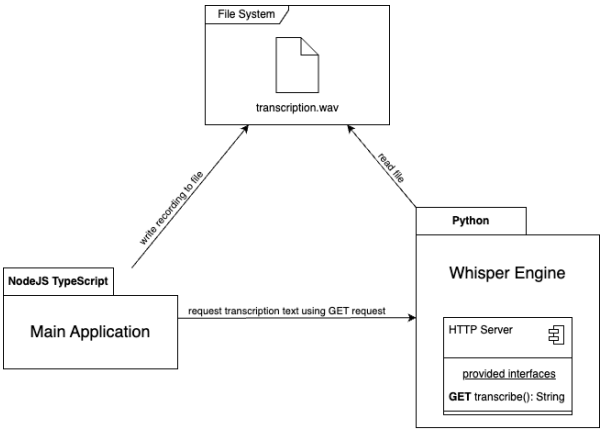

Figure 3.7: Interaction between Main Application (NodeJS) and Whisper Engine (Python)

Because both applications run on the same machine, they can access the same file system. This means there is no need to send the .wav file from the Main Application to the Whisper Engine. Instead, both applications have the path to that file, significantly reducing the transcription time. When the Main Application has finished the recording and saved the .wav file, it sends a GET request to the Whisper Engine with no additional data appended. The Whisper Engine then uses the Whisper Model to transcribe that .wav file and sends the spoken text back to the Main Application as a response to the initial GET request.

As a result, the Whisper Engine application consists of two parts, the Web Server part and the PyTorch part, where the Whisper model is mounted. In further analysis, the PyTorch part will be in focus.

According to the Whisper documentation, an API is provided to interact with the model. It is used in the following way:

```
1 import whisper
2
3 model = whisper.load_model("base")
4 result = model.transcribe("audio.mp3")
5 print(result["text"])
```

[Kim22a]

This will use the standard Whisper model of how OpenAI publishes it. However, some improvements can be made. The most important one is to use CTranslate2, a "Python library for efficient inference with Transformer models" [Kle23]. CTranslate2 "applies many performance optimization techniques such as weights quantization, layers fusion, batch reordering, etc., to accelerate and reduce the memory usage of Transformer models on CPU and GPU" [Kle23]. To provide a reference, if the standard Whisper model requires 4 minutes 30 seconds to transcribe an audio file, the CTranslate2 implementation of Whisper requires only 54 seconds. This means an acceleration of about 4x can be observed when comparing the CTranslate2 and the regular OpenAI implementation, providing the "same accuracy while using less memory" [Kle23].

There is already an open-source implementation of Whisper with CTranslate2, called faster-whisper. When started initially, it downloads OpenAI's Whisper model automatically and applies CTranslate2 to them. It also uses the "PyTorch" machine learning framework. However, the API for interacting with the modified model differs from the default OpenAI way:

```
1 from faster_whisper import WhisperModel
2
3 model_size = "large-v2"
4
5 # Run on CPU
6 model = WhisperModel(model_size, device="cpu", compute_type="int8")
7
8 segments, _ = model.transcribe("audio.mp3")
9 segments = list(segments)  # Transcription will run here.
```

As shown in line 7, the model needs to run on the CPU instead of the GPU in this project. This decision had to be made because the computer inside the car does not have a powerful graphics card, which would have accelerated the transcription process significantly.

3.3 ”Volume Level Analyzer” Module

3.3.1 Approach

As described in the ”Hardware” module section, obtaining the raw byte stream from the microphone is possible. Understanding how these bytes are structured and what they mean allows for computing the volume. This is useful to detect if the user is currently speaking or not by comparing the current volume with a preset threshold.

3.3.2 Structure of the wave format

A .wav file consists of a header and the actual sound data. The header's length is 44 bytes and is structured in the following way:

Description	Size (Bytes)	Usual Contents
RIFF file description header	4 bytes	The ASCII text string "RIFF"
size of file	4 bytes	The file size LESS the size of the "RIFF" description (4 bytes) and the size of file description (4 bytes).
The WAV description header	4 bytes	The ascii text string "WAVE".
fmt description header	4 bytes	The ascii text string "fmt " (note the trailing space).
size of WAV section chunk	4 bytes	The size of the WAV type format (2 bytes) + mono/stereo flag (2 bytes) + sample rate (4 bytes) + bytes/sec (4 bytes) + block alignment (2 bytes) + bits/sample (2 bytes). This is usually 16.
WAV type format	2 bytes	Type of WAV format. This is a PCM header, or a value of 0x01.
mono/stereo flag	2 bytes	mono (0x01) or stereo (0x02)
sample frequency	4 bytes	The sample frequency.
bytes/sec	4 bytes	The audio data rate in bytes/sec.
block alignment	2 bytes	The block alignment.
bits per sample	2 bytes	The number of bits per sample.
data description header	4 bytes	The ascii text string "data".
size of the data chunk	4 bytes	Number of bytes of data is included in the data section.
data	variable length	The audio data.

Figure 3.8: WAV Header Synopsis []

However, in this project it can be ignored since it mainly contains data already set by the programmers before the recording starts. Therefore, in this subsection, the focus is on the data part. Nevertheless, it is necessary to skip the first 44 bytes, since the header does not commit to the actual audio data and, therefore, not to the volume of the audio itself.

Computing the volume

Since the wav file is captured in "linear" and not "mu-law" encoding, which is specified in the "Hardware" module, computing the volume is fairly simple.

According to a StackOverflow response, "in a wav file, the data at a given point in the stream IS the volume (shifted by half of the dynamic range). In other words, if you know what type of wav file (for example 8 bit, mono) each byte represents a single sample. If you know the sample rate (say 44100 HZ) then multiply the time by 44100 and that is the byte you want to look at. The value of the byte is the volume (distance from the middle.. 0 and 255 are the peaks, 127 is zero)" [sna11].

