RWTH Aachen University
Software Engineering Group

Enabling Voice Input for Data Entry through LLMs in MontiGem

Bachelor Thesis

presented by
Wellmeyer, Louis

1st Examiner: Univ.-Prof. Dr. rer. nat. Rumpe, Bernhard Dieter

2nd Examiner: Univ.-Prof. Dr. rer. nat. Lichter, Horst

Advisor: Arkadii Gerasimov, Constantin Buschhaus

The present work was submitted to the Chair of Software Engineering

Aachen, April 2, 2025

Eidesstattliche Versicherung

Wellmeyer, Louis
Name, Vorname

434261
Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit~~/Bachelorarbeit/ ~~Masterarbeit~~* mit dem Titel

Enabling Voice Input for Data Entry through LLMs in MoufiGem

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 02.04.2025
Ort, Datum

L. Wellmeyer
Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 02.04.2025
Ort, Datum

L. Wellmeyer
Unterschrift

Abstract

The shift toward electric vehicles (EV) presents new challenges in establishing a sustainable circular economy. While production continues to scale, the disassembly and recycling of EV batteries remain labor-intensive, costly, and prone to inefficiencies due to the growing variety of battery designs. The DemoRec project at RWTH Aachen University addresses this issue by developing automated solutions for large-scale battery disassembly. A critical part of this process is the accurate documentation of disassembly data, which currently requires workers to frequently switch between manual tasks and digital data entry. To streamline this workflow, a voice-controlled data entry framework was developed, enabling hands-free interaction with MontiGem-generated web applications. By leveraging artificial intelligence technologies, including Azure OpenAI services, Whisper for speech-to-text transcription, and GPT-4o with function calling capabilities, the system transforms spoken commands into executable browser actions, reducing the need for manual input and minimizing disruptions to the disassembly process. Performance evaluations demonstrated that the system maintains high accuracy and efficiency across different voice input modalities. Explicit voice commands achieved a 100% function resolution rate, while natural voice commands provided a faster interaction experience but with lower consistency (60% accuracy). Even in high-volume background noise, the system sustained up to 93.34% function resolution accuracy, ensuring robustness in industrial environments. Notably, inaccuracies were primarily due to unexecuted functions rather than incorrect actions, with misfires occurring in less than 1% of test cases using clean audio inputs. Future research directions include improving command handling for overlapping voice inputs and developing domain-specific language models to further enhance accuracy and usability.

Contents

Chapter 1

Introduction

This chapter introduces the topic of this thesis by outlining its objectives and providing an overview of its structure. The following sections will first present the motivation behind this research, highlighting the challenges and significance of the problem at hand. Subsequently, the research goals of this work will be defined, and the functional and non-functional requirements of the proposed system will be specified. Finally, a structured outline of this work will be provided.

1.1 Motivation

The increasing adoption of electric vehicles (EV) brings forth new challenges in creating a sustainable economy. While the electric automotive industry continues to grow in production, dismantling and recycling batteries become increasingly difficult as the variety widens. Therefore, disassembly for recycling EV batteries remains largely manual, time-intensive, and costly. The DemoRec project at RWTH Aachen University's Chair of Production Engineering of E-Mobility Components (PEM) addresses this challenge by developing automated solutions for industrial scale battery disassembly. [oEMC24]

A critical aspect of the battery disassembly process is the accurate documentation and tracking of disassembly information. Currently, workers must alternate between physical disassembly tasks and manual data entry through a web-based interface generated by MontiGem3 [AMN+20, Rum21], a tool developed at the Chair of Software Engineering at RWTH Aachen University to generate web applications from class diagrams. This frequent alternation between data entry and dismantling not only reduces operational efficiency but also introduces potential for documentation errors and workflow disruptions.

1.2 Objective

To facilitate and simplify the data entry process, this thesis proposes the development of a voice-controlled framework that enables hands-free interactions within the web interface used in the DemoRec project. By leveraging speech recognition and Large Language Models (LLM) with function-calling capabilities, the framework aims to transform spoken commands into executable browser actions, thereby streamlining the data entry process while maintaining accuracy and reliability.

While the immediate aim is the facility for DemoRec, the framework is designed to be modular and adaptable, ensuring its applicability across a wide range of web applications beyond MontiGem-generated ones. The ability to interact with web interfaces through voice input presents advantages across multiple domains. In industrial environments, such as manufacturing and quality control, workers often need to document processes while working on other tasks, making traditional input methods inefficient. Similarly, in healthcare, assistive technologies, and smart home automation, voice interfaces enhance accessibility and ease of use. The automotive sector also benefits from voice-controlled systems by enabling safer interactions with digital interfaces, while education and training environments can leverage such technologies to create more intuitive and engaging learning experiences.

By developing a modular voice-control framework, this research aims to improve accessibility and efficiency in web applications while demonstrating the potential of LLM-driven speech interfaces across different domains. The insights of this research will support the development of more adaptable and efficient voice-based systems, broadening their application across different industries and user scenarios.

1.3 Requirements

To ensure the proposed voice-controlled framework meets the practical demands of DemoRec in industrial environments, it must fulfill several functional and non-functional requirements. Functional requirements define the system's expected capabilities, such as the tasks or actions it should perform, while non-functional requirements focus on the system's performance characteristics, such as reliability, usability, and efficiency. These requirements ensure the system's effectiveness in the DemoRec project and its potential applicability in other industrial contexts.

1.3.1 Functional Requirements

The system must provide reliable speech recognition and command execution to enhance the efficiency of data entry in battery disassembly workflows. First, it should accurately interpret spoken instructions and translate them into executable actions within the web-based interface used in the DemoRec project. The commands should cover a range of functionalities, including data input, navigation within the interface, and execution of predefined tasks. Additionally, the system must support real-time feedback, enabling users to receive confirmation and understanding of recognized commands, and correct misinterpretations if necessary.

While various approaches exist for automating data entry, directly inserting information into the database via voice commands was considered but deemed as not suitable for this application. Direct modification of database records without a clear intermediate step could lead to unintended errors, reduce transparency, and make it harder for users to verify correctness.

To ensure a user-friendly and reliable process, the system will control web elements directly within the existing interface before submission. This approach allows workers to visually confirm the recognized input and make manual corrections if necessary. Providing immediate feedback within the interface ensures users have a clear understanding of what

actions are being executed, improving trust in the system while maintaining alignment with the existing documentation workflow.

Integration with MontiGem-generated web applications is another essential requirement. The framework should be designed in a way that allows seamless interaction with existing MontiGem web interfaces without necessitating major modifications or configurations. Furthermore, given the variability in industrial workflows, the system should provide flexibility by allowing users to define and modify voice input and detected commands to accommodate specific needs.

1.3.2 Non-functional Requirements

Beyond functional correctness, the system must exhibit high recognition accuracy, even in potentially noisy industrial environments. Since battery disassembly settings are expected to have background noise from machinery and other operations, the speech recognition model must be robust enough to maintain a high success rate in correctly understanding commands.

Another key requirement is low-latency processing. The system should execute spoken commands with minimal delay to ensure a smooth and uninterrupted workflow. Any significant delay in processing could reduce usability and decrease the efficiency gains expected.

Security considerations must also be addressed, ensuring that interactions with the web application are controlled and protected against unintended operations. Preventing erroneous command execution is crucial.

Scalability and portability is another important factor, as the developed framework should not be limited to the DemoRec project. The architecture should allow for adaptation to other web-based applications, particularly those requiring hands-free operation in manufacturing and recycling environments. Finally, usability must be considered. The system should be intuitive and easy to use, minimizing the need for extensive configurations and allowing workers to integrate voice commands naturally into their existing workflow.

1.4 Structure

The structure of this bachelor thesis is as follows: Chapter 2 presents the key concepts relevant to this work, including speech recognition technologies, function calling in the context of large language models, and the MontiGem system. These topics provide the necessary theoretical background for understanding the system of the proposed voice-controlled framework. Chapter 3 details the implementation of the system, outlining its architecture, key design decisions, and technical components. It describes the speech recognition pipeline, command execution mechanisms, and the integration of the tool within web interfaces. Chapter 4 evaluates the system's performance, focusing on latency measurements for different communication mechanisms, robustness in noisy environments, and common edge cases relevant to real-world industrial applications. Finally, Chapter 5 concludes the thesis with a summary of the results and an outlook on potential further developments and applications.

Chapter 2

Background and Related Work

Chapter 1 introduced key concepts relevant to this thesis, including speech recognition, function calling, and MontiGem-generated web applications. However, a deeper understanding of these topics is necessary to establish a solid foundation for the proposed system's implementation.

To provide this foundation, this chapter first examines the evolution of Intelligent Virtual Assistants (IVAs), analyzing their historical performance, challenges in voice recognition, and user experience. This discussion serves as a justification and reference point for this works requirements. Afterward, speech recognition technologies are explored in greater depth, highlighting their capabilities, limitations, and considerations for industrial applications. The chapter then introduces function calling in the context of LLMs, explaining how natural language can be converted into executable browser actions. Finally, an overview of MontiGem and its role in generating web applications is presented, detailing how its model-driven approach influences the integration of external functionalities such as the tool developed in this thesis.

2.1 Intelligent Virtual Assistants of the Past

Intelligent Virtual Assistants (IVAs) have become increasingly prevalent in daily lives, with widespread adoption through smartphones enabling access to assistants like Siri on iPhone, Google Assistant on Android devices, Cortana on Windows 10, and Alexa as home speakers [TD18]. These technologies represented a significant advancement in human-computer interaction, allowing users to interact with machines through natural language. Despite their growing popularity, research has identified persistent challenges in three key areas: voice recognition, contextual understanding, and hands-free interaction. A comprehensive survey of 100 users revealed substantial performance differences among major IVAs. While these assistants collectively could answer approximately 17.35% of daily questions, Google Assistant demonstrated superior capability by successfully responding to 59.80% of queries. Siri performed below expectations at 45%, Cortana managed only one-third of questions, and Alexa significantly underperformed at just 7.91% [TD18]. These findings highlight considerable room for improvement in IVA technologies, particularly in handling everyday user inquiries. The survey also revealed a notably poor retention rate among IVA users, with only 25% reporting frequent daily usage [TD18]. This suggests that despite their potential convenience, current implementations fail to deliver sufficient value to maintain

consistent user engagement. However, the research also identified significant potential for IVAs, particularly for users with cognitive disabilities who struggle with forming complete sentences and communication, indicating important accessibility applications.

These findings on prominent IVAs from over half a decade ago provide a critical baseline for my current voice control tool for web applications. By examining whether the identified limitations in voice recognition, contextual understanding, and hands-free interaction have been adequately addressed over the years, my research explores how these persistent challenges can be overcome. Furthermore, the documented low retention rates highlight the importance of identifying what users truly value in voice-controlled interfaces. By applying lessons from these earlier findings to more specialized use cases, my research aims to develop voice control systems that better align with user expectations and needs, providing higher accuracy with the novel approach of LLM function calling, and consequently potentially open new opportunities in education, banking, business, counseling, sales, and other domains suggested by the original research [TD18].

2.2 Speech Recognition Systems

For the success of voice-controlled browser interactions, it is crucial to select a robust and reliable speech recognition system. The chosen system must perform reliably under real-world conditions, including the presence of environmental noise, varied acoustic settings, and diverse speaker characteristics. Recent advancements in Automatic Speech Recognition (ASR) offer valuable benchmarks for assessing system performance, helping to guide the selection of appropriate technology for this research.

The Speech Robust Bench (SRB) [SNH+24], evaluates ASR across 114 different perturbation types and various noisy data sources. This benchmark offers a more thorough view of ASR performance compared to previous evaluations, which were limited to specific perturbation types. The SRB study assessed several popular ASR models, such as different variants of Whisper, Wav2Vec, HuBERT, and Nvidia's Canary. Key findings from the SRB benchmark revealed that Whisper Large demonstrated superior overall robustness across various perturbations, making it particularly suitable for real-world applications. Furthermore, larger models generally exhibited better robustness, contrary to previous assumptions about model size and performance. Different models showed strengths in handling specific perturbations, but Whisper performed consistently well across most categories.

Notably, while Whisper Large shows the best overall performance, certain alternatives demonstrate superior capabilities in specific scenarios. The MMS (Massively Multilingual Speech) model, for instance, excels particularly in social scenarios and far-field recordings, which could be relevant in noisy industrial environments with multiple speakers. Additionally, Nvidia's Canary model, while more sensitive to echo and pitch modifications, demonstrates stronger performance on clean audio data, suggesting it might be preferable in more controlled environments.

2.2.1 Whisper - Web-scale Supervised Pretraining for Speech Recognition

Whisper is an ASR model developed by OpenAI that, unlike many recent ASR advancements that rely on self-supervised learning techniques (e.g., Wav2Vec 2.0 [BZMA20]), is

trained on an extensive dataset of 680,000 hours of labeled audio data [RKX+22]. Of these, 117,000 hours cover 96 languages, making it highly capable in multilingual speech recognition tasks. Whisper does not require self-supervision or self-training techniques, reducing the need for fine-tuning and making it well-suited for zero-shot ASR applications.

The model employs an encoder-decoder transformer architecture, a choice validated for its scalability and robustness [RKX+22]. Whisper also incorporates a multitask training format, supporting functionalities like voice activity detection, speaker identification, inverse text normalization, and language filtering.

A key advantage of Whisper is its ability to perform naturalistic transcription without requiring additional text normalization steps. This is achieved by training the model to predict raw text transcripts directly, leveraging large-scale datasets sourced from the internet. However, to maintain transcript quality, OpenAI implemented automated filtering methods to remove unreliable or low-quality data. [RKX+22]

2.2.2 Considerations

For the DemoRec project, Whisper emerges as an ideal choice for several reasons: Its demonstrated robustness across various perturbation types makes it suitable for industrial environments, such as battery disassembly, where background noise and varying acoustic conditions are expected. Additionally, the SRB benchmark demonstrated that Whisper maintains consistent performance across different speaker characteristics and environmental conditions, making it well-suited for use by multiple workers [SNH+24]. Furthermore, Whisper offers scalability with different model sizes, allowing flexibility in deployment to meet various systems.

Whisper does have limitations, despite its performance. The model can sometimes produce hallucinated transcripts, omit words at the beginning or end of utterances, or generate repeated loops in transcription. Additionally, while Whisper is highly effective in zero-shot settings, its performance can be further improved through fine-tuning on high-quality supervised datasets. Another consideration is that Whisper's performance in certain languages remains weaker due to insufficient training data, highlighting the potential benefits of incorporating self-supervised learning or reinforcement learning approaches in future iterations. [RKX+22]

2.2.3 Conclusion

Whisper achieves state-of-the-art robustness across various noise perturbations, making it particularly suitable for industrial applications requiring reliable speech recognition [SNH+24]. While alternative models may offer advantages in specific scenarios, Whisper generalizes well across multiple situations ensuring suitability for the DemoRec project. Further fine-tuning (see 2.3.3) and unsupervised training techniques could further enhance its performance.

2.3 Function Calling in Large Language Models

LLMs have revolutionized artificial intelligence by enabling machines to understand and generate human-like text. At their core, language models predict the probability of future

or missing words in a sequence [ZZL+24]. While early language models primarily supported tasks like retrieval and speech recognition, modern LLMs, such as GPT-4, serve as general-purpose problem solvers, significantly broadening their applications [ea24].

Recent advancements in scaling model capacity have enhanced LLMs' ability to handle complex tasks with minimal customization. This evolution has fueled discussions on artificial general intelligence (AGI) and AI's broader impact. However, despite their capabilities, models still face challenges [SZX+24] in reasoning and executing precise operations, such as mathematical calculations or logical inferences, that heavily control value and user retention of virtual intelligent assistants [TD18].

With the release of gpt-4-turbo on June 13, 2023 [Ope23], the recent versions of gpt-3.5-turbo and gpt-4, are trained to be able to determine the use of specified tools. This capability addresses several fundamental challenges of LLMs: Although large language models excel at natural language processing and generation, they face significant limitations that impact their reliability and practical utility. These limitations include hallucinations (generating fabricated or incorrect information), poor mathematical reasoning, and limited context windows. Most critically, LLMs struggle with complex computational tasks, specialized domain reasoning, and accessing real-time information beyond their training data [SZX+24].

By utilizing external tools to feed models with real-time data, system interactions, and computation results, models can overcome these limitations. Function calling creates a bridge between the LLM's natural language capabilities and specialized external systems, allowing for enhanced problem-solving abilities.There are several key purposes for function calling [Ope23]: The ability for LLMs to determine defined functions based on natural language input allows them to interact directly with external systems, enabling the models to leverage real-time data and system functionalities. The limitations of LLMs, including reasoning capabilities in specific domains like mathematics and scientific computation [SZX+24] can be overcome by delegating specific tasks to specialized tools. This approach not only enhances reliability but also extends the model's effective capabilities by combining natural language understanding with purpose-built computational tools, verification systems, and domain-specific tasks . Furthermore, with the use of function calling, raw data and natural language can efficiently be transformed into specifically structured data for other purposes. As demonstrated in this work, it also enables direct modification of application user interfaces through user requests. Previously, function calling enables models to determine a single function based on the given query. Introduced with versions of gpt-4 and gpt-3.5-turbo models after November 6, 2023, models can now execute multiple function calls simultaneously, improving the efficiency and decreasing cost as multiple tools can be invoked with only one request [Ope23].

2.3.1 Architecture

Function calling follows a structured communication pipeline between the system and the LLM, as illustrated in Figure 2.1. This interaction consists of the following steps:

1. **Initialization**: The application initiates the process by sending an API request. This request includes the user's prompt as well as definitions of callable functions that the LLM can utilize.

2. **Evaluation**: The LLM processes the input to determine whether it can directly respond to the prompt or if one or more functions should be invoked to fulfill the request. If functions are required, the API returns a structured response containing the names of the selected functions along with their associated argument names and values, which are extracted from the user's prompt.

3. **Function Execution**: The application uses the returned information to execute the selected functions in its local environment, potentially modifying the state of the application or retrieving necessary data.

4. **Optional API Call**: The application can make an additional API call, including the original prompt along with the results of the executed functions. This step allows the LLM to incorporate the outcomes of the function execution into its reasoning.

5. **Optional Final Response** If an additional call is made, the LLM processes the updated input and generates a final response that fulfills the original user request.

