MULTIMODAL DIGITAL ASSISTANT FOR A SMART FOOD PROCESSOR: A DESIGN SCIENCE APPROACH

Amitrajit Sarkar[1], Nicholas Leslie[2], John Ascroft[2], Ralph Bergmann[3], Lilly Heindl[3] and Markus Westner[4]

[1]*ARA Institute of Canterbury, New Zealand*
[2]*Jade Software Corporation, New Zealand*
[3]*BSH Hausgeräte GmbH, Germany*
[4]*OTH Regensburg, Germany*

ABSTRACT

This paper presents the results of a comprehensive research project on the integration of an alternative human-computer interaction interface in home appliances, using a design science research approach. The outcome of the project is a user interface concept and prototype with a focus on multimodal design that incorporates voice commands and manual interaction to reduce friction and improve user experience. The paper presents the design and development of an offline natural language processing (NLP)-based application for a food processor, which allows users to interact with the appliance through voice-based interaction. The research and testing of offline NLP solutions is detailed, and the results of research into alternative user interfaces for smart home appliances are presented. The findings suggest that external application programming interface (API) solutions such as Google's should be used for conversational API services, and that natural language understanding service implementations can be extended to provide additional functionality. The results illustrate an alternative novel and effective approach to user interaction with smart home appliances.

KEYWORDS

Digital Assistant, Design Science Research, Design Artifact Development, Innovation, Artificial Intelligence

1. INTRODUCTION

Nowadays, digital assistants that are primarily operated through voice commands, have become a part of daily interaction of users with computer systems (Fernandes and Oliveira, 2021). Popular and commonly used assistants are provided by the global technology companies such as Amazon (“Alexa”), Apple (“Siri”), or Google. Aside from those appliance- or phone-related assistants, voice-based human-computer interaction, also plays a significant role in other application scenarios, such as cars (Scates, 2022). Another field for, partially artificial intelligence (AI)-based, digital assistants are home appliances.

In 2020/2021 we had an opportunity to be part of an international virtual collaboration project. The multinational project involved two business organizations, and two academic institutions, with equal representation from Germany and New Zealand.

Based on the generic user story of our artifact development we will here narrate the scenario which is like the scenario described by Maedche et al. (2019): It is an illustration of a digital assistant powered by AI interacting in the scenario of a smart kitchen. When cooking in the kitchen, a user can activate a food processor without touching the appliance's switch by interacting with a speech-based conversational agent (such as Alexa or Siri). The AI-powered digital assistant uses the audio recordings as input, analyzes the speech, and attempts to comprehend the language. The anticipated action is chosen as part of dialogue management (e.g., the food processor is turned on via the connected smart sensors). Finally, text-to-speech synthesis is used to produce and convey a confirming answer to the user.

Current interactions with the system are achieved by the front panel touch display. This display shows cooking information and helps guide the user through the cooking process. The food processing system has 140 recipes included and more can be downloaded. The system can then give step-by-step instructions displayed on the panel. The goal of this project is to rethink the user interaction process of the smart home appliance and research the viability of integrating alternative human-computer interaction into home appliances. The project stakeholders have stated four questions concerning rethinking the user interaction process of home appliances:

1. "Can we speed up the setting process by telling the appliance what we need and want?"
2. "How can we easily configure the user settings instead of reading manuals and complicated menus?"
3. "How can we find relevant functions quickly?"
4. "How can we interact with the UI manually and via voice command?"

The primary goal of this paper is to inform the community of researchers and practitioners of how we have conducted, evaluated, and present design science research. The output of this project is to produce a user interface (UI) concept and prototype with a focus on a multimodal design (traditional touch interactions and a new voice-based interaction). This prototype will include voice commands along with manual interaction.

This will have the benefit of offering alternative methods of interacting with the food processor, increase the speed at which the appliance takes to set up, ease configuration, reduce the need for traversing complicated menus, and reduce the need for reading product manuals. This will take the user to the relevant menu quickly and efficiently, and help the user reach their goal quickly allowing them to be up and cooking with little friction.