For recording the .wav file, the developers use 2-channel audio at 16000Hz and a single sample size of 16 bits, formatted using a little-endian.

To get the volume, the little-endian is computed for every 2 bytes in every chunk to compute the average volume. Every value is added to a counter variable starting at 0, and another counter variable is incremented by 1. Based on that, the average for every chunk can be computed. In addition, it is checked if the average value is higher than the threshold value defined in a configuration file. In that way, it is possible to determine whether the user is speaking.

3.3.3 Further Improvements

The approach described above works seamlessly in consistent environments with the same background noise level. This works well for the prototype presentation at BMW Welt. However, if this noise level changes, the threshold must be adopted. As a result, this solution is unsuitable while driving the car because the background noise level changes constantly based on radio volume and wind noises. This leads to further improvements that need to be conducted for the production version of this voice assistant. The developers would need to keep track of the volume change to achieve a more robust solution instead of using a threshold. This can be done by computing the delta in volume change for every chunk and checking if the resulting delta volume is larger than a certain percentage of the background volume. It can be seen that this is a fairly simple solution to avoid consistent background noise affecting the system. Still, radio sound would cause the system to detect a speaking user wrongly. To make the system stable even under these circumstances, a far more complex implementation of a voice activation detection (VAD) algorithm is required, like the Hidden Markov Model (HMM) with Dynamic Time Warping (DTW). Discussing the functionality of that algorithm would break the boundaries of this thesis. However, B.-H. Juang published a great paper showing its functionality called "On the Hidden Markov Model and Dynamic Time Warping for Speech Recognition — A Unified View".

3.3.4 Implementation

The most important code used for the implementation can be found below:

```
1 let isFirstProcessed = false;
2 let startedSpeaking = false;
3 startMicrophoneStream('volume-level-analyzer');
4
```

```
5  getMicrophoneStream('volume-level-analyzer').stdout.on('data', (
       chunk: Buffer) => {
6      //Construct array of 16-bit integers representing the audio
           data
7      let out: any = [];
8      for (let i = 0; i < chunk.length; i += 2) {
9          out.push(Math.abs(chunk.readInt16LE(i)));
10     }
11
12     //Calculate the average volume
13     let average = 0;
14     out.forEach((value: number) => {
15         average += value;
16     });
17     average /= out.length;
18
19     //Skip header if the first chunk
20     if (!isFirstProcessed) {
21         isFirstProcessed = true;
22         return;
23     }
24
25     //Invoke callbacks if necessary
26     if (average > config.volumeThreshold) {
27         startedSpeaking = true;
28         if (timeout != null) {
29             clearTimeout(timeout);
30             timeout = null;
31         }
32     } else {
33         if (timeout == null && startedSpeaking) {
34             timeout = setTimeout(() => {
35                 stopMicrophoneStream('volume-level-analyzer');
36                 onFinish();
37             }, 1000);
38         }
39     }
40 });
```

In a previous subsection for computing the volume, it was explained that the developers use a single sample size of 16 bits, formatted using a little-endian. Line 9 shows that JavaScript already provides a function for directly reading this format from a Buffer. The function returns a JavaScript number holding the read value.

As line 36 shows, an *onFinish* callback is invoked after the user finishes talking. To avoid this callback being executed immediately when starting the volume-level-analyzer, even if the user has not even started talking, a boolean variable, called *startedSpeaking*, is used. This variable is false initially and set to true when the volume level exceeds the threshold for the first time.

Furthermore, it can be noticed that Timeouts are used in this code section. The purpose of doing so is to prevent a too-early callback execution because of a longer thinking time of the user about what to say. Line 37 indicates that a timeout of 1 second is used. This allows the user to make a small break between sentences and to think about the upcoming words.

3.4 ”Seat API” Module

3.4.1 Purpose

This module aims to handle the communication with the seat control units, which were developed by another team at Tratter Engineering. They provided documentation on how to use the API. According to that documentation, the developer must send the motor's identification number, the distance to move that motor as an integer between 0 and 255, and the direction of movement.

This module needs to take the result from the ”Interpreter” module, post-process that result, and perform the Rest-API call at the seat control units.

3.4.2 Functioning

By using a method called function-calling, the GPT model by OpenAI can respond with structured data that is processed in this module. An example of a GPT response can be found below:

```
{
    "name": "move_seat_horizontal",
    "arguments": "{\n\"direction\": true,\n\"distance\": \"medium
        \"\n}"
}
```

As described in the thesis of Leo Kerschbaumer, the GPT model performs poorly at guessing a number used for the distance parameter. To overcome this limitation, the developers figured out that using words instead, like *low*, *medium* or *high*, produces far better outcomes in terms of reliability and correctness. However, the seat control unit requires integer values, as described above. Therefore, a mapping function was introduced to map the keywords to integer values.

Finally, an API request is constructed using the obtained values and sent to the appropriate seat control unit, which can be either on the driver or passenger side, depending on who did the query.

3.4.3 Implementation

An example for the implementation can be found below:

```
export const moveHorizontal = async (seat: 'DS' | 'PS', value:
    number) => {
    return requestHandler({
        HexId: "73636531",
        Input: seat + '#01#' + (value > 0 ? 'p' : 'n') + value
    });
}

const requestHandler = async (params: any) => {
    try {
        const response = await axios({
            method: 'post',
            url: config.Seat_API_Endpoint,
            headers: {
                'Content-Type': 'application/json',
                'Accept': 'application/json',
                'Grpc-Metadata-Authorization': config.
                    Seat_API_Authorization
```

```
17              },
18              data: { ...params }
19          });
20          return response.data;
21      } catch (e) {
22          console.log('Error: ', e);
23          throw e;
24      }
25  }
```

The module contains a private function, named *requestHandler*, which sends the actual API request to the Seat Control Unit. It sends the data received as a parameter and deals with error handling. Line 16 shows that an API key is required as well to authorize the request at the Seat Control Unit.