The specified function definitions that are sent to the LLM for evaluation of use are formatted in JSON schemas following the standard [Tea25] for structure, types, and validation rules. OpenAI extends this standard with custom fields to define function metadata:

- `name` - Name of the function.
- `description` - Explanation of what the function does.
- `parameters` - A detailed schema of the function's parameters according to the JSON standard [Tea25].

For a function to be usable as a tool by the LLM, these three fields are mandatory to define.

Although JSON schemas can be defined manually, [Ope23] highlights that developers can simplify the process by using schema definition tools like Pydantic and Zod. Pydantic [Pyd] is a Python library for data validation that uses Python type annotations and offers automatic conversion to JSON schemas. Zod [Zod], on the other hand, is a TypeScript schema declaration and validation library that can generate JSON schemas. However, it's important to note that not all features of these tools are fully supported in the function calling context. When precise control over the schema structure is needed, defining the schema directly will always remain the most reliable approach.

In the case of frontend manipulation for data entry, a simplified schema for a text field control function can look like Listing 2.1.

Tool Selection Control

As mentioned in Section 2.3.1, recent versions of GPT-4 and GPT-3.5-turbo models support simultaneous execution of multiple function calls, enhancing efficiency and reducing costs by invoking several tools with a single request [Ope23]. The `parallel_tool_calls` parameter controls this behavior. When set to true (the default), the model is allowed to call multiple functions concurrently. Conversely, setting `parallel_tool_calls` to

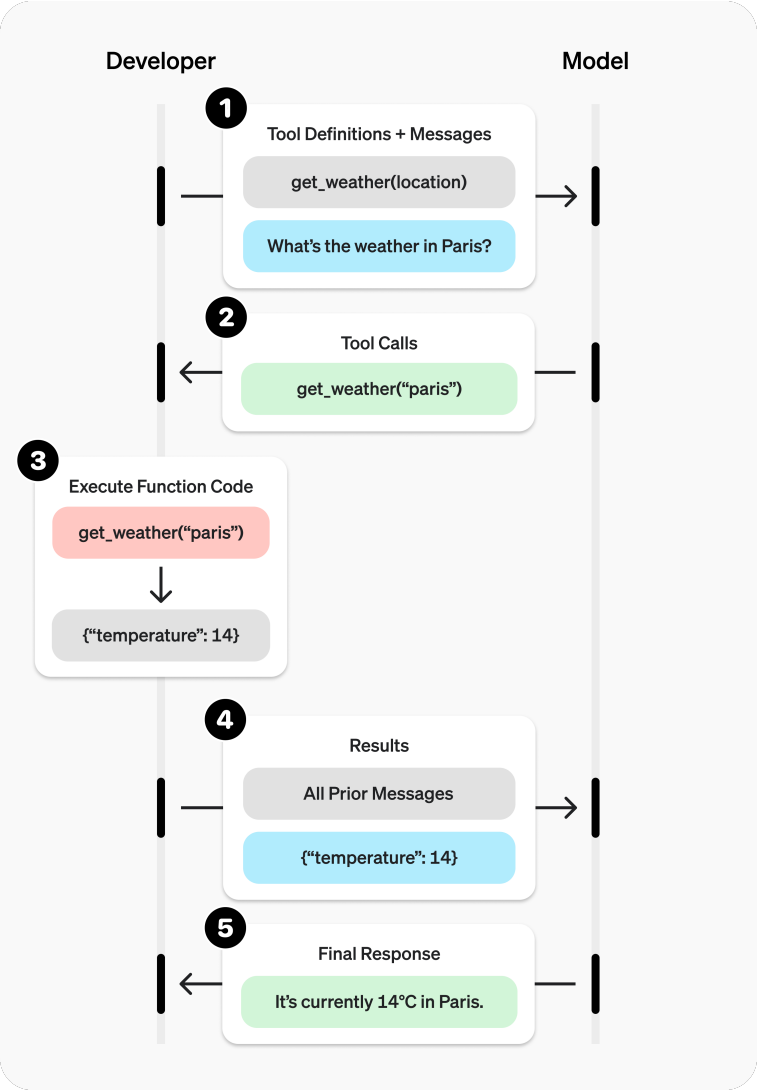

Figure 2.1: OpenAI function calling pipeline [Ope23]

false restricts the model to invoking exactly zero or one tool per turn. Such functionality proves particularly useful in applications that require sequential function execution or when parallel operations could potentially lead to conflicts in the system state.

Furthermore, the LLM's tool selection behavior can be configured using the `tool_choice` parameter, which offers several modes:

- **Auto** (Default): Allows the model to determine whether to use zero, one, or multiple functions based on the context
- **Required**: Forces the model to call at least one function
- **Forced Function**: Forces the model to call exactly one designated function
- **None**: Simulates behavior as if no functions were provided

```
{
    "name": "sayHello",
    "description": "Outputs a greeting message.",
    "type": "object",
    "parameters": {
        "properties": {
            "name": {
                "type": "string",
                "description": "The name of the person to
                   greet. Defaults to 'World'.",
                "default": "World"
            }
        },
        "required": []
    }
}
```

Listing 2.1: Example JSON schema for the function `sayHello` to generate a greeting message based on the optional input name. If no name is provided, it defaults to greeting "World".

The flexibility of this configuration allows developers to implement precise control over the function calling behavior, which is particularly useful when specific tool usage patterns are required.

To ensure strict adherence to function schemas, strict mode can be enabled in the function calling implementation. This feature enforces schema compliance by always responding with JSON structured outputs. This way, the model wont omit required keys or hallucinate invalid enum values, making responses more reliable and simplifying prompts as return formats do not need to be specified. Nevertheless, it introduces certain requirements. Specifically, object parameters must have `additionalProperties` set to `false`, and all fields in properties must be marked as required. Although strict mode is recommended for most implementations, it comes with several limitations that should be considered. These include limited support for certain JSON schema features, additional processing time for schema validation during initial requests, temporary data storage for validation that may violate zero data retention policies, and performance impacts due to schema caching [Ope23]. Therefore these factors are particularly important to consider for systems that require high performance or have specific policies.

Best Practices

Clear and concise function names, parameters, and outputs, as well as specifying each tool's purpose are important. It advises providing examples and edge cases where relevant, combining related functions to simplify workflows, and limiting the number of functions for improved accuracy. Additionally, it is recommended to offload tasks from the model by passing predefined variables instead of parameters [Ope23].

Function schemas can be refined using tools like OpenAI's Playground [Ope25b], and fine-tuning should be considered for more complex tasks (see 2.3.3).

2.3.2 Dummy Project: Textfield UI Control

This section presents a small-scale dummy implementation that demonstrates how function calling can be utilized to manipulate components of a simple user interface. The implementation mimics behavior that could be beneficial for the DemoRec project by showcasing the potential of function calling for controlling graphical user interface (GUI) components. It is important to note that this implementation is not integrated into a MontiGem project and does not involve web-based interfaces. Instead, it serves as a conceptual showcase using a simple JavaFX GUI, illustrating how frontend manipulation through function calling can be envisioned and applied in a practical context.

Implementation

The system architecture comprises a JavaFX-based user interface with text fields, an Azure OpenAI client for communication with OpenAI's GPT-4-mini model, and a main codespace that defines a function-calling framework. This framework leverages the GPT model to invoke functions capable of manipulating the JavaFX text fields based on natural language input.

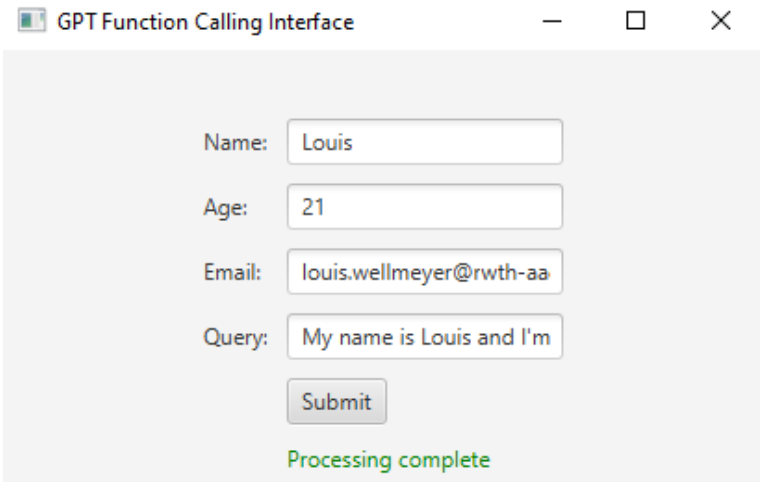

Figure 2.2: JavaFX GUI which textfileds to be controlled via function calling by manually entering prompt.

The GUI is shown in Figure 2.2. It comprises four text fields: three designed to be dynamically altered via function calls - `name`, `age`, `email` - and one for users to input queries that are subsequently sent to the LLM. Additionally, the GUI includes a label that displays the current status of the function call procedure, providing feedback to the user on the ongoing operations.

As described in Section 2.3.1, enabling the models to determine which functions to invoke requires the implementation of JSON schemas. Consequently, these schemas define the available functions, their input parameters, and expected outputs, ensuring that the LLMs responses align with the users requests and that responses have the necessary structure to correctly envoke the evaluated tools (see Listing A.1). As a result, this approach not only standardizes the communication between the application and LLM but also minimizes errors by enforcing strict data validation through schema definitions.

By adding the query as a `ChatRequestMessage` to the `chatMessages` array (Listing A.1) and defining the usable tools through JSON definitions, the LLM can process the query, evaluate the applicable tools, and return response data. The response includes the LLM's outputs as well as metadata such as timestamps, IDs, and token usage. For a comprehensive list of metadata contained in a `chatCompletions` response, refer to the official documentation ([Mic24]). As detailed by the documentation [Mic24], the LLM's responses are accessible through a list of `ChatChoice` objects, with each `ChatChoice` representing a single completion of the prompt. For instance, given the query "I'm 21 years old," a `chatCompletions` object might produce the following choice:
`{"textField":"age","value":"21"}`
Accordingly, the output can then be used to reflectively call the fitting function with the chosen parameters in the local environment.

2.3.3 GPT Function Calling Analysis

In this section, the findings from the previous sections are analyzed, focusing on their implications for the performance and optimization of function calling in voice-controlled web applications.

Performance

Referencing the Java FX example presented in 2.3.2, Listing A.1 illustrates the schema definitions for all input fields of the project's GUI. While defining a schema for each text field is a valid approach, it is advised to minimize the number of tools [Ope23]. Managing numerous text fields becomes impractical due to the increased complexity and risk of errors. Additionally, the volume of information passed to the LLM increases, leading to higher token usage and increased computational costs. To mitigate latency and reduce costs, a parameterized function that handles multiple text fields is introduced instead (see Listing A.2).

Empirical measurements highlight the impact of different function structures on token consumption: Just the system prompt without function calls required 126 tokens, whereas defining separate functions for each text field resulted in a substantial increase to 297 tokens. By contrast, the parameterized function that handles multiple text fields reduced the token count to 192 tokens, achieving a 35% reduction compared to individual text field functions.

This reduction in token usage is significant, as it directly correlates with lower latency and processing costs. The individual function approach introduces unnecessary redundancy, inflating token consumption and increasing the complexity of function resolution for the LLM. In contrast, the parameterized function provides a structured yet efficient alternative, reducing overhead while maintaining functionality. Thus, consolidating multiple individual functions into a single, generic function where possible is a good practices for optimizing performance in LLM-based applications.

Model fine-tuning

When implementing function calling with multiple tools, fine-tuning the underlying model can significantly enhance accuracy and reduce token consumption. Unlike "few-shot learn-

ing", which means relying on including task examples within prompts, fine-tuning enables models to learn from larger datasets. Once a model has been fine-tuned, it can process given examples during prompt processing without requiring explicit demonstration in subsequent calls, thereby reducing latency and token costs while improving accuracy [Ope23].

Furthermore, fine-tuning can be utilized to define style and format of results, cover edge-cases more effectively, and perform tasks that are difficult to articulate in prompts. For detailed information about the training process and the usage of fine-tuned models, refer to the fine tuning guide of the OpenAI documentations [Ope23].

The fine-tuning process typically involves three main steps: preparing and uploading training data, training the model, and evaluating training results. If necessary, this process can be iteratively refined. Proving the claim of benefits in fine-tuned models, [ELJ+24] highlights, while models like GPT-4 and Gemini-1.5 offer powerful function calling capabilities, their large size and high computational requirements can pose challenges:

- Privacy concerns due to data upload requirements
- Dependency on stable internet connectivity
- Increased latency from data transfer

As stated by [ELJ+24], to address these limitations, researchers have begun developing specialized, domain-specific smaller models. However, enabling effective function calling in these models requires careful consideration of several factors. The model must accurately:

- Determine which functions to call
- Identify appropriate input arguments
- Establish the correct order of function calls, considering potential interdependencies

Recent advances in synthetic data generation have helped address the challenge of limited training data for specific domains. Researchers have successfully utilized larger language models to generate high-quality training datasets [ELJ+24]. This approach has been particularly effective when combined with careful filtering datasets to maintain the same or even better performance while reducing data size.

The Low-Rank Adaptation (LoRA) [HSW+21] was introduced as an efficient paradigm for fine-tuning. This method proved itself effective, particularly when incorporating both function descriptions and negative samples in a training process. This approach has yielded promising results, with some implementations achieving success rates up to 4% higher than GPT-4-Turbo [ELJ+24]. To further optimize performance, particularly in terms of latency, research has focused on prompt efficiency. Rather than including all tool descriptions in the prompt, a fine-tuned DeBERTa-v3-small model classifies which functions are required for a specific query. This approach, termed Retrieval-Augmented Generation (RAG) [HGC23], achieves nearly perfect recall (0.998) while reducing prompt size by approximately 50% in token count, significantly improving inference efficiency [ELJ+24]. However, it's important to note that while smaller fine-tuned models have shown promising results, they still carry risks of hallucination and erroneous responses, necessitating appropriate human oversight [ELJ+24].

2.4 MontiGem

While the goal of this research is to develop a tool that is platform independent and compatible with any website, the primary motivation of this thesis is to facilitate the DemoRec disassembly process by integrating speech control into the projects MontiGem3 generated web interface for data management. To achieve this, it is essential to ensure seamless compatibility between the developed tool and MontiGem based web applications.

This section provides an overview of MontiGem, detailing its architecture to understand and identify constraints and suitable integration points for speech control. By analyzing its structural components and interaction mechanisms, the goal is to design a tool that effectively integrates with MontiGem's framework, enabling intuitive and efficient speech-controlled interactions within the generated web interfaces. The frontend of MontiGem-based applications is built with Angular, while the backend is powered by Spring Boot.

About

MontiGem is a model-driven code generation framework designed for enterprise information systems. It follows the principles of Model-Based Software Engineering (MBSE) and is built on MontiCore [Rum21], a language workbench and code generation framework. By leveraging domain specific models, MontiGem automates the development of data centric business applications, reducing manual effort and improving consistency between frontend and backend components [AMN+20].

MontiGem's architecture [AMN+20, Rum21] is to be split into three broad parts of reading, translation, and generation. Domain-specific models are first parsed and converted into an abstract syntax tree (AST) representation using MontiCore's parsing infrastructure. The AST is then transformed and processed using template engines that apply standardized and project-specific templates. Finally, the transformed AST is used to generate essential components such as data structures, database communication layers, business logic, access control mechanisms, and GUIs.

A crucial part to consider for the development of this thesis speech control tool is MontiGem's command-based communication infrastructure, which manages the interaction between the frontend and backend. This system generates commands for each domain and aggregate class, allowing structured data retrieval and modification. By ensuring that frontend to backend interactions are passed through commands, MontiGem maintains data consistency and integrity across the system [AMN+20].

TOP Mechanisms in MontiGem

A key feature of MontiGem's architecture is the TOP-mechanism, which facilitates the integration of handwritten code with automatically generated classes. The TOP mechanism ensures that developers can extend or modify the behavior of generated classes, such as AST nodes, without altering the generated code itself. Instead of manually changing the generated code, which makes maintenance more difficult, the developer can create a handwritten class that extends the generated TOP class, preserving the integrity of the generated code while adding custom functionality. This seamless integration is achieved by MontiGem's code generation tool MontiCore, which automatically generates an abstract

superclass for each class that is replaced by handwritten code. This mechanism is particularly beneficial because it avoids the bad practice of modifying the generated code directly. It allows for cleaner integration of custom code with the generated system, ensuring that any custom modifications do not interfere with future regeneration processes. Moreover, the TOP mechanism can be applied to all generated classes, including AST classes, node builders, and other components, providing a flexible and robust framework for extending the generated application [Rum21].

Additionally, the use of design patterns such as the Multiple Interface Composition Pattern, implemented within MontiCore, further enhances the ability to compose handwritten modifications into the generated system. This allows for overriding generated functionality, ensuring that custom code integrates smoothly with the overall system, maintaining flexibility and extensibility [Rum21].

Communication System in MontiGem

The communication system in MontiGem is based on a command pattern, which facilitates the synchronization between the frontend and backend, as well as between the server and clients. For each action, such as adding or altering a domain object, MontiGem generates a command (Java/TypeScript class) along with corresponding de-serializers (JSON) [Mon25]. When executing an action, these commands are exchanged between the server and clients via a SpringBoot WebSocket endpoint. For information on the usage of MontiGem's command system, refer to Section 3.1.2.

Chapter 3

Tool Implementation

This section outlines the detailed implementation of the voice-controlled tool, describing the architecture, key design decisions, and technical components that enable seamless integration within web interfaces. It covers the speech recognition pipeline, command execution mechanisms, system architecture, and how the solution was tailored to meet the functional and non-functional requirements defined for the DemoRec project in Chapter 1.

3.1 Architecture

The proposed tool is designed to enable voice-controlled data entry in web applications by integrating multiple key components that work together seamlessly. These components form the backbone of the system, facilitating efficient speech recognition, command processing, and interaction with web-based user interfaces. The tool architecture consists of various components, including the SpeechFunctionCaller, DataProcessor, InstanceManager, FunctionResolver, and Transcriber, each playing an independent central role in delivering a cohesive solution. This seperation of concerns will help maintaining the architecture and exchanging components if desired.

Figure 3.1 shows the tool's system at a high level. The proposed tool enables users to interact with web applications through spoken commands, with the underlying tool infrastructure handling real-time audio processing, transcription, and execution of actions. The integration of an LLM for function calling and a transcription service ensures that the system is capable of interpreting complex speech inputs into executable functions.

