SMART HOME AUTOMATION WITH WAKE WORD DETECTION USING ESP32

GAN PEI YI

UNIVERSITI TUNKU ABDUL RAHMAN

SMART HOME AUTOMATION WITH WAKE WORD DETECTION USING ESP32

GAN PEI YI

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Electronics Engineering with Honours

Faculty of Engineering and Green Technology
Universiti Tunku Abdul Rahman

September 2025

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature	:	[illegible]
Name	:	GAN PEI YI
ID No.	:	20AGB04908
Date	:	

APPROVAL FOR SUBMISSION

I certify that this project report entitled **“SMART HOME AUTOMATION WITH WAKE WORD DETECTION USING ESP32”** was prepared by **GAN PEI YI** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Electronics Engineering with Honours at Universiti Tunku Abdul Rahman.

Approved by,

Signature	:	[illegible]
Supervisor	:	Ts Dr Nuraidayani Binti Effendy
Date	:	8/10/2025

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Dr. Nuraidayani Binti Effendy for her invaluable advice, guidance and her enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me encouragement.

SMART HOME AUTOMATION WITH WAKE WORD DETECTION USING ESP32

ABSTRACT

The limitations of traditional smart home automation systems that rely on costly, commercial voice assistants, which often lack customization and raise privacy concerns. The project aims to develop a low-cost, user-friendly smart home automation system that enhances privacy and flexibility to control the lights. The method involved in this project is related to the continuous audio sampling via a microphone connected to the ESP32 microcontroller, with on-device wake word detection using TensorFlow Lite to recognize the trigger word "Marvin" locally, which reduces the reliance on cloud services. After wake word detection, voice commands are transmitted to the Wit.ai cloud service for natural language understanding that enables the system to interpret user intents and control commands. ESP32 then executes the user's commands by managing the connected LED through the GPIO pins. Experiments evaluation showed that the wake word model had a high success rate in intent identification using the NLP service, minimal latency, and over 90% accuracy in controlled environments. This technique effectively protects users' privacy by reducing the reliance on cloud services and cutting the expenses by utilizing affordable hardware. In comparison to traditional voice assistance, the project shows significant enhancements in system responsiveness and user convenience. At the end of this research, a flexible yet affordable and privacy-conscious voice assistance solution is designed and built. This system offers consumer more control over their smart environment. Future work may explore expanding device compatibility and refining natural language processing to enhance system robustness further.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

2D	Two-Dimensional.
API	Application Programming Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
dB	decibel
dBFS	decibels relative to full scale
DCT	Discrete Cosine Transform
DIY	do-it-yourself
DNN	Deep Neural Network
GND	Ground
GPIO	General Purpose Input/Output
GPIO	General-Purpose Input/Output
H5	content created with HTML5
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation, and Air Conditioning
Hz	hertz
I2S	Inter-IC Sound
IoT	Internet of Thing
JSON	JavaScript Object Notation
LED	Light-Emitting Diode
LSTM	Long Short-Term Memory
MEMS	Micro-Electro-Mechanical System
MFCC	Mel-Frequency Cepstral Coefficients
ML	Machine Learning
NLP	Natural Language Processing
PC	Personal Computer
RAM	Random-Access Memory

RNN	Recurrent Neural Network
ROM	Random-Only Memory
SBC	Single-Board Computer
SNR	Signal-to-Noise Ratio
SRAM	Static Random Access Memory
SSID	Service Set Identifier
TLS	Transport Layer Security
USB	Universal Serial Bus
VSCode	Visual Studio Code
Wi-Fi	Wireless Fidelity

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background Study of Smart Home Automation

Technological advancement is rapidly growing. The integration of smart devices and systems in human life has become omnipresent. Smart home technology has completely changed the way humans interact with daily living spaces, with a variety of automation and wireless control options. In layman's terms, an intelligent home system connects different appliances and gadgets via a network that allows users to remotely control and monitor them through voice commands, smartphone, or other communication methods. This seamless communication between devices enhanced the security and convenience of daily life.

The idea of automating household chores has been discussed for some time. From the early concepts that were frequently illustrated in science fiction, imagine a future where smart systems aid human beings in handling everyday chores. This goal took years to become reality, but the desire for productivity and simplicity has always been the motivating factor of designing. In the very beginning of creation, home automation solutions tend to be complicated, costly, and limited in scope. This system used wired configurations with centralized panels to manage basic operations like temperature or lighting (Chen, Zhang and Zhong, 2024). These early systems paved the way for today's more developed and user-friendly technologies.

The Internet of Things (IoT) significantly improved smart home automation, making residences more secure, convenient, and energy-efficient (Kodali *et al.*, 2016). An IoT-based home system is built to improve user interaction by utilizing voice and gesture commands. The device control is easier and more accessible with voice and gesture control, especially for those who have disabilities. User can use smartphones to control their home environment. The most typical components of a smart home framework are smart devices, sensors, connectivity, a control platform, a user interface, and automation rules.

1.2 Importance of the Study

The way individuals interact with their living spaces has changed due to the rapid advancement of smart home technologies. Voice-controlled smart home systems have become an important part of modern households. This feature can bring simplicity, effectiveness, and enhanced security to users. Advanced wake word detection technology provides the chance for users to interact with smart devices more naturally and hands-free.

The study is important because it covers a wide range of smart home technology. First of all, the study prioritizes ease of use and accessibility. The ability to control lighting and other smart home devices using voice commands eliminates the need for physical switches. Both the elderly and people with impairments may gain benefits from these features of the home system (Stefanović, Nan and Radin, 2017). The user experience can be maintained more effectively and easily by employing a straightforward wake word to activate the system.

In addition, the research has made significant contributions to the system to make it more energy efficient. By controlling lighting and HVAC functions in real-time, these can reduce energy consumption in a well-designed home automation system. According to Singh *et al.* (2024), home automation can lead to substantial energy savings when devices are only operated when needed. The home system can also handle devices based on the occupancy or the usage trends.

The ESP32 microcontroller has the advantages of low cost, low power consumption, and a built-in Wi-Fi/Bluetooth connection. Thus, ESP32 is a great choice for an IoT-based system (Litayem, 2024). By using ESP32 as the primary controller in a system, the study illustrates that a smart home system can be economical and effective. This can be important in the home automation field, where users are often hindered by high cost and complex configurations.

An extra level of security and user control is offered by the system. It is said so as the system is integrated with wake word detection as a triggering mechanism. When smart home technology became more widespread, the security concern grew. The primary security issues that emerge are issues like unexpected device activation and unauthorized access to personal information. According to Meng *et al.* (2018), the usage of wake word detection helps to reduce the risks of cyberattacks. The wake word detection system ensures that only authorized users can activate it. The study dives into applying a customized wake word to prevent unintentional or malicious commands from being performed. Thus, it adds a level of security to the smart home systems.

Overall, the importance of the project is to be capable of adding a voice-controlled system to implement in the field of smart home automation. Offering smart solutions that are not only technologically advanced but also economical and practical for daily application. The study focuses on ESP32 as a microcontroller and wake word detection to construct a low-cost solution for a smart home environment. It can boost the quality of human life, and also energy efficiency and security provided.

1.3 Problem Statement

The development of modern civilization grows rapidly, daily activities of individuals become more hectic. The demand for a more effective and more efficient way to handle daily tasks with technology is increasing (Canziani and MacSween, 2021). The traditional methods of operating smart devices that frequently involve manual input can be troublesome and time-consuming to consumers who prefer a seamless, hands-free experience (Irugalbandara *et al.*, 2023). The customer interactions can be more direct by implementing voice assistance, such as Amazon Alexa. Nevertheless, most of the commercial solutions are either proprietary, costly, and offer limited choices for customization (Ghafurian, Ellard and Dautenhahn, 2023).

A DIY voice assistant can be designed to overcome the difficulties by combining an open-source software like Wit.ai for natural language processing and TensorFlow Lite for wake word recognition with inexpensive hardware like an ESP32 microcontroller. The system allows users to automate processes and operate devices by using basic voice commands without depending on proprietary platforms. By incorporating such options, customer gains more privacy, convenience, and assistance to assist in their needs. In a rapidly developing digital world, the do-it-yourself method offers individual to build a customizable, expandable, and affordable voice assistance that improves their daily lives.

1.4 Aim and Objectives

This project aims to develop a customizable and efficient smart home voice assistance system using the ESP32 microcontroller and TensorFlow Lite for on-device processing, where simple voice commands by the user can be done. The objectives of the project include:

i. To design and develop a smart home automation system using an ESP32 microcontroller capable of reliable wake word detection.
ii. To integrate TensorFlow Lite on the ESP32 for efficient on-device wake word detection and voice command processing.
iii. To implement natural language understanding for seamless voice-controlled operation of home lighting.

1.5 Scope and Limitation of the Study

This project involves designing, implementing, and evaluating a smart home automation system with wake-up voice detection using an ESP32 microcontroller, TensorFlow Lite for on-device processing, and a custom wake word, namely “Marvin”. The project focuses on voice-controlled activation and basic smart home device control. The automation system is designed to be a low-cost and customizable alternative to commercial voice assistance. However, the project is limited by not addressing advanced natural language processing (NLP) beyond basic commands, the lack of integration with a wide range of smart home environments, and the absence of robust security features. Scalability testing for a large number of devices or complex home automation will also be limited due to memory constraints.

1.6 Contribution of the Study

This study makes several key contributions to the field of smart home automation, specifically in the area of voice-controlled systems and wake word detection. The two most significant contributions are as follows:

i. The development of an affordable, privacy-focused smart home system.
   This study demonstrated that the integration of a low-cost ESP32 microcontroller coupled with TensorFlow Lite for on-device processing can be used to create a private and voice-controlled home automation system. By processing the wake word locally, the system eliminates the need for cloud-based services that offer a lower memory consumption on the microcontroller and a more secure home experience.
ii. The integration of wake word detection for hands-free control.
   The research contributed to the development of an efficient wake word detection on a resource-constrained microcontroller. This facilitates a smooth hands-free experience by using natural language voice commands to control lighting. The system provides a more affordable solution compared to a proprietary system. At the same time, it enhances the accessibility and adaptability of the smart home technology.

1.7 Outline of the Report

The purpose of this thesis is to provide a comprehensive overview of the completed project and the results achieved. The subsequent paragraphs are the overview for each chapter.

i. CHAPTER 1 provides a general summary of the project, which includes the project's aim, objectives, and approach. This chapter also outlines the structure of the thesis and the research methodology, scope, and limitations.
ii. CHAPTER 2 discusses the literature on smart home automation, voice assistance technology, ESP32 microcontroller, wake-up detection, and TensorFlow Lite applications in an embedded system. It gives background information for the current study and draws attention to the limitations of the system.
iii. CHAPTER 3 describes the method used to build and implement the smart home automation system. In this chapter, the system architecture, hardware configuration, software development, and smart home device control implementation are covered. Additionally, it provides a glimpse into the techniques employed for validation and testing.
iv. CHAPTER 4 illustrates the results of the project, followed by the interpretation of the results obtained. This includes a review of the system responsiveness, speech recognition accuracy, and smart home device control efficacy.
v. CHAPTER 5 concludes this thesis by providing a summary of the results of this project and suggestions for further research and possible system enhancements.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview of Smart Home Automation

The ESP32 has been widely used as the primary controller in recent smart home automation advancements to improve energy efficiency, convenience, and security. According to Syari, Dzaky and Saragih (2025), systems that use ESP32 to handle motion sensors and cameras in real-time have reported detection accuracies of approximately 92%. These developments paved the way for voice control to be implemented in smart surroundings to enable more natural interaction.

In contrast to always-on listening systems, voice control studies highlight wake word detection as a crucial element that enables devices to stay in low-power idle modes until activated, which enhances privacy and saves energy. Systematic wake word models can be used on peripheral devices with lightweight machine learning frameworks like TensorFlow Lite. These methods yield an acceptable accuracy and a low latency under 400ms, according to Kundu *et al.* (2022)

An ESP32-based smart home automation system with wake word detection is intended to establish a voice-activated environment. By taking advantage of the ESP32 capabilities and the dual-core architecture, this solution can significantly increase user comfort and accessibility. This enables voice commands for hands-free control of household equipment (Babiuch *et al.*, 2020).

In contrast to the conventional home automation systems, which rely on manual switches, this technique uses a lightweight machine learning (ML) model that is achievable with TensorFlow Lite. The ML model integrated wake-up word detection directly on the ESP32 (Rashidi, 2022). The device continuously listens for a specific trigger phrase, "Marvin", and only when the wake word is recognized, further processing or initiating communication with cloud-based natural language understanding services, Wit.ai, is activated (Iliev and Ilieva, 2023). This not only saves power but also improves privacy because the system remains idle until the user activates it (Froiz-Míguez, Fraga-Lamas and Fernández-CaraméS, 2023).

2.1.1 Smart Home Control Logic

A smart home control system aims to provide convenience and energy efficiency in automation and remote control of household appliances. Chakraborty *et al.* (2023) highlighted the integrations of technology in living environments to achieve a safer, comfortable, and more secure environment. Besides that, integrating smart devices into home devices can achieve higher energy efficiency. The study emphasized that smart devices could reduce the overall electricity consumption and thus lower the cost and offer better security for the home system.

Another study by Elkholy *et al.* (2022) discussed regarding the design and implementation of a real-time smart home management system that mainly focuses on energy conservation. The paper research on the management system allows users to control their home remotely and locally using smartphones and PCs. Additionally, the logic behind smart home control has developed to incorporate energy-conscious automation, such as occupancy detection and HVAC or lighting adjustments based on environmental factors. The features enhance the user’s life quality and also improve the energy efficiency (Elkholy *et al.*, 2022; Chakraborty *et al.*, 2023).

2.2 Microcontroller

In Internet of Things (IoT) applications, microcontrollers serve as the central processing unit of a system. There are several functions which is to interface with sensors, process input, and communicate with cloud-based services. Various microcontrollers can be integrated with the smart home system. Each of the microcontrollers has different specifications depending on application requirements.

Arduino Uno and Arduino Mega have been utilized for a long time in embedded systems due to the ease of use and vast developer community. However, to design a wireless communication system, Arduino is not a good choice as it does not have built-in Wi-Fi and Bluetooth modules. To implement wireless communication in home automation, additional modules are needed. Moreover, standard Arduino boards have limited processing power. Thus, it is not suitable for tasks involving real-time signal processing and voice recognition.

Raspberry Pi is a single-board computer (SBC) that offers higher processing power and is also supported with a full operating system. According to Goyal (2024), Raspberry Pi can run complex applications like speech recognition and computer vision. Even though Raspberry Pi has good computational performance, it consumes more power than a standard microcontroller. This feature is less appropriate for use in a smart home system that runs on a battery.

ESP32 offers a compact and affordable alternative compared to Arduino, which needs extra modules for Wi-Fi and Bluetooth (Yüksel, 2020). ESP32 offers better internet access with an integrated wireless port, contrasting with basic microcontrollers like Arduino Uno. ESP32 still consumes less power and is smaller than Raspberry Pi and other single-board computers (SBCs). For speech-based controls, the model can effectively function within the memory and latency constraints of ESP32. This combines with a lightweight CNN with MFCC feature extraction (Badade *et al.*, 2025).

According to Yüksel (2020), ESP32 is the most popular microcontroller for IoT. It is due to the low cost and low power consumption with integrated Bluetooth and Wi-

Fi. ESP32 stands out for its high processing capabilities and low cost compared with Arduino and Raspberry Pi. These properties make the ESP32 a good choice for home automation projects (Maier, Sharp and Vagapov, 2017).

2.3 Machine Learning

Voice recognition and smart home automation are just two of the many applications where machine learning (ML) is becoming more prevalent. TensorFlow Lite and other lightweight machine learning frameworks have become popular in executing models on embedded systems. It offers real-time performance, low power, and high efficiency (Warden, 2018). Instead of relying mostly on cloud computing, the characteristics allow wake word detection to be carried out locally on a device with limited resources.

TensorFlow Lite is a common framework for implementing machine learning techniques into practice in wake word detection problems. This is due to its excellent performance for low-power devices and lightweight design with good efficiency. According to Chittepu, Martha and Banik (2025), edge-based segmentation protects the user's privacy by processing the data locally while maintaining system responsiveness. This is crucial for user acceptance and experience.

2.3.1 Types of Neural Networks

In audio classification, Convolutional Neural Networks (CNNs) are commonly used, especially in wake word identification. CNN can extract spatial and temporal patterns from spectrograms and audio data. It can effectively learn the local time-frequency characteristics from the Mel-Frequency Cepstral Coefficients (MFCC) inputs. This characteristic is suitable for low-resource devices like ESP32 (Arik *et al.*, 2017). The lightweight design architecture with balanced precision and computing cost enables real-time wake word detection with low latency.

Other methods have been used for voice recognition in smart home systems in addition to Convolutional Neural Networks (CNNs). The Recurrent neural networks (RNNs), in particular, Long Short-Term Memory (LSTM) networks, are ideal for reproducing the temporal relationships in continuous speech signals (Saputra *et al.*, 2024). This enhanced the accuracy of processing sequential audio data, which includes speech recognition and wake word detection. The networks can identify speech in a noisy environment or in different speaking styles because they can capture long-term contextual information.

Additionally, transformer-based architectures and hybrid Deep Neural Networks (DNNs) that combine CNNs and RNNs have demonstrated potential in reaching cutting-edge performance. The method is through the usage of attention mechanisms for improved noise robustness and feature learning of the systems (Natarajan *et al.*, 2025). Techniques for quantization and model compression are necessary to implement such sophisticated models with limited hardware (Xu *et al.*, 2025).