The function called *moveHorizontal*, is 1 of 6 exported functions to control a specific seat feature. For example, this *moveHorizontal* function is responsible for moving the seat forward and backwards. All exported functions use the *requestHandler* auxiliary function to send the API request. A List of all exported functions can be found below. They all work similarly to the *moveHorizontal* function, however, they use different data to send to the API.

- enableMode
- moveHorizontal
- moveVertical
- moveBackrest
- incline
- strengthen

3.5 "Rest API" Module

3.5.1 Purpose

To understand the purpose of this module, it is necessary to understand the possible states the voice assistant can have. The following state transition diagram shows the different levels of activeness of the voice assistant:

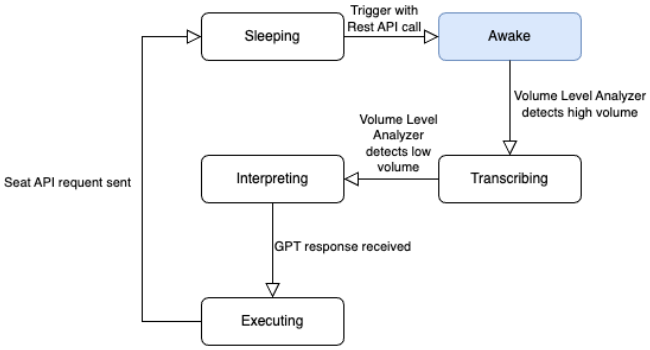

Figure 3.9: States of the Voice Assistant

Explanation of the states:

- **Sleeping**: The Voice Assistant waits to be awakened. In this state, it does not even listen to the microphone input.
- **Awake**: In this state, the system uses the "Volume Level Analyzer" to listen when the user starts talking. The transcribing state is entered as soon as the volume exceeds a certain threshold.
- **Transcribing**: In this state, the system creates the .wav file containing the recording of the user's query. The recording is stopped automatically when the "Volume Level Analyzer" module detects a low volume. After that, the Whisper model is used to transcribe that file.
- **Interpreting**: A prompt is constructed and sent to the GPT API with the transcript of the user's query.
- **Executing**: The result of the GPT model is parsed, and an API request to the seat control unit is made.

It can be observed that this module aims at providing an interface for other developer teams at Tratter Engineering to allow them to put the Voice Assistant into the *Awake* state. To achieve that, a REST API endpoint is made available that can be called by the other computers mounted in the trunk. One of these computers is connected to a button on the centre glove box. Pressing this button three times causes the connected computer to perform an API request on the exposed REST endpoint, which will change the state of the voice assistant to *Awake*.

A REST API was preferred over other communication technologies, like gRPC, because performing HTTP requests is a fairly simple task in almost every programming language.

Introducing a REST API to wake the system offers several other benefits. One is being independent of the software that triggers the wake-call. As mentioned before, this software can be a script that checks whether a button has been pressed. However, this software can also be far more complex, like a "Wake word detection AI". This artificial intelligence model observes the microphone input in real-time and checks if a wake word, like "Hey BMW", is detected. A self-created example implementation of this model using TensorFlow can be found in the appendix at C. Unfortunately, explaining this code breaks the boundaries of this thesis.

To test the Voice Assistant in scenarios where the developers did not have access to the car or the button was not yet implemented, they created a simple website with a button. Pressing that button would send an API call to the exposed endpoint. A screenshot of this website can be found below:

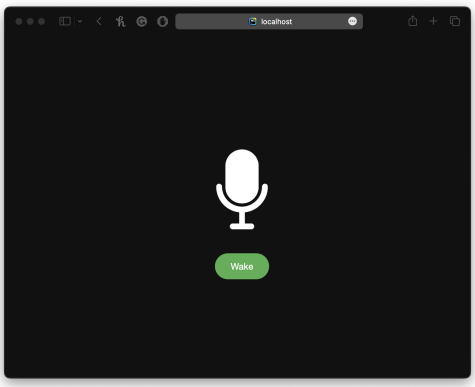

Figure 3.10: Website to simulate the hardware button. Pressing this button will wake the Voice Assistant

3.5.2 Implementation

The express JS framework for NodeJS is used to construct the REST API. The implementation can be seen in the following code snippet:

```
1  //[httpServer.ts]
2  import express from 'express';
3  import apiRouter from "./apiRouter";
4  const app = express();
5  const port = 3000;
6
7  export const startRestAPI = () => {
8      app.use(express.json());
9
10     app.use('/api', apiRouter);
11
12     app.listen(port, () => {
13         console.log(`Rest API listening at http://localhost:${port
               }`);
14     });
15 }
16
```

```
17 //[apiRouter.ts]
18 router.get('/wake', (req, res) => {
19     wake();
20     res.send({status: 'ok'});
21 });
```

The purpose of the file *httpServer.ts* is to create the HTTP server and setup the routing. The file *apiRouter* exposes the endpoint */wake*, which calls the *wake* function (line 19) exposed by the "Main Application" module, which can be found in the appendix at (C). Calling this *wake* function in the "Main Application" module will put the system in the *Transcribing* state according to the diagram (3.5.1).

Chapter 4

Practical Application on a BMW Seat

4.1 Testing the Assistant with BMW Hardware

To evaluate the Voice Assistant under real-world scenarios, the development team chose to build a replica of the inner space of a BMW car at the NOI tech park in Bolzano to perform extended testing on prompt engineering. This replica consists of a car seat taken from a BMW 3 series and a dashboard primarily used to develop the BMW 1 series.

Figure 4.1: Image of the Seat and the Dashboard

Given that this project is independent of Tratter Engineering, BMW sponsored the seat directly, and Professor Haller managed to organize the dashboard from a previous project he conducted in Austria. That way, shipping times could be minimized since no middlemen need to be informed, allowing faster completion time. This represents a vital criterion of this side project because the time

constraints were minimal.