The overall design is aimed at providing a flexible, extensible, and maintainable solution, which can be easily adapted to different web applications, especially in industrial settings such as the DemoRec project. Diagram 3.2 provides a visual representation of the tool's infrastructure, illustrating the various components it consists of.

3.1.1 API Selection

Using external services for transcription and function calling is an optimal choice because they provide powerful, scalable, and well-documented APIs that integrate seamlessly into the system's architecture. Leveraging external services like these allows the tool to focus

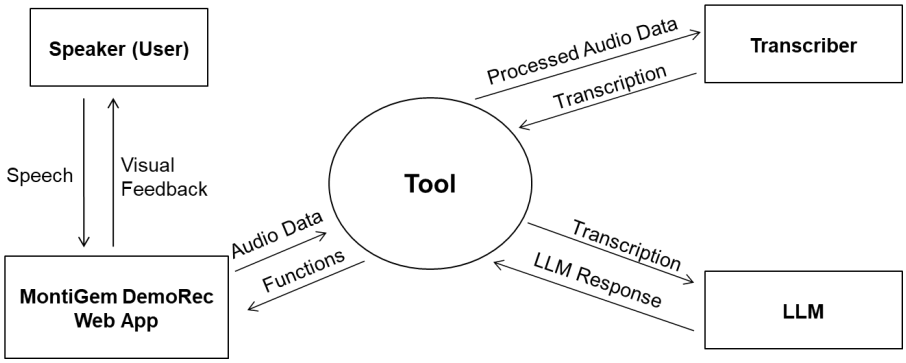

Figure 3.1: Context diagram providing a high-level overview of the thesis' tool's architecture, illustrating how users interact with web applications through spoken commands, which are processed by the tool's core components utilizing external services.

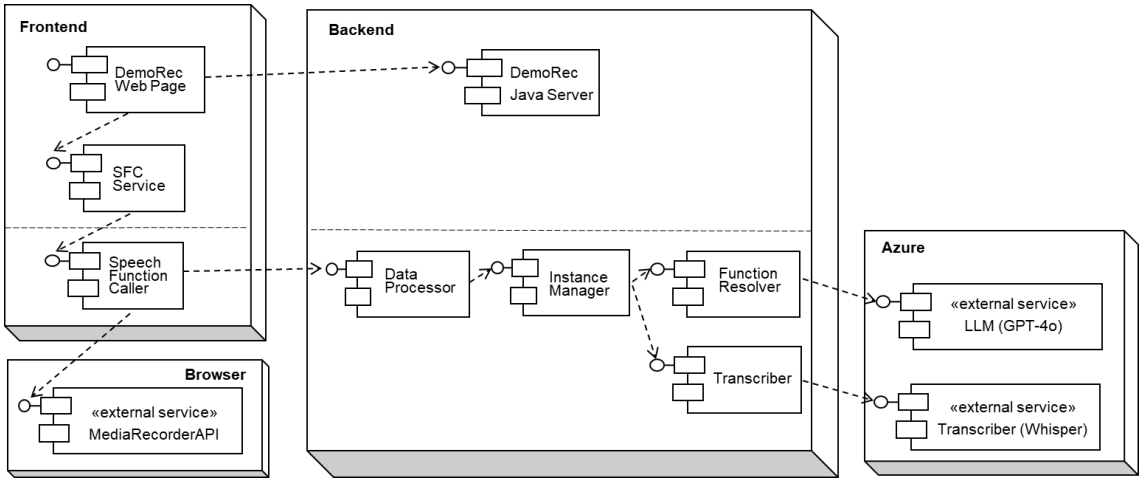

Figure 3.2: Component diagram detailing the internal architecture of the tool, highlighting the various components and their interactions.

on core functionalities and avoids reinvention in areas such as speech-to-text transcription and natural language processing. This choice not only accelerates development but also ensures that the tool can evolve with advancements in AI and machine learning without requiring major internal rework.

The implementation of this thesis' tool leverages the Azure OpenAI Service's Chat Completions API for function calling, utilizing Azure's official Java software development kit (SDK) [Mic24]. This approach was chosen based on several key factors:

- The Azure Java SDK aligns seamlessly with MontiGem's existing Java-based architecture, enabling efficient and consistent integration.
- Java SDK support, including native type safety, error handling, and a well-documented API, simplifying development and ensuring reliability.
- Azure OpenAI client library idiomatic interface and rich integration with the Azure SDK ecosystem facilitating further scalability

Like mentioned in Section 2.2, as for the transcription service, Azure OpenAI Whisper is

utilized. Whisper has demonstrated exceptional robustness across various acoustic conditions, including noisy environments. It maintains consistent performance across different speaker characteristics and environmental variables, making it highly suitable for industrial applications, such as battery disassembly in the DemoRec project.

3.1.2 SpeechFunctionCaller

The SpeechFunctionCaller component forms the backbone of the tool, enabling configuration of the tool's behavior, starting audio capture, processing audio, invoking transcription, and calling functions via the LLM. The core architecture of this component is shown in the class diagram for the SpeechFunctionCaller in Figure 3.3. This diagram provides a detailed overview of the class structure, illustrating how the component is organized and how different elements, like the `AudioManager` and `StatusManager`, interact with the `SpeechFunctionCaller` class.

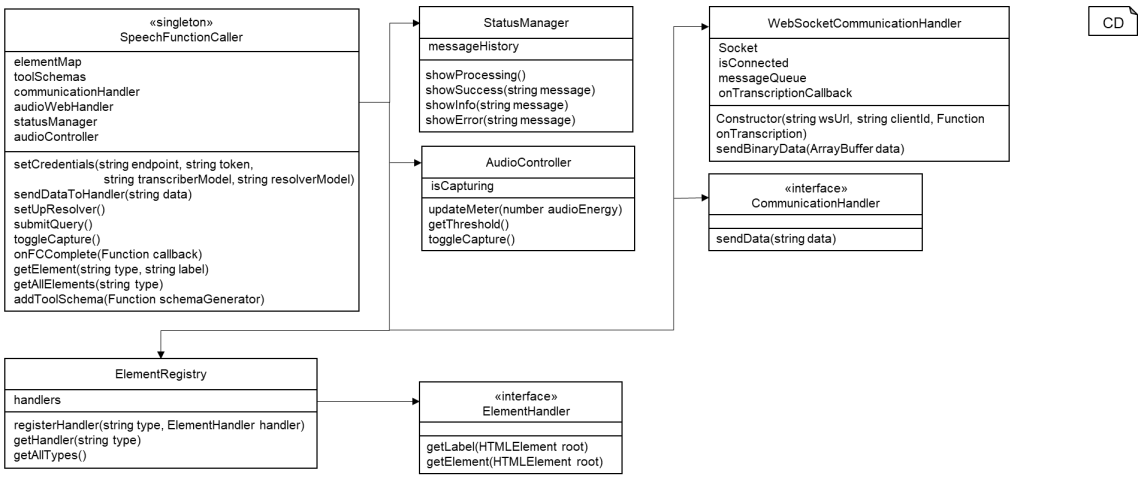

Figure 3.3: Class Diagram for the SpeechFunctionCaller Component, highlighting its key components and interactions.

In addition to its core functionality, the SpeechFunctionCaller creates HTML elements for displaying errors, tool information, and LLM responses, ensuring transparency and clarity regarding the tool's actions. Figure 3.4 provides an example of its UI, which includes a `StatusManager` for tool information and an `AudioController` to manage audio capture.

The client does not handle transcription and function execution directly but instead communicates with backend components, which will be introduced in the following subsections. This backend architecture is essential for several reasons. First, transcription and LLM function calling require significant computational resources that would degrade frontend performance if handled client-side. Second, separating these resource-intensive operations to a dedicated backend enables better scalability as demands increase. Finally, backend processing enables more sophisticated caching strategies and pre-processing of audio data before transcription, improving overall system efficiency.

There are several common methods for frontend-backend communication, including WebSockets, REST APIs, and GraphQL. To keep the tool modular and consistent, an abstract communication protocol is facilitated through the `CommunicationHandler` interface,

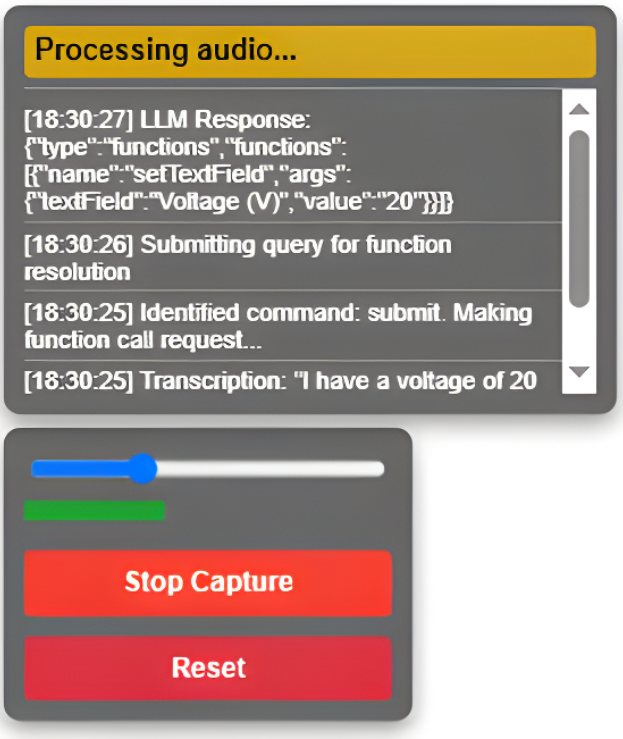

Figure 3.4: The UI provided by the SpeechFunctionCaller component, featuring a `StatusManager` to display tool information and an `AudioManager` to manage audio capture.

which defines a structured approach for sending data as JSON strings. In web applications with a frontend and backend, communication is typically initiated by the frontend, while backend-initiated communication is not always necessary. Therefore, to increase portability, it was decided to generalize communication in this manner, ensuring all interactions are initiated from the frontend and responses are received from the backend as return values.

To integrate this approach within the given environment, the communication handler in this project invokes a newly defined MontiGem command with a string payload, depicted in Listing 3.1, which simply calls `DataProcessor.process(data);`. This method ensures that data is consistently processed within the backend while maintaining compatibility with MontiGem's architecture. It furthermore decreases configuration as only a single new montigem command is required to invoke various functions.

The main part of the component, the `SpeechFunctionCaller` class, is implemented as a singleton, which is a deliberate architectural decision with several advantages in this context. As the central coordinator between various components with different responsibilities (audio capturing, UI management, backend communication), a singleton ensures there is exactly one instance managing these interactions throughout the application lifecycle. This pattern eliminates the need for complex synchronization mechanisms that would be required if multiple instances were competing to access shared resources such as the audio stream or communication channels.

The singleton approach also aligns perfectly with the separation of concerns principle

```java
public class Caller extends CallerTOP {

    private final String data;

    public Caller(String data){
        super(data);
        this.data = data;
    }

    @Override
    public SimpleResult<String> doAction() {
        try {
            Object result = DataProcessor.process(data);
            return new SimpleResult<>(getId(), result != null ?
                result.toString() : "null", String.class.getName());
        } catch (Exception e) {
            return new SimpleResult<>(getId(), e.toString(), String.
                class.getName());
        }
    }
}
```

Listing 3.1: MontiGem command introduced to invoke `DataProcessor.java` backend functions from the frontend.

employed in the system architecture. Various classes (AudioManager, UIController, and user defined ones) can access the singleton instance to request services without needing to maintain their own communication channels or duplicate functionality. This promotes better maintainability as components can be modified or replaced without affecting the central coordination logic. Additionally, the singleton provides a centralized point for configuration changes and state management, making the system easier to debug and extend. Should future requirements necessitate replacing any component, the singleton pattern minimizes the impact on other parts of the system, as only the relevant interfaces to the replaced component would need updating.

Audio Communication between Frontend and Backend

Within the `SpeechFunctionCaller` class, the communication abstraction of generic JSON string data is leveraged to invoke functions and retrieve data from the backend components. The base communication flow is depicted in Figure 3.5. Once audio data has accumulated in the frontend, it is sent to be transcribed on the backend via the Whisper API. Since Whisper transcription is computationally intensive and requires time to process, it must be called asynchronously. While transcription is ongoing, speech must continue to be gathered and processed, necessitating an efficient mechanism to update the frontend with the latest transcription results upon completion.

Since the system is designed for one-way frontend-to-backend communication, periodic polling is the most straightforward method for retrieving transcription updates. However, MontiGem also supports bidirectional streaming via Spring Boot WebSockets. To optimize performance for the DemoRec project, both approaches were evaluated, leading to the adoption of a hybrid solution: the system defaults to polling but can seamlessly switch to real-time WebSocket updates when supported. The WebSocket audio transmission which

can be used instead of the polling mechanism is shown in Figure 3.6.

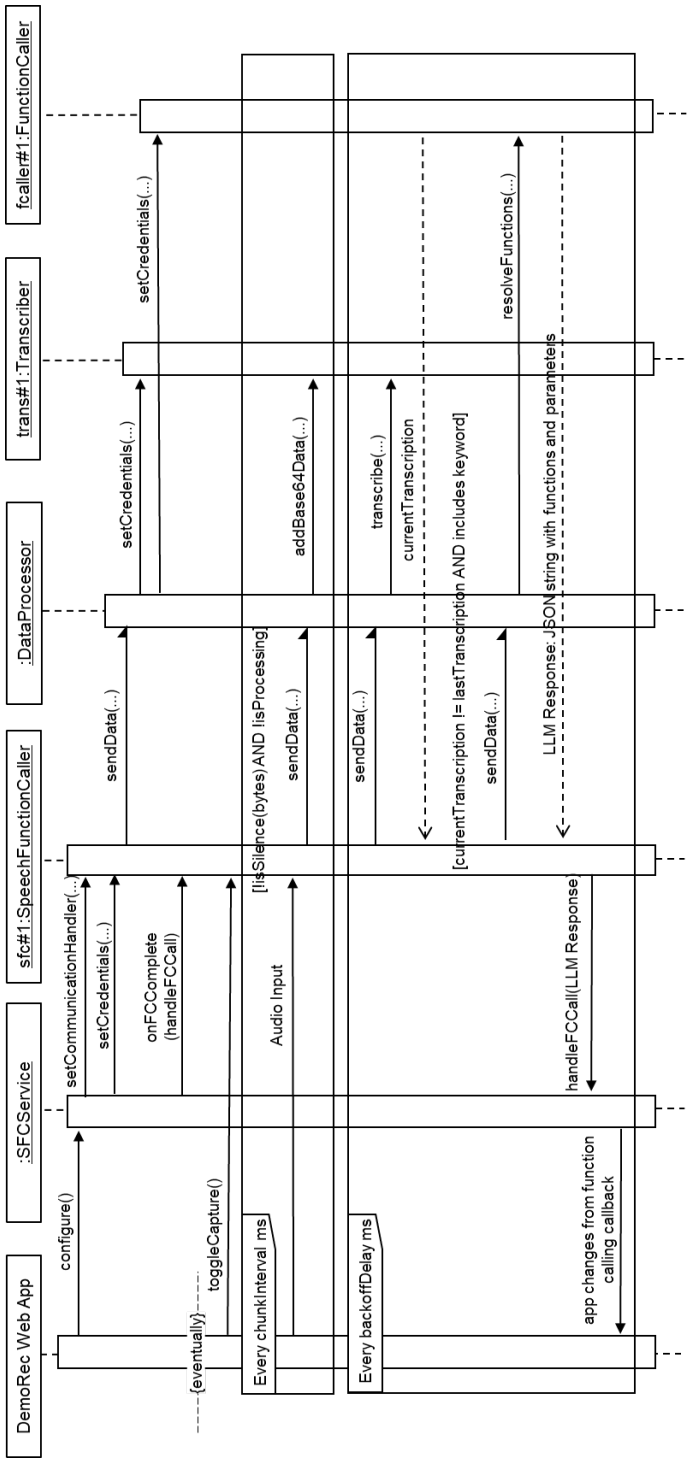

Figure 3.5: Communication flow between frontend and backend, from configuration to audio capture and function resolution. Transcription updates are fetched with polling in intervals.

The advantages of this hybrid approach include:

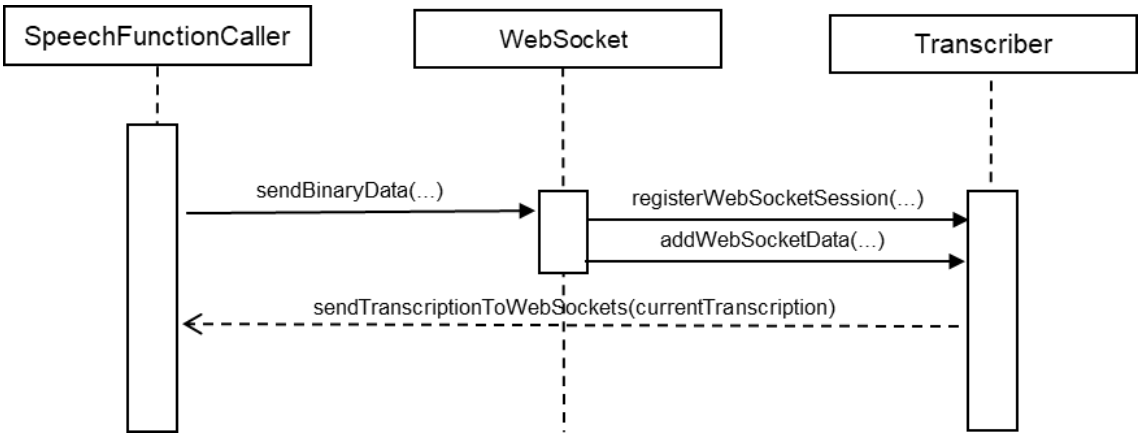

Figure 3.6: Optimized communication workflow using WebSockets for real-time updates.

- **Flexibility:** By supporting both WebSockets and polling, the system can adapt to different environments, ensuring broader compatibility while optimizing for real-time performance when possible.
- **Optimized Communication:** WebSockets enable instant, bidirectional communication, reducing latency and providing a seamless user experience. Polling, on the other hand, works as a fallback mechanism that ensures consistent transcription updates without the need for persistent connections.
- **Seamless User Experience:** Users with WebSocket-enabled environments will experience near-instant transcription updates, while users without WebSocket support will still receive transcription updates at regular intervals through polling.