2. RESEARCH APPROACH

In this research, we adhere to the design-science paradigm (Hevner et al., 2004) which aims to push the boundaries of human and organizational capabilities by creating new and innovative artifacts. According to Hevner et al., 2004, design science paradigm is a problem-solving paradigm: Based on the identification of a problem, a potential solution is derived based on the combination of existing knowledge and new ideas. The potential solution is subsequently iteratively implemented and evaluated and, ultimately, contributes to and extends the current body of knowledge. Hevner et al., 2004, define seven ideas that can guide design science-driven research. We will briefly outline the seven ideas and mention how we implement those in our research.

1. Design Science Research aims at a real result (artifact). This can be a method, a model, or a product. The artifact designed must achieve a practical use. In our project we develop a speech interface for a home appliance.

2. Problem Relevance: The result must solve an existing and current practical problem. In our project, we examine how speech recognition and control can be attached to a device lacking this functionality.

3. Design evaluation: Whether the result solves the problem must be evaluated in a methodologically clean way. The literature provides numerous established methods that can be applied for this purpose - from user testing to case studies to controlled usability lab tests. In the paper at hand, we relied on user testing.

4. Research Contributions: In addition to the artifact solving the practice problem, there must also be a generally applicable contribution. In the paper at hand, we generate the lessons learnt from the use of the artifact in a particular industry or domain and generalize the artifact itself.

5. Research rigor: Correctly apply established methods from the literature in developing and evaluating the outcome. In our paper, we follow proven software development and testing methods.

6. Design as an optimization problem: With Design Science Research, the result is incrementally developed in search of the "perfect" solution that satisfies all constraints. While, due to capacity constraints, we could not strive for the "perfect" solution, we evaluated several solution problems and iterated towards the most effective solution.

7. Communication: The research contribution must be made accessible to the professional public. This is achieved by the publication at hand.

3. DESIGN OF AN ARTIFACT

3.1 Natural Language Processing (NLP) Background

Spoken dialogue systems are an example of an advanced application of spoken language technology. Dialogue systems represent an interface between the user and a computer-based application that allows for interaction with the application in a natural manner based on NLP (Cenek, 2005). NLP was one of the fundamental aspects of the project and one of the cornerstone pieces needed for project success. NLP is concerned with how human language is interpreted by a computer. NLP includes speech recognition (speech to text transcriptions), natural language understanding (the machine's reading comprehension), and natural language generation (turning data into natural, human-like language). Natural language generation was not included in this project. NLP is what makes up the conversational chatbot service at the center of this application.

Speech recognition is the process of transforming spoken language (in our case, recorded audio recorded after triggering the assistant service) into text. This text transcription is used by the later steps in the NLP process.

Natural Language Understanding (NLU) is inferring intent and comprehending what the input text is saying. Text is analyzed to determine things such as concepts, entities, keywords, relationships, and what the purpose of the text is (IBM, 2022). Effectively, NLU leads to a computer understanding what you have said.

These concepts are put into practice to produce any dialogue system where the applications that can perform tasks such as answering questions and can have conversations with humans. For example, a user could ask a dialogue system to update their address stored in the database. The dialogue system in response would be able to comprehend the request, ask follow-up questions such as, "What is your new address?", and forward the request to the appropriate services.

3.2 NLP Terminology

The following paragraphs provide an overview of relevant NLP terminology used in this paper.

Intent: What the user wants to do. For example, alter the screen's brightness or change the current system language.

Entities: Variables that are part of a request sent to the chatbot and extract from a sentence. For example, in the statement, "decrease screen brightness by 50", the entities that would be extracted are "decrease" and "50". These entities can be used by the system later, along with the current intent, "change screen brightness", to alter the screen brightness.

