ESP-Assistant

ESP32 Control Using LLM Generated Hardware Code

Graduation Project Thesis
Submitted to
Computer Engineering Department

Faculty of Engineering
Pharos University in Alexandria

June 2024

ESP-Assistant: ESP32 Control Using LLM Generated Hardware Code

By

Abdelmeniem Nasr Abdelmeniem

Ahmed Bilal Zaki

Khaled Eldesoukey Mohamed

Mahmoud Mohamed Abdo

Mohamed Ahmed Abdellatif

Supervisors

Prof. Dr. Hicham Elmongui

ABSTRACT

This thesis addresses the recent trend of increasing interest in Internet of Things (IoT) among enthusiasts and the accompanying need for entry-level hardware control solutions to facilitate the involvement of newcomers in the field. We identify the ESP32 microcontroller as a prominent starting point due to its popularity, affordability, versatility, and open-source nature. To bridge the gap between novice users and hardware control, we propose a methodology that leverages the ESP32 alongside AWS and Gemini to generate hardware code seamlessly.

Our approach includes the development of a mobile app and installer, providing users with intuitive interfaces to interact with the ESP32 without requiring extensive programming knowledge. By utilizing AWS and Gemini, we enable users to generate hardware code through simple prompts, further lowering the barrier to entry for IoT development.

The results of our methodology are demonstrated through the creation of three applications: a remotely controlled car, a reactive sensor station, and a heart rate monitor. These applications showcase the versatility and accessibility of our no-code tool, highlighting its potential to empower individuals with limited technical expertise to engage in IoT projects.

Moreover, our work underscores the implications of Gemini's capability to understand and address niche use cases through basic prompts. By facilitating the translation of user requirements into functional hardware code, Gemini opens doors for innovation and experimentation in IoT development, ultimately contributing to the democratization of technology.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
INTRODUCTION

I. INTRODUCTION

A. Overview

The world of the Internet of Things (IoT) has experienced a remarkable surge in popularity in recent times [1]. With its promise of connecting devices and enabling data-driven decision-making, IoT applications have attracted interest from various fields. However, a significant barrier to entry for many individuals and industries is the requirement for coding expertise to develop and deploy IoT solutions effectively.

Simultaneously, Language Model Models (LLMs) have emerged as powerful tools in the realm of software development. These models, including Gemini Pro, boast capabilities such as code generation, documentation creation, and error detection [2]. These advancements hold the potential to streamline development processes and make software engineering more accessible to a broader audience.

Gemini Pro, in particular, has garnered attention for its ambitious promises, including advanced code generation capabilities [2]. It claims to revolutionize software development by automating repetitive tasks and enabling developers to focus more on problem-solving than on writing boilerplate code.

There have even been many popular trends of AI Software Developers meant to replace the normal software developer, most notable of which has been Devin [3]. Though it has to be said, claims of the accuracy of Devin have been brought into question [4][5]. Therefore using LLMs as a single source of development seems to be a risky endeavor.

However, Given the gap between the growing demand for IoT solutions and the coding skills required to create them, there is a need to explore alternative approaches to IoT development. Leveraging the capabilities of LLMs presents a promising avenue to address this challenge. By harnessing the power of popular LLM APIs, we aim to create a hardware code generator that can simplify the process of developing IoT applications.

Our decision to investigate Gemini Pro's capabilities stems from its potential to bridge this gap effectively. However, it is essential to critically assess the claims made by such tools and evaluate their practical implications in real-world scenarios. As there have been many cases where AI has decreased code quality [6]. Therefore, we embarked on this study to validate the promises of LLMs in the context of hardware code generation for IoT applications.

B. Background

The Internet of Things (IoT) has rapidly evolved from a futuristic concept to a tangible reality, with the number of connected devices and systems exploding every year [1]. This network of interconnected devices is increasingly impacting our lives, from our homes and workplaces to unforeseen applications [1]. The growth of IoT is driven by the ability of devices to sense, connect, communicate, capture, and interpret data, enabling automation, data-driven insights, and intelligent environments [1].

This rapid adoption of IoT is expected to continue for at least the next five years, leading to significant implications for various sectors [1]. Gartner defines IoT as "a network of dedicated physical objects (things) that contain embedded technology to communicate and sense or interact with their internal states or the external environment" [1]. This network enables companies to capture data and events, learn user behavior and usage patterns, and react with preventive actions or adjust business processes. The IoT is essentially a foundational capability for creating a digital business [1].

As IoT devices proliferate across industries, three key areas will be of increasing interest: workplace safety, employee productivity, and security.

IoT devices are being used for data collection, environmental monitoring, and predictive maintenance, providing insights into potential hazards and enabling proactive safety measures [1]. For example, data collection devices can monitor employee mental and occupational health, while environmental sensors can provide feedback to HVAC systems to ensure comfortable working conditions and energy efficiency [1]. Maintenance sensors can monitor machines and predict potential failures, preventing accidents and downtime [1].

IoT devices are streamlining tasks, reducing manual work, and providing real-time data for optimized workflows [1]. For example, manufacturers in warehouses can accurately track available resources and monitor their supply chain, while smart tags and RFID sensors can efficiently track and control inventory [1]. Wearable devices are even being used to track employee actions and provide data to management, allowing for real-time adjustments to activities on the factory floor or in the conference room [1].

The rise of IoT raises concerns about security, as each connected device poses a potential gateway for cyberattacks [1]. However, IoT devices can also be used to monitor and mitigate security threats, ensuring the safety of both digital and physical assets [1]. This includes 24/7 monitoring of digital devices and networks, sensors and cameras, and wearable devices for employees [1]. These devices can help sense and predict possible cyberattacks, and proactively mitigate harmful situations for employees [1].

Large Language Models (LLMs) represent a significant advancement in artificial intelligence and natural language processing [3]. Models like OpenAI's GPT series and Google's BERT are trained on vast datasets to understand and generate human-like text [2]. LLMs are capable of a wide range of tasks, including text completion, translation, summarization, and even code generation [2]. These models learn intricate patterns in language and context, allowing them to generate coherent and relevant text based on user prompts [2].

Code generation using LLMs involves providing a prompt describing the desired code functionality to the model. The model then generates code snippets based on the input. Prompt engineering plays a crucial role in guiding the model to produce the desired output effectively [2]. This approach allows developers to leverage LLM capabilities for automating certain aspects of software development, accelerating the coding process and minimizing manual effort [2].

However, using LLMs for IoT hardware code generation presents specific challenges. The quality of code generated by LLMs can vary significantly, leading to inconsistencies in performance and reliability [5]. This unpredictability creates a significant obstacle for developers relying on these tools for hardware projects, as it can result in time-consuming debugging and troubleshooting [6]. Moreover, even when the generated code is correct, effectively uploading it to the target hardware can be a challenge, especially in no-code scenarios. Seamless integration between the generated code and the hardware platform is critical for achieving the desired functionality without requiring extensive coding knowledge from end-users.

The rise of AI-powered code assistants like Copilot has led to a significant increase in code generation, raising concerns about potential impacts on code quality [6]. A recent study analyzed over 150 million lines of code and found an uptick in code churn and a decrease in code reuse since the dawn of LLM-driven software development [6]. While correlation does not equal causation, the study suggests that blindly relying on AI outputs without proper oversight can contribute to lower code quality [6].

These challenges highlight the need for a robust and reliable solution to address the inconsistency and integration issues associated with using LLMs for IoT hardware code generation. This research aims to develop a system that can seamlessly generate code for the ESP32, a popular microcontroller commonly used in IoT projects, while ensuring the quality and uploadability of the generated code. By addressing these limitations, this research aims to facilitate wider adoption of LLMs for IoT development, making hardware programming more accessible to a broader range of users, from enthusiasts to professionals.

Chapter II
METHODOLOGY

II. METHODOLOGY

A. Hardware

1) The ESP32

The ESP32 is a powerful and versatile microcontroller that has gained immense popularity in the maker and IoT communities for several reasons [7][9]. First and foremost, its affordability makes it accessible to a wide range of enthusiasts and professionals alike [7][9]. Despite its low cost, the ESP32 boasts a robust feature set, making it suitable for a diverse range of applications [7][9].

Table I [7][8][9][10][11][12][13][14][15][16]
Table of Comparison between many microcontrollers.

Name	Arduino Pro	Arduino Micro	ESP32	ESP8266	Raspberry Pi 4
CPU Speed	8 MHz	16 MHz	240 MHz	160 MHz	1.8 GHz
Analog In/Out	06/0	12/0	18/13	1/0	0/0
Digital IO/PWM	14/06	20/07	34/16	17/05	40/04
EEPROM [kB]	0.512	1	448	0	0
SRAM [kB]	1	2.5	520	160	8000000
Flash [kB]	16	32	4000	1000	0
Bluetooth	-	-	BLE 4.2	-	BLE 5.0
WiFi	-	-	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz
Price in L.E	300	270	400	150	7500

One of the standout features of the ESP32 is its built-in Wi-Fi and Bluetooth connectivity[7]. This allows developers to create IoT devices that can seamlessly connect to the internet and communicate with other devices or cloud services. The integration of Wi-Fi and Bluetooth functionality simplifies the development process and expands the possibilities for creating interconnected systems.

Another factor contributing to the popularity of the ESP32 is its compatibility with the Arduino ecosystem[17]. With extensive support for Arduino libraries and the Arduino Integrated Development Environment (IDE), developers can leverage familiar tools and resources to program the ESP32. This lowers the barrier to entry for those already familiar with Arduino programming and facilitates rapid prototyping and development.

Furthermore, the ESP32 is an open-source platform, which means that the hardware specifications and software resources are freely available for anyone to study, modify, and contribute to. This openness fosters innovation and collaboration within the community, leading to the development of a wide range of projects and applications using a multitude of peripheral devices. Examples of Common Peripheral Devices that we will use during our project are as follows.

In the domain of electronics and robotics, the integration of various components such as servo motors, DC motors, photoelectric sensors, H-Bridge circuits, and DHT sensors with microcontroller platforms like ESP and Arduino has been a cornerstone for developing sophisticated and interactive projects. These components, each with its unique characteristics and functionalities, play pivotal roles in the design and execution of systems aimed at automation, environmental monitoring, and robotics. This discourse aims to delve deeper into the technical aspects, including pin diagrams and descriptions, and explore their applications in ESP and Arduino projects, thereby providing a comprehensive understanding of their significance in the field.

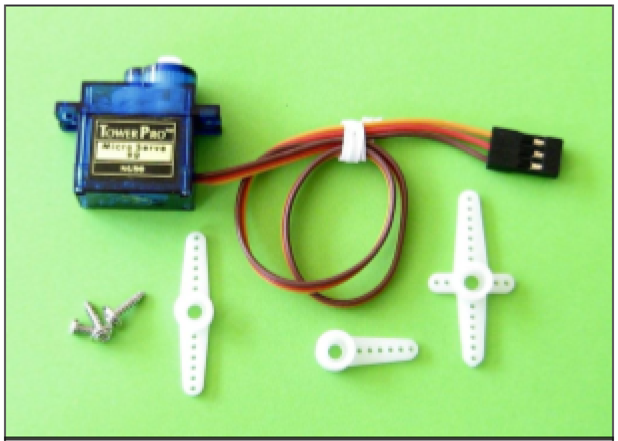
Fig 1. Servo Motor SG-90

Servo motors are distinguished by their ability to provide precise control of angular position, speed, and acceleration[18]. This precision is made possible through a feedback loop that typically involves a potentiometer for position feedback, an internal control circuit, and a DC motor[18]. The standard servo motor comes with three wires: power (red), ground (black or brown), and control (yellow or white). The control wire receives a PWM signal, which dictates the motor's position. In the context of ESP and Arduino, servo motors are controlled using libraries such as the `Servo` library in Arduino, which abstracts the complexities of PWM signal generation, allowing developers to specify the desired angle with simple function calls. Servo motors find extensive applications in projects requiring precise positioning, such as robotic arms, automated doors, and camera gimbals.