While the hybrid approach offers flexibility, it is still useful to compare both methods in terms of their advantages and drawbacks. Polling is a safe and reasonable solution due to the following reasons:

- **Compatibility with Existing Infrastructure:** In most web applications consisting of a frontend and backend, communication is typically initiated from the frontend, while backend-initiated communication is not always necessary. Polling aligns with this structure, ensuring seamless deployment across different web applications without requiring modifications to the core communication model.
- **Minimizing WebSocket Dependency:** While WebSockets enable real-time updates, they require persistent connections, which may not be practical or supported in all deployment scenarios. By defaulting to polling and using WebSockets only when available, the tool remains adaptable without enforcing a continuous connection requirement.
- **Simplicity in Deployment and Maintenance:** Polling makes deployment straightforward. It removes the need for additional WebSocket server configurations and allows developers to maintain their preferred frontend-to-backend communication approach. Additionally, polling simplifies maintenance by eliminating concerns such as managing WebSocket connections, handling state synchronization, or dealing with connection drops and reconnection logic.

- **Frequency:** Speech input is continuous, meaning new audio is frequently available for transcription. Furthermore, to ensure consistency and prevent communication loss, polling can effectively track transcription updates in intervals.

To optimize performance, this thesis' implementation employs an adaptive backoff mechanism that dynamically adjusts polling intervals based on transcription updates, shown in Algorithm 1. This function reduces unnecessary network requests in case transcription updates occur infrequently, while ensuring responsiveness when speech is continuous.

Algorithm 1 Polling with Backoff Mechanism

```
1: function pollWithBackoff()
2: if polling is disabled then
3:     exit
4: end if
5: Send request to retrieve transcription status
6: Parse response to get transcription status
7: if transcription has changed then
8:     Update last transcription value
9:     Reset backoff delay to minimum delay
10: else
11:     Increase backoff delay by 5%, but cap at maximum delay
12: end if
13: if error occurred then
14:     Set backoff delay to maximum delay
15: end if
16: if polling is still enabled then
17:     Schedule next poll with updated backoff delay
18: end if
```

Despite an improved polling system, it is important to acknowledge the advantages that WebSockets provides:

- **Realtime Bidirectional Streaming:** Unlike polling, new WebSockets enable instant updates and reduce latency, providing a more seamless user experience.
- **Limitations in MontiGem:** MontiGem's command system does not support sending audio bytes directly. Therefore, without WebSockets additional conversions on both frontend and backend are required with polling leading to additional complexity, while a WebSocket communication for bytes could just be implemented.
- **Audio Package Size Constraints:** In the current implementation, audio must be split into smaller segments before transmission. A WebSocket based solution could potentially reduce this overhead by supporting direct streaming.

AudioWebSocket

To implement a WebSocket solution for audio streaming, a dedicated WebSocket endpoint (`/audio-transcription`) can be introduced to handle direct audio data transmission.

By leveraging MontiGem's extensibility with Spring Boot, a `WebSocketConfigurer` is defined in a `AudioWebSocketConfig` class to register an `AudioWebSocketHandler`, which processes binary audio data in real-time. This handler receives and forwards audio data to a Transcriber instance while managing WebSocket sessions. On the frontend, the `WebSocketCommunicationHandler` class in SpeechFunctionCaller automatically manages connection handling, reconnection logic, and message processing. Users only need to set the WebSocket endpoint via

```
1  SpeechFunctionCaller.getInstance()
2    .setAudioWebHandler(SERVER_CONTEXT_PATH + "/audio-
       transcription")
```

as all other functionalities, such as buffering, error handling, and transcription updates, are managed internally. The environment must support WebSocket connections and provide a running backend service for the tool to use the WebSocket system properly.

The decision to use SpringBoot's WebSocket implementation offers several technical advantages. It seamlessly integrates with the existing MontiGem Spring ecosystem, while the framework's automatic configurations significantly reduces boilerplate code. Additionally, support for session management and buffer size configuration simplifies development. SpringBoot also facilitates straightforward integration with security mechanisms, enabling authentication and authorization of WebSocket connections if required [Spr25].

However, this implementation choice introduces a limitation: The WebSocket functionality becomes tightly coupled to the SpringBoot framework, creating a dependency that affects portability. Despite this, the benefits outweigh the constraints in this project's context. The productivity gains from reduced development time, consistency with the existing MontiGem SpringBoot codebase, and access to additional features justify this framework dependency. Consequently, to enable WebSocket communication, the environment must implement SpringBoot and support WebSocket connections. In the future, additional methods for transmitting audio may be introduced, or a more abstract communication mechanism could be implemented, allowing users to define their preferred approach if deviating from the provided methods.

In conclusion, while WebSockets provide real-time updates and reduced latency, the polling-based fallback ensures compatibility across a wider range of environments. The hybrid approach adopted in this system leverages the advantages of both, depending on the environment's capabilities. Section 4 presents a comparative complexity and latency analysis of both solutions for the same audio.

Accessing HTML Elements

The tool developed in this thesis is designed to interact with and control web applications. To achieve this, it must access various components that make up the application and invoke their respective functionalities.

A key challenge is identifying and mapping these elements in a manner that allows the LLM to recognize and interact with them for function calling.

While standard HTML elements such as labels, buttons, and input fields exist, they are rarely used in isolation. In many cases, a functional UI component consists of multiple

nested elements. Additionally, the number of possible operations across different elements is vast, making it impractical to equip the LLM with exhaustive knowledge of all potential interactions.

Instead of attempting to process all generic HTML elements and their operations, a targeted approach is necessary. Several strategies can be considered for achieving this:

Tools like Playwright [Mic25] enable web scraping and mimic user actions. However, this approach has several drawbacks. First, it introduces high overhead and complexity for dynamically interacting with web applications. Second, it lacks a structured way to expose functions for the LLM, making it difficult to define reliable interactions. Finally, automation tools often operate at the browser level rather than the application level, making them hardly suitable to be integrated within the tool of this work.

Another approach involves reflectively accessing predefined functions for setting and retrieving values, as well as parsing elements from HTML or UMLP GUI-DSL files. This method, however, requires prior knowledge of function names and application structures. In MontiGem, UMLP files consist of nested components, each can be managed in separate files, making it difficult to traverse upwards in the component hierarchy or locate functions across different files. Additionally, heavy manual configuration would be required to accommodate varying application architectures between different applications. A further downside is that this approach uses user defined function directly, likely modifying variables. As a result, users might not have the direct opportunity to verify changes through visual feedback before they are applied, reducing transparency and control.

The approach used in this thesis involves directly accessing the Document Object Model (DOM) tree through query selection. While this method initially suffers from the aforementioned challenge of selecting all HTML elements, this issue can be mitigated by defining an interface inside SpeechFunctionCaller that allows users to explicitly query only the elements they wish to interact with (see Figure 3.2) by overwriting this interfaces getter-function for the HTML Element that is supposed to be operated on. Additionally, overwriting a label function enables elements to be paired with unique identifiers. This mapping is particularly useful in the context of MontiGem, where elements and their associated functions are generated dynamically and may have arbitrary names. By using this approach, the LLM can reliably identify and select the appropriate elements for function calls.

To specify which operations should be callable by the LLM, users can define function call schemas directly in the code. A decorator system is employed, where functions intended for LLM usage are annotated using the JSON schema definitions, introduced in Section 2.3.1. However, JSON schemas are not directly defined as JSON datatypes or string, instead generator functions are used to ensure function calls remain dynamic in response to site changes while also seamlessly allowing the retrieval of site information, such as available dropdown options. This concept is known as lazy evaluation, where the schemas are only then created when they are needed. Figure A.3 illustrates an example of the decorator usage. It shows a function for setting the value of a text field, annotated to specify it as an LLM accessible tool.

Audio Capture using the MediaRecorder API

Before being able to utilize Whisper to transcribe speech, audio has to first be captured. For this purpose, the MediaRecorder API [Moz25b] is used. The choice of this API for

```
class GemTextInputHandler implements ElementHandler {
  getLabel(root: HTMLElement): string {
    const label = root.querySelector('label');
    return label ? label.textContent?.trim() : "";
  }

  getElement(root: HTMLElement): HTMLElement | null {
    return root.querySelector('input') || null;
  }
}

class GemButtonHandler implements ElementHandler {
  getLabel(root: HTMLElement): string {
    const label = root.querySelector('button');
    return label ? label.textContent?.trim() : "";
  }

  getElement(root: HTMLElement): HTMLElement | null {
    return root.querySelector('button') || null;
  }
}

ElementRegistry.registerHandler('gem-text-input', new
   GemTextInputHandler());
ElementRegistry.registerHandler('gem-button', new
   GemButtonHandler());
```

Listing 3.2: TypeScript interface for getting the actual HTML element to operate on and specifying an identifier directly from the DOM-Tree for MontiGem buttons and textfields.

capturing audio in this research was primarily driven by its broad browser support, ease of use, and efficient handling of streamed audio data. The MediaRecorder provides a straightforward mechanism for capturing and encoding audio directly within the browser, eliminating the need for third-party plugins or external dependencies. Furthermore, MediaRecorder enables real-time audio capture with minimal overhead, making it suitable for applications that require continuous processing [Moz25b].

A key consideration in this project was ensuring compatibility with Whisper processing, which required audio recordings in a 16kHz mono format [RKX+22]. MediaRecorder allows for direct configuration of audio settings, aligning with these constraints [Moz25b].

For recording and storing audio data, WebM is used initially due to its broad browser support and efficient encoding with the Opus codec. Opus, introduced in 2012, is a versatile lossy audio codec designed for both speech and music transmission, offering high quality at low bitrates while maintaining low latency and smaller file sizes [Smi20]. This makes it particularly suitable for real-time processing applications, minimizing storage requirements and latency while ensuring sufficient quality for tasks like accurate transcription.

Whisper supports various audio formats with differing advantages depending on the project's needs. Table 3.1 summarizes the comparison of all supported whisper audio formats [Ope25a]. While MP3 is one of the most commonly used formats, it is an older

lossy codec with lower efficiency at comparable bitrates. AAC, used in M4A and MP4 files, offers better compression efficiency and quality than MP3, but because it requires licensing and is not an open format, it is less commonly used. MPGA and MPEG formats, which utilize MP3 or MP2 encoding, are open-source but are generally not optimized for modern web applications.

On the lossless side, WAV files use PCM encoding, offering uncompressed high-quality audio at the cost of significantly larger file sizes. While WAV is less practical for storage and transmission over the web due to its large size, its uncompressed nature makes it ideal for processing, enabling easier manipulation of the raw audio data for tasks like voice activity detection, silence detection, and other critical operations.

WebM, utilizing Opus, offers significant advantages over older formats. It supports a wide bitrate range from 6 kbit/s to 510 kbit/s, offers both variable and constant bitrate encoding, and maintains a frequency range of up to 48 kHz [Smi20]. Unlike MP3, which has become outdated due to its lower efficiency and lack of ongoing development, Opus achieves superior quality at much lower bitrates, making it ideal for web applications.

However, WebM audio blobs require a leading header to indicate the format due to their compressed nature [Web25]. Since WebM is designed for continuous streaming, splitting the audio into chunks results in fragments that lack the necessary headers. As a result, if we attempt to save these chunks as temporary WebM files for transcription, they become invalid. In this tool, audio is sent to the backend in chunks, and to optimize processing time, only new, relevant chunks should be transcribed. However, because these chunks lack proper headers, this approach is infeasible. Handling them as independent WebM blobs with individual headers or extracting and discarding the variable-size header adds further complexity.

As a result, WebM audio is initially recorded due to its broader support across browsers in the MediaRecorder API [Moz25b] but is then converted to WAV format for processing and transcription. This conversion is necessary to simplify handling the audio data, making it easier to perform operations like voice activity detection, silence checks, and transcription. While the conversion to WAV introduces slightly higher latencies due to the increased data size sent to the backend, it significantly simplifies processing while maintaining high audio quality.

Format	Lossy/Lossless	Open Source	Codec	File Size	Quality at Low Bitrate	OS Support
WebM	Lossy	Yes	Opus/Vorbis	Small	Excellent	Good
MP3	Lossy	No (patent-free since 2017)	MPEG Audio Layer III	Small	Moderate	Excellent
M4A	Lossy	No	AAC/ALAC	Medium	Good	Excellent
MP4	Lossy	No	AAC	Medium	Good	Excellent
MPGA	Lossy	Yes	MP1/MP2	Medium	Moderate	Excellent
WAV	Lossless	Yes	PCM	Large	N/A	Medium
MPEG	Lossy	No	Various	Medium	Varies	N/A

Table 3.1: Audio Formats Supported by Whisper - Comparison Table [Smi20, BRBR18, Web25, Fou25, Moz25a, Gro25]

Audio Chunk Format and Processing

Audio data is recorded in chunks, captured via the `ondataavailable` event, and pushed into a queue for processing. When using the WebSocket approach for audio data transmission, the recorded data is sent directly as binary with a maximum message size of 1 MB. However, this limit is never reached in practice due to the typical size of each audio chunk.

Currently, audio data is captured and transmitted every 500 milliseconds, provided that the current accumulated data is not already processing. The size of each chunk depends on the sample rate, bit depth, number of channels, and encoding format. Adhering to the Whisper PCM audio format:

- **Sample rate:** 16,000 Hz
- **Bit depth:** 16-bit (2 bytes per sample)
- **Channels:** Mono (1 channel)

The total size of one chunk can be computed as:

$$16,000 \times 2 \times 1 \times 0.5 = 16,000 \text{ bytes} = 16 \text{ KB} \tag{3.1}$$

Since this is far below the 1 MB limit, even if the transmission interval were increased (e.g., sending data every 5 seconds), the resulting chunk size would still be manageable:

$$16,000 \times 2 \times 1 \times 5 = 160,000 \text{ bytes} = 160 \text{ KB} \tag{3.2}$$

Furthermore, it is ensured that no message exceeds the WebSocket's capacity, even in unexpected scenarios. This is achieved via client-side chunking. If an audio chunk exceeds a predefined maximum size (`MAX_CHUNK_SIZE = 512 * 1024; // 512 KB`), it is split into smaller segments before transmission:

```
1  const MAX_CHUNK_SIZE = 512 * 1024; // 512 KB
2
3  if (buffer.byteLength > MAX_CHUNK_SIZE) {
4    console.warn("Chunk too large, splitting...");
5    for (let i = 0; i < buffer.byteLength; i += MAX_CHUNK_SIZE
       ) {
6      const chunk = buffer.slice(i, i + MAX_CHUNK_SIZE);
7      this.audioWebHandler.sendBinaryData(chunk);
8    }
9  } else {
10   this.audioWebHandler.sendBinaryData(buffer);
11 }
```

Listing 3.3: Client-Side Chunking

If, however, an individual message still exceeds 1 MB due to more unforeseen conditions, the WebSocket rejects it, and an error is logged. This is enforced in the WebSocket configuration. Additionally, the `handleBinaryMessage` method ensures that the transcriber only receives valid data by discarding oversized messages:

```
1  @Override
2  protected void handleBinaryMessage(WebSocketSession session,
       BinaryMessage message) throws Exception {
3    ByteBuffer data = message.getPayload();
4
```

```
5     if (data.remaining() > BUFFER_SIZE_LIMIT) {
6       System.err.println("Received oversized message: " + data.
            remaining() + " bytes. Ignoring...");
7       return; // Do not process oversized messages
8     }
9
10    byte[] audioData = new byte[data.remaining()];
11    data.get(audioData);
12
13    transcriber.addWebsocketData(audioData);
14    transcriber.registerWebSocketSession(session);
15  }
```

Listing 3.4: Server-Side Validation

This approach ensures that all transmitted data remains transparent within safe limits, avoiding message loss due to WebSocket constraints while maintaining efficient streaming. Once transcription is successfully completed, the result is returned via a registered callback and sent back to SpeechFunctionCaller through the WebSocket for further processing. If WebSockets are unavailable, the system falls back to a Base64-encoded JSON approach. In this case, each audio chunk is encapsulated as:

```
1  interface AudioChunk {
2    sequence: string; // Unique order identifier
3    data: string; // Base64 encoded audio
4    end: boolean; // True if final chunk
5  }
```

The `sequence` field ensures correct ordering, combining a timestamp (`Date.now()`) and an index. The `data` field contains the Base64-encoded audio, and `end` signals the last chunk.

Since JSON does not support raw binary data, encoding the audio in Base64 ensures compatibility while allowing seamless transmission within structured messages. The encoded data is split into fixed-size chunks similar to the edge-case handling of the defined WebSocket and transmitted sequentially.

Silence Detection in Audio Streams

To determine whether an incoming audio chunk contains meaningful speech or is silent, a silence detection mechanism based on audio energy levels is used. This can vastly increase efficiency as silent parts are filtered out. As a result, the audio data does not need to be chunked unnecessarily, converted to Base64 and most importantly does not need to be sent to backend. Furthermore, the accumulated audio data in the backend is ensured to only consist of non-silent audio. Therefore, the data size only depends on the valid audio, decreasing workload and thus latency of Whisper transcription.

The function `isSilence` (depicted in Algorithm 2) processes an array of 8-bit PCM audio bytes and returns a boolean value indicating whether the signal falls below a energy threshold that can be directly set via a slider in the web application. It extracts 16-bit audio samples by combining two consecutive bytes. It then calculates the absolute sum of these samples and normalizes it by the number of samples to estimate the signal's energy. This computed energy is compared against the set energy threshold. If the energy

falls below this threshold, the function returns `true`, marking the segment as silence. Otherwise, it returns `false`.

Since speech-to-text processing is computationally expensive, silence detection helps reduce unnecessary processing by discarding silent segments before transcription. This optimization improves efficiency and reduces latency of network traffic and computation. The threshold for detecting silence may need to be adjusted dynamically depending on environmental factors such as microphone sensitivity and background noise.

Algorithm 2 Silence Detection Algorithm

Input: `bytes` (array of 8-bit PCM audio data)
Output: `true` if silence, `false` otherwise
function isSilence(bytes)
    $numBytesRead \leftarrow$ length of $bytes$
    $sum \leftarrow 0$
    for $i = 0$ **to** $numBytesRead - 1$ **step** 2 **do**
        $sample \leftarrow (bytes[i+1] \ll 8) \mid (bytes[i] \& 0xFF)$
        $sum \leftarrow sum + |sample|$
    end for
    $energy \leftarrow \frac{sum}{numBytesRead/2}$
    updateMeter(energy)
    if $energy <$ getThreshold **then**
        return `true` ▷ Silence detected
    else
        return `false` ▷ Audio contains speech or noise
    end if
end function

In this implementation, silence detection is performed using energy thresholds to determine speech boundaries. While this method is straightforward and computationally efficient, it has several limitations. Variability in background noise levels, differences in speaker volume, and overlapping speech can lead to inaccurate transcriptions, either prematurely cutting off speech or failing to detect pauses correctly.