CNN models are still being optimized for situations with limited resources to recent research. It uses methods like quantization and pruning to cut down the model size and inference time without excessively sacrificing the accuracy. The development has motivated more responsive and private user interactions by boosting the usage of CNN-based wake word engines in edge devices (Zhang *et al.*, 2018).

2.3.2 Feature Extraction Method

There are many alternatives for feature extraction that have been proposed in audio processing. Gammatone Frequency Cepstral Coefficients (GFCCs) are one of the methods used for audio feature extraction. GFCC is designed to be more robust to noise compared to MFCC. GFCC models the auditory periphery more accurately using the gammatone filter banks. According to Dua, Aggarwal and Biswas (2019), the performance of GFCC is better than PLP, MFCC, and hybrid MF-PLP techniques under a noisy condition.

Besides that, another method, which involves wavelet transforms, provides a high time-frequency resolution, and it can also adapt to non-stationary signals. A study by Ali *et al.* (2014) compared the DWT-based features against MFCC recognition for Urdu words. This paper comes to the conclusion that wavelet-based feature performs better when signal noise is present. Bajaj *et al.* (2022) had made a contrast between wavelet and MFCC for emotion recognition and showed that the wavelet method can capture more details in a variable time-frequency localization setting.

On the other hand, the work by Kamińska, Sapiński and Gholamreza (2017) on cepstral and perceptual linear prediction method compares MFCC, PLP, PLC, and perceptually motivated variants under the emotion recognition function. They found that the MFCC technique of feature extraction performed well, which also provides some improvement under particular conditions. In order to improve durability against noise and unpredictability, MFCCs are created using spectral analysis and a mel-scale filter bank that highlights frequencies.

MFCC is chosen for this project as a method for audio feature extraction. This is due to MFCC being widely used and well-developed in the audio processing sector. Besides that, MFCC captures perceptually relevant spectral information via the mel-scale filter bank and cepstral coefficient (Logan, 2000). These parameters tend to map well with the human hearing system. MFCC also has a computationally efficient method compared to the more complex methods, such as the wavelet.

Recent research shows that the longer frame lengths between 200 and 300 ms enhanced the noise robustness. MFCC collects more stable audio patterns without excessively increasing the latency (Zhantleuova, Makashev and Duzbayev, 2025). In a normal household, background noise, the wake word detection became more accurate and dependable when the parameters are adjusted to the distinguishable condition of an environment.

2.4 Natural Language Processing

Natural language processing (NLP) allows the systems to understand and interpret human spoken words. In voice-controlled smart home systems, NLP extracts the intents and entities from user speech, then transforms the spoken commands to perform tasks (Naved *et al.*, no date). When NLP is combined with wake word detection systems, human-device interactions become smoother and natural. Additionally, devices can stay in idle mode until the wake word activates them. This allows the system to save power and protect user privacy.

Depending on the capabilities of the device, NLP implementations have changed over time. Lightweight on-device natural language processing models are designed for resource-constrained devices. It optimized the latency and limited the processing power. However, cloud-based interpretation offers more advanced language understanding with little on-device processing. But this method is at the cost of internet reliance and may have privacy issues (Kang *et al.*, 2014).

2.4.1 Communication Services

Cloud-based NLP systems, hybrid and local processing models are examples of communication services. These models prioritize privacy, latency, and dependability (Schweidtmann, Zhang and von Stosch, 2024). The hybrid model is typically dedicated to advanced natural language understanding on cloud servers. This is to reduce the network congestion and enhance the system responsiveness. The wake word detection and basic command processing are carried out locally on the ESP32 simultaneously (Ezugwu *et al.*, 2025). Lightweight local wake word engines and protocol is utilized to communicate with the cloud server. This approach balanced out the computational constraint and functionality of the system (Ezugwu *et al.*, 2025).

Locally processed NLP systems have also been developed, which execute the whole voice processing pipeline on embedded hardware. This technique improved the system’s privacy and removed the need for internet connectivity. However, this approach often comes at the price of the model's accuracy and linguistic complexity (Shastry and Shastry, 2023). These communication techniques cater to various requirements of the smart home settings, where elements like operational continuity, user experience, and data security are crucial (Lamaakal *et al.*, 2025).

Cloud-based NLP technologies have made it much easier to develop voice-controlled applications. Wit.ai, which is a cloud-based NLP platform, has been chosen for this project. Without having to create and manage complex machine learning models, developers can access robust speech-to-text capability through the services (Opara, 2022). These platforms remotely recognize intent and interpret natural language by processing text or voice data sent over secure Wi-Fi connections from devices like ESP32. While cloud NLP services enable high accuracy and short development cycles, issues about latency, data privacy, and internet stability persist (Vlist, Helmond and Ferrari, 2024).

2.4.2 NLP Services

By utilizing strong remote servers, cloud-based NLP technologies like Wit.ai by Meta, Google Cloud Speech-to-Text, and Amazon Alexa Voice Service offer sophisticated natural language understanding capabilities. These services greatly simplify the creation of smart home apps by enabling contextual comprehension, advanced language processing, and support for a variety of commands.

Google Cloud Speech-to-Text is excellent at accurately transcribing a wide range of languages and dialects, and the models are optimized using Google's extensive data sets. The primary function is speech-to-text conversion. However, developers must incorporate their own layers for intent categorization and Natural Language Understanding (NLU). This might raise on system complexity and latency (Xu *et al.*, 2021). Additionally, transcription accuracy decreases in loud environments or with a poor internet connection (Xu *et al.*, 2021). Since the services are at a cost, the increase in usage might increase the expenses. expenses may increase with increased usage.

Amazon Alexa Voice Service (AVS) offers a more comprehensive solution by integrating dialogue management, NLU, and Automatic Speech Recognition. The Alexa platform can directly communicate with the home automation. As Alexa Voice Service has pre-trained models that are optimized for common smart home activities. This makes it simpler to implement voice-controlled lighting, thermostats, or appliances with little requirement for specialized training. For developers who need modification beyond the Alexa-supported commands may be constrictive. Furthermore, due to the security and compatibility constraints, incorporating Alexa Voice Service onto non-Amazon hardware (such as ESP32-based devices) can be more difficult.

Meta's Wit.ai offers a more developer-friendly balance. It provides a comparatively straightforward REST API-based integration that enables the definition of custom intents and entities. It also accepts both text and voice input. Researchers have demonstrated that Wit.ai can provide good performance with less development overhead in chatbot and educational contexts. For instance, Qaffas (2019) found that using Wit.ai in conjunction with a Word Sequence Kernel improved chatbot semantics in an education domain. Wit.ai had exceeded the capabilities of traditional messenger chatbots. Wit.ai is used for FAQ chatbots in another study by Bagus *et al.* (2021), which also demonstrates the usefulness of the platform for dialogue systems with intermediate complexity. However, Wit.ai still relies on cloud infrastructure, so it needs a dependable internet, bandwidth, and data security.

CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter outlines the methodology of developing a Smart Home Lighting Control System using an ESP32 microcontroller. The system was integrated with wake-up word detection for voice-controlled lighting. The research makes use of the combination of experimental design and functional testing to fulfil the objectives.

3.1.1 Overview of Methodology

The procedure starts with the development and setup of the hardware components. ESP32 is configured to control the lighting in various zones using LEDs. Wi-Fi connection is used to enable communication with the cloud-based NLP service.

Next, lightweight machine learning models are developed with TensorFlow Lite. This model is to train for wake word detection and is deployed on ESP32. Mel-Frequency Cepstral Coefficients (MFCCs) are utilized to extract features from audio input. The MFCCs are then used as the input for a Convolutional Neural Network (CNN) model to recognize the specified wake word "Marvin".

The system captures the user's voice after “Marvin” is detected. ESP32 sends a request via HTTP to connect with the Wit.ai server. This process is to extract the user’s intent using a cloud-based natural language platform. The interpreted intent is then used to trigger specific lighting via the ESP32. Finally, the system functionality is tested to assess the accuracy of wake word detection, response time, and the reliability of the command for the lighting control.

3.2 System Design

The Smart Home Automation System with Wake Word Detection combines hardware and software elements to offer a voice-activated, user-friendly lighting control solution for several areas of a home setting. The central component of the system is the ESP32 microcontroller, which was selected due to its low power consumption, integrated Wi-Fi, and processing power appropriate for wake word detection and device control. The wake word detection module reduces latency and improves privacy by locally recognizing the user's voice instructions on the ESP32. The ESP32 uses connected LEDs to control the illumination in various zones after detecting commands.

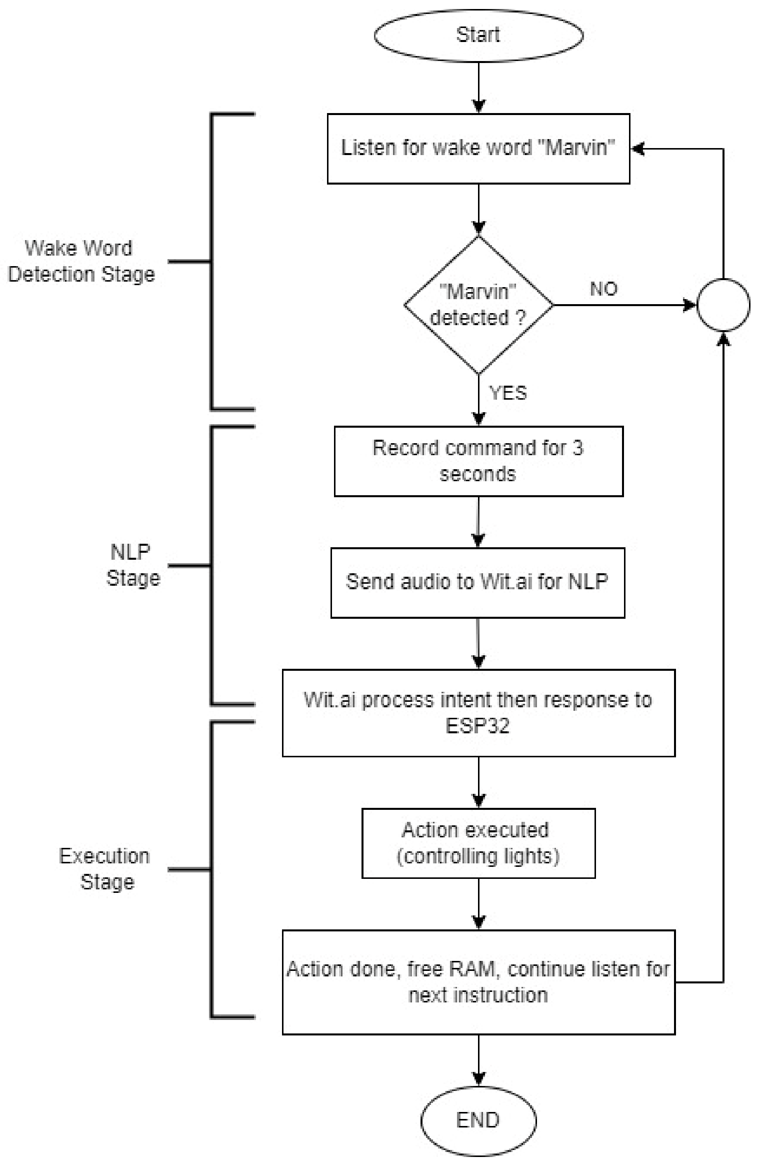

Start
Listen for wake word "Marvin"
Wake Word Detection Stage
"Marvin" detected ?
NO
YES
Record command for 3 seconds
NLP Stage
Send audio to Wit.ai for NLP
Wit.ai process intent then response to ESP32
Execution Stage
Action executed (controlling lights)
Action done, free RAM, continue listen for next instruction
END

Figure 3.1: Flowchart of the overall system

Figure 3.1 is the flowchart for the Smart Home Automation with Wake Word Detection system. The flow for the system can be segmented into three stages, which are the local processing wake word detection stage, the natural language processing stage, and the execution stage. When the system is powered on, it will enter the word detection stage, where the microphone continuously listens for “Marvin” to proceed to any further instruction. This can not only save resources, as the ESP32 only processes after “Marvin” is detected, but also ensures the privacy of the user, as the command processing is cloud-based, which might cause unauthorized access.

Next, the system will capture 3seconds of the next audio right after “Marvin” is detected, which is the stage known as the natural language stage. In this particular stage, Wi-Fi connectivity is needed to connect with an NLP platform called Wit.ai to process the user’s command. The recorded 3seconds of audio is sent to the Wit.ai platform via HTTP protocol, to interpret the user’s command. After the NLP is done, the cloud-based server will send a response back to ESP32 for the next stage, which is the execution stage.

Finally, in the execution stage, the ESP32 received the information from the Wit.ai platform to execute the user’s command of controlling the lights. ESP32 either sends a high signal or a low signal to the respective GPIO pins to enable or disable the lights. After the command is done, the system will clear its RAM and wait for the next wake word detected. This process is important as the ESP32 is a low-power limited limited-resource microcontroller. Then the system will continue to listen for “Marvin” if the microcontroller is still powered on.

3.2.1 System Architecture

The system architecture is to define how the hardware and software components are organized and interact to design a voice-controlled smart home automation. Figure 3.2 illustrates the system architecture for the Smart Home Automation with Wake Word Detection system. Table 3.1 is the elements that are used for the system and their respective functionality.

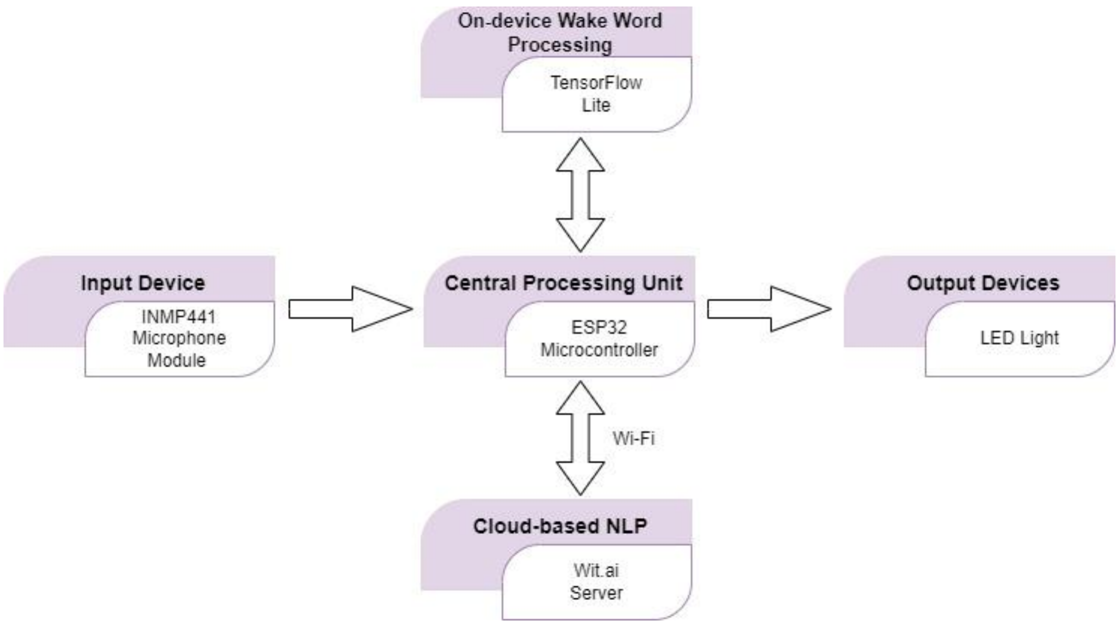


Figure 3.2: Block diagram of the system architecture

Table 3.1: Components used and its respective functionality on the project

Components	Functions
ESP32 Microcontroller	To handle wake word detection, captures audio and communication with Wit.ai
ESP32 Built-in Wi-Fi Module	Allows internet connectivity for data transmission
INMP441 Omnidirectional Microphone Module	To capture audio input for processing
Wit.ai Cloud-based NLP Platform	To process audio data that analyse and extract the user's intent
LEDs	LEDs are used as output devices

In the system architecture, the ESP32 DevKit V1 is used as the brain of the system. This central processing unit handles the wake word detection, audio recording, and communication with Wit.ai cloud services. Hypertext Transfer Protocol (HTTP) facilitates the communication between ESP32 and the Wit.ai server. This application protocol layer ensures that the user's intents are sent to Wit.ai to be analyzed and executed by the ESP32. The Wi-Fi network connection supports HTTP, which allows communication between the ESP32 and Wit.ai server. The built-in Wi-Fi module of the ESP32 provides internet connectivity necessary for transmitting data in real-time.

INMP441 Omnidirectional Microphone Module is the input device that is connected to the ESP32. It is used to control the output devices through voice command. The module captures and records audio input for the system to process. The LED lights are the output devices for the system. They are connected to the GPIO pins of the ESP32 to provide visual feedback or indication of system status.

3.2.2 Hardware Components

The Smart Home Automation System with Wake Word Detection using ESP32 relies on hardware components to function, allowing for voice-controlled lighting in many zones. The core hardware components include the ESP32 microcontroller and INMP441 microphone.