One significant benefit of having a standalone, completely functioning system allowed the developers to fine-tune the prompt sent to the GPT interpreter since they could put themselves into the role of the driver and the passenger. Furthermore, the developers could gain experience working with the preinstalled hardware in the seat and perform extended testing of their voice assistant related to the system's presets, like sleeping, relaxing and driving.

Considering that this side-project aims to fulfil the same task as the construction of the car seat at Tratter Engineering, the project as a whole was able to benefit from this side-project. Since the developers often chose different approaches compared to the engineers at Tratter, it was possible to cherry-pick the best working solutions to be adopted in the final car.

4.2 Modified Hardware Components

Given that the seat uses proprietary connectors and protocols to communicate with the car, changes on the hardware side needed to be performed to expose an API that allows for controlling the seat using self-developed 3rd party software, like this Voice Assistant. Since BMW was unable to share specifications regarding the pinout and protocol for controlling the seat, the developers needed to reverse-engineer the functionality of the seat and find a point of weakness where they could intercept. In the following sub-sections, each necessary change will be explained in detail.

4.2.1 Motors

Overall, there are four motors located in the seat which are responsible for controlling

- the height of the seat
- the distance between seat and steering wheel
- the angle of the backrest
- the inclination of the seat regarding the bottom sitting area

Each motor requires 12 V 10 amperes to function properly. An image of this motor mounted in the seat can be found below:

Figure 4.2: Image of a *Nidec 12V 405.940 (P1-26599-01-03)* motor

Below the motor, a brown connector with four pins can be noticed. Two pins are responsible for powering, and the other two are most likely control pins. The direction of the motor can be changed by swapping those two power pins. Unfortunately, BMW cannot specify how the control pins work, which are most likely for stopping the motor as soon as it approaches its limit. Therefore, a current sensor is used in this project to detect when the motor's energy consumption increases rapidly, which means that the motor is stalling and unable to move further.

4.2.2 Drivers

Since the self-created seat control unit cannot control the motors directly, drivers were needed to accomplish that task. After having difficulties finding drivers that support up to 10 amperes per motor, the MDD20A by the company Cytron Technologies was ordered. Since this driver can already control two motors with 20 amperes, two of these drivers are needed to control all four motors. A detailed image of that driver follows:

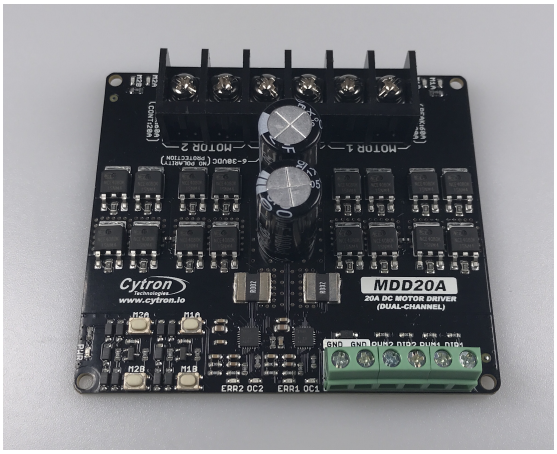

Figure 4.3: Image of an *MDD20A* driver

The four right green pins out of the six green pins in the front of the driver are connected to the GPIO of the seat control unit, which allows for enabling and disabling the two motors as well as for changing direction independently. There are two pins per motor, which need to be configured as shown in the following table:

Pin 1	**Pin 2**	**Result**
LOW	HIGH or LOW	Motor Off
HIGH	LOW	Motor Forward
HIGH	HIGH	Motor Backwards

Figure 4.4: Functioning of the control pins on the *MDD20A*

On the opposite side of the driver, six larger connectors can be seen, which are needed to power the motors. The centre two connectors are connected directly to the 12 V output of the power supply, whereas the left two are connected to one motor and the right two to the other.

4.2.3 Power Supply

A suitable power supply was needed to power all the seat and the seat control unit motors. Since every motor needs about 10 amperes to function correctly even under heavy weight on the seat, the power supply must deliver at least $4 * 10 amperes = 40 amperes$. A power supply from Mean Well USA Inc. was chosen to fulfil these needs, which can be seen in the image below.

Figure 4.5: Image of a *Mean Well HLG-600H-12A* power supply

In particular, this power supply offers a 12 V output with 40 amperes which is exactly what is needed for the motors, but also a 5 V output which perfectly matches with the ESP32 that replaces the pre-installed seat control unit as explained in the next subsection.

It requires 100-240V AC as input, which allows to solder it directly to a Schuko plug. The 12 V output is connected to the two motor drivers presented in the previous section. The 5 V output is soldered to a jumper to power the ESP32.

4.2.4 Seat Control Unit

For building the replica, BMW sent us only the seat and not a complete car, there is no way to work with the already built-in seat control unit. to overcome this challenge, it was decided to use an ESP32 with WiFi and Bluetooth capability that replaces the original seat control unit. An image of that device can be found below:

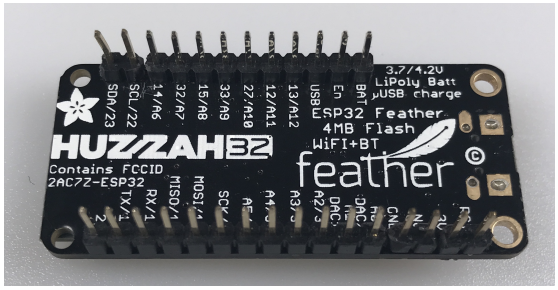

Figure 4.6: Image of a *Huzzah32 feather (ESP32)*

The software part will be discussed in the following section.

4.3 Design of a Seat Control Unit

As mentioned previously, the preinstalled seat control unit was unusable to the developers due to its limited capabilities with communicating to 3rd party software other than the car. Therefore, it was decided to develop an own version of a seat control unit using an ESP32 with WiFi and Bluetooth connectivity. This allows the voice assistant's computer to send commands to the ESP via WiFi to control the installed motors.