A more robust approach would be to incorporate voice activity detection (VAD), which is designed to distinguish between speech and non-speech segments more accurately. As introduced by [CJR+22], modern VAD models leverage machine learning techniques to adapt to different acoustic environments, improving detection reliability. Deep learning approaches, such as Convolutional Neural Networks and Deep Neural Networks, have demonstrated higher accuracy in speech-related tasks compared to traditional methods, recognizing 96–98% voices correctly in some cases [CJR+22]. Integrating a VAD system before the Whisper model could enhance transcription accuracy by ensuring that only relevant speech segments are processed.

However, the use of a VAD model introduces additional computational overhead and latency, especially when employing machine learning approaches. As noted in recent research, deep learning models require significant computational resources due to their multi-layered structure, and their performance heavily depends on factors such as data preprocessing, model architecture, and parameter selection [CJR+22]. The trade-off between improved detection and increased processing time needs to be carefully evaluated.

Future work could explore whether the benefits gained in transcription accuracy and performance outweigh the initial increase in latency due to VAD computation.

3.1.3 DataProcessor

The `DataProcessor.java` component serves as a critical backend module that enables the speech control system. It functions as a reflection bridge, allowing dynamic method invocation based on JSON-formatted requests. This component is responsible for executing logic or data processing on the backend from the client.

The use of reflection in `DataProcessor.java` is a deliberate architectural decision that offers several significant advantages in this speech-controlled system. Reflection enables the system to resolve and invoke methods at runtime based solely on string identifiers, necessitating only a single defined communication channel (in the case of MontiGem only a single command) between frontend and backend for a multitude of various backend functions. Furthermore, this eliminates tight coupling between components, allowing either side to evolve independently without breaking compatibility as long as the string interface remains consistent.

The alternative to reflection would require implementing explicit function dispatch mechanisms, such as large switch statements or lookup tables that must be manually updated with each new function. Reflection eliminates this maintenance burden, as new backend functions become automatically available to the frontend without explicit registration. As the system grows, new voice commands and corresponding backend functions can be added seamlessly. The reflection mechanism will discover and invoke these new methods without requiring changes to the communication infrastructure or speech processing pipeline.

With all method invocations flowing through a single reflection handler, error handling can be implemented comprehensively in one location, providing consistent error reporting and recovery mechanisms regardless of which function is being invoked. Reflection allows the system to perform runtime type checking and conversion between JSON data and the actual parameter types required by backend methods, increasing robustness against type mismatches or data conversion issues. Moreover, since the frontend and backend are implemented in different programming languages (TypeScript for frontend and Java for backend), string-based communication through reflection provides a language-agnostic interface that avoids the complexity of cross-language object serialization.

The component implements comprehensive error checking and exception handling, ensuring robust and reliable operation by preventing invalid logic from being invoked or improperly executed.

The component is structured around a single function, `DataProcessor.process(data)`, which centralizes all logic handling. This function acts as the entry point for processing all incoming requests from the frontend. The implementation parses JSONs to convert the string request into structured data, employs Java's reflection API for dynamic method invocation based on string identifiers, follows a standardized JSON format for requests, implements comprehensive error handling for malformed requests, missing methods, and runtime exceptions, and provides detailed error feedback to the frontend for diagnostic purposes.

The JSON request format consists of three key elements:

- `class` - A string specifying the backend class to be used
- `function` - A string specifying the method name to be invoked on the specified class
- `parameters` - An array containing the attribute values required for the specified function

For example, invoking the `setCredentials` function in the `FunctionResolver.java` backend script is structured as follows:

```
await this.sendDataToHandler(JSON.stringify({
  "class": "FunctionResolver",
  "function": "setCredentials",
  "parameters": [endpoint, token, resolver_model]
}));
```

While reflection provides flexibility, it also requires careful implementation to prevent security vulnerabilities. The DataProcessor ensures that client-specific backend instances are appropriately managed through the use of the InstanceManager (Section 3.1.3). Together with this component strict validation of incoming requests is enforced, ensuring only approved classes and methods can be invoked. The InstanceManager handles this by managing client-specific backend instances and ensuring that only valid instances associated with a given client ID are invoked. Method access control is enforced, and input parameters undergo validation before being passed to reflected methods. The system operates within a controlled environment where the range of available backend classes is predefined and limited, with all backend logic associated with each client handled independently.

It is important to note that, while the system follows this structured approach internally, users do not need to manually invoke functions in this format. All functionality is abstracted and provided through a clear interface in SpeechFunctionCaller, ensuring ease of use and maintaining a clean separation between implementation details and user interaction.

The component's design provides a powerful mechanism for bridging the frontend speech capturing system with the backend processing logic in a maintainable and extensible manner. This approach enables the entire system to grow organically as new functionalities are added, without requiring modifications to the core communication infrastructure.

InstanceManager

The InstanceManager component plays an essential role in enhancing the management of backend instances. It allows for dynamic and flexible handling of client-specific backend logic, with each client being assigned a unique client ID. This client ID can either be generated or set by the user during tool setup (as described in Section 3.2). This ensures that each client interacts with its own isolated instance of backend logic, unless instances are explicitly shared.

Internally, the InstanceManager uses a ConcurrentHashMap to map client IDs to backend instances. This approach enables multiple clients and threads to safely and concurrently access and modify instances, which is essential in a multi-client environment.

The InstanceManager also facilitates seamless interaction between different programming languages. In this tool, client IDs allow TypeScript clients to be mapped with Java backend instances. This dynamic management of instances ensures that backend logic can be correctly identified and handled by the backend, even when communication between the frontend and backend occurs across different languages.

Moreover, the InstanceManager simplifies the complexity of instance management in a distributed environment. By providing a centralized, session-based instance management system, it allows for easy scalability and the independent management of client-specific states, all while avoiding the tight coupling of individual components.

3.1.4 Transcriber

The Transcriber component, as shown in Figure 3.7, is responsible for converting spoken language into text, which is then handled by the FunctionResolver. It manages the audio processing and transcription tasks using Azure's OpenAI Whisper API [Mic24, RKX+22].

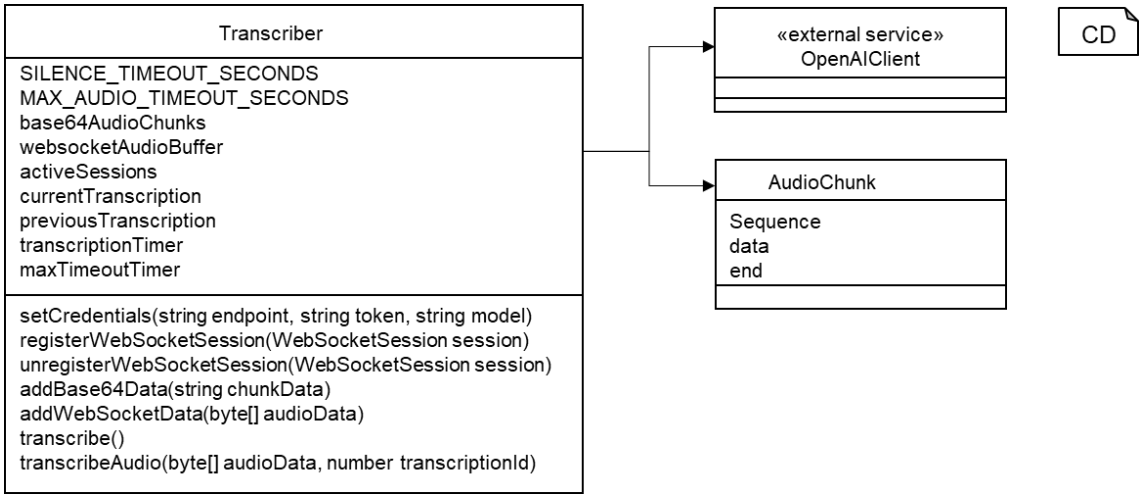

Figure 3.7: Class diagram of the Transcriber component, handling audio in the backend and transcribing to natural language using Whisper [RKX+22]

The most intuitive approach for real-time transcription would be to transcribe each new audio chunk immediately upon arrival. This would minimize latency and allow near-instantaneous feedback. Various techniques, such as multithreading and asynchronous processing, were explored to achieve this goal. However, both approaches quickly ran into Azure OpenAI's API quota limits, leading to frequent rate-limit errors, as depicted here:

```
[trans-6] Error during API call: Status code 429, "{"error":{"code
    ":"429","message": "Requests to the Audio_Transcriptions
    Operation under Azure OpenAI API version 2024-08-01-preview have
    exceeded call rate limit of your current OpenAI S0 pricing tier.
    Please retry after 57 seconds. Please go here: https://aka.ms/oai
    /quotaincrease if you would like to further increase the default
    rate limit."}}"
```

Azure enforces strict rate limits on API calls, as outlined in their documentation [Mic24]. To mitigate these limitations, Microsoft recommends implementing retry logic, avoiding

sharp workload increases, testing different load patterns, and requesting higher quotas. Despite implementing these strategies, rate-limit errors persisted.

To balance responsiveness with efficiency, an optimized audio chunk collection system was implemented. Rather than transcribing each chunk immediately, the system continuously gathers audio chunks until a predefined period of silence is detected. This silence typically indicates the end of a sentence or prompt, triggering transcription only when a complete speech unit is available. To account for noisy environments, audio device issues, or improperly configured input sensitivity, a timeout mechanism is introduced. If no silent period is detected, the system forcefully transcribes the collected data after 15 seconds, ensuring reliability even in unpredictable conditions.

This strategy offers several advantages:

- **Reduced API Calls:** By grouping chunks into larger segments, the number of API calls is significantly reduced, helping to stay within Azure's quota limits.
- **Improved Coherence:** Transcribing entire speech units ensures that context is complete. Transcribing partial or fragments of speech is often unnecessary as it rarely includes the full context for function calling.
- **Retry Mechanism:** In cases where the quota limit is still exceeded, a retry mechanism is employed, following Azure's best practices to avoid API disruptions.

This bulk-processing approach ensures optimal performance while adhering to the API limitations imposed by Azure[Mic24]. The end result is a reliable transcription system that balances real-time processing needs for reliable function calling with API constraints.

A `SingleThreadExecutor` is employed to process transcription tasks sequentially. This approach guarantees that transcription requests are handled one at a time avoiding redundancy, and preventing unnecessarily overwhelming the API, especially if there are rate limits.

By using a `SingleThreadExecutor`, we moreover ensure that each transcription task is processed in the correct order, which is essential for maintaining the integrity of the transcription process. Concurrent transcription attempts could introduce race conditions where audio chunks are processed out of sequence, potentially causing inconsistencies in the transcription states. Furthermore, the `SingleThreadExecutor` avoids such race conditions by ensuring only one transcription process runs at any given time.

Additionally, the use of a single different thread prevents unnecessary blocking of the data collection of this script. While transcription is ongoing, the system can continue collecting new audio chunks, as the execution of transcription tasks does not interfere with the collection process. Thus, the system remains responsive, allowing for continuous data intake without delays caused by the transcription task execution.

Audio Data Handling

The Transcriber component supports both communication protocols introduced: Base64 encoding and raw binary WebSocket data. Incoming Base64-encoded audio chunks are received as JSON-formatted data containing sequence information and the encoded audio.

These chunks are stored in a `TreeMap`, ensuring natural ordering based on timestamps and sequence numbers. In contrast, binary audio data received via WebSockets is stored directly in a `ByteArrayOutputStream` buffer. Despite differences in data formats, the system employs a unified transcription process. After a detected silence period or a timeout, the collected audio data is transcribed regardless of its format. The only distinction lies in preprocessing: for Base64 data, the system iterates through the `TreeMap`, decoding each chunk and appending it to a combined buffer before transcription. In contrast, WebSocket data is directly appended to the buffer.

Sending Transcription Back to the Frontend

In the Base64-based approach, the frontend's SpeechFunctionCaller periodically fetches transcription results via polling. In contrast, WebSockets enable real-time bidirectional communication, allowing transcription results to be delivered immediately upon completion. The AudioWebSocket, responsible for transmitting binary audio data from the frontend to the backend, registers WebSocket sessions within the Transcriber. Consequently, once a transcription is completed, results are instantly pushed to all registered WebSocket clients, significantly improving responsiveness compared to the polling-based method.

The need for this real-time transcription feedback from the backend to the frontend is critical for maintaining an efficient, responsive, and user-friendly voice control system. The frontend plays a pivotal role in initiating function calls based on user inputs, and it is essential that it receives timely and accurate transcription results to trigger appropriate actions.

Transparency in Function Calling is one of the key factors. The frontend should have full visibility into the transcription results and subsequent function calls. By fetching transcription results through WebSockets, we ensure that the frontend receives real-time updates, which allows it to act on the data instantly. This transparency ensures that the user can see how their voice input is processed and provides immediate feedback, enhancing user experience.

Another critical aspect is ensuring the separation of concerns between backend components like the Transcriber and FunctionResolver. If these backend scripts were to communicate directly with each other, it would create unnecessary complexity and dependencies between components. Instead, by receiving real-time transcription results through WebSockets, the SpeechFunctionCaller in the frontend can act as a single controller that decides how to process the transcription data. This allows the frontend to remain in control of the flow, while the Transcriber and FunctionResolver focus solely on their specialized tasks. This approach ensures that each backend component is not tightly coupled with the others, maintaining clear boundaries and providing flexibility to handle various scenarios without requiring modifications to the backend. Moreover, by ensuring a clear separation of concerns, the underlying services can easily be modified or replaced without impacting other components. For instance, the Transcriber and FunctionResolver can be swapped out or updated independently, as their roles are not interdependent. This flexibility is crucial for adapting to new technologies, optimizing services, or integrating with different systems without causing disruptions in the systems workflow, making the system more maintainable and scalable.

3.1.5 FunctionResolver

The FunctionResolver component, as illustrated in Figure 3.8, handles natural language processing and maps inputs to callable functions within web applications using LLM function calling. It processes natural language inputs, maps them to specific functions, and maintains conversational context.

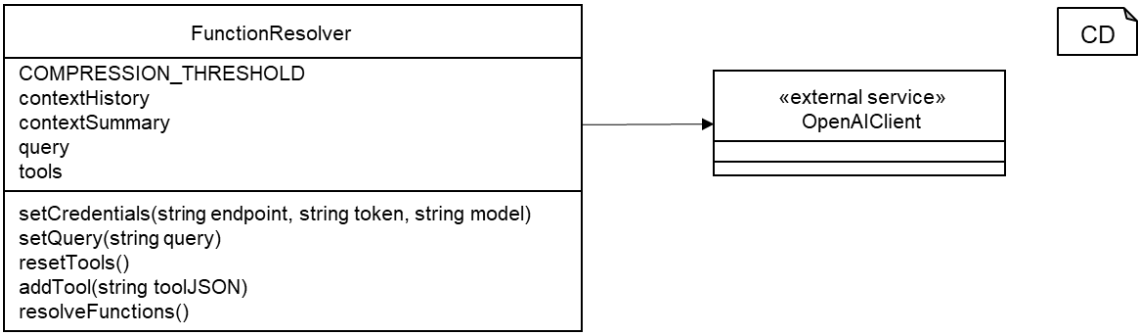

Figure 3.8: Class diagram for the FunctionResolver component.

The component is designed to handle domain-specific understanding, particularly in the context of information extraction. This specialization allows the system to extract relevant values from natural language and map them to form fields accurately. The implementation of contextual memory and conversation history compression is another key design choice, enabling natural, ongoing interactions while managing token usage and reducing latency. The component connects to Azure OpenAI's models to process natural language queries, maps natural language requests to specific functions using the specified tool definitions, and maintains conversation history with a compression mechanism to manage token usage. It processes both function calls and direct message responses, providing structured JSON output for function invocation. Invalid function calls, just direct responses are still important to consider for seamless contexts and potentially to display information to the user if desired.

Context Management and Compression

A key challenge in integrating natural language interfaces into structured applications is managing conversation history efficiently. This is crucial for maintaining a natural conversational flow while optimizing token usage to stay within Azure's API constraints.

The system maintains a conversation history to allow users to reference previous exchanges seamlessly. Without a structured memory mechanism, every user input would be treated in isolation, making complex interactions difficult and unnatural. To prevent excessive token usage while retaining necessary context, a compression mechanism is employed once the history reaches a predefined threshold.

The compression process is triggered when the number of stored interactions reaches `COMPRESSION_THRESHOLD` * 2 (accounting for both user queries and model responses). The system then condenses interactions older than the specified threshold as well as the current summary into a new one by prompting the LLM model. The prompt is as follows:

```
'Summarize the following conversation history in a concise
way, focussing on the key data information.  The summary should
```

`encompass ALL information about any changes and choices made for the key information in the past in order of time.'`

As a result, essential details are retained while redundant or irrelevant information are being removed. The summary is stored separately as `contextSummary` and added to future queries. The context is never cleared unless the user resets the application via the reset-button of the SpeechFunctionCaller component.

Function Resolution with OpenAI's API

The system enables function resolution the exact same way as introduced in Section 2.3.2. The resolution process is handled within the `resolveFunctions` method, which follows these steps:

1. Construct a prompt containing:
    - System instructions defining the AI's role and behavioral constraints.
    - The latest conversation history, including the `contextSummary`.
    - The current user query.
2. Call the OpenAI model via Azure's ChatCompletions API, passing the constructed message sequence along with registered function definitions.
3. If the model determines that a function call is needed, it responds with a structured JSON containing the function name and its arguments.
4. The system parses the function call response, extracts the relevant arguments, and returns them for execution.
5. The last response from the model is stored in the conversation history to maintain contextual continuity.

This approach ensures that function execution is dynamically driven by the model's interpretation of user queries while preserving efficient context management. By combining structured function resolution with intelligent context compression, the system maintains fluid, cross-prompting conversations without exceeding token limits.

3.2 Tool Usage

To integrate the tool into an existing project, users must place the provided files within the appropriate project structure and configure all necessary dependencies. This setup enables interaction between the frontend and backend, as well as the implementation of speech transcription functionalities. A detailed guide on how to correctly import the files and manage the dependencies of the tool is available in the GitHub repository: `https://github.com/Wellbek/SpeechFunctionCaller`

The configuration process is all handled via functions of the SpeechFunctionCaller component. It involves the following key steps:

1. **Define Communication:** First, implement the `CommunicationHandler`. This component is responsible for establishing and managing the communication protocol between the frontend and backend. This allows the tool to seamlessly use the way of communication of the environment without posing requirements for the user.