<u>ESP32 DevKit V1</u>

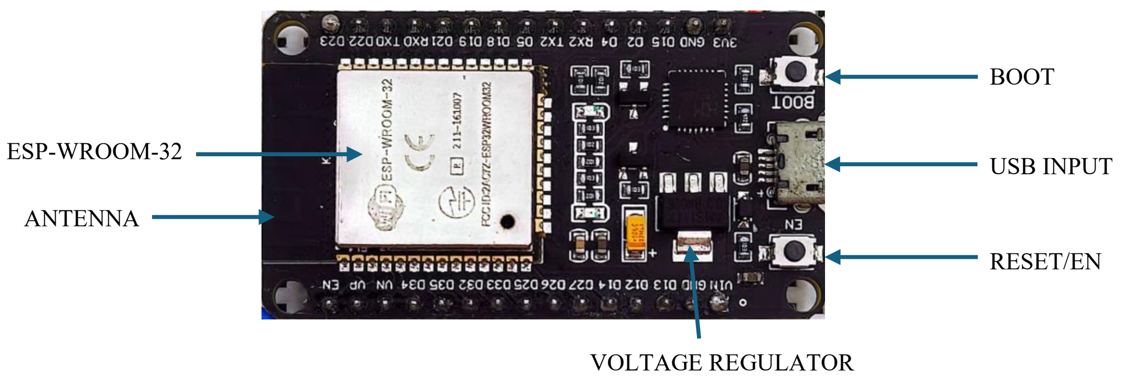


Figure 3.3: The ESP32 DevKit V1 board

The central processing unit of the Wake Word Detection Smart Home Automation System is the ESP32 DevKit V1, shown in Figure 3.3. This microcontroller was chosen because it is affordable, has built-in Bluetooth and Wi-Fi, and supports real-time processing. The Tensilica Xtensa LX6 dual-core processor can operate at up to 240 MHz. It has about 520 KB of SRAM, 448KB of ROM, and 4 MB of flash memory. The ESP32 DevKit V1 is perfect for controlling lights in various zones because it has many GPIO pins for integrating with sensors and actuators. Its integrated Wi-Fi module enables network connectivity for user devices or cloud services to communicate with one another. Based on the official Espressif ESP32 datasheet (*DOIT ESP32 DEVKIT V1 Development Board Details, Pinout*, no date), Table 3.2 provides a summary of the essential elements for comprehensive specifications.

Table 3.2: Specifications of ESP32 DevKit V1 Board

Specification	**Detail(s)**
Processor	Tensilica Xtensa dual-core 32-bit LX6
SRAM	520KB
Flash Memory	4KB
Wireless Connectivity	Wi-Fi: 802.11b/g/n, 2.4 GHz
	Bluetooth: Classic and Low Energy (BLE)
Communication Interfaces	USB, UART, SPI, I2C, PWM, 1-wire
GPIO Pins	36 digital GPIO pins
Power Supply	7-12 V

<u>INMP441 Omnidirectional Microphone Module</u>

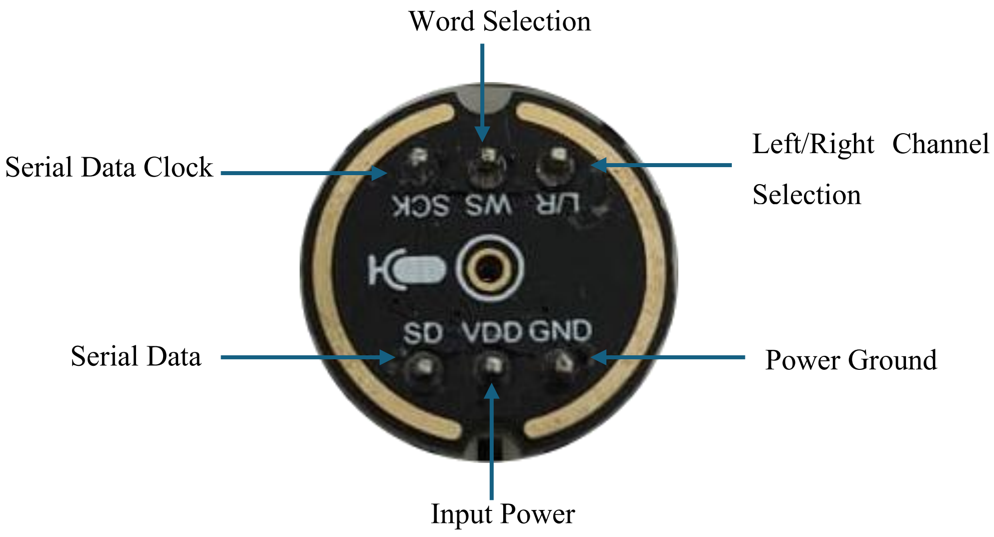


Figure 3.4: INMP441 Omnidirectional Microphone Module

The INMP441, as shown in Figure 3.4, is a digital-output omnidirectional MEMS microphone module with a bottom port that is high-performance and low-power. Its 24-bit high-precision I2S digital interface eliminates the requirement for an audio codec and facilitates direct communication with microcontrollers such as the ESP32. Clear audio capture is ensured by the microphone's high signal-to-noise ratio (SNR) of 61 dB and high sensitivity of −26 dBFS. It has a frequency response of 60 Hz to 15 kHz, allowing it to record real sound for voice detection applications. Furthermore, the module has a low power consumption of approximately 1.4 mA, making it perfect for battery-powered or energy-efficient applications (“INMP441,” no date).

Table 3.3: Specifications of INMP441

Specification	**Detail(s)**
Interface	Digital I^2S interface
Data Output	High-precision 24-bit data
Signal-to-Noise Ratio	61 dBA typical
Sensitivity	-26 dBFS typical
Current Consumption	≈ 1.4mA
Power Supply	1.8V to 3.3V DC

3.2.3 Software Component

In order to control the operation and communication of the physical components, the Smart Home Automation System with Wake Word Detection software is essential is the main integrated development environment used for software development.

Visual Studio Code (VSCode)

Figure 3.5: The Visual Studio Code logo

Figure 3.6: The PlatformIO logo

VSCode is used to build the wake word detection model for deployment onto embedded devices. Jupyter Notebook (.ipynb file) is developed to train the model. The trained model is then converted into a TensorFlow Lite (.tflite) format. This is to reduce the file size of the trained model to be loaded onto the ESP32. The ESP32 microcontroller is programmed and implemented using the PlatformIO extension in Visual Studio Code. PlatformIO offers a smooth development experience with capabilities that were designed especially for embedded systems like the ESP32. PlatformIO includes code compilation, firmware uploading, and code debugging features. From the model training to firmware implementation, the development workflow is streamlined by combining VSCode with Jupyter and PlatformIO extensions. Using only one main software makes the project successful and manageable.

3.3 Hardware Design and Setup

The Smart Home Automation System with Wake Word Detection hardware setup involved the integration of several components. In this section, the GPIO configurations and peripheral connections on the ESP32 are described.

3.3.1 Peripheral Connections

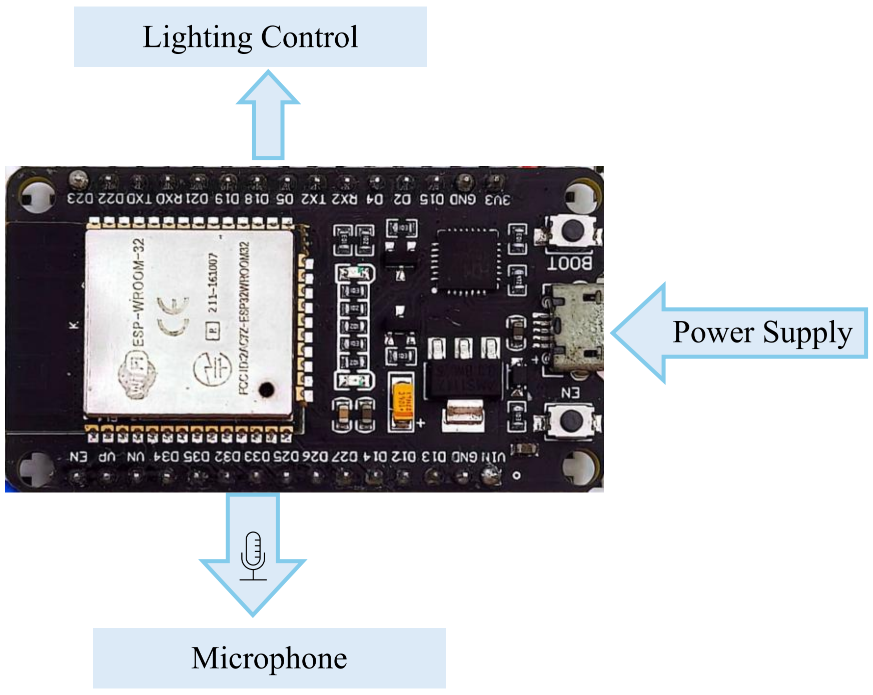


Figure 3.7: Peripheral connections to ESP32

The core controller is the ESP32 DevKit V1, and Figure 3.7 shows the connections to several external devices. The ESP32 and its related components are powered by a steady power source that is supplied via the micro-USB port. Via the I2S interface, ESP32 is linked to the INMP441 omnidirectional microphone module, which records speech commands for wake word detection. Several GPIO pins on the ESP32 are connected to LEDs, which serve as the lighting control outputs. These pins are set up as outputs to control when the lights in various zones are turned on and off. The following subsections will discuss the pins' configuration.

3.3.2 GPIO Configuration

Individual LEDs that represent various illumination zones are controlled by designated GPIO pins on the ESP32 in the system. ESP32 sends a high and low signal to turn on and off the LED as an output for the system. The GPIO assignments and their respective function is summarized in Table 3.4 and Table 3.5.

Table 3.4: The GPIO pin assignment and functions of voice input

GPIO Pin	Function(s)
INMP441 Connections	
D25	Word Select(WS) signal for the I2S interface. It indicates the left channel for digital audio data being transmitted and synchronizes with the data stream.
D33	Outputs the Serial Data (SD) in digital audio form via the I2S interface. It sends the left channel audio samples to the microcontroller.
D32	The Serial Data Clock (SCK) signal is used to synchronize the data bits on the SD line. It controls the timing of the audio data bits.
VDD	Power supply 3.3V to activate and operate the module
GND	Ground pin to complete the electric circuit and serve as the voltage reference.

Table 3.5: The GPIO pin assignment and functions of lighting control

GPIO Pin	Function(s)
LEDs Control	
D4	To control **bathroom** lighting
D5	To control **kitchen** lighting
D21	To control **bedroom** lighting
D23	To control **table** lighting

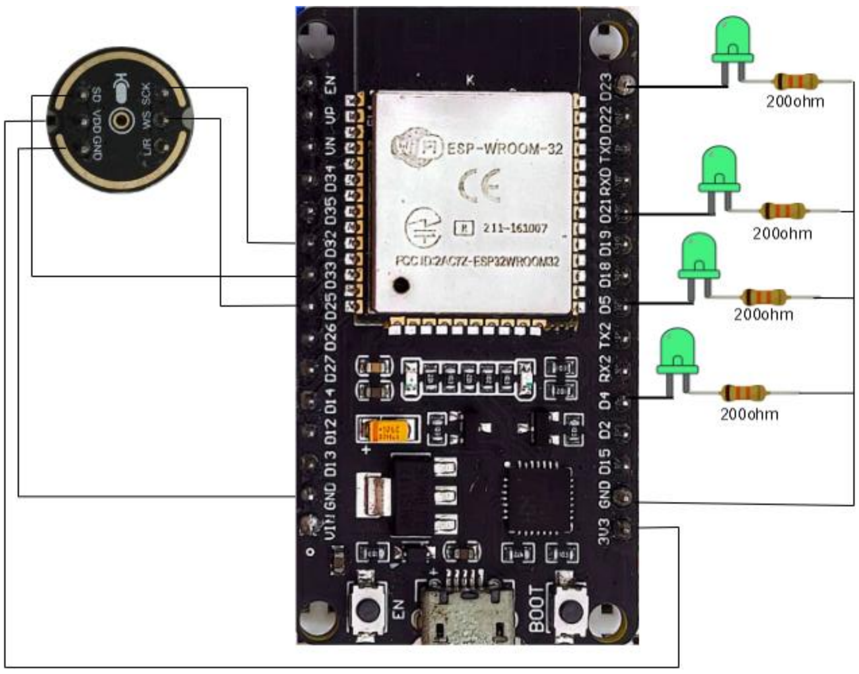

SD VDD GND
SCK WS L/R
EN VP VN D34 D35 D32 D33 D25 D26 D27 D14 D12 D13 GND VIN
D23 D22 TXD RXD D21 D19 D18 D5 TX2 RX2 D4 D2 D15 GND 3V3
ESP-WROOM-32
EN
BOOT
200ohm
200ohm
200ohm
200ohm

Figure 3.8: The schematic diagram of the system

The specific GPIO pins are connected to the LED with a 200Ω resistor in series, which acts as a current-limiting component to avoid excessive current damaging the LED. The other end of the resistor is connected to the ground pin to complete the circuit. The complete hardware integration is shown in Figure 3.8.

3.3.3 Power Supply and Wi-Fi Connectivity

This section explains how ESP32 and other components receive power and how the system connects to a Wi-Fi network for cloud communication.

The ESP32 microcontroller and its peripheral components are powered via a USB port, which supplies a 5V input to the built-in voltage regulator, as illustrated in Figure 3.3, which steps down to the 3.3V required by the ESP. The ESP32 normally draws roughly 250mA during Wi-Fi transmissions and boot procedures, so it is critical to have a stable power source, using a USB to microUSB cable that is capable of transferring at least 500mA, to ensure smooth operation. Proper power management includes safeguarding against voltage fluctuations and incorporating the appropriate capacitors to provide a smooth power supply.

The network SSID and password are hardcoded into the firmware running on the ESP32 to establish a connection with the Wi-Fi router or mobile hotspot. This connection allows communication with Wit.ai, a cloud-based NLP service, via REST APIs. Network failures can affect the system's responsiveness, which may lead to delays or difficulties in processing the voice command. Thus, a stable and reliable Wi-Fi connection is required.

ESP32 will stay in idle mode to consume less power during the period of inactivity. However, this feature has to be properly programmed to preserve the functionality of wake word detection. Overall, a continuous power supply and a stable Wi-Fi connection are critical to maintain the consistency and effective operation of the smart home automation system.

3.4 Software Implementation

The Smart Home Automation with Wake Word Detection used an integrated development environment to set up for seamless lighting control and cloud-based NLP communication. This section outlines the setup of the Wake Word Model, Natural Language Processing, and Command Execution Logic.

3.4.1 Wake Word Detection Model

This section describes the method of implementing wake word detection on ESP32 using CNN machine learning.

3.4.1.1 MFCC Feature Extraction

Raw audio waveform is loaded from a pre-recorded .wav file for feature extraction. This is to ensure that only valid .wav files are handled. The invalid files will be removed from the prerecorded audio list as shown in Figure 4.4. In order to enable localized frequency analysis, the waveform is segmented into short and overlapping frames. This process utilized windowing and framing techniques on the audio signals. These segments are then used to generate the Mel-Frequency Cepstral Coefficients (MFCCs) spectrum to capture and analyze the key properties in the spectrum.

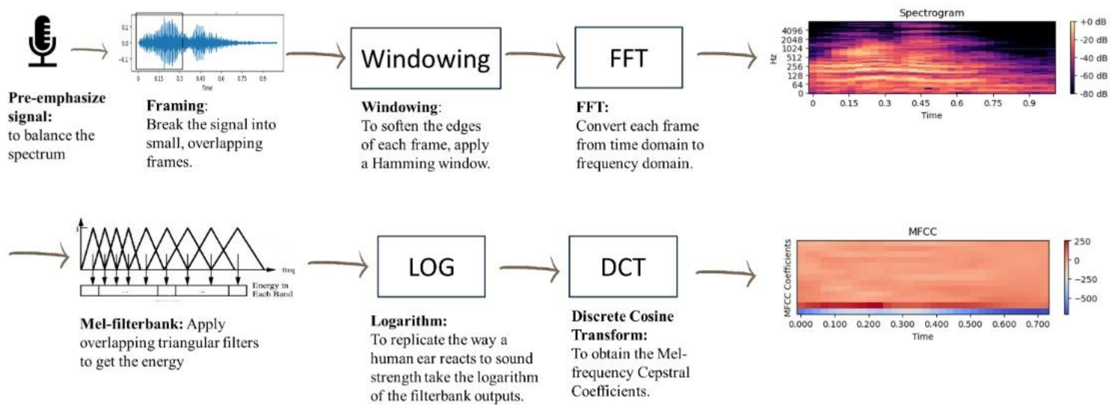

Pre-emphasize signal: to balance the spectrum
Framing: Break the signal into small, overlapping frames.
Windowing
Windowing: To soften the edges of each frame, apply a Hamming window.
FFT
FFT: Convert each frame from time domain to frequency domain.
Spectrogram
Mel-filterbank: Apply overlapping triangular filters to get the energy
Energy in Each Band
LOG
Logarithm: To replicate the way a human ear reacts to sound strength take the logarithm of the filterbank outputs.
DCT
Discrete Cosine Transform: To obtain the Mel-frequency Cepstral Coefficients.
MFCC

Figure 3.9: The process of converting the raw audio signal into MFCC spectrum

The process of converting raw audio into a spectrogram includes pre-emphasis, framing, windowing, the Fourier Transform, mapping frequencies to the Mel scale, calculating the power spectrum's logarithm, and using the Discrete Cosine Transform (DCT) for feature decorrelation, as shown in Figure 3.9. In order to preserve consistency for model input, irregular samples are eliminated, and only MFCC sets of a predetermined length are kept. These processed MFCC arrays prepare the input data for the next stage of model training by forming the feature vectors that are stored with their corresponding labels, which indicate whether the wake word is present or not.