Utterances: These are the phrases the dialogue service is trained upon and uses to determine intent. They are different ways to say the same thing. For example, for the "change screen brightness" intent, the utterances could be "decrease screen brightness", "alter screen brightness", "dim the screen", "reduce brightness".

3.3 Artifact Design Steps

During the development phase of this project, many applications and options were investigated. We created a system that utilized all-online services, such as Google's APIs, but later transitioned the entire project to all-offline.

The application was developed in two components. One was the backend; the section that would handle the applications settings, supply the audio recording and playback features, support the Speech-to-Text (STT), NLU, and Text-to-Speech (TTS) processes, and query external APIs. The front-end was developed as a separate application to decouple it from the backend.

Communication between the two applications was formed through Socket.IO, a WebSocket technology used for real-time communication between a server and a client (Socket.IO, 2022). This allowed for two-way communication between the applications. When a setting was altered from the backend, it would emit an event to the frontend informing it of the change, and vice-versa.

The application supports switching between online-only services and offline-only services, as well as a hybrid of the two. For example, taking advantage of the accuracy of Google's STT allow with the speed provided from the offline NLU and TTS.

The backend was developed using TypeScript and NodeJS. Hotword detection provided by Snowboy, Speech-to-Text provided by Google's STT and Mozilla DeepSpeech, NLU/Chatbot services provided by Google's Dialogflow and NLP.js, Text-to-Speech provided by Google's TTS and Espeak, Authorisation service provided by ExpressJS.

The backend has been developed to ensure modularity. This ensures that all features can be replaced with alternatives. This helped ease the development process when transitioning from online to offline.

It displays a recreation of the pre-existing UI found on the food processor. The UI was created using TypeScript and VueJS 2, along with Vue-i18n (for internationalization support). Special attention was made to ensure the front end mimicked the existing UI in as much detail as possible. This includes animations, loading screens, page transitions, dropdown menus, and icons.

The UI was created using web technologies, as opposed to developing it with Qt (a UI framework for C++ currently used for the food processor), due to the speed at which we could develop the UI with VueJS. The development team did not have experience with C++ or the Qt framework, and since the speed of development was important, the development team decided to focus on what they knew as opposed to learning new technology.

Each screen, found on the Food Processor, was recreated on the UI concept, with focus made on the Recipes page and Settings page.

The UI introduced an "Assistant" pop-up, like the Siri animation on iPhones. This was provided by the graphics department at one of the participating organisations. When the user activates the assistant, either from the backend or frontend, an animated ring appears in the middle of the screen. This is to inform the user that the assistant is listening and provides feedback when the assistant has stopped listening. Figure 1 shows the completed UI screen examples.

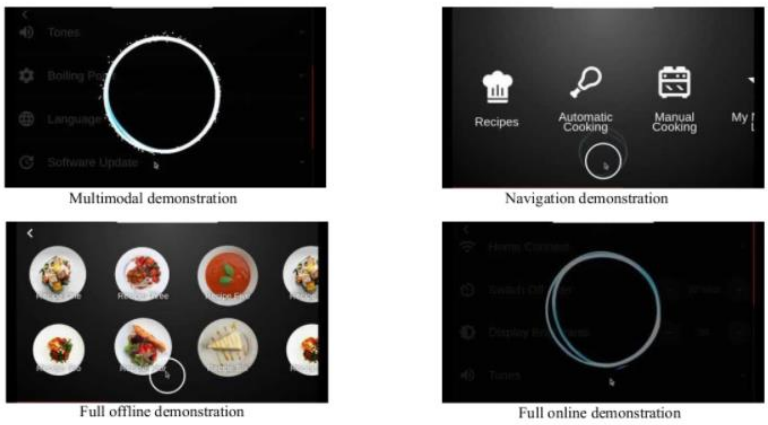

Figure 1. UI Screen examples

A minimal investigation was performed when developing the original, online-only, application. This was based on the recommendations of the industry supervisor. Since they had experience with online technology from previous projects, they recommended utilized the Google Cloud APIs. These included Google's Speech-to-Text, Google's Dialogflow, and Google's Text-to-Speech.