2. **Register HTML Elements:** The next step is to identify and register the HTML elements that will be interactable for the tool. This is accomplished by implementing the `ElementHandlers` interface. During this phase, pairs of queries for labels and corresponding queries for the relevant HTML elements need to be specified. This registration process allows the tool to recognize which elements can trigger functions or be controlled through speech input.

3. **ElementRegistry Setup:** Once the HTML elements have been registered, the next step is to add them to the `ElementRegistry`. This registry serves as a centralized store for all registered elements. By adding each element to the `ElementRegistry`, the system can easily access and manipulate these elements as needed.

4. **Schema Registration:** Callable functions must be decorated with the appropriate schema definitions. This is achieved by applying a decorator pattern to each function. As introduced in Section 2.3.1, the schema ensures that the tool understands the specific requirements for each function, such as when and how the function should be called, as well as the attributes that should be applied.

5. **Handle Function Call Result:** After setting up the callable functions and their corresponding schemas, it is necessary to handle the results of function calls. To do this, a callback function should be registered using `onFCComplete(callback: (transcription: string) => void)`. The callback will be triggered upon completion of a function call, and the result of the function call will be passed to the handler for further processing as a JSON string.

6. **Initialize Communication:**
   The next step is to initialize the communication between the frontend and backend by calling `setCommunicationHandler(handler: CommunicationHandler)`. This function binds the frontend to the backend, with the previously defined `CommunicationHandler` ensuring that all elements and functions are ready for interaction. If WebSockets and Spring Boot are supported, users can optionally specify the URL of the `WebSocketCommunicationHandler` to enable audio streaming.

7. **Initialize AZURE Client:**
   The last step before the tool can be used is to initialize the AZURE client. This is done by calling the method `setCredentials(endpoint: string, token: string, transcriberModel: string, resolverModel: string)` depicted in Listing A.4. This configuration step equips the backend with the models and API credentials.

8. **(OPTIONAL): Customize Command Keywords:**
   To customize the keywords that trigger function resolution, use the method `setCommandKeywords(["your", "custom", "keywords"])`. By default, the list contains the keyword "submit", which triggers the function resolution process. This optional step allows you to modify or extend the list of keywords to suit the specific needs of your application.

9. **Invoke Speech Functions:** Once the system is ready, speech transcription and function calling can be initiated. To begin, either press the 'Start Capture' button on the tools UI or call the function `toggleCapture()`. Afterward, functions will be triggered on recognized keywords, or by directly submitting the gathered natural language transcription via `submitQuery()` for function execution. These functions will fully manage the transcription process and establish communication with the backend using the preferred frontend-backend communication method.

3.2.1 Easy Deployment in MontiGem: MontiGemSFCUtilities

Working with the DOM requires specialized knowledge, and defining function calls can be inefficient, error-prone, or challenging, especially in complex web applications. The system mitigates these difficulties for MontiGem users by providing predefined utilities like schema generators, function calls and communication handlers as a importable library, while still allowing for extension or customization. This support is particularly valuable in cases where standard DOM operations are problematic, such as with MontiGem's dropdown components. In these components, available options cannot be fetched until the dropdown is actually opened in the interface, making static function definitions insufficient. This library ensures that users can implement common operations quickly using predefined behaviors, while retaining the ability to customize when facing unique requirements. MontiGem users can set up the entire system in very little time, without needing to write any function definitions on their own, significantly reducing implementation effort while maintaining the option for future customization.

As a result, configuring MontiGem web applications is both straightforward and efficient. MontiGem users can simply import the `MontiGemSFCUtilities` library and call the `configureSpeechFunctionCaller` function:

```
configureSpeechFunctionCaller({
  endpoint: this.ENDPOINT,
  token: this.TOKEN,
  transcriberModel: this.TRANSCRIBER_MODEL,
  resolverModel: this.RESOLVER_MODEL,
  clientId: "",
  audioWebHandler:
    "ws://localhost:8081/umlp/api/audio-transcription",
  context: this // Context where functions are executed
});
```

Next, function calls can be defined using the library's built-in utilities:

```
@FunctionCall(getTextFieldSchema())
public setTextField(textField: string, value: string): void
    {
  // Use the built-in implementation
  MontiGemSFCFunctions.setTextField(textField, value);

  // And/Or add custom functionality
}
```

`MontiGemSFCUtilities` assumes the existence of the Caller, command shown in Listing 3.1, to communicate with the backend.

With these minimal setup steps, the tool is fully integrated and ready to use within MontiGem web interfaces, significantly reducing the deployment effort shown in Section 3.2 while still allowing full customization.

3.2.2 Integration into the DemoRec Project

To ensure seamless integration of the SpeechFunctionCaller within the frontend architecture, a dedicated Angular service, `SFCService`, has been introduced for the DemoRec project.

The `SFCService` is implemented as an Angular `@Injectable` with `{ providedIn: 'root' }`, ensuring it is globally available throughout the application. This allows the service to persist across different Angular components and even across page navigations within the application. By utilizing this approach, the speech interaction tool remains functional even as the user navigates between different views, eliminating the need for redundant reconfiguration.

To integrate this service, `SFCService` is injected directly into the Angular `AppComponent`, which invokes the SFCService's `configure()` method during initialization. This configuration method performs the necessary aforementioned setup steps.

Chapter 4

Results & Discussion

This chapter presents a comprehensive evaluation of the audio-based interaction system implemented for the DemoRec project. The analysis begins with a comparative assessment of the two audio transmission methods, the Base64 polling approach and the WebSocket-based solution, examining the overall and their individual performance characteristics and efficiency. Following this technical comparison, the chapter delves into a systematic evaluation of the tool's effectiveness within the DemoRec environment, focusing on functional correctness, user experience, and operational robustness under various conditions.

4.1 Base64 Polling Vs. WebSocket Audio Transmission Methods

A test was conducted to compare the performance of the two audio transmission solutions introduced in Chapter 3: the default Base64 approach with polling and the WebSocket-based transmission method.

Both transmission methods were implemented in the same underlying system with identical hardware configurations. The first approach used a `CHUNK_SIZE` of 5000 characters per packet to adhere to MontiGem's existing command constraints, while the second approach used a `BUFFER_SIZE_LIMIT` of 1MB. In both methods, a 1-second interval was maintained between transmissions.

Five independent test runs were conducted for each transmission method. In each run, a 39-second audio file was played through a microphone via a virtual cable to simulate real speech. The audio was then processed in the frontend and sent to the backend for further processing and transcription via the Whisper API. Various timing metrics were logged throughout the process, including:

- Audio size (bytes)
- Temporary file creation and audio conversion time (ms)
- Whisper API call time (ms)
- Chunk splitting

- Transmission time (ms)
- Queue processing time (ms)

4.1.1 Results

The average results of the 5 runs for each method is depicted in Table 4.1.

Metric	Base64 with Chunking	Direct WebSocket
Average Audio Size	1,246,848 bytes	1,220,519 bytes
Temp File Creation	31.12 ms	28.8 ms
Whisper API Call	4,507 ms	3,993 ms
Chunking	5 to 10	1
Total Transmission Time	274.3 ms	43.0 ms
Queue Processing Time	321.9 ms	69.1 ms
Total Time	5,134.32 ms	4,133.9 ms

Table 4.1: Performance comparison between Base64 with Chunking and Direct WebSocket

The most significant performance difference is observed during the data transmission phase, where the direct WebSocket approach is 84.3% faster than the Base64 chunking method. This substantial improvement can be attributed to the larger payload size in the WebSocket approach, which does not require individual chunks to be sent separately. This reduces the number of network packets needed and, consequently, lowers transmission overhead. Furthermore, Base64 encoding adds additional overhead. According to [Jos06], Base64 encoding converts every 3 bytes of binary data into 4 bytes of text, resulting in a 33.3% size increase.

The queue processing time also shows a similar improvement. The WebSocket method benefits from reduced transmission time and does not need to handle the chunking logic and Base64 encoding, unlike the Base64 approach. Other metrics did not vary significantly, as both methods share the same underlying processing system.

It is important to note that in the case of the Base64 approach, backoff polling is used, adding additional network traffic and delays. The minimum delay `minDelay` and maximum delay `maxDelay` intervals for polling further extend the time until the transcription is received. In contrast, the WebSocket method transmits the transcription immediately to the client once it is available.

4.1.2 Conclusion

While the WebSocket solution is significantly more efficient than the Base64 solution, it requires maintaining a consistent connection. Additionally, WebSocket support must be available in the underlying system. As a result, a hybrid system was introduced, allowing users to take advantage of WebSockets when supported, while falling back to the Base64 solution if needed.

In practice, the 39-second speech file resulted on average in a difference of only about 1 second between the two methods, which does not significantly impact the user experience.

Therefore, both solutions are valid and can be used depending on the specific system requirements and environment.

Additionally, this test highlighted that even long user requests (around 40 seconds of audio) achieve results in just a few seconds, underscoring the responsiveness of the system, ensuring a good user experience.

4.2 Tool Effectiveness in DemoRec

This section presents a systematic evaluation of the tool's performance, focusing on functional correctness, scalability, robustness, and user-experience. However, transcription accuracy and function calling performance are not assessed here, as these aspects are inherently dependent on the underlying OpenAI services — Whisper for transcription and GPT-4o for function execution. Similarly, latency measurements are not explored in depth beyond the comparative analysis provided in Section 4.1.

The following evaluations are conducted within the DemoRec Site environment, meaning the results primarily reflect the tool's behavior in this specific implementation rather than its general applicability across different sites. While the core logic of the tool remains consistent across implementations, its integration within DemoRec introduces environment-specific factors, such as UI structure, available elements, and external influences like perturbations during recording, that all may impact performance and user-experience. All tests are conducted five times, and the average is taken to ensure reliable results. This section examines these factors in detail.

All subsequent tests utilize the WebSocket communication protocol to transmit audio to the backend. Unless specified otherwise, the tests are conducted on the Start-new-Process page within the DemoRec web interface, as it integrates various MontiGem components. Figure 4.1 provides an illustration of this web interface.

4.2.1 Task Completion Testing Methodology

This subsection presents a systematic evaluation of the tool's effect in completing realistic predefined workflows using different input modalities. The primary objective was to assess both the accuracy and efficiency of the tool across various interaction methods, to identify usage patterns and assess how well the tool can facilitate data entry for the DemoRec project.

Testing Scenarios

Four distinct interaction scenarios were tested to comprehensively evaluate the tool's capabilities:

1. **Manual Input:** The user manually entered all required information through traditional keyboard and mouse interactions. This scenario served as the control condition against which the voice-based modalities were compared.
2. **Sequential Voice Input:** The user provided voice commands in a step-by-step manner, dictating one field at a time.

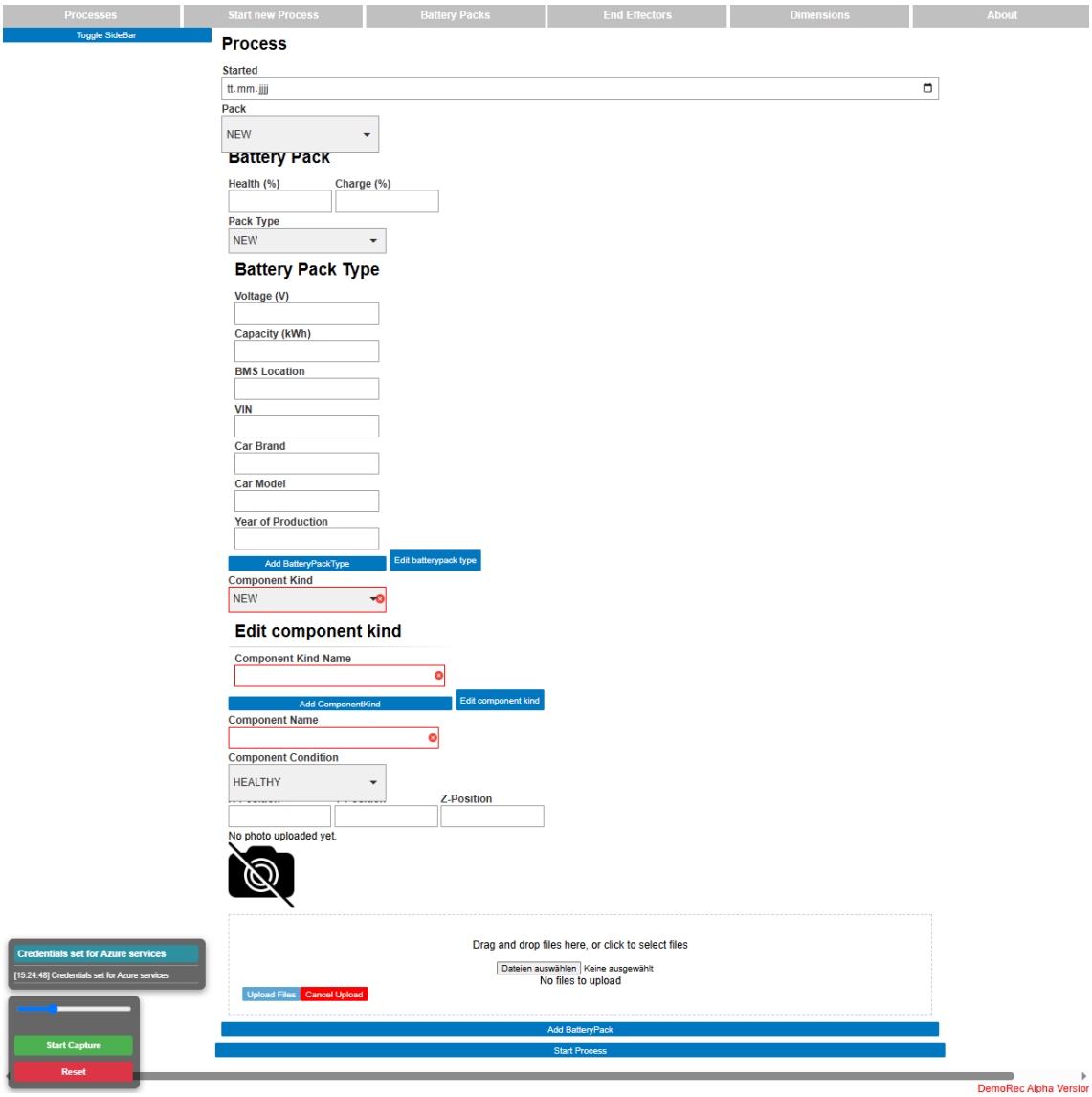

Figure 4.1: DemoRec Start-new-Process Page

3. **Batch Voice Input:** The user dictated all required information in a single voice command, requiring the system to automatically parse and distribute inputs to appropriate fields. This method tested the system's capability to process natural language and understand contextual relationships between data points.

4. **Specific HTML Voice Input:** The user provided voice commands with explicit references to target HTML elements. This approach combined natural language with structural knowledge of the interface, potentially offering a balance between flexibility and precision.

Test Case Design

To ensure consistency across testing scenarios, a standardized test case was developed that simulated a realistic battery disassembly workflow in the DemoRec Site environment:

> *"I started my battery disassembly on 29th of January, 2025. My car is a 2025 VW ID5 with a voltage of 750V and a capacity of 100 kWh. The Battery Management System is located in the engine compartment. The battery health is currently at 70%, and the Vehicle Identification Number is 000. The component is a not healthy VW Pack. Start the process."*

This test case was designed to include multiple types of inputs, such as numbers with units, names, and dates, categorical selections that require interaction with dropdown menus, and button presses for user interactions.

For each scenario, five test runs were conducted to account for variability and to establish reliable performance metrics. The evaluation focused on three key dimensions:

- **Accuracy:** Measured as the percentage of correctly populated fields across different element types (text fields, dropdowns, and buttons).
- **Efficiency:** Measured as the total time required to complete the workflow, including processing time.
- **Processing Overhead:** For voice-based interactions, the proportion of time spent on Whisper processing was measured to identify potential bottlenecks.

The complete scripts used for sequential voice input and specific HTML voice input testing can be found in the appendix (see Figure B.1 and Figure B.2 respectively).

4.2.2 Task Completion Results

Expected Outcomes

Prior to testing, several hypotheses were formulated regarding the expected performance of each interaction modality:

1. Manual input was expected to offer high accuracy, but potentially lower efficiency due to the mechanical nature of keyboard and mouse interactions. Unfortunately, this test does not account for the time spent switching between disassembly and manual data entry, limiting the insights it provides.

2. Sequential voice input was expected to offer high accuracy at the cost of efficiency, as each field would require individual attention.

3. Batch voice input was expected to provide the highest efficiency but potentially at the cost of accuracy, particularly for complex or ambiguous inputs.

4. Specific HTML voice input was expected to balance accuracy and efficiency by combining natural language with structural guidance.

Analysis

The results of this test are visualized in Figure 4.2. Furthermore, the average accuracies for each testing scenario are presented in Table **??**.

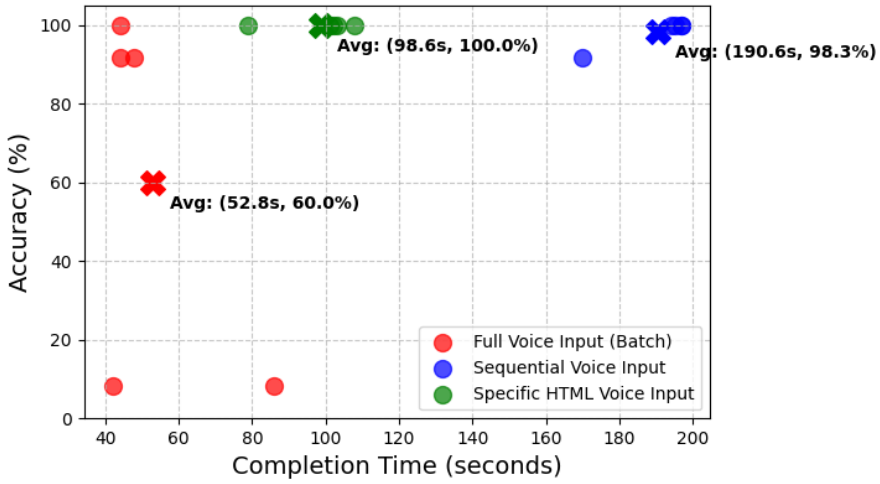

Figure 4.2: Accuracy and Time Results for the Task Completion Test

Input Modality	Text Fields	Dropdowns	Buttons	Overall
Manual Input	100%	100%	100%	100%
Sequential Voice Input	100%	100%	80%	98.33%
Specific HTML Voice Input	100%	100%	100%	100%
Batch Voice Input	64.4%	60%	20%	60%

Table 4.2: Accuracy Results by Input Modality and Element Type

Naturally, manual input achieves perfect accuracy across all element types. The accuracy results revealed several notable patterns in the voice input scenarios.