Table 3.6: Specification of the MFCC parameters used in the system

Sampling Rate	Audio is resampled to 8kHz
Number of MFCC coefficients	16 coefficients per frame (`num_mfcc = 16`)
Frame/Window Length	256 milliseconds applied on audio frames (`winlen=0.256` seconds)
Frame Step Length	50 milliseconds (`winstep=0.050` seconds), which defines how often the window moves forward
Number of Mel Filters	26 triangular bandpass filters used to approximate human auditory perception
FFT Size	2048 data points for the Fourier transform (`nfft=2048`), which defines the frequency resolution.

Pre-emphasis Filter	It is disabled for the system (`preemph=0.0`)
Window Function	A Hanning window is applied to each frame to reduce spectral leakage
Length of MFCC Frames per sample	16 (`len_mfcc = 16`), used to filter valid samples for further processing

3.4.1.2 CNN Model Development

The MFCC feature arrays are used as input to the network's input layer, which takes two-dimensional representations of the extracted features to create the convolutional neural network (CNN) model. The network design consists of several convolutional layers that use spatial filtering to automatically learn pertinent spectral and temporal audio patterns. Max-pooling layers are added after convolutional layers to reduce feature dimensionality and improve model robustness through translational invariance.

After extracting spatial features, fully connected (dense) layers interpret the filtered patterns to conduct classification. Finally, the network has an output layer that performs binary classification, predicting whether the wake word is present or not present. The model is trained on a labeled dataset that includes publicly available speech command datasets, obtained from *speech_commands | TensorFlow Datasets* (no date), as well as personalized wake word recordings. Accuracy and loss measures are used to assess the performance of the trained model in order to guarantee dependable and accurate detection capabilities.

3.4.1.3 TensorFlow Lite Deployment

For implementation on embedded hardware, the trained CNN model is quantized to reduce the parameters to 8-bit integers. This approach reduces the model's size and speeds up the inference, making it appropriate for resource-constrained devices such as the ESP32. The quantized model is then transformed into TensorFlow Lite (.tflite) format for use with lightweight inference processors. The TensorFlow Lite Micro library is utilized on the ESP32 to effectively execute the model within the firmware of the device.

In a continuous loop, the embedded application constantly records audio, extracts incoming frames for MFCC in real time, and carries out wake word detection inference. This integrated pipeline offers low-latency, high-accuracy wake word recognition completely on-device, removing the requirement for cloud connectivity and shortening response times for practical voice-activated applications.

3.4.2 Natural Language Processing

Natural Language Processing (NLP) allows the system to recognize and interpret human voice commands that are more than simple keyword recognition. NLP converts spoken language into structured data, which allows ESP32 to take appropriate actions based on the user's intent. The following subsections will go through Wit.ai integration for NLP processing, as well as handling of HTTP requests and intent interpretation by the system.

3.4.2.1 Integration with Wit.ai

Wit.ai is known as a free cloud-based natural language processing tool that developers may utilize to extract information from voice or text input. It provides rapid and efficient voice recognition and intent extraction. Thus, Wit.ai is for real-time smart home applications that do not require a lot of local processing power. The command audio is transmitted to Wit.ai over a Chunked HTTP Streaming API after the wake word is activated. Wit.ai extracts intentions like "turn on" or "turn off" and entities such as kitchen or bedroom from the audio signal. This data format allows exact command execution based on natural language.

3.4.2.2 HTTP Request and Intent Handling

ESP32 uses HTTP POST requests to allow communication between the recorded voice with Wit.ai. It sends together with the permission tokens and the relevant content-type headers. The system parses the intent and entity values after getting a JSON response, such as { `"intent"`: `"turn_on"`, `"entity"`: `"bedroom"` }. This interpreted response is used by the ESP32 firmware to turn on or off the specific smart home components.

3.4.3 Command Execution Logic

Smart home lighting is controlled by the ESP32 command execution logic that decodes the extracted NLP intents and entities, then maps user voice instructions to the particular hardware actions. This decision-making mechanism allows for the immediate and accurate activation or deactivation of lights in response to recognized instructions.

3.4.3.1 Mapping User Intents to Lighting Control

GPIO control commands are mapped to user intents that reflect actions like "turn on" or "turn off," which send high and low signals to the connected LED relevant to specific lighting zones. The GPIO pin that is of interest is determined by entities that correspond to room names or zones (for example, bedroom, kitchen). The ESP32 firmware makes use of if-else statements to effectively convert the received intents and entities into hardware control signals.

3.5 System Communication

This section discusses the method of ESP32 setup to maintain Wi-Fi connectivity. This is important to connect to the internet to allow communication with external cloud services for natural language processing. This section also describes the establishment of a network using native libraries and chunked HTTP POST requests to communicate with Wit.ai speech recognition API and manage the execution logic.

3.5.1 ESP32 and Network Communication

ESP32 is connected to Wi-Fi networks via a hardcoded SSID and password controlled by the WiFi.h library. The device attempts to connect and continuously monitors its status upon booting. ESP32 has an automatic reconnection attempt system to ensure network reliability. This is to guarantee a dependable performance of the firmware and also manage some typical networking problems like delays and timeouts.

3.5.2 Communication with Wit.ai Server

For real-time speech recognition, ESP32 used chunked transfer encoding HTTP POST requests to deliver the raw audio data in discrete chunks to the `/speech` endpoint of Wit.ai. A zero-length chunk is sent to indicate that it is the end of the stream after several audio data chunks have been sent. Each chunk is preceded by its size in hexadecimal notation. The `WiFiClientSecure` class utilizes TLS to establish secure communication. However, certificate validation is turned off when using `setInsecure()`.

After audio streaming is ended, ESP32 analyzes the server's JSON response and parses essential parameters like intents, entities, and traits using the Arduino JSON library and an efficient deserialization filter. The structured response controls the command execution logic of the system. Robust error handling and connection status monitoring are used to delicately manage potential transmission faults while maintaining a reliable operation.

3.6 Summary

This chapter described the overall techniques of developing a smart home lighting control system with wake word detection using an ESP32 microcontroller. The hardware configuration, wake word detection with MFCC and TensorFlow Lite, Wit.ai for natural language processing, and the communication between components are covered in this chapter. The system was created to provide a quick, dependable, hands-free voice management of the home lighting. The following chapter will discuss the results of the process and assessment of the system's performance in terms of functionality, accuracy, and user experience.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 General Introduction

This chapter describes the overall technique of developing a smart home lighting control system with wake word detection using an ESP32 microcontroller. Hardware configuration, wake word detection with MFCC, and TensorFlow Lite will be covered. Besides that, Wit.ai for natural language processing and the communication between components will also be discussed. The following chapter will discuss the results of the processes and assess the performance of the system in terms of functionality and accuracy.

4.2 Wake Word Detection

TensorFlow Convolutional Neural Network (CNN) model is used to create the wake word detection module that was trained on Mel-Frequency Cepstral Coefficients (MFCC) as the input of the model. The MFCCs are processed from audio inputs. The system was developed to recognize the wake word "Marvin". It was also configured to incorporate with ESP32 microcontroller for real-time testing.

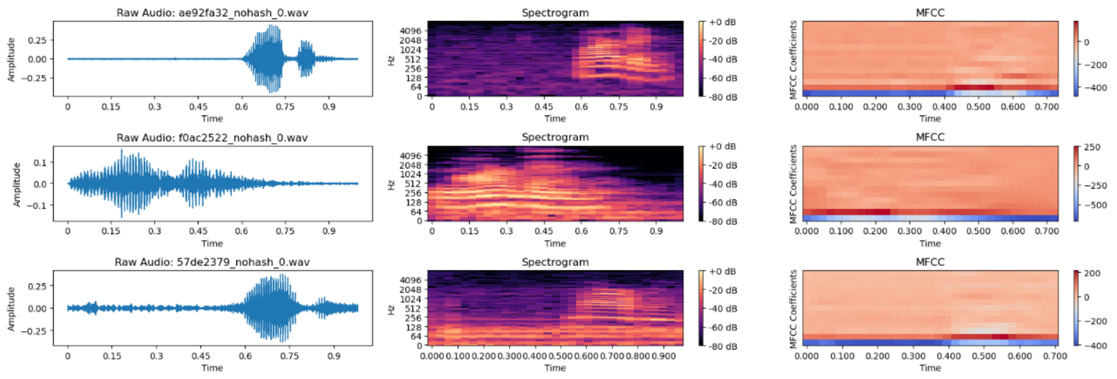


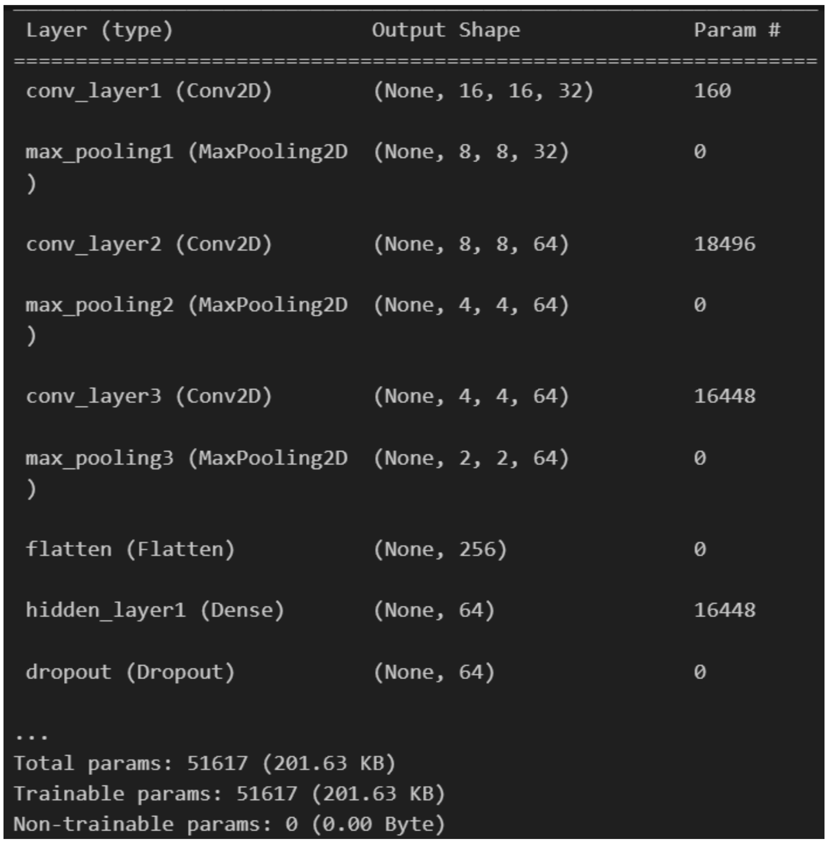
Figure 4.1: The process of converting raw audio signal (for the word "Marvin") into the MFCC spectrum for feature extraction

The Mel-frequency cepstral coefficient (MFCC) is a common technique that is used for speech processing. It converts a raw audio data signal into a compact representation to only captures the key feature of an audio. Three sets of "Marvin" audio files are taken in Figure 4.1 to illustrate how the system transforms raw audio data into MFCC spectrum, which feeds as the input of the CNN model for feature extraction. There are a total of 2100 audio files that were used to train for the word "Marvin". The parameter that is used for this model is stated in Table 3.6.

```
model = Sequential([
    Conv2D(32, (2, 2),
          activation='relu',
          padding='same',
          name='conv_layer1',
          input_shape=sample_shape),
    MaxPooling2D(pool_size=(2, 2), name='max_pooling1'),

    Conv2D(64, 3,
          activation='relu',
          padding='same',
          name='conv_layer2'),
    MaxPooling2D(pool_size=(2, 2), name='max_pooling2'),

    Conv2D(64, (2, 2),
          activation='relu',
          padding='same',
          name='conv_layer3'),
    MaxPooling2D(pool_size=(2, 2), name='max_pooling3'),

    Flatten(name='flatten'),
    Dense(64, activation='relu', name='hidden_layer1'),
    Dropout(0.5, name='dropout'),
    Dense(1, activation='sigmoid', name='output')
])
```

Figure 4.2: The code segmentation of the CNN model.

The above code segmentation is the part where the convolution of the audio takes place. The model is structured with multiple convolutional layers, max-pooling layers, flattening, and fully connected (dense) layers. A summary of the model is shown in Figure 4.3.



```
 Layer (type)                Output Shape              Param #
=================================================================
 conv_layer1 (Conv2D)        (None, 16, 16, 32)        160

 max_pooling1 (MaxPooling2D  (None, 8, 8, 32)          0
 )

 conv_layer2 (Conv2D)        (None, 8, 8, 64)          18496

 max_pooling2 (MaxPooling2D  (None, 4, 4, 64)          0
 )

 conv_layer3 (Conv2D)        (None, 4, 4, 64)          16448

 max_pooling3 (MaxPooling2D  (None, 2, 2, 64)          0
 )

 flatten (Flatten)           (None, 256)               0

 hidden_layer1 (Dense)       (None, 64)                16448

 dropout (Dropout)           (None, 64)                0

...
Total params: 51617 (201.63 KB)
Trainable params: 51617 (201.63 KB)
Non-trainable params: 0 (0.00 Byte)
```

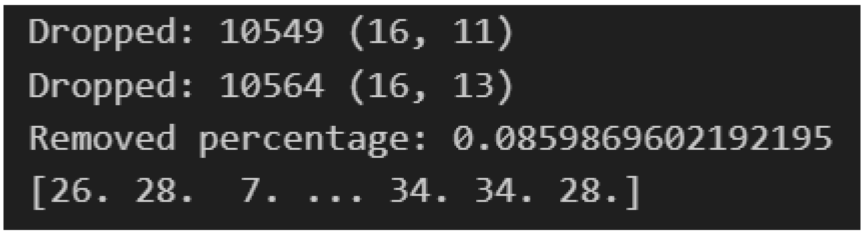
Figure 4.3: A summary of the trained model.

In the first layer `conv_layer1`, the input shape is `(None, 16, 16, 1)`, which means a 2D image with height and width of 16 and a single colour channel image is fed to this layer. Then, for the output is `(None, 16, 16, 32)`, meaning the size of the image remains the same as the input for this layer, but with 32 feature maps. Each of the feature maps corresponds to a learned 2×2 filter.

Next, the `max_pooling1` layer reduces the spatial size of the feature maps by a factor of 2. The max-pooling operation takes a 2×2 grids in each feature map to reduce computation and extract more abstract features while keeping the important information. Thus, the output shape is reduced from the initial size of 16×16 to 8×8, as seen in `(None, 8, 8, 32)`. The `Conv2D` and `MaxPooling2D` layers are repeated 3 times to reduce the size of the feature maps, preparing for the next classification layer.

After the third convolutional and pooling layers, the features are flattened from 2D feature maps into a one-dimensional vector. The input shape is from `(None, 2, 2, 64)` to the output shape of `(None, 256)`, which is then fed into the fully connected layer for classification. Then, in the `hidden_layer1`, which performs a dense layer, learns the high-level representation of the feature extracted from the convolutional layers. The parameters shown `16,448` mean that the dense layer has 64 neurons. The model learns to connect 256 features from the flattened layer to the 64 outputs. This layer is also known as the "decision-making" layer, which processes all the features to produce the final output, which is the `dropout`. The dropout is used to prevent overfitting during the training.

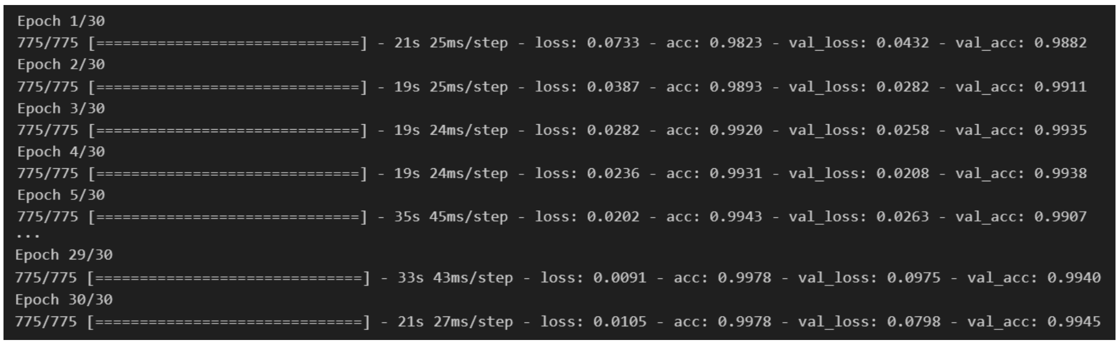
Lastly, the total parameters that are computed, as seen in Figure 4.3, which is the value `51,617` is means that these are the trainable parameters that the model successfully learned during the training. The non-trainable parameters are equal to zero, it is because before converting the raw audio data into MFCC spectrum to feed into the CNN model, the problematic dataset is being filtered and eliminated in the function of `extract_features`.



```
Dropped: 10549 (16, 11)
Dropped: 10564 (16, 13)
Removed percentage: 0.0859869602192195
[26. 28.  7. ... 34. 34. 28.]
```

Figure 4.4: Approximately 8.599% of audio data is removed from the sample dataset, which does not fulfill the desired length to perform MFCC.