These three services provided the underlying chatbot services. The user would speak into a microphone, it would be recorded and streamed to the Google Speech-to-Text API. A response would be returned to what the user said. This transcription would be forwarded to Dialogflow. Dialogflow is an NLU service provided by Google. From here, an intent was inferred, entities extracted, and returned to the application. If Dialogflow required more information, it would trigger the Text-to-Speech service. This would turn a follow-up request from Dialogflow into an audio file that would be played to the user. This would prompt the user to continue speaking so Dialogflow could get all the information required. From here, the request would alter the system settings, update the current screen, or query the Home Connect service.

During the implementation of these processes, we discovered that remote users (the testers in Germany) needed to recreate the cloud infrastructure on their end, due to the security issues arising from transmitting the Google credentials files (a file used to ensure you are a valid user that can call that Google services. This file is unique to an organization and losing track of it could cause the particular organisation to infer increased costs of these cloud services). Sharing this file is considered "bad practice" (Steffres, 2018).

What was done to remedy this situation was to create "API-middlemen". API-middlemen are small servers that handle incoming requests, from a user, and make the request to a service on behalf of that user. The small server knows the credentials needed to access that service and can then return the results from that service. These small servers could be full, traditional servers hosted on a computer somewhere, or (in our case) cloud functions.

Cloud functions are a serverless framework that automatically runs the backend code when a request is made to it and deactivates the backend after a short period (Firebase, n.d.). This works like traditional servers but is designed for small, lightweight processes. This would be perfect for something that would just forward the user's requests and return results. We used Firebase Cloud Functions for these ephemeral API-middleman servers.

Dialogflow also supports a streamlined process of providing it with an audio file, it will be able to transcribe that into text, process it through the Dialogflow chatbot, and return an audio file in response. This process was implemented to speed up the online processes and reduce the need for querying three different online services. This increased performance substantially but couples the solution to one service. This is an option that can be picked if needed.

Before we transitioned to all-offline, we spent numerous weeks researching and testing different offline solutions. For the Speech-to-Text solutions, we investigated Mozilla's DeepSpeech, Kaldi, and PocketSphinx by Carnegie Mellon University. For the NLU service, we investigated NLP.js by AXA, and Botpress from Botpress, Inc. For the Text-to-Speech service, we investigated Espeak, Flite, Picotts, Nanotts, MaryTTS, and Mozilla Voice TTS.

For the Speech-to-Text, we investigated multiple options but decided upon Mozilla's DeepSpeech. This was chosen for its ease of installation, the large development community, ease of extension to support new languages and new language models, and its overall accuracy during testing. Mozilla's DeepSpeech has issues with the provided language models. These models are, mostly, trained on audio provided by American males in quiet environments (Morais, 2020). This will need to be considered if this service was to be used in future iterations of the application.

For the NLU service, NLP.js was chosen. This was picked because NLP.js was easy to install, open-source, extensive documentation, lightweight and performant for low-computational environments (such as a Raspberry Pi), easy to use and implement new intents and entities, and come with an optional UI that could be added in, if needed. The benefit of having the UI be an optional extra meant it wouldn't take up resources in an already constrained environment. The offline NLU service performed similarly to Google's Dialogflow and worked exceptionally well for our use case.

For the Text-to-Speech, Espeak was chosen. Espeak was picked for its ease of installation and low computational overhead. Espeak is not as human as the online version (Google's Text-to-Speech), but most on-device options were poor quality to reduce the computational power required. Mozilla's TTS was investigated but due to the lack of documentation, it was discarded.

During the development of this application, we contributed to three open-source repositories. These changes have been merged into the master codebase or waiting for pull-request to be integrated.

Improvements were made to the "Dialogflow-NLP-to-nlpjs", an application to convert a Dialogflow chatbot into a format that can be read by NLP.js. Improvements to defensive checks of input data ensured any language could be selected and ensured that asynchronous code was called correctly.