DC motors, characterized by their two-wire connection (positive and negative), convert electrical energy into mechanical motion with a speed that is proportional to the applied voltage[31]. Controlling these motors' speed and direction in ESP and Arduino projects necessitates the use of H-Bridge circuits and PWM signals. An H-Bridge circuit, comprising four switches arranged in an H configuration, enables current to flow through the motor in both directions, thus allowing for directional control[19]. Libraries such as `Adafruit_MotorShield` for Arduino simplify the implementation of speed and direction control by providing high-level functions to manage the underlying H-Bridge circuitry. DC motors are ubiquitous in projects that require movement, including mobile robots, fans, and pumps.

Photoelectric sensors, which operate by emitting and detecting light, are used to sense the presence, absence, or distance of objects. These sensors typically have three connections: power, ground, and signal output. The signal output changes based on the interruption of the light beam, which can be easily interfaced with ESP and Arduino digital pins to trigger events or actions in a project. Applications of photoelectric sensors are vast, ranging from automated manufacturing lines and object sorting systems to user interaction devices where gesture control is desired.

The H-Bridge circuit's role in controlling the direction of DC motors has been briefly touched upon. It is worth noting that the circuit can also be used to implement braking and speed control through PWM. The ability to control motor direction and speed precisely is crucial in applications such as autonomous vehicles, robotic arms, and any system requiring controlled mechanical movement.

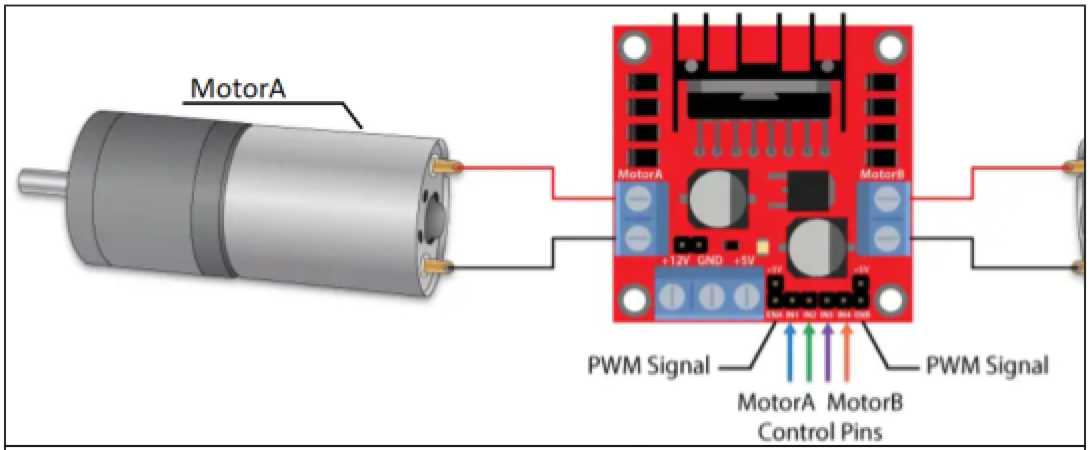

Fig 2. H-bridge with motor attached to it

DHT sensors, specifically the DHT11 and DHT22, offer a simple yet effective solution for measuring temperature and humidity[20]. These sensors have a minimalistic interface, requiring only a single digital pin for data communication, in addition to power and ground. The digital output from these sensors can be directly read by ESP and Arduino devices using libraries like `DHTlib`, which parse the data into temperature and humidity values. This simplicity and ease of use make DHT sensors ideal for environmental monitoring projects, smart agriculture systems, and home automation solutions where climate control is necessary.

The HC-SR04 ultrasonic sensor is a staple in distance measurement and object detection applications[21]. This sensor operates by emitting an ultrasonic sound wave at a frequency of 40 kHz, which travels through the air. If there is an object in its path, the sound bounces back to the sensor, which then calculates the time taken for the echo to return. Given the speed of sound in air, the distance to the object can be accurately calculated.

The HC-SR04 has four pins: VCC (power), Trig (trigger), Echo (receive), and GND (ground)[21]. To use the sensor, the Trig pin is pulsed for 10 microseconds, which sends out the ultrasonic burst. The Echo pin then goes high until the sound wave returns, and the duration for which this pin remains high is proportional to the distance. In Arduino applications, this sensor is widely used for obstacle avoidance in robotics, liquid level measurement, and as a component in automated parking systems. Libraries and functions within the Arduino IDE simplify the process of calculating distance, making the HC-SR04 a popular choice for hobbyists and professionals alike.

The AD8232 is a specialized heart rate monitor, designed for electrocardiography (ECG)[22]. It is an integrated signal conditioning block for ECG and other biopotential measurement applications. This module is adept at amplifying the small electrical changes that

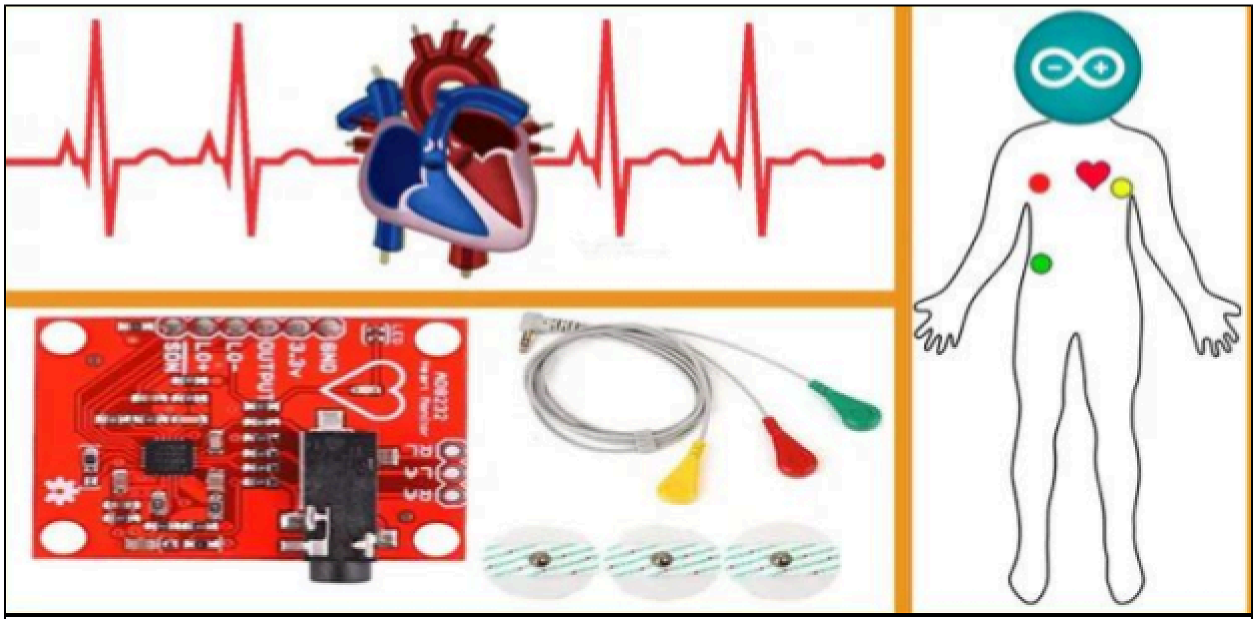

Fig 3. ECG AD8232 Heart Rate Monitor

are a consequence of the heart muscle's electrophysiological pattern.

The AD8232 module typically features pins for connecting electrodes, a 3.3V power supply, ground, and analog output pins that interface with a microcontroller's analog input pins. The output of the AD8232 is an analog signal that corresponds to the electrical activity of the heart. This signal can be processed and analyzed to monitor heart rate and to detect abnormalities in the ECG pattern. When used with platforms like Arduino, the AD8232 enables the development of portable health monitoring devices, fitness trackers, and even sophisticated diagnostic tools. The simplicity of interfacing and the availability of open-source libraries for signal processing make it an invaluable tool for projects at the intersection of technology and healthcare.

Oximeters, particularly pulse oximeters, are devices that measure the oxygen saturation level (SpO2) in the blood and heart rate. These devices work by passing light through a body part, usually a fingertip or earlobe, and measuring the amount of light absorbed by the blood. This absorption varies with the level of oxygen saturation, allowing the device to calculate SpO2 levels.

While standalone pulse oximeters are common in medical settings, integrating oximetry functionality into Arduino or ESP projects involves using modules like the MAX30100 or MAX30102[23]. These modules combine two LEDs, a photodetector, optimized optics, and low-noise analog signal processing to detect pulse oximetry and heart-rate signals. They typically feature an I2C interface, making them easy to connect to microcontrollers. The data obtained can be used for health monitoring in wearable devices, remote patient monitoring systems, and in projects that require real-time monitoring of blood oxygen levels alongside other physiological parameters.

In conclusion, the synergy between these components and microcontroller platforms such as ESP and Arduino enables the creation of complex, interactive systems that can sense and respond to their environment in intelligent ways. Whether it's through the precise control of motors, the detection of objects, or the monitoring of environmental conditions, these components are instrumental in pushing the boundaries of what's possible in electronics, robotics, and automation projects. Their versatility and ease of integration underscore their importance in both educational contexts and professional applications, fostering innovation and creativity in the field.

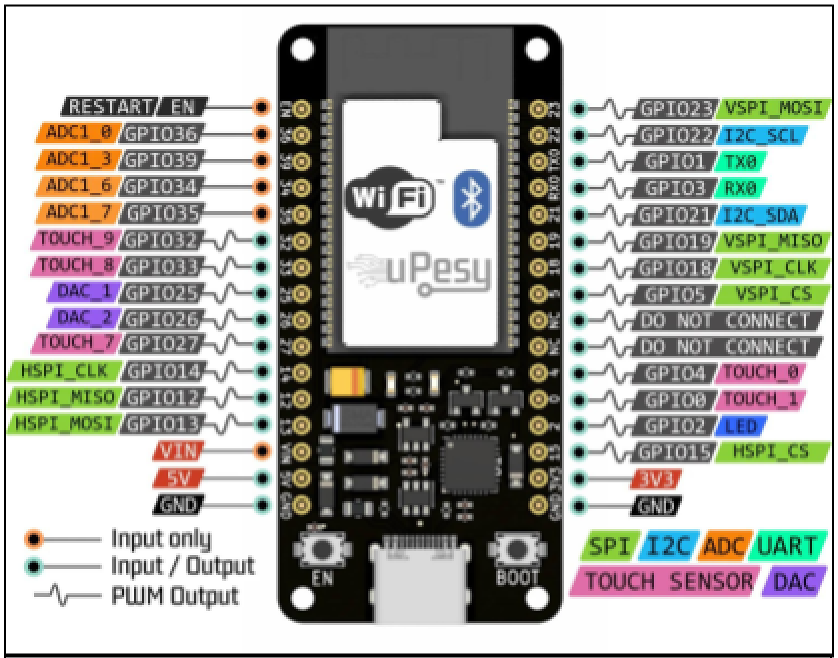

Fig 4. ESP32 Pinout

2) WifiPair Functionality

However to interface our generated code with these peripherals, we need a method to send this code to our ESP32, which requires a wifi connection. Therefore, the WifiPair functionality plays a critical role in simplifying the Wi-Fi configuration process for our ESP32 device[7], particularly when aiming for seamless integration with mobile applications.