In order to establish a connection, the idea was to host a RestAPI on the ESP32 that mimics the specification of seat control unit of Tratter. By doing so, only the endpoint of the "Seat API" module had to be changed to control the prototype at the NOI tech park.

The obtained Seat Control Unit API documentation from Tratter can be found below:

---

title: BMW CS HYPERVISOR author: - name: Roland A. Burger

BMW CS

Draft Technical Notes

BMW HYPERVISOR API This is a preliminary draft note for testing the testing environment of the BMW-CS HYPERVISOR1. Subject to change. Further documentation will follow.

TEST API CALL curl -X POST –header 'Content-Type: application/json' –header 'Accept: application/json' –header 'Grpc-Metadata-Authorization: 1[redacted]19' -d '{"HexId": "73636531", "Input": "DS#01#p100"}' 'https://bmw1.hypercontrol.org/api/scene/runwithcallback'

Figure 4.7: Usage of the Seat Control Unit API

To prevent the leaking of sensitive information, sensitive codes have been blacked out. The most important part in this document is the "Input" property in the second last line. It specifies which seat to move, what motor to move and how much to move the motor.

The value in front of the first # specifies the seat, which can either be *DS* for the driver seat or *PS* for the passenger seat. After that, the motor needs to be clarified. Each motor has its own ID. In this example, *01* moves the seat forward and backwards. The last value after the last # specifies the distance to move the motor. Therefore, *p* specifies a positive number and *n* a negative number. In the given example, *p100* moves the seat forward by 100 units.

To understand how the ESP works internally, the used code will be analyzed in the following paragraph. The whole code can be found in the GitHub repository (`https://github.com/EliasBinder/ESP32SeatAPI`). This section, however, concentrates on the most important code snippets.

To create a Web Server, the Arduino WebServer library is used. The following code snippet explains how the WebServer is set up.

```
1 WebServer server(80);
2 server.on("/api/scene/runwithcallback", HTTP_POST, handleRequest);
3 server.begin();
```

The Server runs on port 80 and exposes one POST endpoint at */api/scene/runwithcallback*. The *handleRequest* method is used to process requests on that path.

```
void handleRequest() {
  if (server.hasArg("plain") == false) {
    Serial.write("Invalid body!");
  }

  DynamicJsonDocument doc(1024);
  String body = server.arg("plain");
  deserializeJson(doc, body.c_str());

  String input = doc["Input"];

  String id = input.substring(3, 5);

  String value = input.substring(6, value.length() - 1);

  String annotation = value.substring(0, 1);
  int valToMove = value.substring(1, value.length() - 1).toInt();

  if (id.equals("00")) {
    allMotorsOff();

  } else if (id.equals("01")) {
    if (annotation.equals("p")){
      motorForward(0, valToMove);
    } else {
      motorBackward(0, valToMove);
    }


  ...

  } else if (id.equals("04")) {
    if (annotation.equals("p")){
      motorForward(3, valToMove);
    } else {
      motorBackward(3, valToMove);
    }
  }

  String resp = "{\"Output\": \"OK-STATUS#BMWCS-HYP1: #0xC8-
      CONFIRMED " + input + "\"}";

  server.send(200, "application/json", resp.c_str());
}
```

By using the *ArduinoJson* library, the POST request body can be processed to determine the motor ID and the distance to move the motor. After calling the auxiliary methods to act, a response is sent back that mimics the same format as the response from the API of Tratter Engineering. For controlling the motors, the code snippet below is used.

```
static void allMotorsOff() {
  Serial.write("\nAll motors off");
  for(int i = 0; i < 4; i++) {
    motorOff(i);
  }
}

```

```
8
9  static void motorOff(int id) {
10   digitalWrite(motorPins[id][0], LOW);
11   digitalWrite(motorPins[id][1], LOW);
12 }
13
14 static void motorForward(int id, int value) {
15   digitalWrite(motorPins[id][0], HIGH);
16   digitalWrite(motorPins[id][1], LOW);
17 }
18
19 static void motorBackward(int id, int value) {
20   digitalWrite(motorPins[id][0], HIGH);
21   digitalWrite(motorPins[id][1], HIGH);
22 }
```

For every motor, two control pins are used each. If both pins are LOW, the motor is not moving. To move the motor forward, pin 1 must be HIGH, the other LOW. To change direction and move the motor backwards, pin 1 needs to be LOW and pin 2 HIGH. The control pins are directly connected to the motor drivers, which then coordinate the 12 volts that drive the motor.

4.4 Challenges

During the development of the replica, the developers encountered several difficulties. The most challenging ones will be discussed in the following paragraph, and solutions for these problems will be provided.

First, finding working drivers for the motors turned out to be tough. As explained before, the motors require 10 amperes each. The motors did not work correctly because most ESP-compatible drivers support up to 2 amperes. Running the preinstalled motors with 2 amperes, they could barely move the seat and stopped working entirely as soon as a person was sitting on the seat. This problem was resolved by performing extended research by the developers. After ordering working drivers and a suitable power supply to power four motors simultaneously, all motors worked as expected.

Due to missing hardware specifications, the developers had to reverse-engineer the functioning of the seat. Therefore, the cabling from the preinstalled seat control unit to the motors has been analyzed, and the decision was made to replace the control unit with a self-made one. After a time-consuming trial and error testing session, the essential wires were found and connected to the upgraded seat control unit.

The most challenging task, however, was to stop the motors automatically when the seat could not move any further. Due to the lack of documentation mentioned earlier, the two unknown pins of the motor, which most likely are responsible for exactly that task, could not be used. Instead, the developers needed to install a current sensor and monitor the energy consumption of the motors. As soon as they hit their limit, the motors stall, so energy consumption increases drastically. With the current sensor, this consumption can be monitored, and as soon as it exceeds a predefined limit, the motors are stopped.