Sequential voice input demonstrated near-perfect accuracy (98.33% overall), with occasional failures in button activation (80% success rate). Its high accuracy can be attributed to the isolated handling of each function call, minimizing contextual ambiguity.

Specific HTML voice input performed flawlessly, achieving perfect accuracy across all element types and matching the performance of manual input. Explicitly specifying elements eliminated ambiguity in the LLM's interpretation, ensuring that function calls aligned precisely with their intended descriptions. However, during one test, the LLM mistakenly matched the car model to a similarly named existing pack type and incorrectly selected it from the dropdown. This required additional prompting to revert to the correct state. Fortunately, with the implemented context-gathering mechanism, restoring previous values was seamless.

In contrast, batch voice input exhibited significantly lower accuracy (60% overall), with particularly poor performance in button activation (20% success rate). The primary challenge lay in parsing multiple pieces of information from a single prompt, especially when commands did not precisely match predefined labels. However, despite the lower accuracy, no incorrect data was entered into the wrong fields. Interestingly, structured inputs, such as "My car is a 2025 VW ID5", were often processed correctly, with "VW" assigned as the brand, "ID5" as the model, and "2025" as the production year. Accuracy in batch voice input varied widely, ranging from 100% in the best case to just 8.33% in the worst case. Notably, while some function calls were missed, no erroneous actions were executed, an outcome preferable to excessive incorrect function executions, which would require additional corrective actions.

The reduced accuracy in button function calls can be attributed to the strict function calling descriptions defined. This approach was implemented to prevent the LLM from mistakenly invoking button logic for potential synonyms. Since buttons serve critical functions, such as submitting data, saving information, navigating between pages, or sending inputs, erroneous activations could lead to frustrating user experiences.

Despite lower accuracy, out of the total 180 function calls to be made during testing, only a single call was incorrect, yielding a misfire rate of less than 1% under clean audio conditions. This highlights the robustness of the system in handling function calls reliably. From a user experience perspective, the low misfire rate means fewer disruptions and corrections, making interactions smoother and more intuitive.

The efficiency results for each testing scenario are presented in Table 4.3.

Input Modality	**Average Time (s)**	**Best Case (s)**	**Worst Case (s)**
Manual Input	45.4	34	73
Sequential Voice Input	190.6	170	197
Specific HTML Voice Input	98.6	79	108
Batch Voice Input	52.8	42	86

Table 4.3: Efficiency Results by Input Modality

For voice-based modalities, the proportion of time spent on Whisper transcription processing is presented in Table 4.4.

Voice Input Modality	**Average Whisper Time (s)**	**Percentage of Total Time**
Sequential Voice Input	70.4608	36.9%
Specific HTML Voice Input	28.051.8	28.4%
Batch Voice Input	9.3576	17.7%

Table 4.4: Whisper Processing Overhead by Voice Input Modality

Sequential voice input proved to be the least efficient, with an average completion time of 190.6 seconds. A significant portion of this time (36.9%) was spent on Whisper transcription processing, reflecting the cumulative overhead of handling multiple discrete commands individually.

Specific HTML voice input performed more efficiently, completing tasks in an average of 98.6 seconds. This approach struck a better balance between accuracy and efficiency, with a reduced Whisper processing overhead of 28.4%, indicating improved batch handling of commands compared to sequential input.

Batch voice input was the most efficient among voice-based approaches, with an average completion time of 52.8 seconds, closely matching the speed of manual input. This efficiency gain stemmed from a more natural language approach that required less detailed prompting. Whisper processing overhead was minimal at 17.7%, suggesting that handling audio in larger segments is significantly more efficient than processing multiple smaller ones.

Despite its efficiency, batch voice input showed considerable variability. In the best-case scenario, completion time dropped to 42 seconds, rivaling even the fastest manual input methods. However, in the worst case, it extended to 86 seconds, making it substantially slower, which is attributed to Whispers variability. Nevertheless, this fluctuation closely mirrored the accuracy results, indicating that when batch processing is successful, it delivers both high accuracy and efficiency, but when it fails, both metrics decline.

4.2.3 Discussion and Implications

The comprehensive evaluation of task completion across different input modalities has yielded valuable insights into their respective strengths and limitations, offering important implications for the tools usage.

Optimal Use Cases for Each Modality

Each input modality exhibits distinct advantages, making them suitable for specific use cases. Manual input remains the most reliable method for tasks requiring high accuracy, offering consistent performance with moderate efficiency. However, the major drawback of frequent switching between disassembly and data entry was not directly measured in this study, despite being the motivation for this research.

Sequential voice input delivers near-perfect accuracy but at the expense of efficiency. It is particularly suited for scenarios where precision is essential, and time constraints are less important. Moreover, the results highlight the tools performance for isolated commands and less complex prompts.

The approach of specifically stating the HTML elements strikes an optimal balance between accuracy and efficiency, making it best-suited for users familiar with the interface structure who require hands-free operation with high reliability. By explicitly specifying elements, this approach minimizes ambiguity in the LLM's interpretation, ensuring precise function resolution.

Batch voice input offers the highest efficiency among voice-based methods but exhibits inconsistent accuracy. It is best suited for routine tasks with simple inputs, where occasional unidentified function calls can be easily repeated through additional prompts.

This modality is particularly effective in scenarios where speed is prioritized over absolute precision.

Best Practices and Recommendations

The evaluation results suggest several best practices for improving voice-based interaction. Structured voice commands significantly enhance accuracy, as clear field designations help minimize misinterpretations, even in batch mode. Users should be informed to provide explicit and well-structured inputs to maximize the effectiveness of voice interactions.

When accuracy is critical, specifying target elements explicitly leads to the most reliable outcomes. The Specific HTML Voice Input modality demonstrated how element specification reduces ambiguity, ensuring that function calls align precisely with their intended descriptions.

A hybrid interaction approach may be the most effective strategy: Users can begin with batch input for efficiency and switch to sequential or specific input for correcting misinterpreted fields. This adaptive method allows for a balance between speed and precision, optimizing the user experience.

Additionally, optimizing the underlying speech processing system could significantly enhance efficiency. A considerable portion of time, particularly in sequential input, was spent on Whisper transcription. Exploring alternative services or refining the processing pipeline could reduce latency and improve overall performance.

4.2.4 Environmental Robustness

Even though transcription accuracy in noisy environments is primarily determined by the underlying Whisper model (as discussed in Section 2.2), this evaluation assesses the tool's current robustness under different audio conditions, as noises are expected in an industrial disassembly environment. The same set of spoken commands used in the specific HTML test in 4.2.1 is tested under varying levels of background noise, and transcription accuracy is measured along with the system's ability to correctly execute the intended function.

Clean Audio

This baseline test assesses transcription accuracy in an ideal environment with no background noise. As previously examined in 4.2.1, the tool performed flawlessly under these optimal conditions. The system achieved a perfect 100% accuracy rate across all five test cases, confirming that under optimal conditions, the system is capable of perfect recognition and execution.

Low Volume Background Noise

To assess the impact of mild environmental noise, white noise with an amplitude of 0.8 was introduced at a low volume (voice at +36dB, noise at -36dB). The tool maintained a flawless accuracy rate, correctly transcribing 100% of test cases. This result confirms that

Whisper effectively filters out minor background disturbances without significant loss in transcription reliability.

The primary difference compared to clean audio conditions is that, when voice sensitivity is not ideally configured, silent periods may go undetected. However, this does not disrupt functionality, as the system relies on the fallback timeout mechanism, which, while potentially increasing response latency slightly, ensures reliable operation.

High Volume Background Noise

A more challenging scenario was introduced by increasing the background white noise to a level louder than the speaker's voice (voice now at +0dB, noise at -17dB). Under these conditions, the average function resolution accuracy across all test cases was 93.34%. While Test Cases 1, 3, and 4 maintained perfect 100% accuracy, Test Case 2 showed a reduced accuracy of 75%, as only the text fields were set, and Test Case 5 achieved 91.7% accuracy by not evoking the button press.

Despite the high noise levels causing occasional transcription errors, function execution remained highly reliable. For instance, while some words were misinterpreted, the system correctly identified key command structures and successfully executed nearly all intended functions. One notable example of a noisy transcription was:

> *"Set the starter tax grid value to the 29th of January 2025. Set the car brand tax grid value to Volkswagen. Set the carbon model tax grid value to ID5. And set the year of production tax grid value to 2025. Set the voltage tax grid value to 750. Set the capacity tax grid value to 100 kWh. And the D&F location tax grid value to engine compartment. Set the health tax grid value to 70%. Set the vehicle identification number tax grid to 000. From the component time dropdown select the VW type. From the component condition dropdown select other. Press the start button. That's it."*

Although minor transcription errors were present, function execution accuracy remained perfect in this case with an accuracy of 100%, demonstrating the robustness of the tool in noisy environments.

Test Case	Clean Audio	Low Noise	High Noise
1	100%	100%	100%
2	100%	100%	75%
3	100%	100%	100%
4	100%	100%	100%
5	100%	100%	91.7%
Average	100%	100%	93.34%

Table 4.5: Function Resolution Accuracy Under Different Audio Conditions

4.2.5 Edge Case Testing

This section evaluates how well the tool handles non-standard inputs and unexpected scenarios that might occur during usage in the DemoRec context. Edge case testing

is crucial for ensuring system robustness and proper error handling across various user interaction patterns.

Ambiguous Commands

This test examines how the system handles commands that could have multiple interpretations, leading to possible confusion in execution.

Test Scenarios and Expected Behavior: Two primary test cases were chosen to evaluate the system's response to ambiguous commands. In the first scenario, the command "Click button" was issued in a context where multiple buttons with identical labels were present on the screen. For the second scenario, the system to "Select the first option of the Component Kind dropdown" was said when faced with a dropdown containing multiple options.

In both cases, it was expected that the system would handle the ambiguity in one of two acceptable ways: either by requesting clarification from the user (such as asking "Which button do you want to click?") or by selecting a default action based on contextual information if available. Most importantly, whatever approach the system took it was expected to be consistent and provide clear feedback to the user about how the ambiguity was resolved.

Results: In the case of trying to invoke a button where there are multiple ones with the same label, the last element in the DOM tree is used. This occurs because buttons are internally saved with the label as the key, and with duplicates, previous entries are simply overwritten. While this implementation is straightforward, it presents limitations in handling identical labels. A more sophisticated approach would require an additional indexing system that maintains unique identifiers for each element regardless of label duplication. The current architecture relies on label-based identification as the primary mechanism for the LLM to seamlessly locate the correct HTML Element, making duplicate handling challenging without substantial architectural changes.

In cases of complete ambiguity where it is unclear which button to activate because no specific label was provided, the LLM behaves as expected by requesting clarification from the user. This interactive clarification process significantly enhances user experience by providing immediate feedback about the system's understanding, as depicted in Figure 4.3.

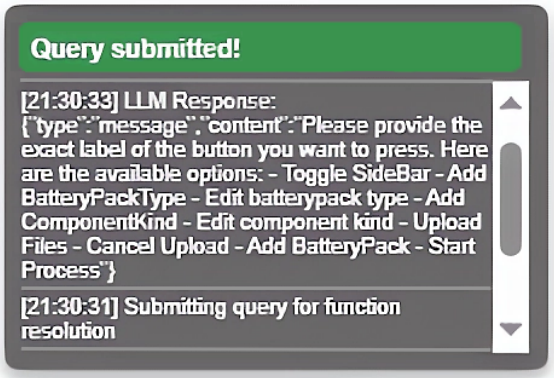

Figure 4.3: LLM UI Feedback for "Press the Button. submit." Query

When attempting to select dropdown items by index rather than by label, the system demonstrated inconsistent behavior. Despite thorough analysis, no clear pattern of success or failure could be established. This inconsistency is attributed to the underlying GPT-4o language model, as the elements are provided to the LLM as an iterable list, which should theoretically support index-based selection without difficulty. A potential enhancement would be to explicitly append index information to element labels, providing the LLM with alternative selection methods. However, this approach requires comprehensive testing and would remain dependent on the underlying LLM's interpretation capabilities.

Overlapping Commands

This section examines how the system processes multiple voice commands when issued either simultaneously or with minimal time separation.

Test Scenarios and Expected Behavior: To evaluate how the system handles overlapping commands, two test scenarios were designed. In the first case, the command "Set Voltage to 50" was issued followed immediately by "Press the Start Process button," introducing two distinct commands with minimal time separation between them. For the second scenario, true command overlap was tested by beginning to speak the command "I have a capacity of 100." while simultaneously stating "Set Voltage to 50" and "My Vehicle Identification Number is 000". The system was expected to mix the inputs and result in invalid transcription. The resolution should be clear to the user.

Results: The test results denied the assumption that overlapping speech consistently leads to command ambiguity and recognition failures. Instead, the system demonstrated varied behaviors depending on the nature of the overlap, as summarized in Table 4.6.

When two commands related to Capacity and Voltage were issued simultaneously, the system exhibited a strong preference for processing only one command rather than both. In 80% of test cases, the Capacity command was correctly identified, while the Voltage command was ignored. The Voltage command prevailed in 20% of cases. This suggests that in simultaneous speech scenarios, the system does not queue or process both commands but instead prioritizes one based on recognition confidence.

For commands issued with a slight delay, where the Voltage instruction began first, the results varied. In 40% of test cases, both commands were correctly processed in the intended sequence. However, in another 40%, only the Capacity command was recognized despite being issued second, while in the remaining 20%, only the Voltage command was processed. These findings indicate that temporal order does not consistently determine command execution priority. Supporting this inconsistency, when the sequence was reversed, with the Capacity command issued first and the Voltage command following, the system produced hybrid responses in 100% of cases. Specifically, the system correctly identified the Voltage command but mistakenly assigned it the value intended for the Capacity command.

The system's performance degraded further when three overlapping commands (Capacity, Voltage, and Vehicle Identification Number) were introduced simultaneously. In all test cases, the system failed to execute any function calls, resulting in a 100% failure rate. Even with slight timing offsets, no significant improvement in accuracy was observed. The only noticeable difference was a marginal increase success-rate to identify values when commands were more isolated, but no clear pattern emerged.

These findings are illustrated in Figure 4.4, which visualizes the system's behavior when handling overlapping and near-simultaneous commands.

Test Scenario	Both Commands	First Command	Second Command	Failed/Mixed
Simultaneous (Capacity & Voltage)	0%	20%	80%	0%
Offset (Voltage first)	40%	20%	40%	0%
Offset (Capacity first)	0%	0%	0%	100%
Three Commands Simultaneous	0%	0%	0%	100%

Table 4.6: Command Recognition Accuracy with Overlapping Instructions

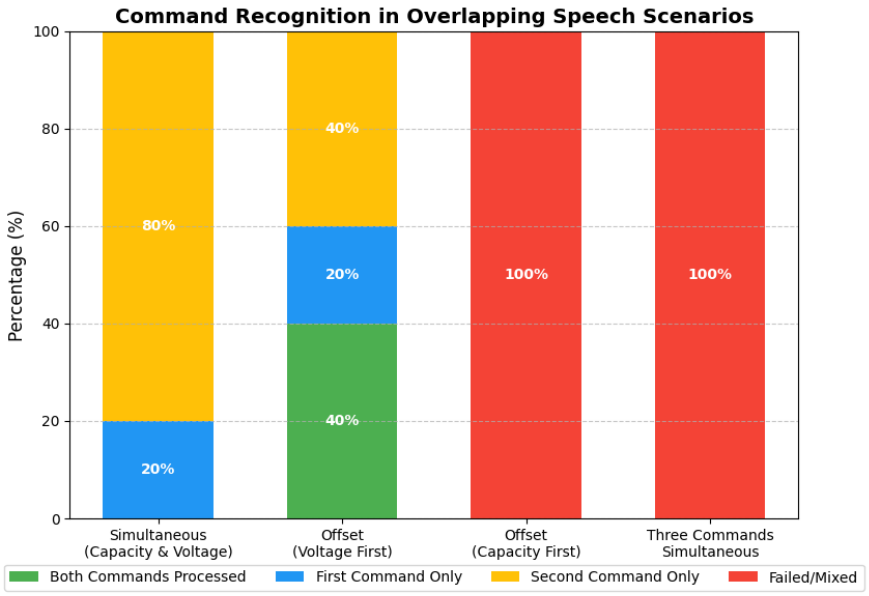

Figure 4.4: Command Recognition Patterns in Overlapping Speech Scenarios. The diagram illustrates the system's behavior when processing multiple simultaneous or near-simultaneous commands with slight offsets.

Contrary to initial expectations, the system demonstrated occasional success in recognizing concurrent commands, particularly when only two instructions were involved. However, in general, the anticipated outcome of hybrid or mixed responses was observed, particularly as command complexity increased. These results have important implications for real-world applications, such as for the DemoRec battery disassembly. In a production setting, inconsistent command recognition could lead to a frustrating user experience or operational inefficiencies. A robust mechanism to handle overlapping commands, such as prioritization rules, or speaker segmentation, would be necessary to ensure reliable system performance in practical scenarios, where there might be multiple speakers.

Out-of-Scope Inputs

Here, the system's reaction to when the user tries to input commands that are outside the tool's expected functionality or reference non-existent UI elements is evaluated.

Test Scenarios and Expected Behavior: To assess the system's response to out-of-scope inputs, two test cases involving non-existent functionality were tested. In the first

scenario, the system was instructed to "Set the car weight to 300," referencing a text field that does not exist in the interface. For the second test, the system was asked to "Compute the amount", a function that is not implemented within the system. In both cases, the system was expected to clearly notify the user that the requested command is not supported, with responses such as "Sorry, I can't help with that" or "That's outside of my capabilities." Additionally, it was anticipated that these notifications would be straightforward and potentially offer guidance toward available alternatives when appropriate.