The initial approach focused on Bluetooth pairing, where both the phone and the ESP32 subscribed to the same "point" to establish communication. However, this method presented a significant obstacle, Bluetooth pairing, while seemingly convenient, resulted in a considerable increase in memory usage on the ESP32. This poses a challenge, especially for resource-constrained microcontrollers like the ESP32, where memory limitations can restrict the complexity and functionality of projects. Implementing Bluetooth pairing might have necessitated sacrificing other functionalities or features due to memory constraints as can be seen from (Table II).

To overcome the memory limitations associated with Bluetooth pairing, the project adopted WifiPair, a Wi-Fi-based solution. WifiPair leverages the ESP32's built-in Wi-Fi capabilities to establish a connection between the device and the mobile application. The steps for pairing are as follows.

The initial step involves configuring the ESP32 to operate in station mode. In this mode, the ESP32 acts as a Wi-Fi client, actively searching for available Wi-Fi networks.

To facilitate the connection process, the ESP32 creates a temporary Wi-Fi hotspot. This hotspot is typically unsecured and serves the sole purpose of enabling the mobile application to establish a temporary connection with the ESP32.

Once the mobile application connects to the ESP32's hotspot, it sends a GET request to a specific IP address on the ESP32. This IP address is pre-configured on the ESP32 and signifies the endpoint for receiving the GET request. Upon receiving the request, the ESP32 scans for nearby Wi-Fi networks and compiles a list of their SSIDs (network names).

The mobile application receives the list of available Wi-Fi networks from the ESP32. The user can then select the desired network from this list within the mobile application interface. Additionally, the user enters the corresponding password for the chosen network.
The mobile application transmits the selected network SSID and password back to the ESP32. Armed with these credentials, the ESP32 attempts to connect to the specified Wi-Fi network.

If the connection is established successfully, the ESP32 is now integrated into the user's existing Wi-Fi network. Communication between the mobile application and the ESP32 can now proceed over this network, enabling further interaction and control from the mobile app.
WifiPair offers several advantages over Bluetooth pairing in this specific scenario. By leveraging Wi-Fi, WifiPair eliminates the memory overhead associated with maintaining a Bluetooth connection. This frees up valuable resources on the ESP32, allowing for a more feature-rich and complex codebase.

Table II.
BLE vs WIFI vs Cellular Data Comparison [24][25]

Feature	BLE (Low power Bluetooth)	WIFI	Cellular Data (4G)
Range	Short (less than 10 m)	Longer (Up to 45 m)	Longest (Up to 600 m)
Data Transfer Speed	Moderate (Up to 3 Mbps)	High (Up to 10 Gbps)	Highest (Up to 20Gbps)
Security	Moderate	Strong	Strong
Power Usage	Low	Moderate to High	Slightly higher
User-Friendly Setup	Simple pairing process	May involve configuration	Certain hardware needed
Mobility	High mobility good for on-the-go	Limited mobility, Best for stationary and semi-stationary devices	Limitless moblility (Especially 5G)
Location Detection	Provides location data , but less precise	Provides location data, but accuracy varies	Provides location data, with accurate measurement
Compatibility with Existing Infrastructure	Integrates well with existing devices	Compatible with existing WiFi networks	Compatible with existing cell towers
Cost-Effectiveness	Most cost-effective	Cost-effective ,but adds firmware complexity	More expensive on a per-bit
Total Firmware Size	Large (1,475,717 Bytes)	Small (886,105 Bytes)	Most memory efficient (260,645 Bytes)

WifiPair provides a user-friendly experience for configuring the ESP32's Wi-Fi connection. Users can simply select their desired network from a list displayed on the mobile app and enter the password, eliminating the need for manual configuration on the ESP32 itself.

WifiPair is not limited to a single Wi-Fi network. The mobile application can potentially display a list of multiple networks within range, allowing users to choose the most suitable one based on their needs. This flexibility is beneficial in scenarios where multiple Wi-Fi networks might be available.

3) OTA Connectivity

This section delves into the initial exploration of ThingsBoard, its strengths and limitations, and the ultimate need for a more automated approach. How we went to other vendors, but ultimately, decided on making our own solution.

ThingsBoard emerged as the initial candidate for managing OTA updates. It's an open-source IoT platform that offers a comprehensive suite of features for device management, data visualization, and rule engine functionalities. Here's a closer look at ThingsBoard and its architecture:

ThingsBoard empowers users to connect various IoT devices, collect and visualize sensor data, create custom dashboards for real-time monitoring, and even implement rule-based actions based on sensor readings. It provides a user-friendly interface for managing these aspects of IoT projects.

ThingsBoard employs a multi-tenant architecture. This means it can accommodate multiple independent users or organizations (tenants) within a single instance. Each tenant has its own isolated workspace for managing their devices, data streams, and dashboards. This segregation ensures data privacy and security for different users.

ThingsBoard utilizes the concept of telemetry for ingesting data from connected devices. Telemetry refers to the collection and transmission of sensor data or other measurements from a device to a central server or platform. In the context of this project, the ESP32's sensor data could have been uploaded to ThingsBoard via telemetry, potentially providing insights into environmental conditions or device status.

While ThingsBoard offered a seemingly attractive solution, it presented several challenges for the project's specific needs. ThingsBoard's user management system would have necessitated each user to create an account on the platform[26]. Additionally, separate "things" (entities representing devices) would have been required for both the mobile phone and the user within ThingsBoard[26]. This would have introduced a cumbersome registration process and potential confusion for end-users.

Integrating ThingsBoard with the existing backend infrastructure built on AWS would have required complex synchronization. We would have needed to coordinate user creation on both the AWS backend and the ThingsBoard platform, adding an extra layer of development effort.

ThingsBoard's API, crucial for programmatic interaction, was considered outdated at the time. More importantly, the desired OTA functionality – provisioning the generated code to the ESP32 – lacked an automated implementation via the API[26]. ThingsBoard primarily offered a GUI-based approach for OTA updates, which wouldn't have aligned with the project's requirement for seamless automation.

OTA updates, also known as firmware updates, refer to the process of delivering and installing new software or firmware onto a device wirelessly. This eliminates the need for physical access to the device, streamlining the update process and ensuring all devices are running the latest version of the code.

OTA updates are prevalent in various consumer electronics today. Smartphones, tablets, smart TVs, and even wearables frequently receive software updates over the air. These updates can introduce new features, address security vulnerabilities, or improve performance.

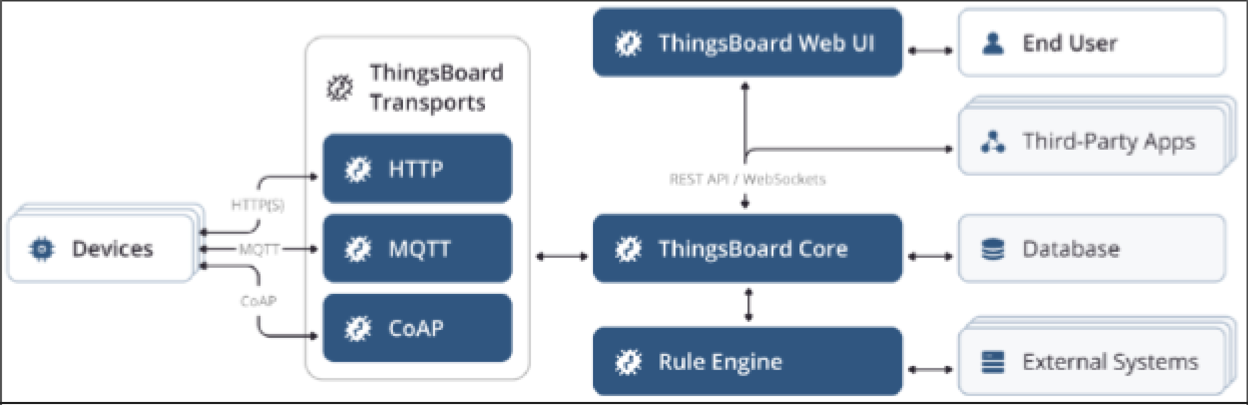

Fig 5. Thingsboard Architecture

In the context of this project, the goal of OTA updates was to wirelessly upload the code generated by the LLM onto the ESP32 device. This would allow for dynamic updates to the device's functionality without requiring manual code flashing or physical interaction with the ESP32.

However, as discussed, ThingsBoard's limitations in terms of automated code provisioning via API necessitated the exploration of alternative solutions to achieve a more streamlined and automated OTA update process for the ESP32 device.

Following the limitations encountered with ThingsBoard, we explored other platforms to establish a robust OTA update mechanism. We found Arduino Cloud which was another popular platform for IoT telemetry using ESP32 devices. It provides features for collecting data from connected devices, visualizing data in dashboards, and triggering actions based on predefined rules. However, similar to Thingsboard, Arduino Cloud lacks OTA capabilities, which means it does not support over-the-air firmware updates for ESP32 devices.

While Arduino Cloud offers a user-friendly interface and integration with Arduino IDE, its limitations in OTA functionality made it unsuitable for our project's requirements[27]. A crucial limitation for this project was the lack of robust OTA capabilities within Arduino Cloud. While Arduino Cloud offers basic functionalities for updating firmware, it wouldn't have fulfilled the specific requirement of uploading dynamically generated code created by the LLM.

Given the limitations of existing platforms, We opted to develop a custom OTA update solution leveraging Amazon Web Services (AWS). Here's a breakdown of the key components employed within the custom AWS-based solution.

The Compiler backend acts as the processing engine for the generated code. It receives the code produced by the LLM and performs the necessary compilations. The compiled code, typically in the form of binary files (bin files), is then uploaded to a secure storage solution within AWS.

AWS S3 serves as the storage repository for the compiled bin files. S3 offers scalable and reliable object storage, making it well-suited for housing the firmware updates destined for the ESP32 devices[28]. The ESP32 can be configured to download these bin files from S3 storage when an OTA update is initiated.