Under normal indoor conditions, the model can achieve an approximate accuracy of 90%. It correctly detected the wake phrase in most of the trials. The detection was most efficient within a 60cm radius of the microphone. The accuracy is reduced when the distance exceeds 70cm radius or in the presence of significant background noise. The system's processing latency remained less than one second, indicating that the model was lightweight enough to allow real-time inference on the resource-constrained ESP32 platform.



```
Epoch 1/30
775/775 [==============================] - 21s 25ms/step - loss: 0.0733 - acc: 0.9823 - val_loss: 0.0432 - val_acc: 0.9882
Epoch 2/30
775/775 [==============================] - 19s 25ms/step - loss: 0.0387 - acc: 0.9893 - val_loss: 0.0282 - val_acc: 0.9911
Epoch 3/30
775/775 [==============================] - 19s 24ms/step - loss: 0.0282 - acc: 0.9920 - val_loss: 0.0258 - val_acc: 0.9935
Epoch 4/30
775/775 [==============================] - 19s 24ms/step - loss: 0.0236 - acc: 0.9931 - val_loss: 0.0208 - val_acc: 0.9938
Epoch 5/30
775/775 [==============================] - 35s 45ms/step - loss: 0.0202 - acc: 0.9943 - val_loss: 0.0263 - val_acc: 0.9907
...
Epoch 29/30
775/775 [==============================] - 33s 43ms/step - loss: 0.0091 - acc: 0.9978 - val_loss: 0.0975 - val_acc: 0.9940
Epoch 30/30
775/775 [==============================] - 21s 27ms/step - loss: 0.0105 - acc: 0.9978 - val_loss: 0.0798 - val_acc: 0.9945
```

Figure 4.5: The training log of the CNN model during the training process.

In epoch 1, the model is performing and learning quite effectively on both training and validation sets. As shown in Figure 4.5, the word "`loss`" and "`acc`" are the training metrics, while `val_loss` and `val_acc` are the validation metrics. The training metrics show the model is learning from the training data. The model will adjust its parameters to reduce the loss while improving the accuracy during the training. On the other hand, validation metrics show the model performs validation on the cross-validation dataset. It evaluates how well the model generalized to the new and unseen data.

As the model completely passes through the entire training dataset 30 times, it can be seen that the loss is reduced from 0.0733 to 0.0105 and the accuracy of the training is increased from 98.23% to 99.78%. As for the validation sets, the accuracy is improving from 98.82% to 99.45%. But the validation loss is slightly higher than at some earlier epochs. It might be due to the model slightly overfitting, but overall, the performance of the validation sets is acceptable since the validation accuracy performs well.

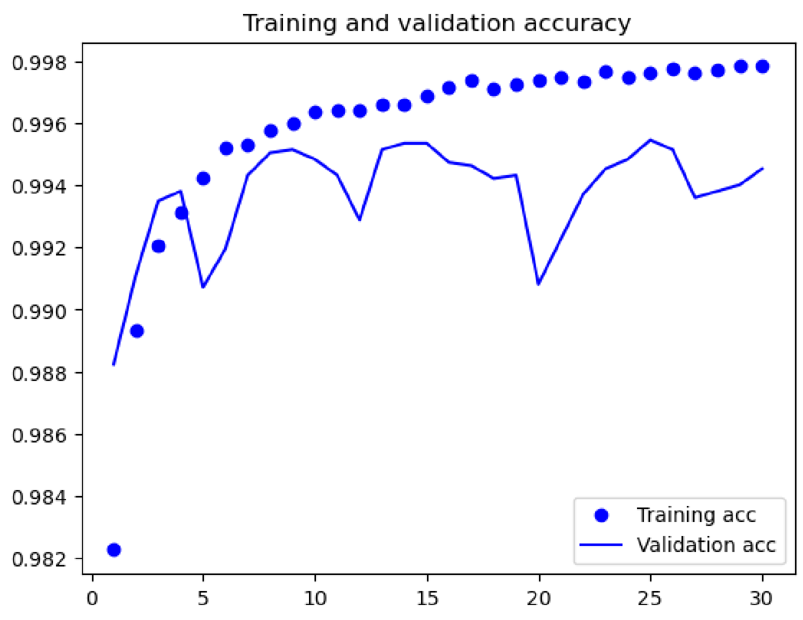


Figure 4.6: Training and validation accuracy of the wake word model.

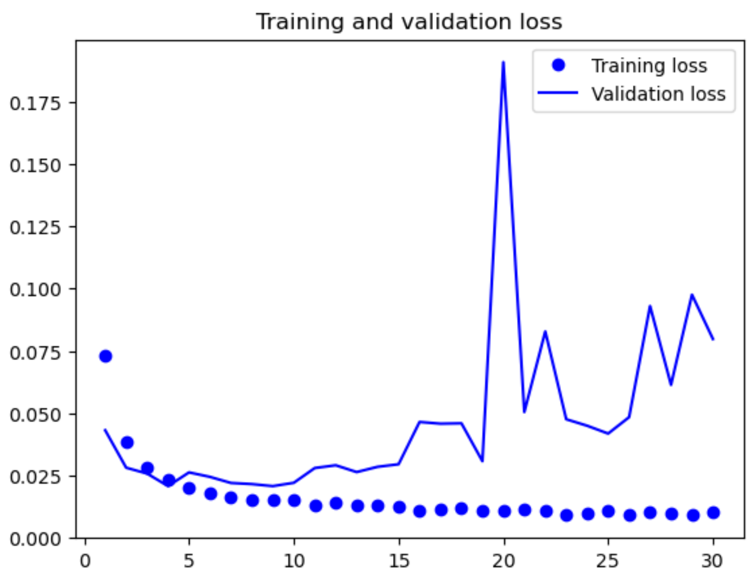
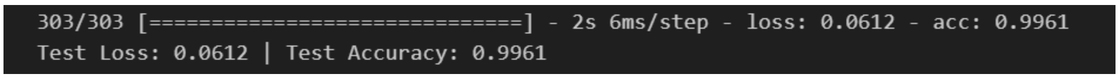


Figure 4.7: Training and validation loss of the wake word model.

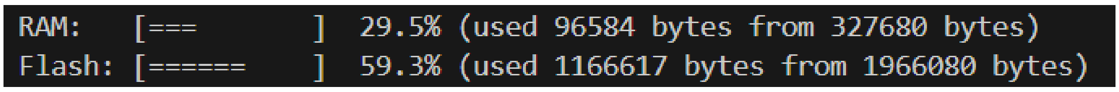
The accuracy and loss of the epochs are plotted as shown in Figure 4.6 and Figure 4.7. The training accuracy and loss are plotted using dots while the cross-validation accuracy is given by the solid line in Figure 4.6 and Figure 4.7. There is a total of 30 epochs as seen in Figure 4.5. The training and loss accuracy and loss converged around the separate value after a few epochs. The model performs better on the training data than on the validation data. This means that the model starts to learn the features unique to the samples in the training sets that do not generalize properly to the unseen data. This is known as overfitting and may cause issues to occur if the model has a bigger gap between the training and validation data. The accuracy of the model is above 96% means that the dataset does not overfit the model too much.



```
303/303 [==============================] - 2s 6ms/step - loss: 0.0612 - acc: 0.9961
Test Loss: 0.0612 | Test Accuracy: 0.9961
```

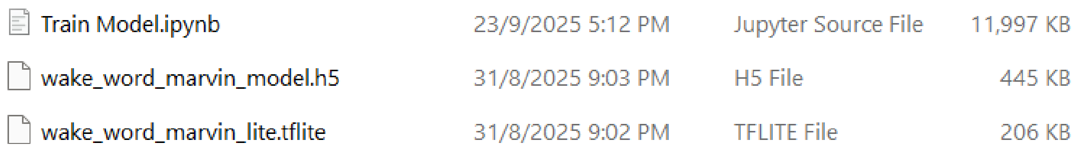
Figure 4.8: Evaluation test on the model, to see how well the model performs on the test case to predict "Marvin"

Then an `evaluate` function is called to evaluate on the test data that is separated from the trained data to observe the accuracy and loss of the model generated. Figure 4.8 shows the score of the accuracy and loss of the test case, which has an accuracy score of 99.61% and a loss of 0.0612. The model performs very well, as the prediction accuracy is close to the validation accuracy, as shown in Figure 4.5.



```
RAM:   [===       ]  29.5% (used 96584 bytes from 327680 bytes)
Flash: [======    ]  59.3% (used 1166617 bytes from 1966080 bytes)
```

Figure 4.9: The total RAM and Flash memory used on ESP32 for the entire system.



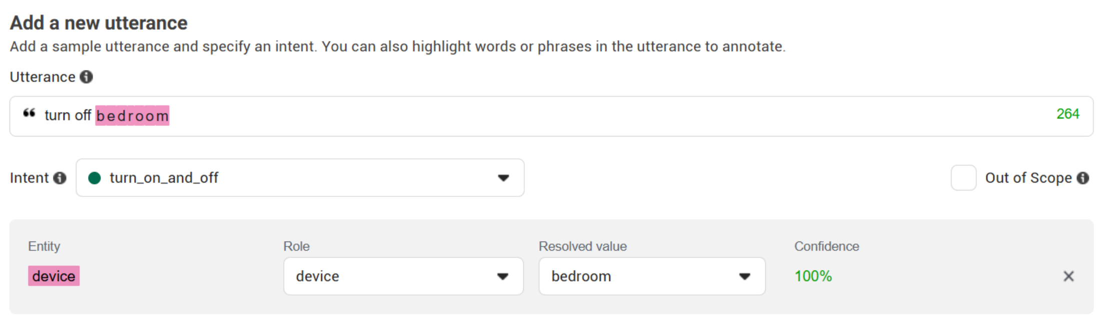
Train Model.ipynb	23/9/2025 5:12 PM	Jupyter Source File	11,997 KB
wake_word_marvin_model.h5	31/8/2025 9:03 PM	H5 File	445 KB
wake_word_marvin_lite.tflite	31/8/2025 9:02 PM	TFLITE File	206 KB

Figure 4.10: Converting the model into a lightweight model to deploy onto ESP32

Due to the ESP32 microcontroller being a low-power microcontroller with limited computational resources, in terms of CPU, RAM, and storage, TensorFlow Lite is used to design a lightweight version of Keras for resource efficiency. The TFLite model is compact, faster, and more optimized for ESP32 to reduce the memory and computational overhead, as seen in Figure 4.9. Figure 4.10 shows that the file size of the TFLite file is two times smaller than the H5 file. The model training for the wake word “Marvin” detection is compressed to embedded on ESP32.

4.3 Voice Command Interpretation via NLP

The results of the voice command interpretation system are discussed in this section. The system integrates natural language processing (NLP) after wake word detection to perform real-time actions based on voice commands. This process relies on Wit.ai, an NLP platform, to process and extract the user's voice input, identify the intents, and map the instructions to the corresponding entities. The system demonstrated an excellent level of accuracy and efficiency in interpreting the voice command, although there were challenges, such as accent variation and unclear pronunciations.


Add a new utterance

Add a sample utterance and specify an intent. You can also highlight words or phrases in the utterance to annotate.

Utterance

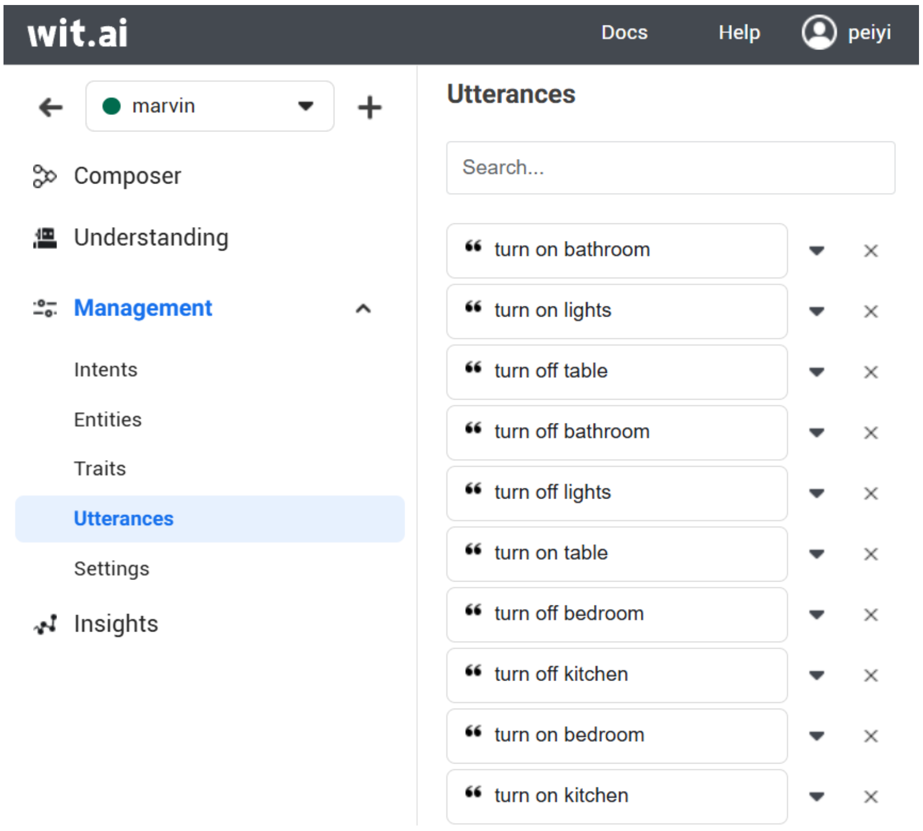
turn off bedroom 264

Intent: turn_on_and_off

Out of Scope

Entity	Role	Resolved value	Confidence
device	device	bedroom	100%

Figure 4.11: The accuracy and confidence level of adding and training for a new utterance can be up to 100% by the Wit.ai platform

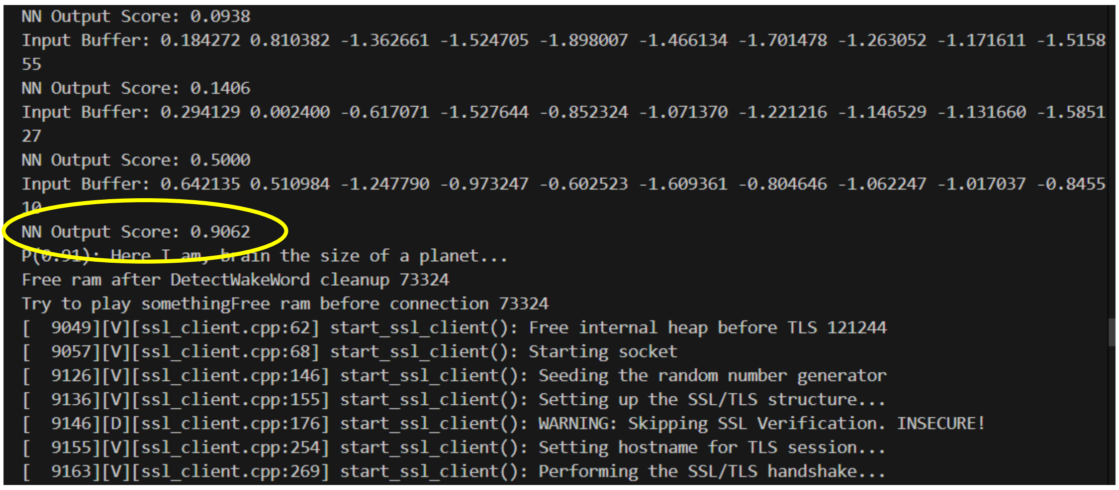


Utterances

- turn on bathroom
- turn on lights
- turn off table
- turn off bathroom
- turn off lights
- turn on table
- turn off bedroom
- turn off kitchen
- turn on bedroom
- turn on kitchen

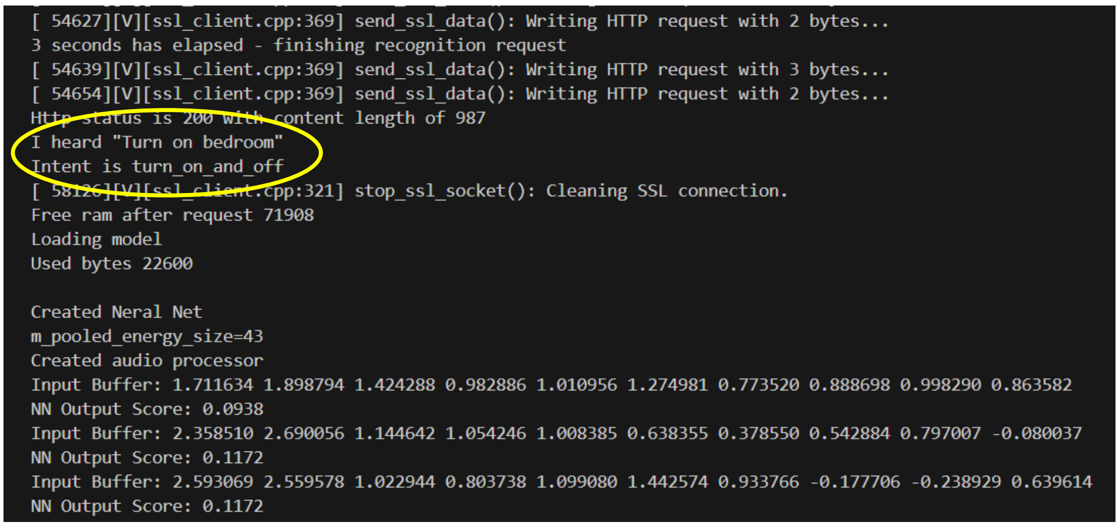
Figure 4.12: The utterances that are trained for this particular system