Results: Systematic testing revealed that when presented with undefined functions or non-existent interface elements, the system consistently showed appropriate no-behavior rather than attempting to approximate or guess at user intent. Moreover, when tasked to perform actions on non-existent elements or execute undefined functions, the system appropriately returned clear notifications indicating that the requested function or element was unavailable. This robust error handling represents a significant usability advantage, as the system avoids a frustrating pitfall of attempting to execute the closest available approximation of an invalid command, which could lead to unintended consequences or user confusion. Instead, by providing explicit feedback about command invalidity, the system maintains transparency and helps guide users toward correct interactions. An example of this feedback mechanism is shown in Figure 4.5.

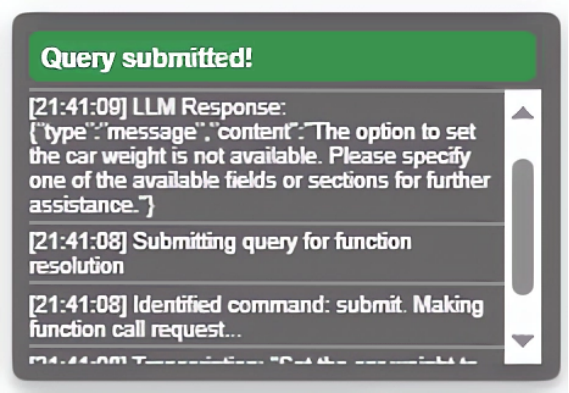

Figure 4.5: LLM UI Feedback for "Set the car weight to 300. submit." Query

Additional Considerations and Conclusion

It is important to note that numerous other edge cases will occur during normal system operation in real-world practice. Many of these, such as speech recognition errors from mispronunciations, speech cutoffs due to microphone issues, background noise interference, and variations in accent or dialect, ultimately result as forms of ambiguous commands within the system.

Furthermore, the system's ability to handle these edge cases is fundamentally linked to the capabilities of the underlying speech transcription and language model services. Performance in these scenarios is expected to improve with future iterations of these services. However, the current implementation already demonstrates a reasonable level of robustness against common edge cases, particularly in its ability to request clarification when commands are ambiguous, maintain sequential execution order for multiple commands, and provide clear feedback when requested actions cannot be performed.

Chapter 5

Conclusion

This thesis presents the development of a tool that enables voice-controlled data entry in MontiGem-generated web interfaces, applied specifically to the battery disassembly process of the DemoRec project. While staying close to MontiGem's requirements and architecture, the tool extends beyond this use case and can be integrated into any web application with a frontend and backend, requiring minimal configuration. It leverages novel artificial intelligence approaches including Azure OpenAI services, Whisper for speech-to-text transcription, and GPT-4o for mapping natural language to user-defined functions with function calling, while allowing users to define their own communication protocols.

5.1 Contributions

This research enhances accessibility and efficiency in human-computer interaction by introducing a scalable voice control solution for web applications. Its hands-free operation is particularly valuable in industrial environments such as battery disassembly, maintenance, and logistics, where manual data entry can be disadvantageous. Beyond industrial applications, the tool holds significant potential in fields like healthcare, assistive technologies, smart home automation, and automotive interfaces, enabling more seamless and intuitive interactions.

By demonstrating the feasibility of integrating AI-driven voice recognition into web applications, this thesis bridges the gap between theoretical advancements in language models and their practical implementation. The system's architecture does not require backend-to-frontend communication, ensuring compatibility with various environments through a string-based messaging approach.

5.2 Performance

The tool shows promising real-time appliances as even for longer queries up to a minute in input length, the system is able to show results in a couple of seconds, achieving a pleasant user experience. To enhance efficiency, the tool includes optional support for a WebSocket-based setup using Spring Boot, reducing network load and improving latency. Compared to Base64 polling-based retrieval, direct byte streaming over WebSockets decreases audio

package sizes by 25% and improves transmission latency by 84.3% on average. Polling remains a fallback option, though it introduces potential delays depending on the configured intervals. Silent detection mechanisms decrease the number of API calls and enhance performance by triggering transcription only after three seconds of silence or a 15-second timeout. Additionally, a keyword-based activation system ensures that API calls occur only when necessary, similar to common smart assistant systems.

5.2.1 Optimal Input and Usage Patterns

The comprehensive evaluation of different input modalities revealed significant insights into their respective strengths and limitations. Saying instructions sequentially reaches near-perfect accuracy (98.33%) but at the expense of efficiency, as the input duration will increase drastically. This approach is particularly suited for scenarios where precision is key and time constraints are less important. Specifically mentioning the HTML Element strikes an optimal balance between accuracy and efficiency, achieving 100% accuracy during testing with an lower completion time. By explicitly specifying elements, this approach minimizes ambiguity in the LLM's interpretation, ensuring precise function resolution. Natural batch voice input offers the highest efficiency among voice-based methods, rivaling direct matching manual input speed, but exhibits inconsistent accuracy (60% overall). It is best suited for routine tasks with simple inputs, where occasional unidentified function calls can be easily repeated through additional prompts. The evaluation results suggest several best practices for improving voice-based interaction. A hybrid interaction approach may be the most effective strategy: users can begin with batch input for efficiency and switch to sequential or specific input for correcting misinterpreted fields. This adaptive method allows for a balance between speed and precision, optimizing the user experience.

5.2.2 Robustness

While perturbation robustness depends almost exclusively on the underlying transcription model, environmental noise in industrial settings like battery disassembly are expected. Therefore, it is important for the tool to work beneficial even under difficult conditions. Testing under varying audio conditions demonstrated the tool's robustness in real-world scenarios. In clean audio environments, the system achieved perfect 100% accuracy and a less than 1% function call misfire rate. This performance was maintained even with low-volume background noise. Under high-volume noise conditions, the system maintained a respectable 93.34% average function resolution accuracy, demonstrating its resilience in challenging environments.

Edge case testing revealed important patterns in the system's handling of ambiguous commands, overlapping inputs, and out-of-scope requests. For ambiguous commands, the system appropriately requests clarification from the user. However, the handling of dropdown items by unconventional means rather than label showed inconsistent behavior. When processing overlapping commands, the system demonstrated varied behaviors depending on the nature of the overlap. For two simultaneous commands, the system typically processed only one, showing a preference based on recognition confidence rather than temporal order. With three overlapping commands, the system consistently failed to execute any function calls, indicating a limitation in handling complex overlapping speech.

For out-of-scope inputs, the system consistently provided clear notifications indicating that the requested function or element was unavailable, avoiding the pitfall of attempting to approximate invalid commands.

5.3 Future Research

Future research should focus on employing voice-activity-detection systems or implementing effective speaker separation practices to better manage challenges with overlapping speech and background noise. Speaker differentiation would be particularly valuable in collaborative environments where multiple users might interact with the system simultaneously. Advanced audio processing techniques could potentially segment overlapping speech into distinct command streams, enabling more reliable recognition and execution.

The system's architecture could evolve to enable LLMs to directly read DOM trees to correctly identify elements for functions to call regardless of their representation and labeling. This would enhance the system's flexibility in handling complex interfaces with non-standard element structures or duplicate identifiers.

Domain-specifically trained models, fine-tuned on battery disassembly terminology and common command patterns within the DemoRec environment, will further improve the accuracy of the overall system. Leveraging specialized training datasets significantly enhances model performance by reducing ambiguities and improving task handling. For example, fine-tuning can optimize the recognition of technical terms, enabling the system to better interpret domain-specific tasks that general-purpose models may struggle with. Additionally, fine-tuning could lead to a reduction in latency and token consumption, making the system more responsive in real-world applications.

The integration of multimodal interactions, such as combining voice with gestures or eye-tracking, could further expand the usability of voice-driven interfaces across various fields, providing users with more flexible and intuitive ways to interact with complex systems. These complementary input methods could help resolve ambiguities in voice commands by providing additional contextual information.

As speech processing and language model technologies continue to evolve, future iterations of this system could implement adaptive learning mechanisms to refine speech recognition over time, potentially customizing to individual users' speech patterns and domain-specific vocabulary. Such personalization would further enhance the system's robustness and usability in specialized contexts like battery disassembly and other industrial applications.

Bibliography

[AMN+20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise information systems in academia and practice: Lessons learned from a mbse project. In *40 Years EMISA 2019*, pages 59–66. Gesellschaft für Informatik e.V., Bonn, 2020.

[BRBR18] Vladimir Britanak, KR Rao, Vladimir Britanak, and KR Rao. Audio coding standards,(proprietary) audio compression algorithms, and broadcasting/speech/data communication codecs: overview of adopted filter banks. *Cosine-/Sine-Modulated Filter Banks: General Properties, Fast Algorithms and Integer Approximations*, pages 13–37, 2018.

[BZMA20] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *CoRR*, abs/2006.11477, 2020.

[CJR+22] Neelam Chandolikar, Chaitanya Joshi, Prateek Roy, Abhijeet Gawas, and Mini Vishwakarma. Voice recognition: A comprehensive survey. In *2022 International Mobile and Embedded Technology Conference (MECON)*, pages 45–51, 2022.

[ea24] OpenAI et al. Gpt-4 technical report, 2024.

[ELJ+24] Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Tinyagent: Function calling at the edge, 2024.

[Fou25] Xiph.Org Foundation. Xiph.org, 2025. Accessed: 2025-23-02.

[Gro25] The Moving Picture Experts Group. Mpeg, 2025. Accessed: 2025-01-04.

[HGC23] Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing, 2023.

[HSW+21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[Jos06] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.

[Mic24] Microsoft. Azure openai client library for java - version 1.0.0-beta.13, 2024. Accessed: 2025-01-03.

[Mic25] Microsoft. Playwright, 2025. Accessed: 2025-02-16.

[Mon25] MontiCore. Monticore umlp gitlab - wiki, 2025. Accessed: 2025-02-16.

[Moz25a] Mozilla. Media types and formats for image, audio, and video content, 2025. Accessed: 2025-01-04.

[Moz25b] Mozilla. Mediastream recording api, 2025. Accessed: 2025-02-23.

[oEMC24] RWTH Production Engineering of E-Mobility Components. Demorec, 2024. Updated: 2024-07-10.

[Ope23] OpenAI. Openai developer platform, 2023. Accessed: 2024-10-26.

[Ope25a] OpenAI. Openai help center, 2025. Accessed: 2025-03-23.

[Ope25b] OpenAI. Openai playground: Chat, 2025. Accessed: 2025-01-18.

[Pyd] Pydantic. Pydantic: Data validation and settings management using python type annotations `https://docs.pydantic.dev/latest/`. Accessed: 2025-01-18.

[RKX+22] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022.

[Rum21] Bernhard Rumpe. Monticore language workbench and library handbook: Edition 2021. 2021.

[Smi20] Tetiana Smirnova. Comparative analysis of modern formats of lossy audio compression. 2020.

[SNH+24] Muhammad A. Shah, David Solans Noguero, Mikko A. Heikkila, Bhiksha Raj, and Nicolas Kourtellis. Speech robust bench: A robustness benchmark for speech recognition, 2024.

[Spr25] Spring by VMware Tanzu. *Spring Documentation*, 2025.

[SZX+24] Jiankai Sun, Chuanyang Zheng, Enze Xie, Zhengying Liu, Ruihang Chu, Jianing Qiu, Jiaqi Xu, Mingyu Ding, Hongyang Li, Mengzhe Geng, Yue Wu, Wenhai Wang, Junsong Chen, Zhangyue Yin, Xiaozhe Ren, Jie Fu, Junxian He, Wu Yuan, Qi Liu, Xihui Liu, Yu Li, Hao Dong, Yu Cheng, Ming Zhang, Pheng Ann Heng, Jifeng Dai, Ping Luo, Jingdong Wang, Ji-Rong Wen, Xipeng Qiu, Yike Guo, Hui Xiong, Qun Liu, and Zhenguo Li. A survey of reasoning with foundation models, 2024.

[TD18] Amrita S Tulshan and Sudhir Namdeorao Dhage. Survey on virtual assistant: Google assistant, siri, cortana, alexa. In *International symposium on signal processing and intelligent recognition systems*, pages 190–201. Springer, 2018.

[Tea25] JSON Schema Team. Json schema, 2025. Accessed: 2025-01-03.

[Web25] WebM. The webm project, 2025. Accessed: 2025-02-23.

[Zod] Zod. Zod: Typescript-first schema declaration and validation library `https://zod.dev/`. Accessed: 2025-01-18.

[ZZL+24] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2024.

Appendix A

Listings

```java
public void performFunctionCall(String query) {
  ...

  String nameSchema = "{"
      + "\"type\": \"object\","
      + "\"properties\": {"
      + "    \"name\": {"
      + "        \"type\": \"string\","
      + "        \"description\": \"The name to enter in the name
          input field\""
      + "    }"
      + "},"
      + "\"required\": [\"name\"]"
      + "}";

  String ageSchema = "{"
      + "\"type\": \"object\","
      + "\"properties\": {"
      + "    \"age\": {"
      + "        \"type\": \"string\","
      + "        \"description\": \"The age to enter in the age
          input field\""
      + "    }"
      + "},"
      + "\"required\": [\"age\"]"
      + "}";

  String emailSchema = "{"
      + "\"type\": \"object\","
      + "\"properties\": {"
      + "    \"email\": {"
      + "        \"type\": \"string\","
      + "        \"description\": \"The email to enter in the email
          input field\""
      + "    }"
      + "},"
      + "\"required\": [\"email\"]"
      + "}";

  List<ChatCompletionsToolDefinition> tools = Arrays.asList(
      new ChatCompletionsFunctionToolDefinition(
         new ChatCompletionsFunctionToolDefinitionFunction("setName"
             )
             .setDescription("Sets the name in the name input field"
                 )
             .setParameters(BinaryData.fromString(genericSchema))
      ),
      // same for age and email
      ...
  );

  List<ChatRequestMessage> chatMessages = new ArrayList<>();
  ...
  ChatCompletionsOptions chatCompletionsOptions = new
     ChatCompletionsOptions(chatMessages);
  chatCompletionsOptions.setTools(tools);
  ChatCompletions chatCompletions = client.getChatCompletions("gpt-4
     o-mini", chatCompletionsOptions);
  ...
}
```

Listing A.1: Tool schema definition for functions to change each textfield.

```
public void setTextField(String textField, String value){
    switch (textField) {
        case "name":
            setName(value);
            break;
        case "age":
            setAge(value);
            break;
        case "email":
            setEmail(value);
            break;
        default:
            break;
    }
}

public void performFunctionCall(String query) {
  ...
  String genericSchema = "{"
  + "\"type\": \"object\","
  + "\"properties\": {"
  + "    \"textField\": {"
  + "        \"type\": \"string\","
  + "        \"enum\": [\"name\", \"age\", \"email\"],"
  + "        \"description\": \"The text field to enter the value in
      \""
  + "    },"
  + "    \"value\": {"
  + "        \"type\": \"string\","
  + "        \"description\": \"The text to enter inside the
      specified textfield\""
  + "    }"
  + "},"
  + "\"required\": [\"textField\", \"value\"]"
  + "}";
  ...
  new ChatCompletionsFunctionToolDefinition(
      new ChatCompletionsFunctionToolDefinitionFunction("
          setTextField")
          .setDescription("Sets the extracted value (name, age,
              email) into the corresponding text field")
          .setParameters(BinaryData.fromString(genericSchema))
  )
  ...
}
```

Listing A.2: One parameteized funnction definition instead of many to decrease token amount and latency.

```
1  @FunctionCall(
2    function () {
3    return {
4      name: "setTextField",
5      description: "Sets the given value into the textfield of
           name provided by the textField parameter.",
6      parameters: {
7        type: "object",
8        properties: {
9            textField: {
10               type: "string",
11               enum: SpeechFunctionCaller.getInstance().
                    getAllElements("gem-text-input"),
12               description: "The text field to enter the
                    value in"
13           },
14           value: {
15               type: "string",
16               description: "The text to enter inside the
                    specified textfield"
17           }
18       },
19       required: ["textField", "value"]
20     }
21   };
22 })
23 public setTextField(textField: string, value: string): void
     {
24   const inputElement = SpeechFunctionCaller.getInstance().
       getElement("gem-text-input", textField);
25
26   if (inputElement instanceof HTMLInputElement) {
27     inputElement.value = value;
28   } else {
29     console.error("Input element of label: '" + textField +
         "' not found or is not an input element.");
30   }
31 }
```

Listing A.3: Decorator-based function for setting text field values, making it accessible to the LLM.

```
/**
 * Configures credentials for backend AZURE service where
    transcriber model and resolver resides
 * @param endpoint AZURE service URL
 * @param token AZURE Authentication token
 * @param transcriber_model Model for speech transcription
 * @param resolver_model Model for function resolution
 */
public async setCredentials(endpoint: string, token: string,
   transcriberModel: string, resolverModel: string) {
    await this.sendDataToHandler(JSON.stringify({
        "function": "FunctionResolver.getInstance().
           setCredentials",
        "parameters": [endpoint, token, resolverModel]
    }));

    await this.sendDataToHandler(JSON.stringify({
        "function": "Transcriber.getInstance().setCredentials"
           ,
        "parameters": [endpoint, token, transcriberModel]
    }));
}
```

Listing A.4: Configures credentials for backend AZURE service where transcriber model and resolver resides

Appendix B

Figures

"I started my battery disassembly on 29th of January, 2025. Submit. [wait for function resolution (wffr)]
My car is a 2025 VW ID5. Submit. [wffr]
It has a voltage of 750V. Submit. [wffr]
The capacity is 100 kWh. Submit. [wffr]
The Battery Management System is located in the engine compartment. Submit. [wffr]
The battery health is currently at 70%. Submit. [wffr]
The Vehicle Identification Number is 000. Submit. [wffr]
The component kind is a VW Pack. Submit. [wffr]
The component condition is not healthy. Submit. [wffr]
Start the process. Submit."

Figure B.1: Script of the Sequential-Voice-Input test.

"Set the 'Started' textfield value to the 29th of January, 2025. Set the 'Car Brand' textfield value to 'Volkswagen', set the 'Car Model' textfield value to 'ID5', and set the 'Year of Production' textfield value to 2025. Set the 'Voltage' textfield value to 750, set the 'Capacity' textfield value to 100 kWh, and the 'BMS Location' textfield value to 'engine compartment'. Set the 'Health' textfield value to 70%. Set the 'Vehicle Identification Number' textfield to 0-0-0. From the 'Component Kind' dropdown, select 'VW Pack', from the 'Component Condition' dropdown, select 'other'. Press the start process button. Submit."

Figure B.2: Script for the Specific-HTML-Voice-Input test.