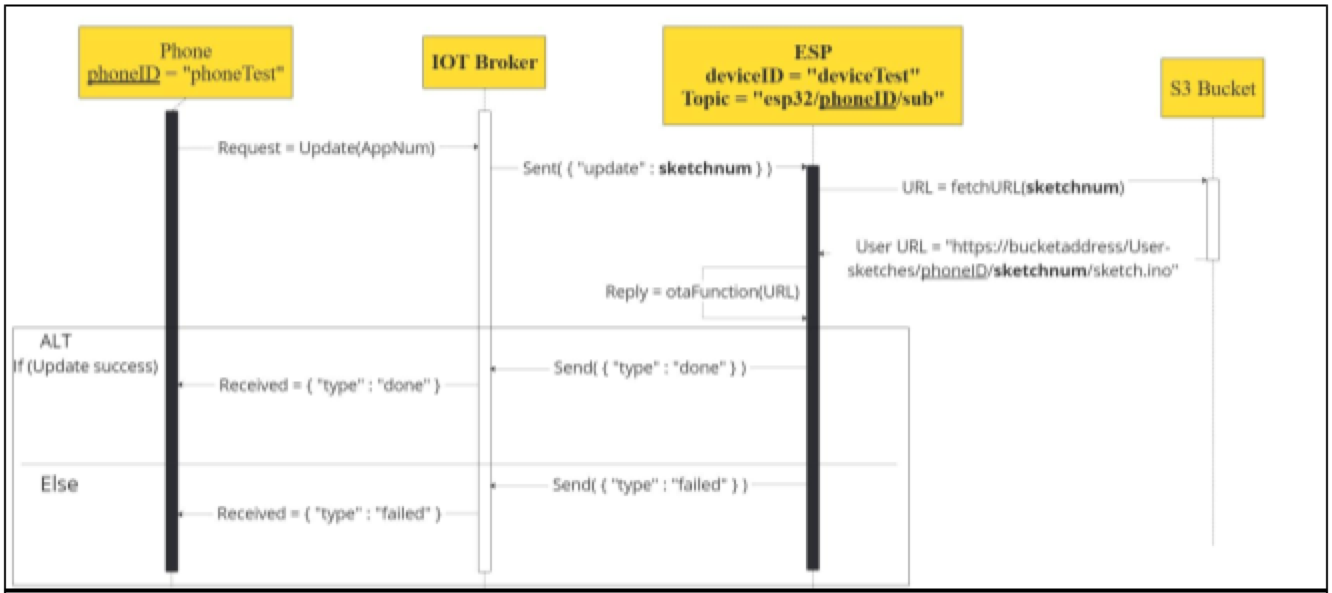
Phone
phoneID = "phoneTest"
IOT Broker
ESP
deviceID = "deviceTest"
Topic = "esp32/phoneID/sub"
S3 Bucket
Request = Update(AppNum)
Sent({ "update" : sketchnum })
URL = fetchURL(sketchnum)
User URL = "https://bucketaddress/User-sketches/phoneID/sketchnum/sketch.ino"
Reply = otaFunction(URL)
ALT
If (Update success)
Received = { "type" : "done" }
Send({ "type" : "done" })
Else
Send({ "type" : "failed" })
Received = { "type" : "failed" }

Fig. 6: OTA Update Steps

AWS IoT Core is a managed service designed for securely connecting and managing IoT devices at scale. In this project[29], IoT Core acts as the intermediary between the ESP32 devices and the backend infrastructure hosted on AWS. The ESP32 devices establish a secure connection with IoT Core, enabling communication and facilitating OTA updates.

In the code of the ESP32, topics for subscribing and publishing to AWS IoT Core are "hard-coded" during code generation. This means that the topics are predefined and cannot be changed dynamically at runtime. Additionally, all the certificates required for connecting to AWS IoT Core are hard-coded into the ESP32 code.

To ensure that the code we created is not overwritten during generation, we need to consider the structure of the code. The code generation process must preserve specific sections of the code that are essential for connecting to AWS IoT Core and other functionalities, while allowing for dynamic generation of other parts of the code.

4) Code Structure

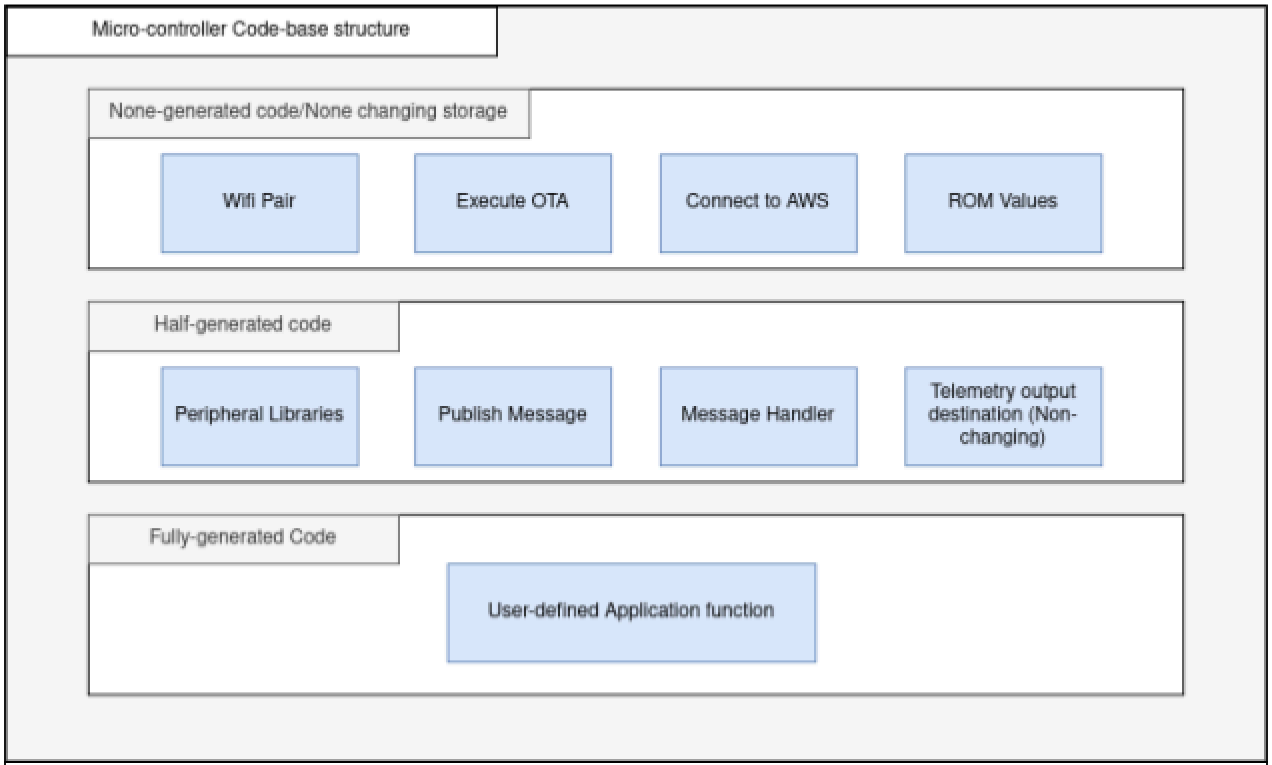
Micro-controller Code-base structure
None-generated code/None changing storage
Wifi Pair
Execute OTA
Connect to AWS
ROM Values
Half-generated code
Peripheral Libraries
Publish Message
Message Handler
Telemetry output destination (Non-changing)
Fully-generated Code
User-defined Application function

Fig. 7: Microcontroller Code structure

As we can see from (Figure 2.1.4.a) to maintain code integrity during generation, we categorize functions into three distinct types: fully generated, half-generated, and unaltered. In Fully Generated Functions we have the Custom Function Section in Setup Function. This section houses custom functions that are entirely generated during the code generation process.

These functions are tailored to the project's specifications and typically include initialization routines, configuration settings, or any other specific logic required for the application to run effectively. For instance, setting up MQTT connections, initializing sensors, or configuring display interfaces can all fall under this category.

Next is the Half-Generated section, Part of this section are libraries which are partially generated. It includes both fully generated and half-generated parts. Libraries such as WebServer, Update, WiFi, HTTPClient, etc., are included based on project requirements. While the inclusion of these libraries is automated, the LLM may need to add specific libraries or update versions based on the prompt.

Similar to libraries, global variables are partially generated. Some variables, such as AWS IoT endpoint configuration and S3 bucket configuration, are fully generated based on project settings. However, other variables, like pin declarations and sensor initialization, may be modified or added by the user as needed for their specific hardware setup.

The setup function is half-generated. While some parts, such as Wi-Fi setup and AWS IoT connection, are crucial and generated based on project settings, other parts, such as peripheral initialization and custom setup routines, are left for the user to define. This allows users to customize the setup process according to their project requirements.

Publish Message and Message Handler functions are also half-generated. The core functionality for sending and handling MQTT messages is provided, but specific logic related to data processing and device control may be customized by the user. For instance, the LLM can define how to handle incoming messages from AWS IoT Core based on their application's needs.

Finally Unaltered Functions, Execute OTA function remains unaltered during code generation. It handles the logic for performing over-the-air updates and is independent of project-specific configurations. The OTA logic ensures that firmware updates can be applied seamlessly to the ESP32 devices without manual intervention.

The Wi-Fi Pair function remains unaltered as well. It is responsible for handling the pairing of the ESP32 with Wi-Fi networks as discussed above. Similarly, the ConnectAWS function remains unaltered and handles the connection to AWS IoT Core. It sets up the necessary client certificates and establishes a secure connection to the MQTT broker on the AWS endpoint, ensuring reliable communication between the ESP32 and AWS services.

The loop function remains unaltered and provides the main execution loop for the program. It handles routine tasks such as MQTT message processing, Wi-Fi connection management, and other periodic operations, ensuring the continuous operation of the ESP32 device.

More information about how this distinction helps in preventing overwriting during generation will be discussed in the next section.

B. Software

1) Models & Architecture

Before we delve into the applications, we need to get familiar with these common models that serve as the building blocks for managing peripherals, configurations, certificates, and user information, ensuring a cohesive and efficient data flow throughout the application.

First one of these is the Peripheral Model which serves as the digital representation of the physical components connected to the ESP32. It encapsulates the essential attributes of each peripheral,

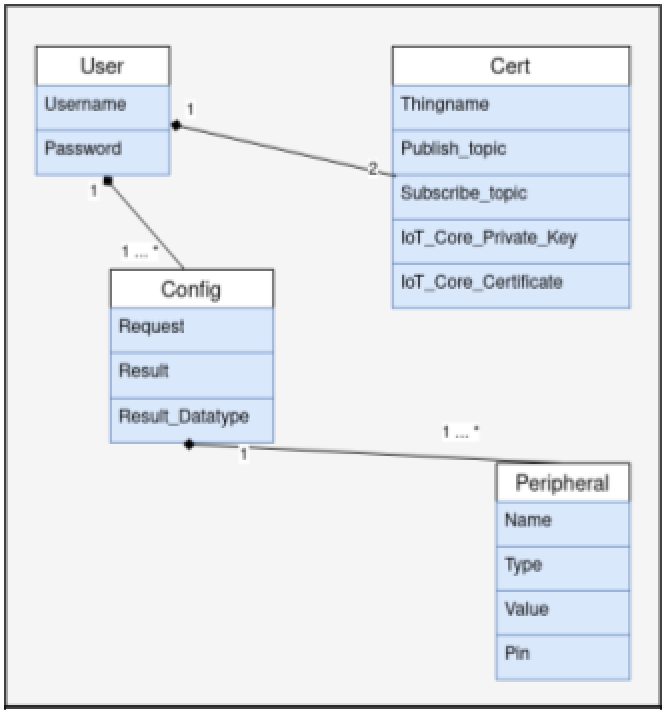

Fig. 8: UML Class Diagram

providing a structured way to manage and interact with the hardware. The model typically includes the following fields.

Pin, this field denotes the physical pin on the ESP32 to which the peripheral is connected. It serves as a unique identifier for the peripheral and facilitates communication with the hardware.

Name, this field provides a human-readable name for the peripheral, making it easier for users to identify and interact with the component within the application. Value, this field represents the current value or state of the peripheral. For sensors, it may hold the measured data, while for actuators, it may indicate the desired state or control signal.