Chapter 5

Limits and Future Prospects

5.1 Limits

Although voice input fundamentally simplifies the car's operation, giving precise commands is not always possible. Using a voice assistant does not allow the user to fine-tune the configuration of the seat since it would be hard for the operator to express precisely where to move every single motor. It is unrealistic to assume that the user formulates queries like "Move my seat forward by 14 centimetres" or "Change the angle of my backrest to 143 degrees". Instead, the user may use phrases like "Move my seat a little forward". The system must guess what the user means by saying "little" and move the seat accordingly. Obviously, this is not a reliable method the change the seat configuration. As a result, the user may prefer other input methods, such as a physical slider for precisely moving the seat.

Another limit is the dependency on internet connectivity. As described in previous sections, this voice assistant works best with a stable and fast internet connection, since this eliminates the need of running local models. However, assuming these circumstances in any case is a huge mistake. Therefore, the developers must think of fallback solutions that work offline. Regarding the Whisper model, a suitable solution would be to run a smaller-sized model locally. For the GPT interpretation part, an open-source LLM, like Llama2 by Facebook, can be considered as a replacement to ensure offline functionality. The problem that arises then is the missing computational resources inside the car, to run two highly demanding AI models simultaneously and in real-time. It can be argued that the capabilities of a car in terms of computational power are increasing a lot since autonomous driving, which many manufacturers claim to have as a goal, requires many resources.

5.2 Future Prospects

In conclusion, it can be stated that there are still a few construction sites in the software, which will be presented in this section.

First, the interpreter used for this voice assistant is the basic, unmodified GPT model. Given that OpenAI provides functionality to fine-tune the GPT model with domain-related data, the boundaries of the interpreter module can be pushed again. Data that refers to the structure of a car or additional names for seating positions can be added to make badly formulated queries work more reliably.

Another improvement that can be made is compressing the transcription audio wave file. Since no compression algorithm is used for saving the microphone input, sending this .wav file to a data centre of OpenAI for transcription takes a while. This can be improved by applying a compression algorithm before submitting the file. Since audio quality is not important as long as Whisper is able to transcribe correctly, even a lossy compression algorithm can be chosen.

To improve the correctness of the self-created Wakeword AI, which detects the phrase "Hey BMW", a larger training set is recommended. Until now, the sample data comprises 100 audio samples from two persons pronouncing "Hey BMW". Using more persons of different ages and genders will confront the model with a larger tone variety of this phrase, resulting in an improved recognition score.

Bibliography

[] *2.1.5 Speech Files: Microsoft's WAV Format*. Tech. rep. The Institute for Signal and Information Processing. URL: https://isip.piconepress.com/projects/speech/software/tutorials/production/fundamentals/v1.0/section_02/s02_01_p05.html.

[Ale] etc. Alec Radford Jong Wook Kim. *Robust Speech Recognition via Large-Scale Weak Supervision*. Tech. rep. OpenAI. URL: https://cdn.openai.com/papers/whisper.pdf.

[AS20] Hornby A.S. *Oxford advanced learner's dictionary of current English 10th Edition*. Oxford University Press, 2020.

[Cha23] Avi Chawla. "LLM training and fine-tuning". In: *toloka.ai Blog* (June 29, 2023). URL: https://toloka.ai/blog/how-llms-are-trained/.

[KG] Spectra GmbH Co. KG. "Spectra PowerBox 300 Advanced 2". In: (). URL: https://industrie-pc.de/en/Box-PC-Systems/Spectra-PowerBox-300-Advanced-2.

[Kim22a] Jong Wook Kim. *Python usage*. Version 20230314. Oct. 21, 2022. URL: bit.ly/3ZesAkz.

[Kim22b] Jong Wook Kim. "Whisper: Available models and languages". In: *Whisper GitHub Repository* (2022). URL: https://github.com/openai/whisper#available-models-and-languages.

[Kle23] Guillaume Klein. *CTranslate2*. Aug. 30, 2023. URL: https://github.com/OpenNMT/CTranslate2/blame/d240717abadad2f89fa77339548a924707088fe0/README.md.

[Ltd18] miniDSP Ltd. *UMA-8 V2 USB MICROPHONE ARRAY WITH EMBEDDED DSP User Manual*. English. Version 1.4. miniDSP Ltd, Hong Kong. Sept. 14, 2018. 18 pp. URL: https://www.minidsp.com/images/documents/UMA-8%20v2%20User%20manual.pdf.

[Mal] Anton Malyy. *Overview of Modular Architecture*. URL: https://triare.net/insights/modular-architecture-2/.

[Mic19] Microsoft. *Which digital assistant are you using?* Tech. rep. Microsoft, 2019. URL: https://www.statista.com/statistics/1134020/digital-assistants-usage-worldwide/#:~:text=Apple's%20Siri%20and%20Google%20Assistant,by%2019%20percent%20of%20respondents..

[Ope22] OpenA. "Introducing Whisper". In: *OpenAI research* (Sept. 21, 2022). URL: https://openai.com/research/whisper.

[Res20] Juniper Research. *Number of digital voice assistants in use worldwide from 2019 to 2024 (in billions)*. Tech. rep. Voicebot.ai; Business Wire, 2020. URL: https://www.statista.com/statistics/973815/worldwide-digital-voice-assistant-in-use/.

[Sea22] Andrew Seagraves. "Benchmarking Top Open Source Speech Recognition Models: Whisper, Facebook wav2vec2, and Kaldi". In: *Deepgram* (2022). URL: https://deepgram.com/learn/benchmarking-top-open-source-speech-models.

[She21] Riya Maurya / Priti Koutikkar / Saaish Gulekar / Gyan Shetty. *EVOLUTION OF VOICE ASSISTANT*. Tech. rep. International Research Journal of Engineering and Technology (IRJET), 2021. URL: https://www.irjet.net/archives/V8/i4/IRJET-V8I4963.pdf.