In most cases, the system successfully identified both intent and entities of the utterance as shown in Figure 4.11. The utterance of “turn off bedroom” consists of the intent, “turn off”, and the entity, “bedroom”. Figure 4.12 shows the utterances that are set by the user to control four areas, namely “bedroom”, “bathroom”, “kitchen”, and “table”. The utterance “turn on/off lights” is to control the four areas of the home system.



```
NN Output Score: 0.0938
Input Buffer: 0.184272 0.810382 -1.362661 -1.524705 -1.898007 -1.466134 -1.701478 -1.263052 -1.171611 -1.5158
55
NN Output Score: 0.1406
Input Buffer: 0.294129 0.002400 -0.617071 -1.527644 -0.852324 -1.071370 -1.221216 -1.146529 -1.131660 -1.5851
27
NN Output Score: 0.5000
Input Buffer: 0.642135 0.510984 -1.247790 -0.973247 -0.602523 -1.609361 -0.804646 -1.062247 -1.017037 -0.8455
10
NN Output Score: 0.9062
P(0.91): Here I am, brain the size of a planet...
Free ram after DetectWakeWord cleanup 73324
Try to play somethingFree ram before connection 73324
[  9049][V][ssl_client.cpp:62] start_ssl_client(): Free internal heap before TLS 121244
[  9057][V][ssl_client.cpp:68] start_ssl_client(): Starting socket
[  9126][V][ssl_client.cpp:146] start_ssl_client(): Seeding the random number generator
[  9136][V][ssl_client.cpp:155] start_ssl_client(): Setting up the SSL/TLS structure...
[  9146][D][ssl_client.cpp:176] start_ssl_client(): WARNING: Skipping SSL Verification. INSECURE!
[  9155][V][ssl_client.cpp:254] start_ssl_client(): Setting hostname for TLS session...
[  9163][V][ssl_client.cpp:269] start_ssl_client(): Performing the SSL/TLS handshake...
```

Figure 4.13: A snapshot from the serial monitors where the system is continuously listening for the wake word



```
[ 54627][V][ssl_client.cpp:369] send_ssl_data(): Writing HTTP request with 2 bytes...
3 seconds has elapsed - finishing recognition request
[ 54639][V][ssl_client.cpp:369] send_ssl_data(): Writing HTTP request with 3 bytes...
[ 54654][V][ssl_client.cpp:369] send_ssl_data(): Writing HTTP request with 2 bytes...
Http status is 200 with content length of 987
I heard "Turn on bedroom"
Intent is turn_on_and_off
[ 58126][V][ssl_client.cpp:321] stop_ssl_socket(): Cleaning SSL connection.
Free ram after request 71908
Loading model
Used bytes 22600

Created Neral Net
m_pooled_energy_size=43
Created audio processor
Input Buffer: 1.711634 1.898794 1.424288 0.982886 1.010956 1.274981 0.773520 0.888698 0.998290 0.863582
NN Output Score: 0.0938
Input Buffer: 2.358510 2.690056 1.144642 1.054246 1.008385 0.638355 0.378550 0.542884 0.797007 -0.080037
NN Output Score: 0.1172
Input Buffer: 2.593069 2.559578 1.022944 0.803738 1.099080 1.442574 0.933766 -0.177706 -0.238929 0.639614
NN Output Score: 0.1172
```

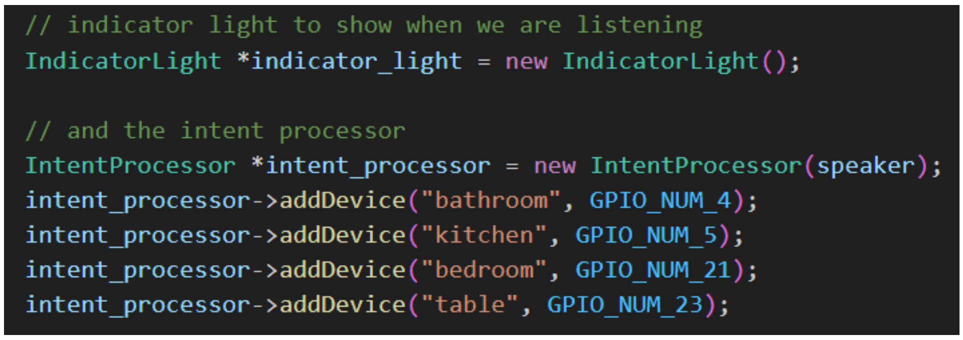
Figure 4.14: Feedback from Wit.ai showing it detected the user's command of "turn on bedroom", and the intent is to “turn on/off”

The system will consistently capture audio from the microphone to recognise “Marvin” through the MFCC spectrum. After the model detects the prediction wake word from the audio is above 90%, it will connect to the Wit.ai server through HTTP protocol, as shown in Figure 4.13. Then the audio that is 3seconds after “Marvin” is detected will be captured to allow Wit.ai to process and extract the user’s instruction. After Wit.ai has processed, it will feed back to ESP32 to execute the action as in Figure 4.14.

The response time for processing an instruction starting from wake word detection is an average of 8 seconds. After "Marvin" is detected, the system listens for 3 seconds, then transmits a request to Wit.ai and receives the response takes about 3seconds. Lastly, ESP32 takes less than 1seconds to control the lighting. But there are cases where the internet connection is very poor, and having delays connecting to the Wit.ai server might cause the system to delay or unrecognised intent to occur.

4.4 Lighting Control Output

Once the user's voice command has been analysed and the correct intent has been determined, ESP32 acts by controlling the GPIO pins to toggle the corresponding LED lights. A current-limiting resistor connects each LED directly to a GPIO pin, enabling dependable and secure operation without the need for extra relay modules. Voice commands like "turn on kitchen" and "turn off bedroom" were mapped to the particular GPIO pins during testing.



```
// indicator light to show when we are listening
IndicatorLight *indicator_light = new IndicatorLight();

// and the intent processor
IntentProcessor *intent_processor = new IntentProcessor(speaker);
intent_processor->addDevice("bathroom", GPIO_NUM_4);
intent_processor->addDevice("kitchen", GPIO_NUM_5);
intent_processor->addDevice("bedroom", GPIO_NUM_21);
intent_processor->addDevice("table", GPIO_NUM_23);
```

Figure 4.15: The snapshot of the written code for controlling the LEDs after the intent is processed.

The system responded appropriately in most of the test cases, with LEDs turning on or off as intended. The output state changed almost immediately after intent parsing, with minimal latency. This demonstrated the GPIO control logic's efficiency and portability. Transparency and traceability were provided during testing by the real-time serial monitor logs, which verified the accurate mapping between spoken intents and physical outputs.

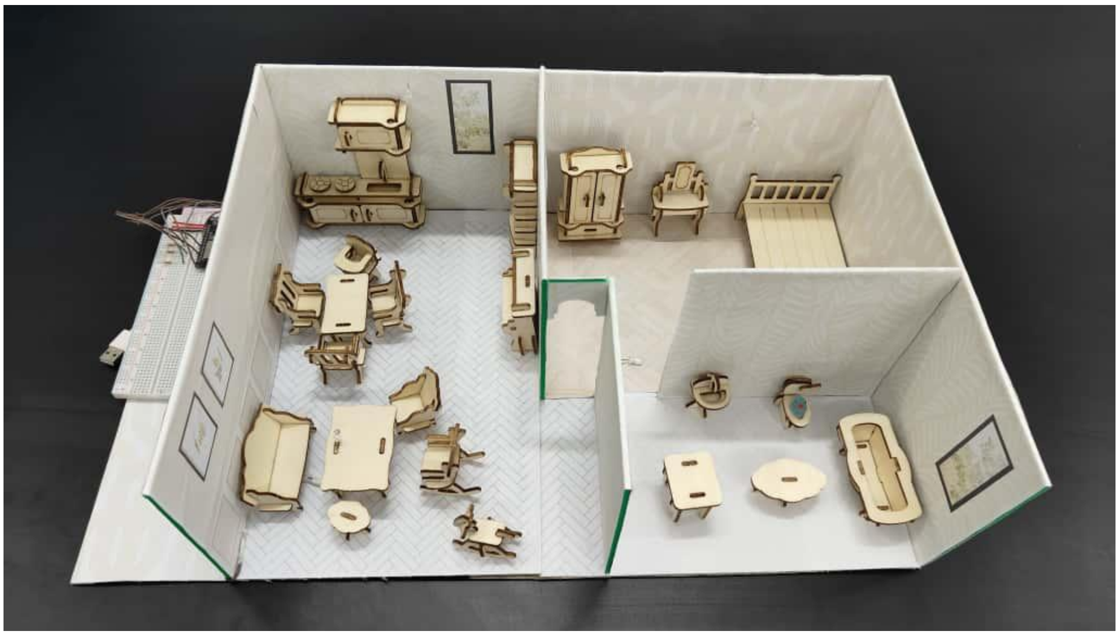


Figure 4.16: A prototype mimicking a real home environment. There are four areas which include kitchen, living room, bedroom and bathroom

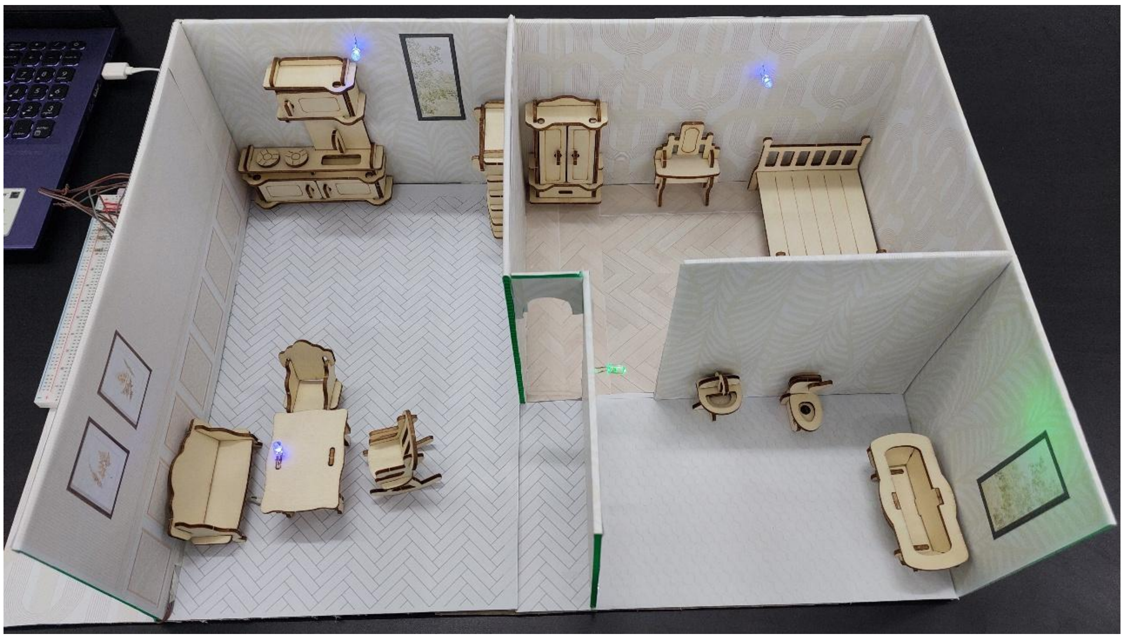


Figure 4.17: All lights in the areas are turned up using the command “turn on lights”

4.5 Performance Evaluation

The performance of the Smart Home Automation with Wake Word Detection system was evaluated based on a series of key metrics that focused on the accuracy and responsiveness of wake word detection, NLP intent recognition and lighting control. The assessment aims to evaluate the performance of the system under various conditions to identify areas for improvement.

4.5.1 Wake Word Detection

The wake word detection metric focuses on the ability of the system to correctly listen and identify the activation word, “Marvin”. The distance impact and also the environmental conditions will be evaluated.

Table 4.1: Wake word detection success rate by manipulating the distance

Distance (cm)	Trials	Success Rate	Accuracy
5	20	20	100%
25	20	19	95%
60	20	17	85%
100	20	13	65%

From Table 4.1, the accuracy of the local processing wake word detection varies with a distance of 5cm. 25cm, 60cm, and 100cm away from the microphone. The fixed variable for this trial-based approach is the word “Marvin” using recorded voice and the fixed environment condition with approximately 75dB noise surrounding. The ability of the system to capture and identify the word activation decreased when the distance is further away from the microphone increased. The accuracy of the word activation may be affected by the distance due to the mixing of background noise when the distance is further away.

Table 4.2: Wake word detection success rate by manipulating the surrounding conditions

Environment Condition	**Trials**	**Success Rate**	**Accuracy**
Normal room environment (≈ 75 dB)	20	20	100%
White noise is playing (≈ 85 dB)	20	19	95%
Music is playing (≈ 93 dB)	20	17	85%

Table 4.2 shows the success rate of detecting “Marvin” under three environmental conditions, which are in normal room conditions, and background music playing in white noise and music. The environmental conditions are evaluated to see how well the voice detection performs under different environmental conditions, with a constant distance of 20cm from the pre-recorded voice input from the microphone. Music playing in the background will disrupt the accuracy of the model slightly compared to the white noise and normal room conditions. This suggests that the surrounding environment will affect the accuracy of voice detection.

4.5.2 NLP Interpretation

The intent recognition metric evaluates the ability of the system to interpret the user's command after detecting "Marvin". The accuracy of the NLP is crucial for command execution based on the user's instructions.

The Wit.ai response time to process the command and return a response takes around 3 LED blinking on the ESP32 board. Which means that the process of sending HTTP requests and receiving the response from Wit.ai takes less than 3seconds, provided that the internet connection is stable.

Table 4.3: The success rate of each utterance of variable pronunciation by one person

Utterance	**Trials**	**Success Rate**	**Accuracy**	**Remarks**
"turn on/off bedroom"	15	15	100%	The system interpreted well
"turn on/off table"	15	13	86.67%	Misinterpreted as "TV", "paper"
"turn on/off kitchen"	15	10	66.67%	Mostly misinterpreted as "kitten"
"turn on/off bathroom"	15	15	100%	The system interpreted well

Table 4.3 shows the success rate of each utterance of variable pronunciation by one person and a constant environmental condition with a constant distance between the microphones. For the lighting control of "bedroom" and "bathroom", Wit.ai processes and transcribes them really well. But for the "table" and "kitchen", the accuracy is 86.67% and 66.67% respectively, where "kitchen" is quite poorly interpreted. From the real-time serial monitor log, it can be observed that the a misunderstanding of "kitchen" and "kitten" as shown in Table 4.4. This might be due to the user's pronunciation, or the accent used is not detected by Wit.ai.

Table 4.4: Compilation of misinterpretation and other faults.

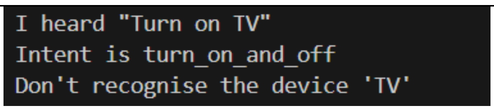
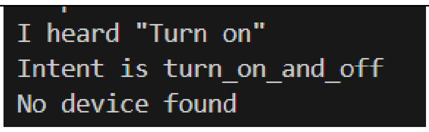
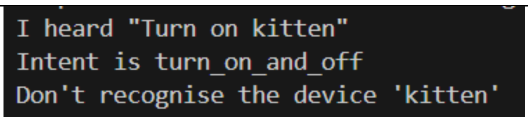
Faulty	Explanation
Misinterpretation “table” to “TV”. Since no device as “TV	 `I heard "Turn on TV"` `Intent is turn_on_and_off` `Don't recognise the device 'TV'`
Wit.ai does not capture the user’s command, so it responds that no device is found.	 `I heard "Turn on"` `Intent is turn_on_and_off` `No device found`
Wit.ai misinterpreted the voice command into “kitten” instead of “kitchen”.	 `I heard "Turn on kitten"` `Intent is turn_on_and_off` `Don't recognise the device 'kitten'`

Table 4.5: Compilation of utterances set by the user and the snapshot of each utterance on the serial monitor log

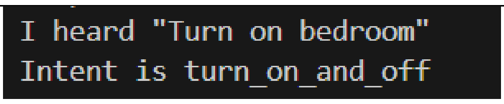
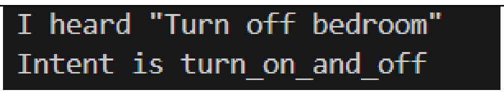
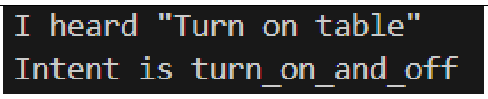
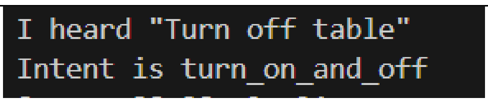
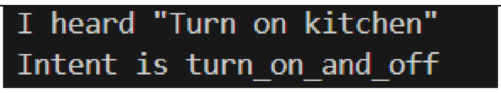
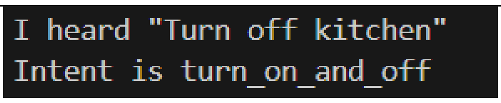
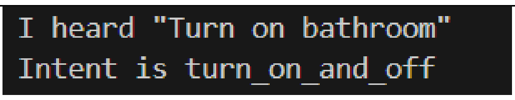
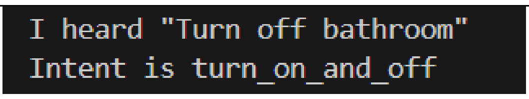
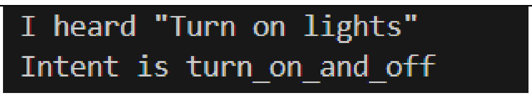
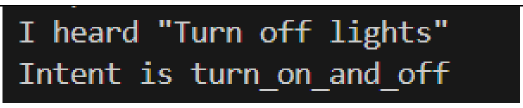
Area	Respond from Wit.ai
Bedroom	 `I heard "Turn on bedroom"` `Intent is turn_on_and_off`
	 `I heard "Turn off bedroom"` `Intent is turn_on_and_off`
Table	 `I heard "Turn on table"` `Intent is turn_on_and_off`
	 `I heard "Turn off table"` `Intent is turn_on_and_off`
Kitchen	 `I heard "Turn on kitchen"` `Intent is turn_on_and_off`
	 `I heard "Turn off kitchen"` `Intent is turn_on_and_off`
Bathroom	 `I heard "Turn on bathroom"` `Intent is turn_on_and_off`
	 `I heard "Turn off bathroom"` `Intent is turn_on_and_off`
Lights (control all lights of the area)	 `I heard "Turn on lights"` `Intent is turn_on_and_off`
	 `I heard "Turn off lights"` `Intent is turn_on_and_off`

Table 4.5 is the result of the system, where each of the lights of different areas can be controlled individually. Wit.ai responded to the user’s intention by turning on or off the desired area of the home environment. The device “lights” is used to turn on or off the lights for all areas.