Type, this field specifies the type of peripheral, such as a sensor, actuator, or communication module. This information is crucial for understanding the capabilities and behavior of the peripheral and ensuring appropriate interactions.

The Peripheral model plays a vital role in various aspects of the application, for example, code generation where the peripheral information is used during code generation to tailor the generated code to the specific hardware configuration of the user's project.

As well as a control interface, the peripheral model provides the necessary data for displaying and interacting with the peripherals within the control screen, enabling users to monitor values, send commands, and control their devices.

Most importantly, Data Exchange where the model facilitates the exchange of data between the application and the ESP32, ensuring that the correct values are sent to and received from each peripheral.

Next is the Config Model which acts as a repository for application-specific configurations, storing essential information about the generated code and associated peripherals. This model typically includes the following components.

Peripherals: This field contains a list of Peripheral models, representing the complete hardware configuration for the generated application. Request: This field stores the user's prompt or request that was used to generate the code, providing context for the generated functionality.

Result: This field holds the generated code itself, ready to be deployed to the ESP32 device. Result Data Type: This field specifies the expected data type of the result returned by the custom function defined in the generated code. This information is important for interpreting the output of the function and displaying it appropriately within the application.

The Config model plays a crucial role in managing the generated application, ensuring that all necessary information is readily available and enabling efficient deployment and execution of the code.

Afterwards, the Cert Model which stores the certificates required for secure communication between the application and AWS IoT Core. This model ensures that only authorized devices and applications can exchange data with the cloud platform. It typically includes the following fields, Thing Name: This field identifies the ESP32 device within the AWS IoT Core ecosystem.

Private Key: This field stores the private key associated with the device certificate, used for authentication and secure communication. IoT Core Certificate: This field holds the device certificate issued by AWS IoT Core, verifying the identity of the device.

Pub Topic: This field defines the MQTT topic used for publishing messages from the application to the ESP32 device. Sub Topic: This field defines the MQTT topic used for subscribing to messages from the ESP32 device.

The Cert model plays a critical role in establishing a secure connection with AWS IoT Core, enabling functionalities such as OTA updates and data exchange between the application and the device. By ensuring secure communication, the Cert model protects sensitive information and maintains the integrity of the IoT system.

Finally, The User model represents an individual user within the application and stores their relevant information and preferences. This model may include the following fields.

Credentials: This field stores the user's login credentials, such as username and password, ensuring secure access to the application and user-specific data. Configs: This field maintains a list of Config models associated with the user, representing the various applications they have generated and their corresponding configurations.

Certificates: This field stores the Cert models for the user's ESP32 devices and mobile application, enabling secure communication with AWS IoT Core. Preferences: This field may store user-specific preferences related to the application's behavior or appearance, allowing for personalized experiences.

The User model facilitates personalized interactions within the application, manages access control to sensitive data, and provides a central repository for user-related information. These models are common throughout the whole application due to the architecture we used, where all parts need to be agreeing on what messages they must receive.

To do this we must agree on the protocol used between the endpoints. We agreed on using MQTT for communication between the Phone and the ESP32, and HTTP for communication between the Backend and anything else. MQTT, short for Message Queuing Telemetry Transport[31], is a lightweight messaging protocol designed for efficient data exchange in constrained environments and over unreliable networks.

It's perfect for scenarios where devices need to communicate small bits of information frequently, like sensor readings or control commands. MQTT uses a minimal message header and a publish/subscribe model, reducing bandwidth consumption and processing overhead[31]. This is crucial for resource-constrained devices like the ESP32 with limited memory and processing power.

The efficient design of MQTT minimizes energy usage, making it ideal for battery-powered IoT devices like ESP32 where extending battery life is critical. MQTT allows for near real-time data exchange, enabling rapid responses and immediate action based on received information. This is perfect for applications that require quick reactions to changing conditions which our application can generate.

MQTT offers different Quality of Service (QoS) levels to ensure message delivery even in unreliable network conditions. This makes it suitable for situations where data loss is unacceptable, such as critical control signals. MQTT can handle a large number of concurrent connections, making it suitable for large-scale IoT deployments with numerous devices.

Studies have shown that MQTT message delivery can be up to 25 times faster than HTTP[32], especially for small messages. This is due to the lightweight protocol and the efficient publish/subscribe model.

MQTT requires significantly less data overhead compared to HTTP, reducing bandwidth consumption and network congestion[32]. This is particularly beneficial for devices on limited data plans or in areas with poor network coverage.

Considering the ESP32's limited resources and its typical use cases in IoT applications, MQTT emerges as the ideal communication protocol. Its lightweight nature, low power requirements, real-time capabilities, and reliability perfectly align with the needs of ESP32-based projects[33].

On the other hand, HTTP, or Hypertext Transfer Protocol, is the foundation of data communication on the web. It's a request-response protocol where a client sends a request to a server, and the server responds with the requested data[34].

While MQTT excels in specific scenarios, HTTP remains a powerful and versatile protocol with its own advantages. HTTP is universally supported by virtually all devices and browsers, making it readily accessible for integration with various platforms and services[33].

HTTP can handle various data formats and complex requests, making it suitable for diverse applications beyond simple data exchange. It can also be secured with TLS/SSL encryption, ensuring data confidentiality and integrity during transmission.

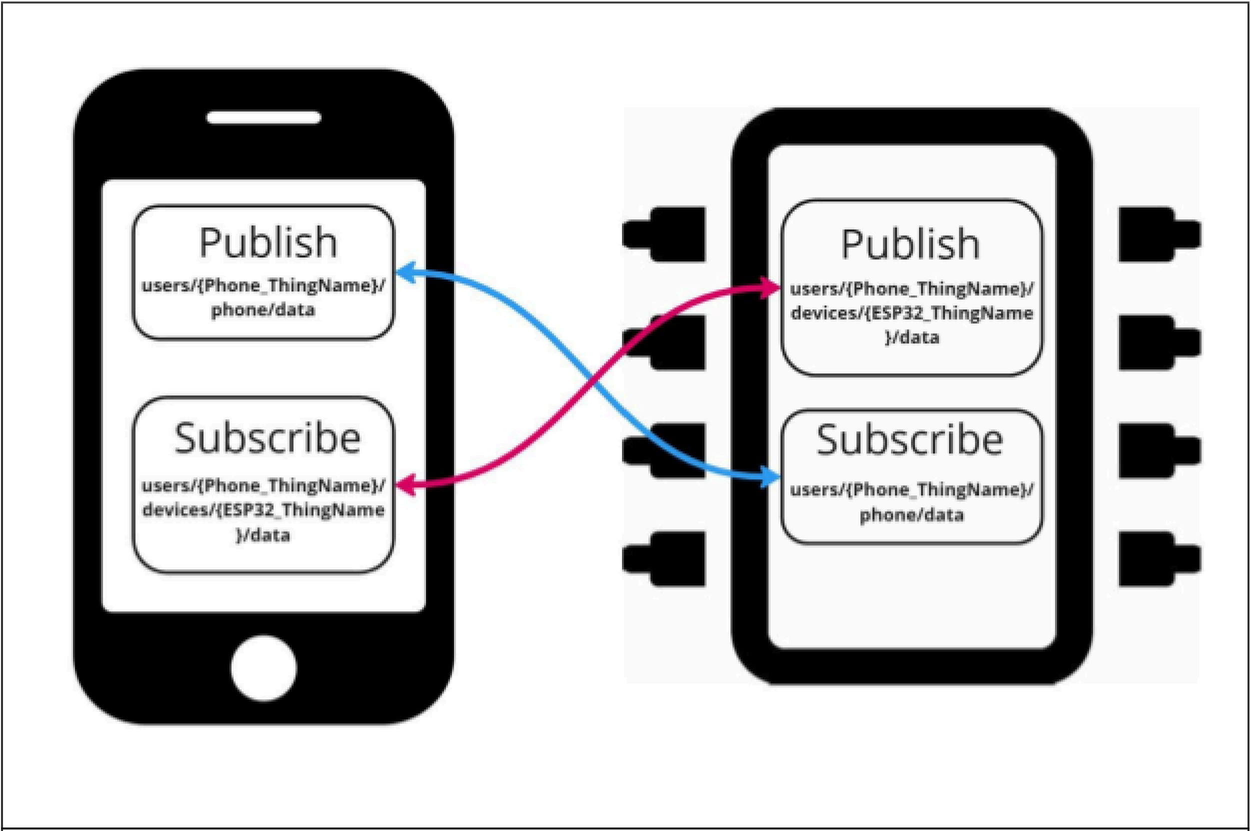
Publish
users/{Phone_ThingName}/phone/data
Subscribe
users/{Phone_ThingName}/devices/{ESP32_ThingName}/data
Publish
users/{Phone_ThingName}/devices/{ESP32_ThingName}/data
Subscribe
users/{Phone_ThingName}/phone/data

Fig. 9: IoT Topics Design

HTTP is the foundation for RESTful APIs, which are widely used for building web services and application integrations. If your application requires transferring large amounts of data, HTTP might be more efficient than MQTT, as it's optimized for handling bigger payloads[33].

Also, when an application involves complex interactions with servers or requires specific functionalities like file uploads or downloads, HTTP offers greater flexibility and capabilities. Which is why we still use HTTP for our own OTA solution, not MQTT.For ESP32 projects and many IoT applications, MQTT often emerges as the preferred choice due to its lightweight nature, efficiency, and real-time capabilities. However, HTTP remains a valuable tool for scenarios involving larger data transfers, complex interactions, and integration with existing web infrastructure. We stand in a unique position where we integrate both.

Therefore for this integration, alongside the models we need to agree on topics that exist for communication between the user and their own ESP. This agreement is assigned during user creation and code generation so that the user never has access to changing his topics to prevent eavesdropping. This along with his certificates creates a high level of security against snooping information.

With all the components common throughout our application done, we can now talk about the application lifecycle.