Additional code was added to "bumblebee-hotword-node", a NodeJS package used to listen to hotwords uttered by a user. Changes were made to ensure an audio device could be selected for Unix operating systems (Leslie, ncpleslie/bumblebee-hotword-node, 2020).

Further contributions to the open-source community were made with "node-speaker", a NodeJS package for playing audio through a hardware speaker. Contributions were made to their TypeScript declarations to ensure all features were supported when in a TypeScript environment.

The extensive focus was made to ensure all services support both English and German languages, at a minimum.

The assistant and backend processes, including STT, NLU, TTS, can support multiple languages. The total number of languages each service can support is discussed in Findings and Recommendations. The bottleneck for language support will be the NLU process. This is due to having to create custom chatbot intents, entities, and responses for each language. The STT and TTS will support a large assortment of languages out of the box or will require updating their models to support more languages. Adding new languages to the NLU chatbot service is documented in the guides provided with the source code and is as simple as adding new intents to the chatbot service.

Frontend internationalization currently supports only the German and English languages. This is facilitated by provided language files along with the UI. The process of adding new languages is documented in the documentation provided with the code. Switching the displayed language can be triggered by tapping the preferred language in the settings menu, or through the voice assistant by saying, "Change language". This will refresh the UI and display the newly selected language.

4. FINDINGS AND RECOMMENDATIONS

Hotword recognition is a low-power, low-cost process. The findings of the hotword detection services we investigate (Snowboy and Porcupine) were both fast enough to provide instant activation. The major differences between the two services were:

Snowboy offered custom hotword support but must be manually trained, with large datasets (500 instances for English, 2000 instances for German (Snowboy, 2022). That means that the custom wake words will require voice recordings of the hundreds of different people in multiple sound environments. This is fine for personal projects, but for commercial products, this could be expensive and bothersome. Snowboy can run on many architectures. We developed with Snowboy on an ARM processor, but it also tested on an x86 processor.

Porcupine offers out-of-the-box universal hotwords (will support many user's voices). These phrases include "bumblebee", "grasshopper", "porcupine", and "hey, Edison", and will work on most device architectures (ARM and x86). This will work well for personal projects but not excellent for commercial products. Porcupine does offer custom wake words, but due to the costs involved with training custom hotwords for ARM processors, their custom wake word service was unable to be tested (Picovoice, 2022). This option could be explored if a commercial product is to be released.

We investigated and implemented two speech-to-text options, Google's STT and Mozilla's DeepSpeech. We found that both were able to perform the task of speech-to-text on the Raspberry Pi. The quality of their transcriptions was vastly different.

Mozilla DeepSpeech performed the worst of the two, with the accuracy of transcriptions far below Google's offering. During testing, phrases were repeated to the device and DeepSpeech would detect differing statements, even though the same sentence was said. When the phrase, "Decrease screen brightness by 50" was uttered to the device, the results returned would include phrases such as, "Dennis, screen brighter bye thief tee". This is most likely caused by the voice models being trained on American male voices and lacking support for New Zealand accents. The DeepSpeech models can be extended and trained if additional voice data is provided. Considerations will have to be made for each new language and country Bosch wished to ship the product. The English voice model is over 1 gigabyte in size (Steffres, 2018).

Google's Speech-to-Text service provided excellent transcription will minimal computation required on the device. Google's STT service can provide transcriptions in 71 different languages and 127 different locales (Barnes, 2020), including New Zealand English, Indian English. This will reduce the need to train custom models or rely on the open-source community to train them for you.

Considerations will need to be made with data privacy. Since DeepSpeech is all performed on-device, data is kept on the device. Google's STT services do log data, but "no logging" and on-premises options are available where data privacy is required (Google Cloud, n.d.).