4.6 Summary

This chapter presented and reviewed the findings from the smart home lighting control system implementation and testing, which used an ESP32 microcontroller with wake word detection. Using a lightweight convolutional neural network (CNN) model and MFCC feature extraction, the system was able to detect the wake word "Marvin" with great accuracy under controlled conditions. Upon successful verification, the user's voice command was processed by Wit.ai for intent recognition and entity extraction, which displayed consistent performance and a high success rate across different test situations.

ESP32 executes the interpreted commands accurately with minimal latency to turn on and off the LEDs that were directly attached to the GPIO pins. The system performed well in terms of latency and accuracy for all main components. The components include wake word detection, NLP interpretation, and the physical outcome based on the performance evaluation. The results demonstrated that the system achieves its goals and provides a responsive and easy-to-use voice-activated home automation solution.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

5.1 Summary of Research Objectives

The project was motivated by three major objectives. The objectives are achieved. The first objective is to design and develop a smart home automation system that is capable of reliable wake word detection using ESP32. A lightweight convolutional neural network (CNN) model that was trained to identify the wake word "Marvin" was integrated to perform the detection. In normal indoor environments, the accuracy of the system in detecting the wake word is 100% given by using a constant voice as input.

The second objective involved integrating TensorFlow Lite into the ESP32 environment for efficient on-device wake word detection. The model and MFCC feature extraction worked well together for real-time inference on the ESP32 microcontroller with a low latency of less than 8seconds.

The goal was to enable natural language understanding for smooth voice-controlled home lighting. This was accomplished by processing spoken commands after wake word activation using Wit.ai. With an average accuracy of 88.335%, the NLP engine correctly identified user intent and targeted devices like the “bathroom" and “bedroom”. Each recognized command was reliably mapped to a GPIO pin on the ESP32 to control the respective LED.

5.2 Conclusion

The ESP32 voice-activated smart home lighting system demonstrates that it is possible to integrate cloud-based natural language processing, lightweight machine learning, and GPIO hardware control into a small and affordable solution. By enabling the on-device wake word detection, the system ensures that user commands are only executed when they are specifically triggered. This improved privacy and also reduced the unnecessary processing. Furthermore, the incorporation of natural language processing with Wit.ai enabled customizable voice command variations without overloading the ESP32 hardware resources. For smart home environments, the system offers a hands-free, low-latency lighting control experience. Overall, the results demonstrate that the objectives were met. The system offers an excellent foundation for future extension into a comprehensive home automation solution.

5.3 Recommendations for Future Work

There are a few improvements that can be made for future work. The precision of wake word detection is one key aspect. The accuracy and responsiveness of the wake word model could be better by incorporating a range of voice kinds, accents, and background noises. The vast dataset may improve the rate of detection in a variety of contexts. Furthermore, employing noise filtering strategies or better microphones would improve detection reliability even more, particularly in larger spaces or busy surroundings.

It is advised to incorporate extra control features, such as physical switches, fans, air conditioners, or other home equipment, to increase the system's functionality. As a result, the system would be more adaptable and appropriate for a wider range of smart home uses.

Moreover, the implementation of feedback mechanisms might improve the user experience. For example, when a command is successfully performed, auditory confirmation, such as a buzzer, may be used. In the current system, the feedback is given in terms of the visualization of a blinking LED on the ESP32 board. Users would feel more confident that their request was received and handled appropriately as a result.

The inclusion of an emergency manual ON/OFF switch is a critical safety and usability consideration. A physical switch that is attached straight to the ESP32 GPIO pins would allow users to physically control the lighting even in the case of no internet connectivity. This guarantees that crucial lighting features continue to function even in the event of connectivity issues.

REFERENCES

Ali, H. *et al.* (2014) 'DWT features performance analysis for automatic speech recognition of Urdu', *SpringerPlus*, 3, p. 204. Available at: https://doi.org/10.1186/2193-1801-3-204

Arik, S.O. *et al.* (2017) 'Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting.', arXiv. Available at: https://doi.org/10.48550/arXiv.1703.05390

Babiuch, M. and Postulka, J. (2020) 'Smart Home Monitoring System Using ESP32 Microcontrollers', in *Internet of Things*. IntechOpen. Available at: https://doi.org/10.5772/intechopen.94589

Badade, A. *et al.* (2025) 'Smart Home Automation System with Voice Control Integration', *International Journal of Scientific Research and Engineering Development*, 8(3), pp. 2753-2762. Available at: https://ijsred.com/volume8/issue3/IJSRED-V8I3P441.pdf (Accessed: 27 September 2025)

Bagus, F.A., Puspa, E. and Safaruddin, H. A. I. (2021) 'Perancangan Aplikasi Chatbot Menggunakan Wit.Ai pada Sistem SPP-IRT Berbasis Web', *Jurnal Informatika Universitas Pamulang*, 6(4), pp. 785–794. Available at: https://doi.org/10.32493/informatika.v6i4.13327

Bajaj, A. *et al.* (2022) 'Comparative Wavelet and MFCC Speech Emotion Recognition Experiments on the RAVDESS Dataset', *Mathematical Statistician and Engineering Applications*, 71(3), pp. 1288–1293. Available at: https://doi.org/10.17762/msea.v71i3.468

Canziani, B. and MacSween, S. (2021) 'Consumer acceptance of voice-activated smart home devices for product information seeking and online ordering', *Computers in Human Behavior*, 119, p. 106714. Available at: https://doi.org/10.1016/j.chb.2021.106714

Chakraborty, A. *et al.* (2023) 'Smart Home System: A Comprehensive Review', Journal *of Electrical and Computer Engineering*, 2023(1), p. 7616683. Available at: https://doi.org/10.1155/2023/7616683

Chen, Y., Zhang, H. and Zhong, S. (2024) 'Design and implementation of smart home system based on IoT', *Results in Engineering*, 24, p. 103410. Available at: https://doi.org/10.1016/j.rineng.2024.103410

Chittepu, S., Martha, S. and Banik, D. (2025) 'Empowering voice assistants with TinyML for user-centric innovations and real-world applications', *Scientific Reports*, 15, p. 15411. Available at: https://doi.org/10.1038/s41598-025-96588-1

DOIT ESP32 DEVKIT V1 Development Board Details, Pinout (no date). Available at: https://www.espboards.dev/esp32/esp32doit-devkit-v1/#tools (Accessed: 22 September 2025).

Dua, M., Aggarwal, R.K. and Biswas, M. (2019) 'GFCC based discriminatively trained noise robust continuous ASR system for Hindi language', *Journal of Ambient Intelligence and Humanized Computing*, 10(6), pp. 2301–2314. Available at: https://doi.org/10.1007/s12652-018-0828-x

Elkholy, M.H. *et al.* (2022) 'Design and Implementation of a Real-Time Smart Home Management System Considering Energy Saving', *Sustainability*, 14(21), p. 13840. Available at: https://doi.org/10.3390/su142113840

Ezugwu, A.E. *et al.* (2025) 'Smart Homes of the Future', *Transactions on Emerging Telecommunications Technologies*, 36(1), p. e70041. Available at: https://doi.org/10.1002/ett.70041

Froiz-Míguez, I., Fraga-Lamas, P. and Fernández-CaraméS, T.M. (2023) 'Design, Implementation, and Practical Evaluation of a Voice Recognition Based IoT Home Automation System for Low-Resource Languages and Resource-Constrained Edge IoT Devices: A System for Galician and Mobile Opportunistic Scenarios', *IEEE Access*, 11, pp. 63623–63649. Available at: https://doi.org/10.1109/ACCESS.2023.3286391

Ghafurian, M., Ellard, C. and Dautenhahn, K. (2023) 'An investigation into the use of smart home devices, user preferences, and impact during COVID-19', *Computers in Human Behavior Reports*, 11, p. 100300. Available at: https://doi.org/10.1016/j.chbr.2023.100300

Goyal, Y. (2024) 'Comparative study of microcontrollers: Arduino UNO, Raspberry Pi 4, ESP32', International *Journal for Research in Applied Science & Engineering Technology*, 12(VII), pp. 588-592. Available at: https://doi.org/10.22214/ijraset.2024.63598

Iliev, Y. and Ilieva, G. (2023) 'A Framework for Smart Home System with Voice Control Using NLP Methods', *Electronics*, 12(1), p. 116. Available at: https://doi.org/10.3390/electronics12010116

“INMP441” (no date) *TDK InvenSense*. Available at: https://invensense.tdk.com/products/digital/inmp441/ (Accessed: 22 September 2025).

Irugalbandara, C. *et al.* (2023) 'A Secure and Smart Home Automation System with Speech Recognition and Power Measurement Capabilities', *Sensors (Basel, Switzerland)*, 23(13), p. 5784. Available at: https://doi.org/10.3390/s23135784

Kamińska, D., Sapiński, T. and Gholamreza, A. (2017) 'Efficiency of chosen speech descriptors in relation to emotion recognition', *EURASIP Journal on Audio Speech and Music Processing*, 2017(3). Available at: https://www.researchgate.net/publication/313861727_Efficiency_of_chosen_speech_descriptors_in_relation_to_emotion_recognition (Accessed: 28 September 2025).

Kang, S. *et al.* (2014) 'Lightweight Morphological Analysis Model for Smart Home Applications Based on Natural Language Interfaces', *International Journal of Distributed Sensor Networks*, 10(6), p. 570634. Available at: https://doi.org/10.1155/2014/570634

Kodali, R.K. *et al.* (2016) 'IoT based smart security and home automation system', in *2016 International Conference on Computing, Communication and Automation (ICCCA). 2016 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 1286–1289. Available at: https://doi.org/10.1109/CCAA.2016.7813916

Kundu, A. *et al.* (2022) 'HEiMDaL: Highly Efficient Method for Detection and Localization of wake-words', arXiv. Available at: https://doi.org/10.48550/arXiv.2210.15425

Lamaakal, I. *et al.* (2025) 'Tiny Language Models for Automation and Control: Overview, Potential Applications, and Future Research Directions', *Sensors (Basel, Switzerland)*, 25(5), p. 1318. Available at: https://doi.org/10.3390/s25051318

Litayem, N. (2024) 'Scalable Smart Home Management with ESP32-S3: A Low-Cost Solution for Accessible Home Automation', in *2024 International Conference on Computer and Applications (ICCA). 2024 International Conference on Computer and Applications (ICCA)*, pp. 1–7. Available at: https://doi.org/10.1109/ICCA62237.2024.10927887

Logan, B. (2000) 'Mel Frequency Cepstral Coefficients for Music Modeling' *Proc. 1st Int. Symposium Music Information Retrieval.* Available at: https://www.researchgate.net/publication/2552483_Mel_Frequency_Cepstral_Coefficients_for_Music_Modeling (Accessed: 25 September 2025)

Maier, A., Sharp, A. and Vagapov, Y. (2017) 'Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of thing', in *2017 Internet Technologies and Applications (ITA). 2017 Internet Technologies and Applications (ITA)*, pp. 143–148. Available at: https://doi.org/10.1109/ITECHA.2017.8101926

Meng, Y. *et al.* (2018) 'WiVo: Enhancing the Security of Voice Control System via Wireless Signal in IoT Environment', in *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*. New York, NY, USA: Association for Computing Machinery (Mobihoc '18), pp. 81–90. Available at: https://doi.org/10.1145/3209582.3209591

Natarajan, S. *et al.* (2025) 'Deep neural networks for speech enhancement and speech recognition: A systematic review', *Ain Shams Engineering Journal*, 16(7), p. 103405. Available at: https://doi.org/10.1016/j.asej.2025.103405

Naved, M. *et al.* (eds.) (2022) *IoT-enabled Convolutional Neural Networks: Techniques and Applications*. River Publishers. Available at: https://ieeexplore.ieee.org/book/9997412 (Accessed: 28 September 2025).

Opara, E. (2022) 'CLOUD-BASED MACHINE LEARNING AND SENTIMENT ANALYSIS', *Electronic Theses and Dissertations* [Preprint]. Available at: https://digitalcommons.georgiasouthern.edu/etd/2515

Qaffas, A.A. (2019) 'Improvement of Chatbots Semantics Using Wit.ai and Word Sequence Kernel: Education Chatbot as a Case Study', International *Journal of Modern Education and Computer Science*, 11(3), pp. 16–22. Available at: https://doi.org/10.5815/ijmecs.2019.03.03

Rashidi, M. (2022) *Application of TensorFlow lite on embedded devices : A hands-on practice of TensorFlow model conversion to TensorFlow Lite model and its deployment on Smartphone to compare model's performance*. Available at: https://urn.kb.se/resolve?urn=urn:nbn:se:miun:diva-46160 (Accessed: 19 September 2025).

Saputra, N.A. *et al.* (2024) 'A Systematic Review for Classification and Selection of Deep Learning Methods', *Decision Analytics Journal*, 12, p. 100489. Available at: https://doi.org/10.1016/j.dajour.2024.100489

Schweidtmann, A.M., Zhang, D. and von Stosch, M. (2024) 'A review and perspective on hybrid modeling methodologies', *Digital Chemical Engineering*, 10, p. 100136. Available at: https://doi.org/10.1016/j.dche.2023.100136

Shastry, K.A. and Shastry, A. (2023) 'An integrated deep learning and natural language processing approach for continuous remote monitoring in digital health', *Decision Analytics Journal*, 8, p. 100301. Available at: https://doi.org/10.1016/j.dajour.2023.100301

Singh, S. *et al.* (2024) 'Empowering homes through energy efficiency: A comprehensive review of smart home systems and devices', *International Journal of Energy Sector Management*, 19(4), pp. 887–912. Available at: https://doi.org/10.1108/IJESM-07-2024-0044

speech_commands | TensorFlow Datasets (no date) *TensorFlow*. Available at: https://www.tensorflow.org/datasets/catalog/speech_commands (Accessed: 22 September 2025).