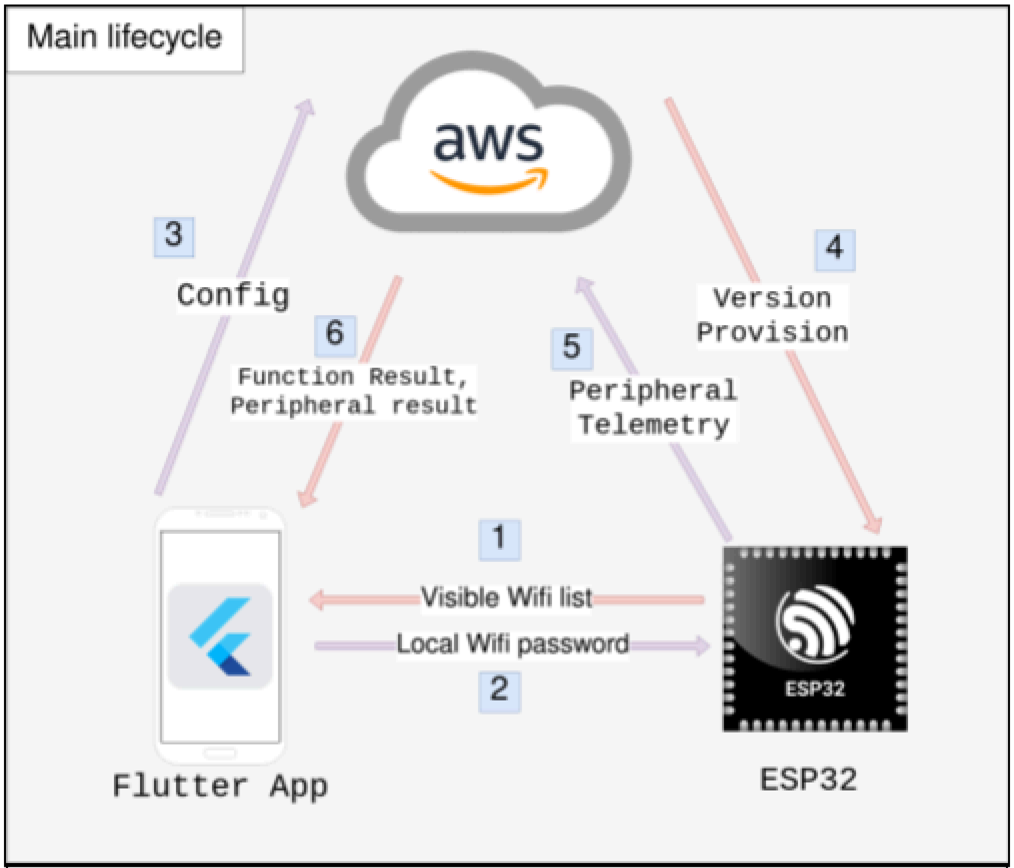

Fig. 10: Application lifecycle

As we can see from (Fig. 10), there are 3 main parts to the application lifecycle, we have already discussed one of the frontend components, the ESP32. Next we shall talk about the Flutter application, and through it we will talk about how it contributes to the cycle.

2) Flutter Mobile App

The hardware capabilities of the ESP32, as discussed, lay a solid foundation for versatile IoT applications. However, the true power of this microcontroller is unlocked through its software interface, allowing users to harness its potential with ease. This section delves into the software aspects of the research, exploring the application lifecycle and the mechanisms that enable seamless interaction.

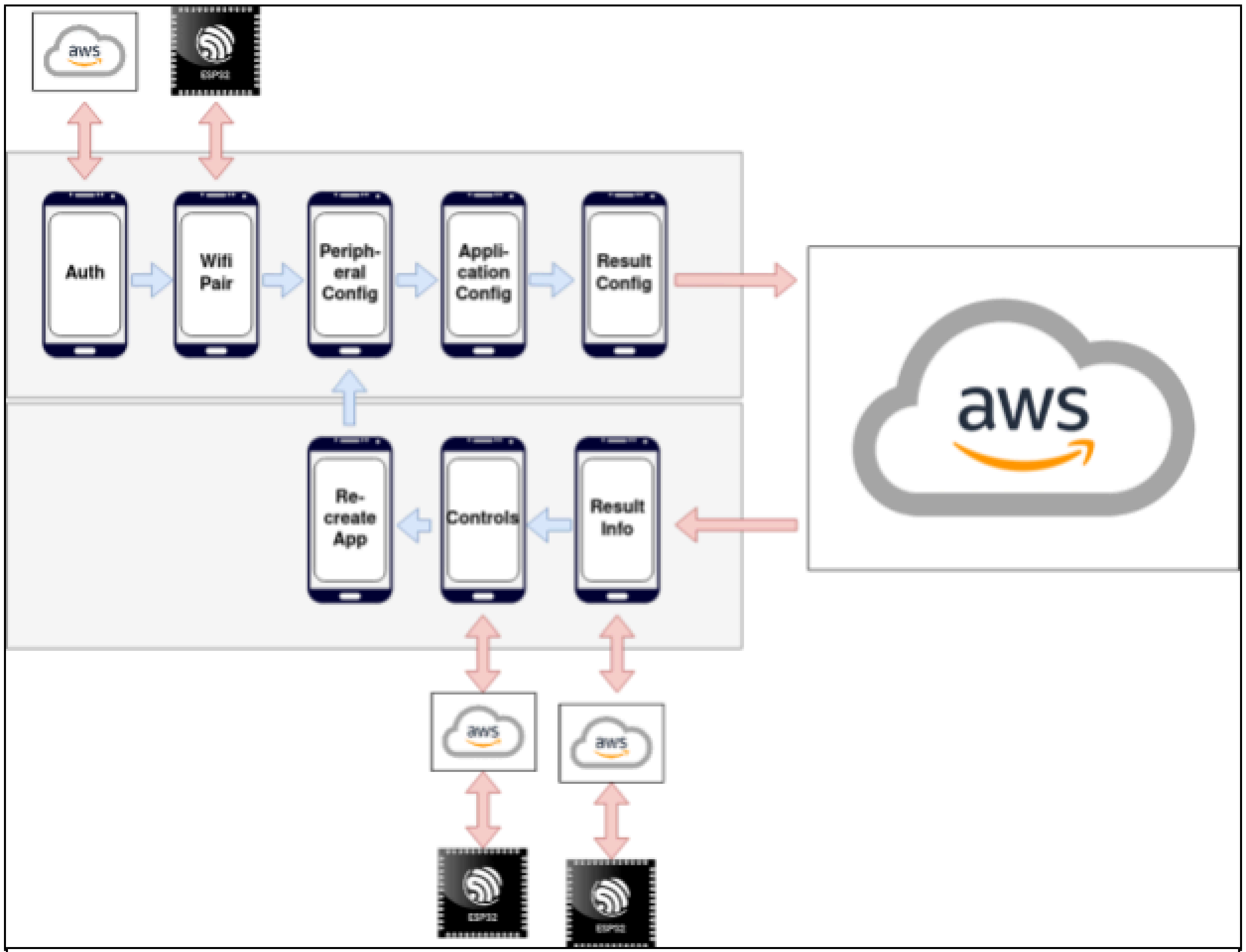

Fig. 11: Flutter Mobile App Lifecycle

As can be seen from (Fig. 11), the user journey through the application is meticulously designed to be intuitive and efficient, ensuring a smooth experience from account creation to device control.

The initial step for any user is establishing their presence within the application (Fig. 10). New users can create an account by providing the necessary credentials, while existing users can simply sign in using their established credentials. This step lays the groundwork for personalized experiences and secure access to project data and device management functionalities.

Once the user is logged in, the next crucial step is connecting their ESP32 device to the Wi-Fi network. As discussed in the hardware section, the WifiPair functionality plays a pivotal role here. By leveraging the ESP32's Wi-Fi capabilities and creating a temporary hotspot, WifiPair allows the user to seamlessly connect the device to their chosen network through the mobile application. This eliminates the need for manual configuration on the ESP32 itself, simplifying the process and enhancing user convenience.

The WifiPair process is optimized for efficiency and user-friendliness. The mobile application automatically detects the ESP32's temporary hotspot and guides the user through the connection process. Additionally, the ESP32 stores previously connected networks, allowing for quick reconnection in future sessions without user intervention making this ideally a one time process.

With the ESP32 connected to the network, the user proceeds to declare the peripherals connected to their device(Fig 10). This involves filling out a form within the application, specifying the type and configuration of each peripheral. This information is crucial as it informs the code generation process, ensuring that the generated code is tailored to the specific hardware setup of the user's project.

The heart of the application lies in the prompt and code generation stage. Here, users articulate their desired functionality through a textual prompt. This prompt details what they expect the custom function to accomplish, including the expected behavior, inputs, and outputs. The application then leverages a large language model (LLM) to interpret the prompt and generate the corresponding code for the ESP32. This code incorporates the user's specified functionality while adhering to the declared peripheral configuration, creating a customized solution for their project.

Upon successful code generation, the user enters the control screen, where they can interact with their ESP32 device and execute the generated code. The control screen offers two modes. Basic Control, This mode provides a simplified interface for users to trigger the custom function and observe its outputs. It is ideal for straightforward interactions and quick testing of the generated code.

As well as Advanced Control, For users who require more repeated control, the advanced mode offers a wider range of options. This includes the ability to create custom buttons that do custom actions with the peripherals. The advanced mode empowers users to explore the full potential of their projects and fine-tune their applications.

The Flutter app utilizes the MVC (Model-View-Controller) design pattern to manage communication and interaction between the user, the mobile app, and the ESP32 microcontroller. This creates the previously mentioned lifecycle, here we’ll delve into the models that are used not just in the Mobile app but throughout the entire system. But as well the controllers that allow the seamless communication between the ESP32 and the Mobile application.

Models form the foundation of the MVC pattern, encapsulating the data and state of the application. In this project, several models are used to represent different aspects of the system and we’ll discuss each one in the following.

Peripheral model stores information about each peripheral connected to the ESP32, including its pin assignment, name, current value, and type (e.g., sensor, actuator). This model ensures a clear representation of the hardware configuration and facilitates communication with the ESP32.

Config model holds the configuration data for the generated application, including the list of peripherals, user prompt, generated code, and expected data type of the result. The config model serves as a central repository for application-specific information, ensuring consistency and enabling efficient code generation.

Cert model stores the necessary certificates for secure communication with AWS IoT Core, including the thing name, private key, and IoT Core certificate. This model plays a vital role in establishing a secure connection and enabling OTA updates and data exchange.

User model encapsulates user information, including credentials and potentially project-specific preferences or settings. The user model facilitates personalized experiences and manages access control within the application.

MQTT models define the structure of MQTT messages used for communication between the mobile application and the ESP32 device. Publish models specify the format for sending commands or data updates to the ESP32, while subscribe models define the format for receiving data or status updates from the ESP32. These models ensure consistent and reliable communication between the application and the device.

Views are responsible for displaying information to the user and capturing user interactions. In this project, views take the form of various screens and interface elements within the mobile application.