Our recommendation is to use an external API, such as Google. It will provide a well-trained conversational API, will support multiple languages, will use low computational power, and provide accuracy of their service. But an "on-prem" solution could be considered if data privacy is paramount.

We implemented two NLU services in this application, Google's Dialogflow and AXA's NLP.js. In terms of the accuracy of intent detection, we found similar results between the two.

The major differences between the two come down to how you extend the functionality of the NLU services. Since Google's Dialogflow is an online service, extending it can be simply down through the browser. This has the benefit of updating the service for all users using the service. NLP.js will need to be provided with an update to support new intents. New functionality for NLP.js can be included by updating a JSON file included with the source code.

Google's Dialogflow can support all the processes required to make a functioning chatbot service. This includes Speech-to-Text, NLU, and Text-to-Speech. This will reduce the processing times but couple the chatbot service to one solution.

Our recommendations are to pick whichever option is the most convenient. Both options perform exceptionally well, require low computation overhead, and can be upgraded relatively simply. Considerations will need to be made with Google's services as data logging will occur.

During development, we investigated multiple Text-to-Speech options. These included Google's Text-to-Speech, Mozilla's TTS, Espeak, Flite, Picotts, Nanotts, MaryTTS. We concluded that all on-device offerings we of much lower quality (less human-sounding) than those options offered by Google's Text-to-Speech.

Google's TTS was fast and high quality, with life-like voices (Google Cloud, 2022). It also supports more languages than any of the other services we investigated. Google supports over 40 languages and 220 different voices.

Of the on-device TTS options that were investigated: Mozilla TTS, Espeak, Flite, Picotts, Nanotts, MaryTTS, one was difficult to install (Mozilla) and the rest were low quality and robotic sounding. The research we made determined that high-quality, humanlike TTS would require large computational power. Some research said that the Raspberry Pi could be six times slower than real-time (Stoker, 2020), well beyond any practical use.

Although Speech-to-Text is not a large component of the application, only being required when follow-up information is required or providing feedback on the current cooking process, it is still part of the process to give the user a pleasant experience. If high-quality is required, we recommended using Google's STT services.

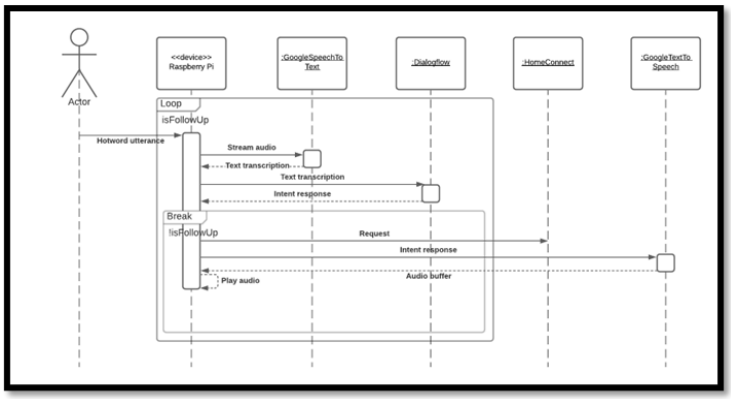

Figure 2. Sequence of the assistant process and how it updates the UI

Although everything can be performed on the device, these details mentioned previously should be considered. To continue further with our recommendations, considerations should be made, when calling external APIs (Google Cloud, etc), for latency. During development, we found that calling these APIs would introduce delays in response. Some queries could take upwards of five seconds to complete. This was due to how we called these external services. Each external query would be sent back to the Raspberry Pi before being sent off to the next external service. This introduced some latency. Hosting the entire application in the cloud could reduce this latency. This would reduce the need of having API middlemen, reduce latency since it will share tenancy with the external APIs it needs, and reduce the need to provide updates to the device should you want to extend the functionality of the service. A simple API call could be made on the device and return a response. These services we investigated, such as STT and TTS, require some computational power. The Raspberry Pi could keep up with them, but research showing it is slower than the cloud offerings provided by Google. If performing these services all on-device is required, this could increase the cost of the hardware shipped with every Food Processor. Figure 2 represents a Sequence diagram of the current system's flow of a general request, from hotword utterance, updating the Home Connect API and responding to the user through the speaker.