Stefanović, I., Nan, E. and Radin, B. (2017) 'Implementation of the wake word for smart home automation system', in *2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin). 2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, pp. 271–272. Available at: https://doi.org/10.1109/ICCE-Berlin.2017.8210649

Syari, M.A., Dzaky, R.F. and Saragih, R. (2025) 'Integration of the Internet of Things in Smart Home Information Systems to Improve Security and Convenience', *Journal of Artificial Intelligence and Engineering Applications (JAIEA)*, 4(3), pp. 1772–1777. Available at: https://doi.org/10.59934/jaiea.v4i3.1010

Vlist, F. van der, Helmond, A. and Ferrari, F. (2024) 'Big AI: Cloud infrastructure dependence and the industrialisation of artificial intelligence', *Big Data & Society*, 11(1), p. 20539517241232630. Available at: https://doi.org/10.1177/20539517241232630

Warden, P. (2018) 'Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition', arXiv. Available at: https://doi.org/10.48550/arXiv.1804.03209

Xu, B. *et al.* (2021) 'A Benchmarking on Cloud based Speech-To-Text Services for French Speech and Background Noise Effect', in. *6th National Conference on Practical Applications of Artificial Intelligence*, Bordeaux, France: arXiv. Available at: https://doi.org/10.48550/arXiv.2105.03409

Xu, H. *et al.* (2025) 'Effective and Efficient Mixed Precision Quantization of Speech Foundation Models', arXiv. Available at: https://doi.org/10.48550/arXiv.2501.03643

Yüksel, M.E. (2020) 'Power Consumption Analysis of a Wi-Fi-based IoT Device', *ELECTRICA*, 20(1), pp. 62–70. Available at: https://doi.org/10.5152/electrica.2020.19081

Zhang, Y. *et al.* (2018) 'Hello Edge: Keyword Spotting on Microcontrollers', arXiv. Available at: https://doi.org/10.48550/arXiv.1711.07128

Zhantleuova, A.K., Makashev, Y.K. and Duzbayev, N.T. (2025) 'Optimizing MFCC Parameters for Breathing Phase Detection', *Sensors*, 25(16), p. 5002. Available at: https://doi.org/10.3390/s25165002

APPENDICES

Appendix A: Main Voice Control Script

```
#include <Arduino.h>
#include <WiFi.h>
#include <driver/i2s.h>
#include <esp_task_wdt.h>
#include "I2SMicSampler.h"
#include "ADCSampler.h"
#include "I2SOutput.h"
#include "config.h"
#include "Application.h"
#include "SPIFFS.h"
#include "IntentProcessor.h"
#include "Speaker.h"
#include "IndicatorLight.h"

// Define constants for the microphone setup
#define SAMPLE_BUFFER_SIZE 1600
#define SAMPLE_RATE 8000
#define I2S_MIC_CHANNEL I2S_CHANNEL_FMT_ONLY_LEFT

// Define GPIO pins for the INMP441 microphone
#define I2S_MIC_SERIAL_CLOCK GPIO_NUM_32
#define I2S_MIC_LEFT_RIGHT_CLOCK GPIO_NUM_25
#define I2S_MIC_SERIAL_DATA GPIO_NUM_33
```

```
// Filter window size for the moving average (e.g., 5 samples for a smoother signal)
#define FILTER_WINDOW_SIZE 5
// Data scaling factor to reduce the raw signal
#define DATA_SCALE_FACTOR 1000

// i2s config for using the internal ADC
i2s_config_t i2s_config = {
  .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX),
  .sample_rate = SAMPLE_RATE,
  .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,
  .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,
  .communication_format = I2S_COMM_FORMAT_STAND_I2S,
  .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
  .dma_buf_count = 4,
  .dma_buf_len = 1024,
  .use_apll = false,
  .tx_desc_auto_clear = false,
  .fixed_mclk = 0
};

// i2s config for reading from both channels of I2S
i2s_config_t i2sMemsConfigBothChannels = {
  .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX),
  .sample_rate = 16000,
  .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,
  .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,
  .communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_STAND_I2S),

  //.communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_I2S),
  .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
  .dma_buf_count = 4,
  .dma_buf_len = 64,
  .use_apll = false,
  .tx_desc_auto_clear = false,
.fixed_mclk = 0};

// i2s microphone pins
i2s_pin_config_t i2s_mic_pins = {
    .bck_io_num = I2S_MIC_SERIAL_CLOCK,
    .ws_io_num = I2S_MIC_LEFT_RIGHT_CLOCK,
    .data_out_num = I2S_PIN_NO_CHANGE,
    .data_in_num = I2S_MIC_SERIAL_DATA};

// i2s speaker pins
i2s_pin_config_t i2s_speaker_pins = {
    .bck_io_num = I2S_SPEAKER_SERIAL_CLOCK,
    .ws_io_num = I2S_SPEAKER_LEFT_RIGHT_CLOCK,
    .data_out_num = I2S_SPEAKER_SERIAL_DATA,
    .data_in_num = I2S_PIN_NO_CHANGE};

int32_t apply_moving_average_filter(int32_t* samples, int size)
{
    long sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum += samples[i];
    }
    return sum / size; // Return the average of the samples
}

// This task does all the heavy lifting for our application
void applicationTask(void *param)
{
  Application *application = static_cast<Application *>(param);

  const TickType_t xMaxBlockTime = pdMS_TO_TICKS(100);
  while (true)
```

```
    {
      // wait for some audio samples to arrive
      uint32_t ulNotificationValue = ulTaskNotifyTake(pdTRUE, xMaxBlockTime);
      if (ulNotificationValue > 0)
      {
        application->run();
      }
    }
}

void setup()
{
  Serial.begin(115200);
  delay(1000);
  Serial.println("Starting up");

  // start up wifi
  // launch WiFi
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PSWD);
  if (WiFi.waitForConnectResult() != WL_CONNECTED)
  {
    Serial.println("Connection Failed! Rebooting...");
    delay(5000);
    ESP.restart();
  }
  Serial.printf("Total heap: %d\n", ESP.getHeapSize());
  Serial.printf("Free heap: %d\n", ESP.getFreeHeap());

  // startup SPIFFS for the wav files
  SPIFFS.begin();
  if (!SPIFFS.begin(true)) {  // 'true' will auto-format if mount fails
    Serial.println("SPIFFS Mount Failed");
```

```
     return;
    }
   Serial.println("SPIFFS Mount Successful");

   File file = SPIFFS.open("/wake_word_marvin_lite.tflite");   //CHANGE FROM
my_model.tflite
   if (!file) {
    Serial.println("Failed to open file");
   }
   // make sure we don't get killed for our long running tasks
   esp_task_wdt_init(10, false);

   // start up the I2S input (from either an I2S microphone or Analogue microphone
via the ADC)
  #ifdef USE_I2S_MIC_INPUT
   // Direct i2s input from INMP441 or the SPH0645
   I2SSampler *i2s_sampler = new I2SMicSampler(i2s_mic_pins, false);
  #else
   // Use the internal ADC
      I2SSampler    *i2s_sampler    =    new    ADCSampler(ADC_UNIT_1,
ADC_MIC_CHANNEL);
  #endif

   // start the i2s speaker output
   I2SOutput *i2s_output = new I2SOutput();
   i2s_output->start(I2S_NUM_1, i2s_speaker_pins);
   Speaker *speaker = new Speaker(i2s_output);

   // indicator light to show when we are listening
   IndicatorLight *indicator_light = new IndicatorLight();
   // and the intent processor
   IntentProcessor *intent_processor = new IntentProcessor(speaker);
   intent_processor->addDevice("bathroom", GPIO_NUM_4);
```

```
    intent_processor->addDevice("kitchen", GPIO_NUM_5);
    intent_processor->addDevice("bedroom", GPIO_NUM_21);
    intent_processor->addDevice("table", GPIO_NUM_23);

    // create our application
     Application *application = new Application(i2s_sampler, intent_processor,
speaker, indicator_light);
    // set up the i2s sample writer task
    TaskHandle_t applicationTaskHandle;
     xTaskCreate(applicationTask, "Application Task", 8192, application, 1,
&applicationTaskHandle);
    // start sampling from i2s device - use I2S_NUM_0 as that's the one that supports
the internal ADC
  #ifdef USE_I2S_MIC_INPUT
          i2s_sampler->start(I2S_NUM_0,          i2sMemsConfigBothChannels,
applicationTaskHandle);
  #else
    i2s_sampler->start(I2S_NUM_0, adcI2SConfig, applicationTaskHandle);
  #endif
  }
  int32_t raw_samples[SAMPLE_BUFFER_SIZE];

  void loop()
  {
    size_t bytes_read = 0;
        i2s_read(I2S_NUM_0,     raw_samples,     sizeof(int32_t)     *
SAMPLE_BUFFER_SIZE, &bytes_read, portMAX_DELAY);
    int samples_read = bytes_read / sizeof(int32_t);

    // Print raw data with scaling for easier visualization
    Serial.print("Raw Data: ");
    for (int i = 0; i < samples_read; i++)
    {
      // Scale the data to make it easier to visualize
      int32_t scaled_value = raw_samples[i] / DATA_SCALE_FACTOR;
      Serial.print(scaled_value);
      Serial.print(", ");
    }
    Serial.println(); // Print a new line after each batch of data

    // Apply the moving average filter and print the filtered values
    Serial.print("Filtered Data: ");
    for (int i = 0; i < samples_read; i++)
    {
      int start_index = max(i - FILTER_WINDOW_SIZE + 1, 0);
                                          int32_t             filtered_value             =
apply_moving_average_filter(&raw_samples[start_index],      min(i      +      1,
FILTER_WINDOW_SIZE));
       int32_t scaled_filtered_value = filtered_value / DATA_SCALE_FACTOR; //
Apply scaling for filtered data
      Serial.print(scaled_filtered_value);
      Serial.print(", ");
    }
    Serial.println(); // Print a new line after each batch of filtered data

    vTaskDelay(1000);
  }
```

Appendix B: Intent Processor Script

```
#include <Arduino.h>
#include "IntentProcessor.h"
#include "Speaker.h"

IntentProcessor::IntentProcessor(Speaker *speaker)
{
  m_speaker = speaker;
}
IntentResult IntentProcessor::turnOnDevice(const Intent &intent)
{
  if (intent.intent_confidence < 0.8)
  {
    Serial.printf("Only %.f%% certain on intent\n", 100 * intent.intent_confidence);
    return FAILED;
  }
  if (intent.device_name.empty())
  {
    Serial.println("No device found");
    return FAILED;
  }
  if (intent.device_confidence < 0.8)
  {
            Serial.printf("Only   %.f%%   certain   on   device\n",   100   *
intent.device_confidence);
    return FAILED;
  }
  if (intent.trait_value.empty())
  {
    Serial.println("Can't work out the intent action");
    return FAILED;
  }
```

```
    if (intent.trait_confidence < 0.8)
    {
        Serial.printf("Only %.f%% certain on trait\n", 100 * intent.trait_confidence);
        return FAILED;
    }
    bool is_turn_on = intent.trait_value == "on";
    // global device name "lights"
    if (intent.device_name == "lights")
    {
        for (const auto &dev_pin : m_device_to_pin)
        {
            digitalWrite(dev_pin.second, is_turn_on);
        }
    }
    else
    {
        // see if the device name is something we know about
        if (m_device_to_pin.find(intent.device_name) == m_device_to_pin.end())
        {
            Serial.printf("Don't recognise the device '%s'\n", intent.device_name.c_str());
            return FAILED;
        }
        digitalWrite(m_device_to_pin[intent.device_name], is_turn_on);
    }
    // success!
    return SUCCESS;
}
IntentResult IntentProcessor::tellJoke()
{
    m_speaker->playRandomJoke();
    return SILENT_SUCCESS;
}
IntentResult IntentProcessor::life()
```

```
{
   m_speaker->playLife();
   return SILENT_SUCCESS;
}

IntentResult IntentProcessor::processIntent(const Intent &intent)
{
   if (intent.text.empty())
   {
      Serial.println("No text recognised");
      return FAILED;
   }
   Serial.printf("I heard \"%s\"\n", intent.text.c_str());
   if (intent.intent_name.empty())
   {
      Serial.println("Can't work out what you want to do with the device...");
      return FAILED;
   }
   Serial.printf("Intent is %s\n", intent.intent_name.c_str());
   if (intent.intent_name == "turn_on_and_off")
   {
      return turnOnDevice(intent);
   }
   return FAILED;
}

void IntentProcessor::addDevice(const std::string &name, int gpio_pin)
{
   m_device_to_pin.insert(std::make_pair(name, gpio_pin));
   pinMode(gpio_pin, OUTPUT);
}
```

Appendix C: Wit.ai Chunked Uploader Script

```
#include "WitAiChunkedUploader.h"
#include "WiFiClientSecure.h"
#include <ArduinoJson.h>

WitAiChunkedUploader::WitAiChunkedUploader(const char *access_key)
{
   m_wifi_client = new WiFiClientSecure();

   m_wifi_client->setInsecure();

   m_wifi_client->connect("api.wit.ai", 443);

   Serial.println("Connected to server");
   char authorization_header[100];
   snprintf(authorization_header, 100, "authorization: Bearer %s", access_key);
   m_wifi_client->println("POST /speech?v=20200927 HTTP/1.1");
   m_wifi_client->println("host: api.wit.ai");
   m_wifi_client->println(authorization_header);
   m_wifi_client->println("content-type: audio/raw; encoding=signed-integer;
bits=16; rate=16000; endian=little");
   m_wifi_client->println("transfer-encoding: chunked");
   m_wifi_client->println();
}

bool WitAiChunkedUploader::connected()
{
   return m_wifi_client->connected();
}

void WitAiChunkedUploader::startChunk(int size_in_bytes)
{
   m_wifi_client->printf("%X\r\n", size_in_bytes);
}

void WitAiChunkedUploader::sendChunkData(const uint8_t *data, int
size_in_bytes)
{
   m_wifi_client->write(data, size_in_bytes);
}

void WitAiChunkedUploader::finishChunk()
{
   m_wifi_client->print("\r\n");
}

Intent WitAiChunkedUploader::getResults()
{
   // finish the chunked request by sending a zero length chunk
   m_wifi_client->print("0\r\n");
   m_wifi_client->print("\r\n");
   // get the headers and the content length
   int status = -1;
   int content_length = 0;
   while (m_wifi_client->connected())
   {
       char buffer[255];
       int read = m_wifi_client->readBytesUntil('\n', buffer, 255);
       if (read > 0)
       {
           buffer[read] = '\0';
           // blank line indicates the end of the headers
           if (buffer[0] == '\r')
           {
               break;
           }
           if (strncmp("HTTP", buffer, 4) == 0)
           {
               sscanf(buffer, "HTTP/1.1 %d", &status);
           }
           else if (strncmp("Content-Length:", buffer, 15) == 0)
           {
               sscanf(buffer, "Content-Length: %d", &content_length);
           }
       }
   }
   Serial.printf("Http status is %d with content length of %d\n", status,
content_length);
   if (status == 200)
   {
       StaticJsonDocument<500> filter;
       filter["entities"]["device:device"][0]["value"] = true;
       filter["entities"]["device:device"][0]["confidence"] = true;
       filter["text"] = true;
       filter["intents"][0]["name"] = true;
       filter["intents"][0]["confidence"] = true;
       filter["traits"]["wit$on_off"][0]["value"] = true;
       filter["traits"]["wit$on_off"][0]["confidence"] = true;
       StaticJsonDocument<500> doc;
       deserializeJson(doc, *m_wifi_client, DeserializationOption::Filter(filter));

       const char *text = doc["text"];
       const char *intent_name = doc["intents"][0]["name"];
       float intent_confidence = doc["intents"][0]["confidence"];
       const char *device_name = doc["entities"]["device:device"][0]["value"];
       float device_confidence = doc["entities"]["device:device"][0]["confidence"];
       const char *trait_value = doc["traits"]["wit$on_off"][0]["value"];
       float trait_confidence = doc["traits"]["wit$on_off"][0]["confidence"];
       return Intent{
           .text = (text ? text : ""),
           .intent_name = (intent_name ? intent_name : ""),
           .intent_confidence = intent_confidence,
           .device_name = (device_name ? device_name : ""),
           .device_confidence = device_confidence,
           .trait_value = (trait_value ? trait_value : ""),
           .trait_confidence = trait_confidence};
   }
   return Intent{};
}

WitAiChunkedUploader::~WitAiChunkedUploader()
{
   delete m_wifi_client;
}
```

Appendix D: Detect Wake Word State Script

```
#include <Arduino.h>
#include "I2SSampler.h"
#include "AudioProcessor.h"
#include "NeuralNetwork.h"
#include "RingBuffer.h"
#include "DetectWakeWordState.h"

#define WINDOW_SIZE 320
#define STEP_SIZE 160
#define POOLING_SIZE 6
#define AUDIO_LENGTH 16000

DetectWakeWordState::DetectWakeWordState(I2SSampler *sample_provider)
{
   // save the sample provider for use later
   m_sample_provider = sample_provider;
   // some stats on performance
   m_average_detect_time = 0;
   m_number_of_runs = 0;
}
void DetectWakeWordState::enterState()
{
   // Create our neural network
   m_nn = new NeuralNetwork();
   Serial.println("Created Neral Net");
   // create our audio processor
   m_audio_processor = new AudioProcessor(AUDIO_LENGTH, WINDOW_SIZE,
STEP_SIZE, POOLING_SIZE);
   Serial.println("Created audio processor");
   m_number_of_detections = 0;
}
```

```
bool DetectWakeWordState::run()
{
  // Time how long this takes for stats
  long start = millis();
  // Get access to the samples that have been read in
  RingBufferAccessor *reader = m_sample_provider->getRingBufferReader();
  // Rewind by 1 second
  reader->rewind(16000);
  // Get hold of the input buffer for the neural network so we can feed it data
  float *input_buffer = m_nn->getInputBuffer();
  // Process the samples to get the spectrogram
  m_audio_processor->get_spectrogram(reader, input_buffer);

  Serial.print("Input Buffer: ");
  for (int i = 0; i < 10; i++) {
    Serial.print(input_buffer[i], 6);  // Print the first 10 values with 6 decimal places
    Serial.print(" ");
  }
  Serial.println();
  // Finished with the sample reader
  delete reader;
  // Get the prediction for the spectrogram
  float output = m_nn->predict();

  // Debug the prediction score
  Serial.printf("NN Output Score: %.4f\n", output);  // Add this line
  long end = millis();
  // Compute the stats
  m_average_detect_time = (end - start) * 0.1 + m_average_detect_time * 0.9;
  m_number_of_runs++;
  // Log out some timing info
  if (m_number_of_runs == 100)
```

```
    {
        m_number_of_runs = 0;
        Serial.printf("Average detection time %.fms\n", m_average_detect_time);
    }

    // Check if the prediction exceeds threshold
    if (output > 0.85)              //change
    {
        m_number_of_detections++;
        if (m_number_of_detections > 1)
        {
            m_number_of_detections = 0;
            // Detected the wake word in several runs, move to the next state
            Serial.printf("P(%.2f): Here I am, brain the size of a planet...\n", output);
            return true;  // Trigger state change
        }
    }
    // Nothing detected, stay in the current state
    return false;
}

void DetectWakeWordState::exitState()
{
    // Create our neural network
    delete m_nn;
    m_nn = NULL;
    delete m_audio_processor;
    m_audio_processor = NULL;
    uint32_t free_ram = esp_get_free_heap_size();
    Serial.printf("Free ram after DetectWakeWord cleanup %d\n", free_ram);
}
```