[Shi23] Rebecca U. Shin. "Large Language Models 101: Everything You Need to Know". In: *coveo* (June 6, 2023). URL: https://www.coveo.com/blog/what-are-large-language-models/.

[shi23] shime. *play-sound/index.js*. Version 1.1.6. Aug. 24, 2023. URL: https://github.com/shime/play-sound/blob/25989f8bcd71665456176174d70c8c75f55bad81/index.js.

[sna11] snapfractalpop. *Finding the 'volume' of a .wav at a given time*. 2011. URL: https://stackoverflow.com/a/8282438.

Appendix A

Motivation

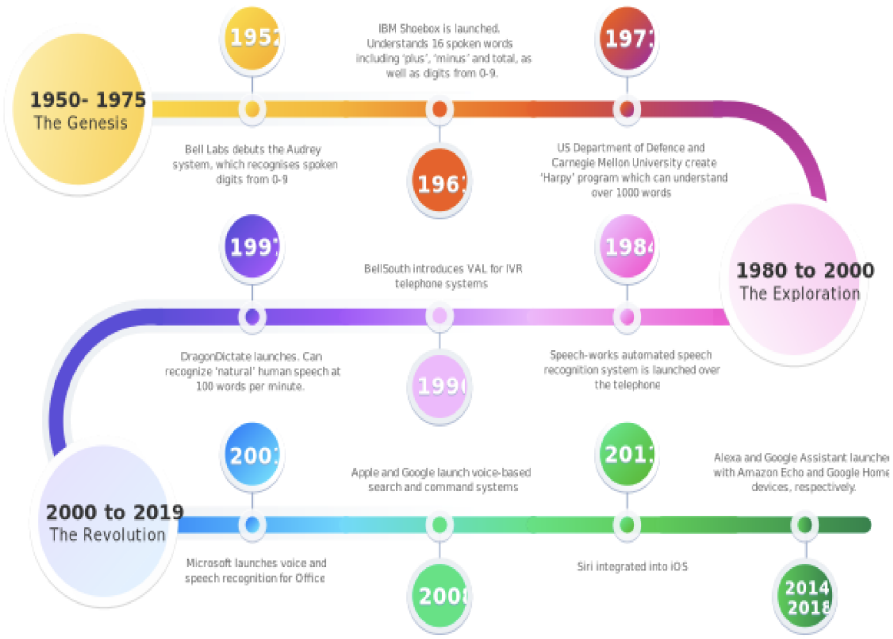

Figure A.1: Decades of Voice Assistants [She21]

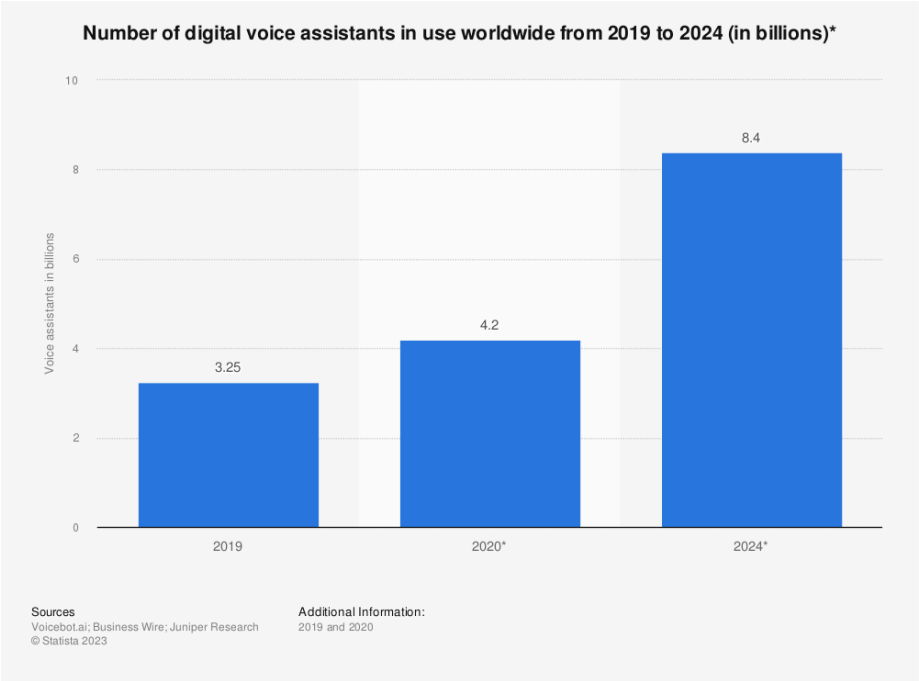

Figure A.2: Number of digital voice assistants in use worldwide from 2019 to 2024 (in billions) [Res20]

In 2020, there will be 4.2 billion digital voice assistants being used in devices around the world. Forecasts suggest that by 2024, the number of digital voice assistants will reach 8.4 billion units – a number higher than the world's population. [Res20]