5. CONCLUSION

This study investigated how to rethink the user interaction process of smart home appliances and examined the potential of incorporating alternative human-computer interfaces into these devices. The results showed that the implementation of speech-based interaction provided a more efficient and alternative way of interacting with home appliances. The research also looked at two hotword recognition services, Snowboy and Porcupine, and two speech-to-text options, Google's STT and Mozilla's DeepSpeech. Based on the results, it was suggested that for conversational API services, external API solutions such as Google's should be used to handle numerous languages, offer accuracy, and require less processing capacity. Finally, NLU service implementations such as Google Dialogflow and AXA's NLP.js can be extended to provide additional functionality. Overall, the study presents a novel user interaction process for smart home appliances that can provide an efficient and user-friendly experience.

ACKNOWLEDGEMENT

The authors acknowledge the highly effective collaboration between the two industry partners and the two academic institutions. We also acknowledge the use of ChatGPT and DeepL Write for improving the fluency and clarity of the text regarding language.

REFERENCES

Barnes, C., 2020. *Enhanced Models And Features Now Available in New Languages on Speech-to-Text*. Retrieved from https://cloud.google.com/ blog/products/ai-machine-learning/new-features-models-and-languages-for-speech-to-text.

Cenek, P., 2005. *A Framework for Rapid Multimodal Application Design*. In *Text, Speech and Dialogue*, Hutchison, d. et al. (Eds.). Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 393–403. DOI: https://doi.org/10.1007/11551874_51.

Fernandes, T. and Oliveira, E., 2021. *Understanding Consumers' Acceptance of Automated Technologies in Service Encounters. Drivers of Digital Voice Assistants Adoption*. Journal of Business Research, Vol. 122, pp. 180–191. DOI: https://doi.org/10.1016/j.jbusres.2020.08.058.

Google Cloud, 2022. *Text-to-Speech*. Retrieved from https://cloud.google.com/text-to-speech.

Hevner, A. et al., 2004. *Design Science in Information Systems Research*. MIS Quarterly, Vol. 28, No. 1, pp. 75–105.

IBM, 2022. *Watson Natural Language Understanding*. Retrieved from https://www.ibm.com/cloud/watson-natural-language-understanding.

Maedche, A. et al., 2019. *AI-Based Digital Assistants*. Business & Information Systems Engineering, Vol. 61, No. 4, pp. 535–544. DOI: https://doi.org/10.1007/s12599-019-00600-8.

Morais, R., 2020. *DeepSpeech 0.9.0*. Retrieved from https://github.com/mozilla/DeepSpeech/releases/tag/v0.9.0.

Picovoice, 2022. *Procupine*. Retrieved from https://github.com/Picovoice/porcupine.

Scates, K., 2020. *How Mercedes Benz's MBUX Voice Control Revolutionizes the User Experience*. Retrieved from https://www.soundhound.com/voice-ai-blog/how-mercedes-benzs-mbux-voice-control-revolutionizes-the-user-experience/.

Snowboy, 2022. *Snowboy Hotword Detection - Hey Bosch*. Retrieved from https://snowboy.kitt.ai/hotword/54265.

Socket.IO, 2022. *What Socket.IO is*. Retrieved from https://socket.io/docs/v4/.

Steffres, 2018. *Best Practice to Share Google Drive API Credentials for Being Used by a Script*. Retrieved from https://security.stackexchange.com/ questions/186329/best-practice-to-share-google-drive-api-credentials-for-being-used-by-a-script.

Stoker, N., 2020. *Installing Mozilla TTS on a Raspberry Pi 4*. Retrieved from https://levelup.gitconnected.com/installing-mozilla-tts-on-a-raspberry-pi-4-e6af16459ab9.