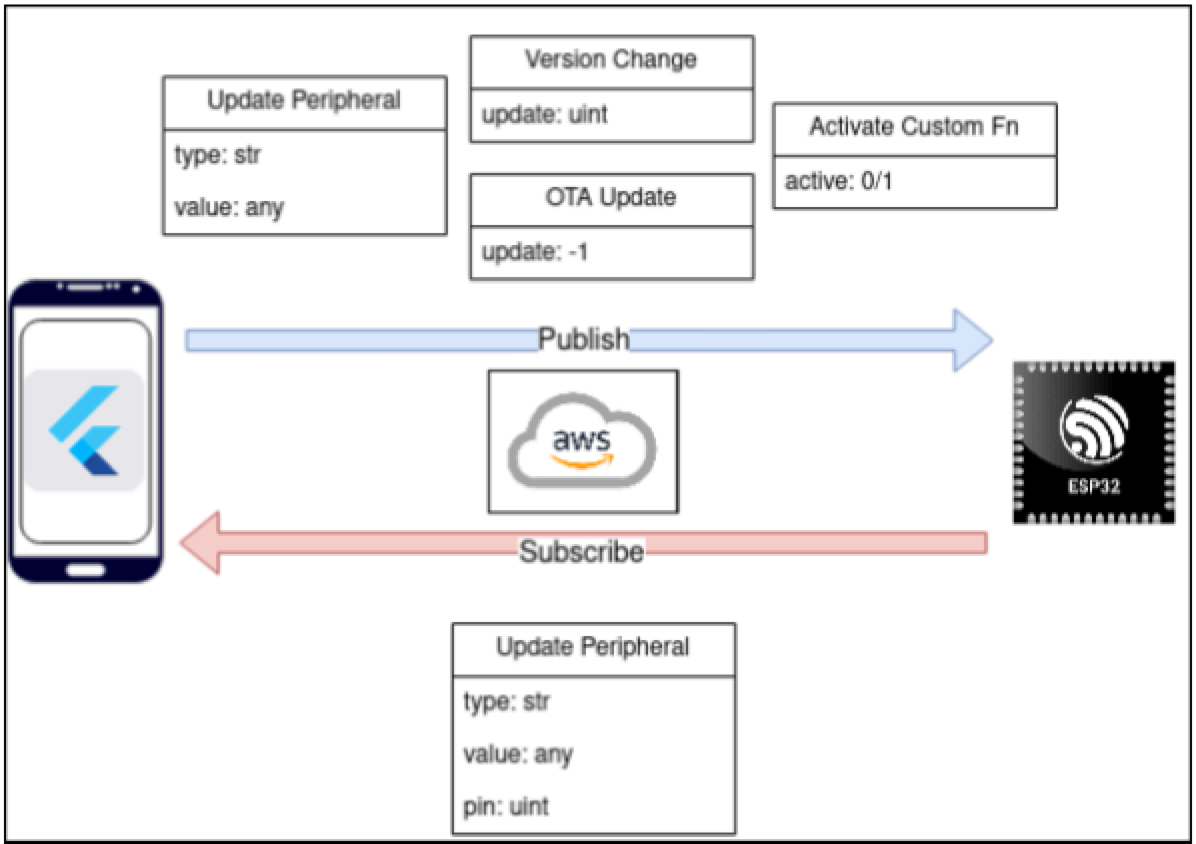

Fig. 12: MQTT Message Formats

Authentication view handles user login and registration, providing input fields for credentials and facilitating account creation. WifiPair view guides the user through the WifiPair process, displaying available Wi-Fi networks and enabling network selection and password input. Peripheral Declaration view presents a form for the user to declare the peripherals connected to their ESP32, allowing them to specify the type and configuration of each peripheral. Prompt view provides an interface for the user to enter their prompt, describing the desired functionality of the generated code.

Control view offers both basic and advanced control options for interacting with the ESP32 device, including buttons for triggering commands, input fields for setting values, and displays for visualizing sensor data. All three of the previous views are combined into one screen, making it easier for the user to remember what application they want to create, allowing them to also correct any errors in the peripheral section without going out of the screen.

The controller acts as the intermediary between models and views, handling user input, updating models, and triggering view updates. In this project, the controller performs several key functions which we will discuss now.

Auth controller manages user authentication, verifying credentials and handling login and registration processes. The WifiPair controller interacts with the WifiPair functionality, retrieving the list of available Wi-Fi networks, sending the selected network and password to the ESP32, and managing the connection process.

Prompt Creation controller processes user input from the prompt view, inserts the prompt data into the config model, and sends the config data to the backend for code generation. Basic/Advanced Command Controller: This controller handles user interactions within the control view. For basic commands, it sends individual MQTT publish messages to the ESP32 based on user input. For advanced commands, it allows users to create custom buttons that trigger sequences of MQTT messages, enabling complex control sequences with a single tap.

3) Backend

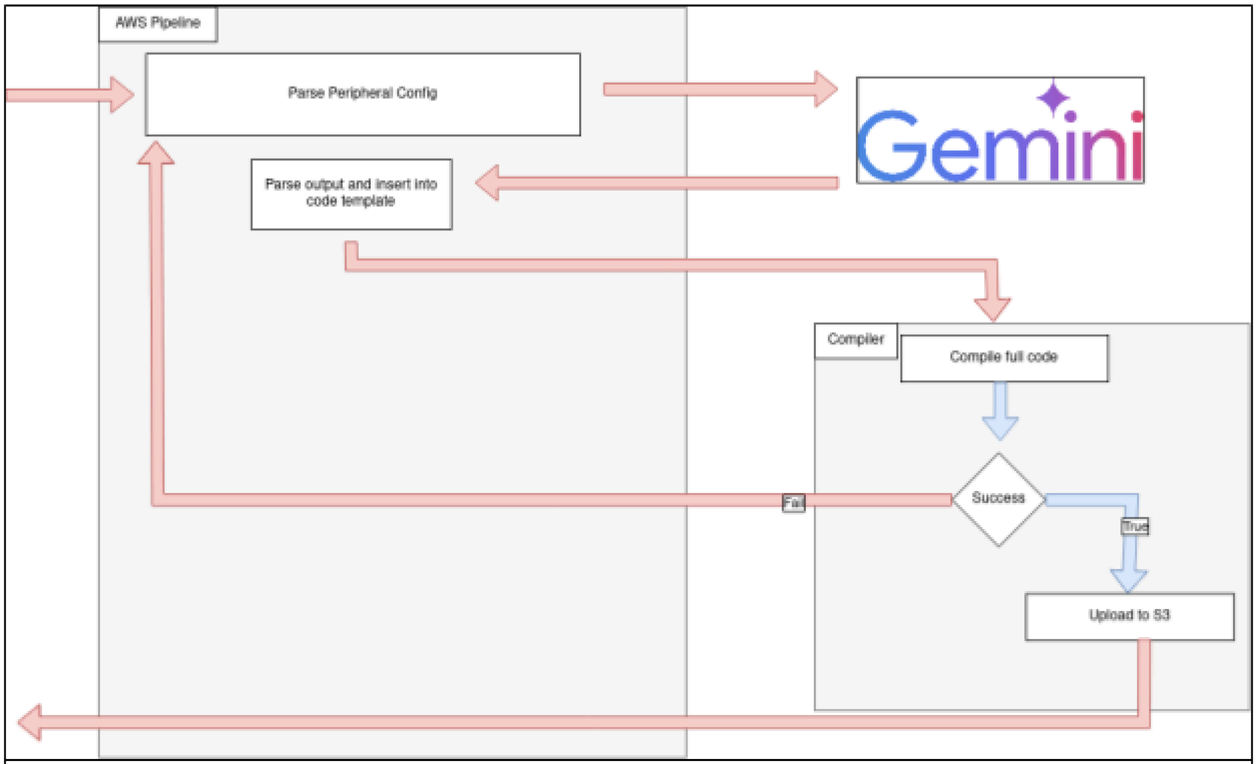

Fig. 13: Backend cycle of prompt request

As we can see from (Fig. 13), the abstract view of the system is dependent on two backends. To host these two backends, the system leverages a diverse set of technologies including Golang, Docker, Amazon EC2, DynamoDB, VPC, S3, and IoT Core, and integrates seamlessly with the powerful language model, Gemini, to create a robust and efficient workflow.

The choice of technologies was driven by specific needs for scalability, performance, and ease of integration. Golang was selected as the primary backend language due to its inherent advantages in concurrency, efficient memory management, and a readily available SDK for seamless interaction with AWS services[35]. This simplifies the automation of tasks, replacing manual processes with programmatic solutions.

Docker serves as the containerization platform for hosting the compiler, which was sourced and adapted from a GitHub repository[36]. Docker ensures environment consistency and portability, simplifying deployment and management of the compiler as a microservice. The decision to modify the compiler into a Flask web server was motivated by the desire to leverage HTTP status codes for error reporting. This facilitates a feedback loop with Gemini[37], allowing for iterative code generation and refinement based on compiler feedback.

The cloud infrastructure leverages various AWS services to achieve optimal performance and user experience. Amazon EC2 provides the virtual server environment for hosting the Golang backend application[38]. The selection of the server location within the United States ensures compatibility with the Gemini API, as European regulations currently restrict access.

DynamoDB, a NoSQL database service, stores user data including ESP certificates, topics, and thing names. Its low latency and scalability make it ideal for rapid retrieval of information required during code generation and deployment[39]. Locating the database in the US minimizes latency for communication with the Golang backend. [40]

Table III

Average ping times between locations

	Cairo	Frankfurt	Washington
Cairo	-	51.351ms	133.905ms
Frankfurt	51.535ms	-	83.793ms
Washington	133.779ms	83.838ms	-

A Virtual Private Cloud (VPC) is employed to create a secure and isolated network environment for the EC2 instance and other AWS resources[30]. This ensures controlled access and enhances the overall security posture of the system. Amazon S3 provides scalable and reliable object storage for compiled code and firmware updates.

IoT Core facilitates secure communication and device management for ESP32 devices. These services are strategically located in Europe to minimize latency for end-users during code download and OTA updates. While this introduces a slight latency increase for uploading compiled code to S3, the impact is negligible compared to the overall compilation time.

The Gemini API plays a pivotal role in generating and customizing code based on user requirements and hardware configurations. The process begins by fetching relevant user data from DynamoDB, including ESP32 certificates, topics, and thing names. This information is crucial for personalizing the generated code and ensuring proper device communication.

A predefined hardware code template serves as the foundation for code generation. User-specific data, along with a link to the S3 bucket for future OTA updates, is inserted into the template. Gemini, with its advanced language understanding and generation capabilities[2], is tasked with creating code specific to the user's chosen peripherals and desired functionalities. This step involves providing Gemini with a comprehensive prompt that encompasses user requirements and hardware specifications.

The generated code from Gemini is integrated into the previously populated template. The final code, along with required libraries and the S3 upload link, is then passed to the compiler API for further processing. The compiler API, built on Docker and incorporating the Arduino CLI, facilitates code compilation and error reporting.

A Docker image, based on a modified GitHub repository, provides a consistent and isolated environment for the compiler[36]. The image is customized to utilize the ESP32 toolchain instead of the standard Arduino tools.

The compiler is transformed into a web server, enabling communication via HTTP requests and responses. This facilitates the transmission of compilation errors as HTTP status codes, which are subsequently fed back to Gemini for potential code correction and regeneration. This feedback loop significantly enhances the robustness of the code generation process.

Upon receiving code, libraries, and the S3 upload link from the Golang backend, the compiler first installs the required libraries. The compilation process then commences, with initial runs taking approximately three minutes due to dependency installations. Subsequent compilations, leveraging cached dependencies, complete within roughly 50 seconds.

If compilation errors occur, the API communicates them back to the Golang backend per (Fig.13) for potential code adjustments and re-submission. In the case of successful compilation, the resulting binary is uploaded to the designated S3 bucket. The Golang backend subsequently receives a success response, allowing the user to proceed with OTA updates if desired.

The previous section detailed the intricate backend infrastructure, leveraging a combination of Golang, Docker, and various AWS services to deliver a seamless user experience. However, the true magic lies in the integration of the powerful language model, Gemini, and the meticulous art of prompt engineering. This section delves deeper into the crucial role of prompt engineering in enabling Gemini to understand user requirements and generate customized code effectively.

The success of the code generation process hinges on Gemini's ability to comprehend the task at hand. This is where prompt engineering takes center stage, acting as the bridge between user intent and Gemini's generative capabilities.

The initial step involves preparing the base template code. This template acts as a foundation for Gemini to build upon. To optimize comprehension, unnecessary elements, such as ESP certificates and static functions, are removed. This ensures Gemini focuses on the core structure and logic of the code.

The template is enriched with comments that serve as explicit instructions for Gemini. These comments delineate the sections where code generation is required and specify areas where existing code should remain untouched. Additionally, examples are embedded within the comments to illustrate the desired code structure and functionality. This provides Gemini with a clear roadmap for the generation process.