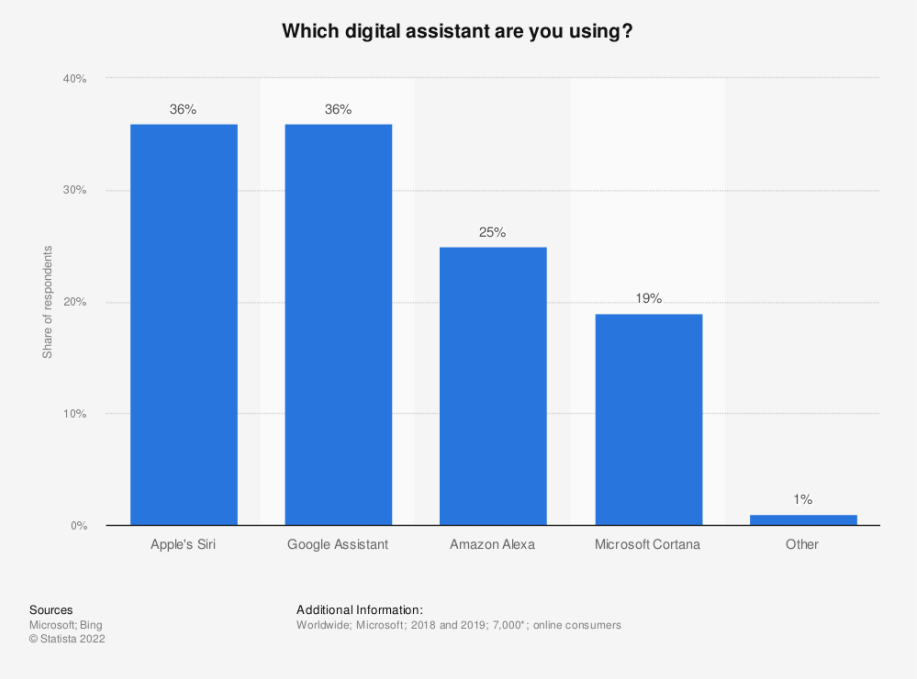

Figure A.3: Which digital assistant are you using? [Mic19]

Apple's Siri and Google Assistant were the most popular digital assistants worldwide as of 2019, each used by 36 percent of the survey respondents. Amazon Alexa was used by 25 percent and Microsoft Cortana was used by 19 percent of respondents. [Mic19]

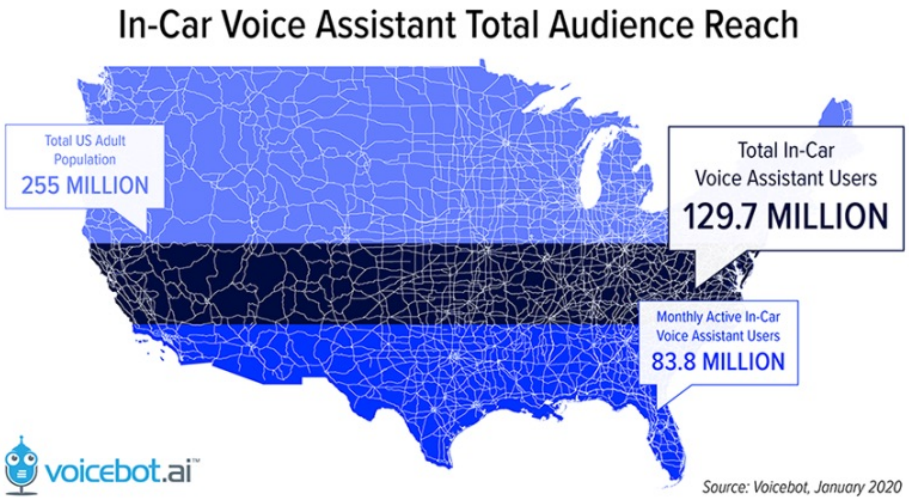

Figure A.4: In-Car Voice Assistant Total Audience Reach

Appendix B

Project Structure

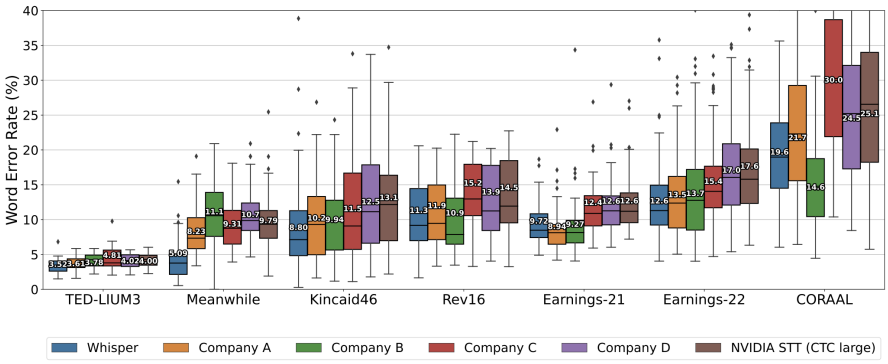

Figure B.1: Whisper is competitive with state-of-the-art commercial and open-source ASR systems in long-form transcription. [Ale]

The distribution of word error rates from six ASR systems on seven long-form datasets are compared, where the input lengths range from a few minutes to a few hours. The boxes show the quartiles of per-example WERs, and the per-dataset aggregate WERs are annotated on each box. Our model outperforms the best open source model (NVIDIA STT) on all datasets, and in most cases, commercial ASR systems as well. [Ale]

Appendix C

Modules

Wakeword AI
The self-created Wakeword AI based on the official TensorFlow Guide (`https://www.tensorflow.org/tutorials/audio/simple_audio`) can be found in the following GitHub repository: `https://github.com/EliasBinder/BMWSeat-WakewordAI`

Main Application Module

```
1  //Setup Rest API
2  startRestAPI();
3
4  export const wake = () => {
5      analyzeStream(async () => {
6          console.log('System is not listening...');
7          const text = await stopTranscriptionMicrophone();
8          //const direction = await stopFetchMicrophoneInterrupts();
9          console.log('Transcription: ', text);
10         if (text.trim() !== '')
11             interpretCommand(text, 1);
12     });
13
14     transcribeMicrophone();
15     console.log('System is awake!');
16     //fetchMicrophoneInterrupts();
17 }
18
19 const interpretCommand = async (command: string, direction: number)
       => {
20     try {
21         const gptResponse = await interpretMessage(command);
22         console.log('GPT Response: ', JSON.stringify(gptResponse));
23         processResponse(gptResponse, direction >= 0 ? 'DS' : 'PS');
24     } catch (e) {
25         playAudio('error.mp3');
26         console.log('GPT Response JSON: ', e);
27     }
28 }
```