A series of tests are conducted to identify the most effective wording for prompting Gemini. The objective is to find the language that best conveys the desired outcome. Generally, the prompt instructs Gemini to first analyze the code's purpose, then overwrite specific functions labeled as "half-generated" while adhering to the user's requirements. This approach allows for flexibility while maintaining a structured generation process.

User-specific information, such as peripheral names, values, types, and compatible libraries, is dynamically integrated into the prompt using string formatting. This ensures the generated code aligns perfectly with the user's hardware configuration and desired functionalities. The library selection is based on a curated list of Arduino-ESP compatible libraries, ensuring compatibility and avoiding conflicts during installation.

The final prompt instructs Gemini to generate only the code, avoiding any extraneous text or explanations. Each generated function is placed under a specific label, ensuring consistency and facilitating easy parsing and integration into the main code template. Additionally, the temperature setting is adjusted to 0.4 to strike a balance between creativity and adherence to the desired format. Parameters like topP are left at default values as they produce satisfactory results[41].

The generated code is then formatted and inserted into the main template, and the compilation process proceeds as usual. The beauty of this model lies in its ability to pinpoint and address compilation errors efficiently. If an error occurs, the faulty function, identified by its label, is sent back to Gemini for correction. This iterative feedback loop significantly enhances the robustness and accuracy of the code generation process.

By combining the structured framework of Golang and Docker with the generative capabilities of Gemini and the precision of prompt engineering, the system achieves a remarkable level of automation and customization. Users can effortlessly generate code tailored to their specific needs, empowering them to bring their IoT projects to life with ease.

4) Website Platform

A fundamental challenge arises when catering to users who lack prior experience with ESP32 programming: the initial setup and code installation process. Traditionally, this stage necessitates familiarity with code editors, compilers, and flashing tools, creating a significant barrier to entry for newcomers. To address this a new approach was created, the main point being around user-friendliness and seamless onboarding.

The initial point of contact for the user is a dedicated website. Here, new users undergo a simple registration process, providing the essential information to the User model. This triggers the backend system, creating a unique user profile. This profile serves as a central repository for managing future projects, device configurations, and access credentials, ensuring a personalized experience for each individual. To be used later on when using the application.

Upon successful registration, the backend system automatically preloads a set of basic code files into an Amazon S3 bucket. These files, constituting a foundational template, encompass core functionalities essential for any ESP32 project. This includes modules for Wi-Fi connection management, establishing communication protocols like MQTT, and basic device operations, providing a solid starting point for users to build upon. The utilization of S3, a scalable and reliable object storage service, ensures the availability and accessibility of these code files for subsequent deployment.

The next stage involves the setup application, a software tool designed to bridge the gap between the user's computer and the ESP32 device. Downloaded from the website and installed on the user's system, the application serves as a user-friendly interface for configuring the ESP32 and deploying the preloaded code.

The process begins with the user connecting their ESP32 to the computer via a USB cable. The application automatically detects the connected device and prompts the user to select the appropriate communication port. This step, while seemingly simple, is crucial for establishing a reliable connection between the application and the ESP32.

Once the port is selected, the application initiates the automated code upload process. The preloaded code files from the S3 bucket are seamlessly transferred and installed onto the ESP32, eliminating the need for manual code handling or complex flashing procedures. A visual cue, in the form of a flashing blue LED on the ESP32, confirms successful code deployment and indicates the device's readiness for further interaction.

A key advantage of this approach lies in its forward-thinking design. By embedding Over-the-Air (OTA) update capabilities within the preloaded code, the system is equipped to handle future firmware updates and functionality expansions without requiring physical access to the ESP32. The intuitive interface and automated processes remove the need for prior programming knowledge or familiarity with complex development tools. This empowers users of all skill levels to engage with the ESP32 and explore its capabilities. Which we will discuss in the Results.

Chapter III
RESULTS

III. RESULTS

A. Code switching

A defining characteristic of the architecture lies in its ability to facilitate dynamic code switching through Over-the-Air (OTA) updates. This feature empowers users to effortlessly transition between different configurations and functionalities on their ESP32 devices without the concern of losing previous settings or encountering compatibility issues. The underlying mechanism hinges on a meticulously orchestrated synchronization process between the backend system, the mobile application, and the ESP32 device itself.

The process commences with the generation of a new configuration within the mobile application. As the user defines the parameters for this new configuration, including peripheral settings, desired functionalities, and custom code, the application simultaneously prepares for a future code switch. It embeds information about the next version – the newly generated configuration – as the designated target for a subsequent OTA update. This information acts as a roadmap, guiding the ESP32 towards the intended configuration when the update command is issued.

However, the system's flexibility extends beyond a simple linear progression of configurations. The mobile application possesses the capability to dynamically alter the designated target version. Through specific update messages, the application can instruct the ESP32 to switch to any previously generated configuration, offering users the freedom to revisit and utilize past projects as needed. This dynamic switching capability opens doors for a variety of use cases, including the following. Users can seamlessly switch between two different configurations to compare their performance, evaluate functionalities, and identify the optimal solution for their specific needs.

The ESP32 can adapt to different environments or scenarios by switching to preconfigured settings tailored to those specific contexts. For instance, a device could switch to a low-power mode when battery life is critical or activate additional sensors when entering a particular location.

In case of unexpected issues or errors with a new configuration, users can easily revert to a previous, stable version without the need for manual reconfiguration or code flashing.

To enhance the user experience and provide practical guidance, the system comes preloaded with three previously generated applications. These applications serve as illustrative examples, showcasing the capabilities of the platform and offering users a starting point for their own projects. By exploring these preconfigured applications, users can gain valuable insights into the next items.

The examples demonstrate how different peripherals, such as sensors, actuators, and communication modules, can be integrated into an ESP32 project and utilized for various purposes.

Users can observe how custom code is implemented to achieve specific functionalities and adapt these techniques to their own projects. The preloaded applications provide a hands-on experience with the control interface, allowing users to familiarize themselves with the process of sending commands, receiving data, and monitoring device status.

These preconfigured applications act as stepping stones, helping users transition from theoretical understanding to practical implementation. They inspire creativity, provide a foundation for experimentation, and empower users to confidently explore the vast potential of the ESP32 platform.

This dynamic code-switching functionality is made possible by the harmonious interplay of various technologies and design principles. The backend system maintains a comprehensive record of all generated configurations, including version history and associated metadata. This enables the mobile application to accurately identify and select the desired target version for OTA updates. The system relies on reliable communication protocols, such as MQTT, to ensure seamless and error-free exchange of update messages and configuration data between the mobile application and the ESP32 device.

The OTA update mechanism is designed with security in mind, employing encryption and authentication protocols to protect against unauthorized access and ensure the integrity of the firmware updates.

By combining these elements, the system empowers users to dynamically switch between configurations, fostering flexibility and adaptability within their ESP32 projects. This functionality not only enhances the user experience but also opens doors for innovative applications and creative exploration within the IoT domain.

B. Generated Applications & their Implications

To demonstrate the reach of capabilities of our project, we decided to theorize four different applications. Each application represents a side of our generation capabilities. From reactivity, to ease of control, to diversity of generation, each application helps create a new perspective of usage for our application.

In each application, red wires represent VCC wires and blue wires represent Ground wires.

A) Application One: Car

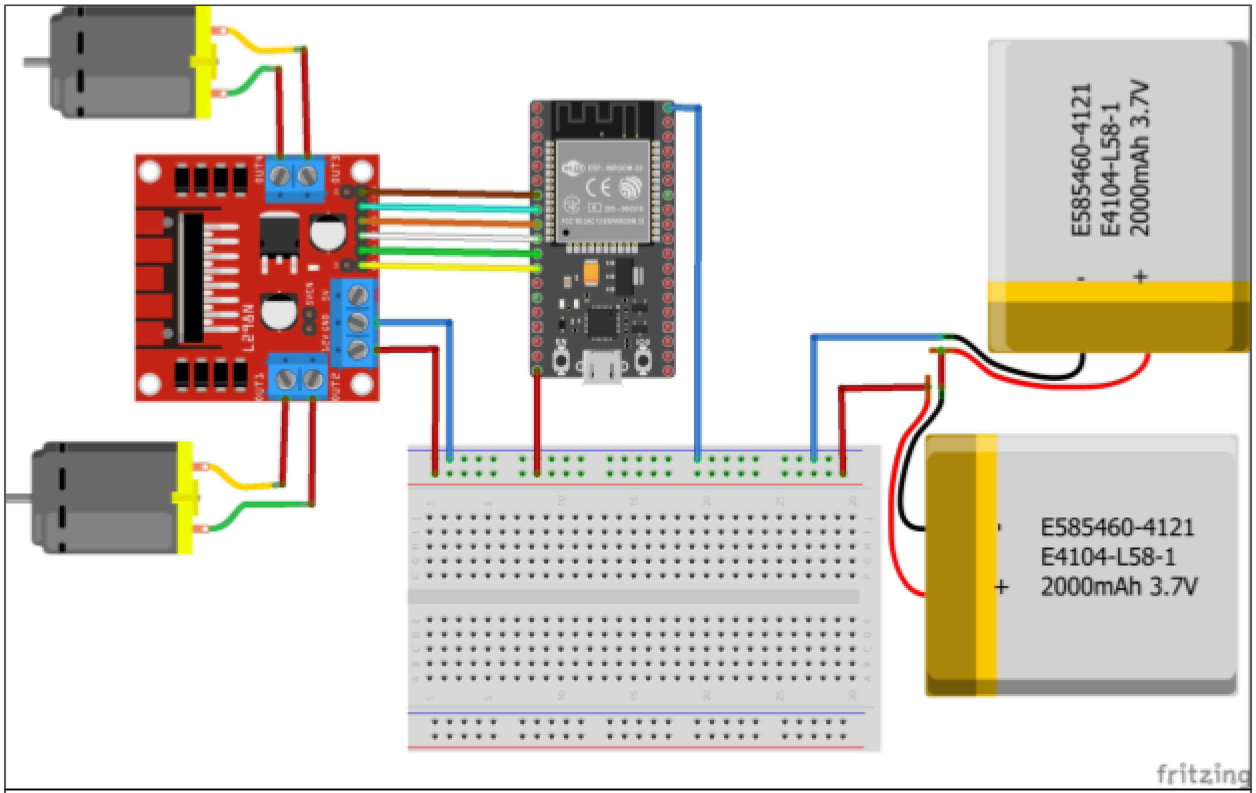

Fig 14. Car Application Schematic

Table IV:

Car Application Pinout

ESP Pins	Motor Drive Pins	Wire Color
Pin 32	Pin ENB	Brown
Pin 33	Pin IN4	Cyan
Pin 25	Pin IN3	Orange
Pin 26	Pin IN2	White
Pin 27	Pin IN1	Green
Pin 14	Pin ENA	Yellow

Peripherals needed to create:

- ESP32-Wroom-32
- 2x li-ion batteries 3.7V
- Motor driver L298N
- 2x 5V motors

Prompt to generate it: "Create code for a car controlled by two DC motors, one for each wheel, using an L298N motor driver. The car should be controllable through a mobile app with buttons for forward, backward, left, and right movement."

What it demonstrates: (Advanced Control) This application showcases the advanced control capabilities of the system. Users can create custom buttons within the mobile app to trigger specific actions on the ESP32, controlling the car's movement with precision. This highlights the potential for complex and interactive projects beyond basic functionality.

B) Application Two: Sensor station

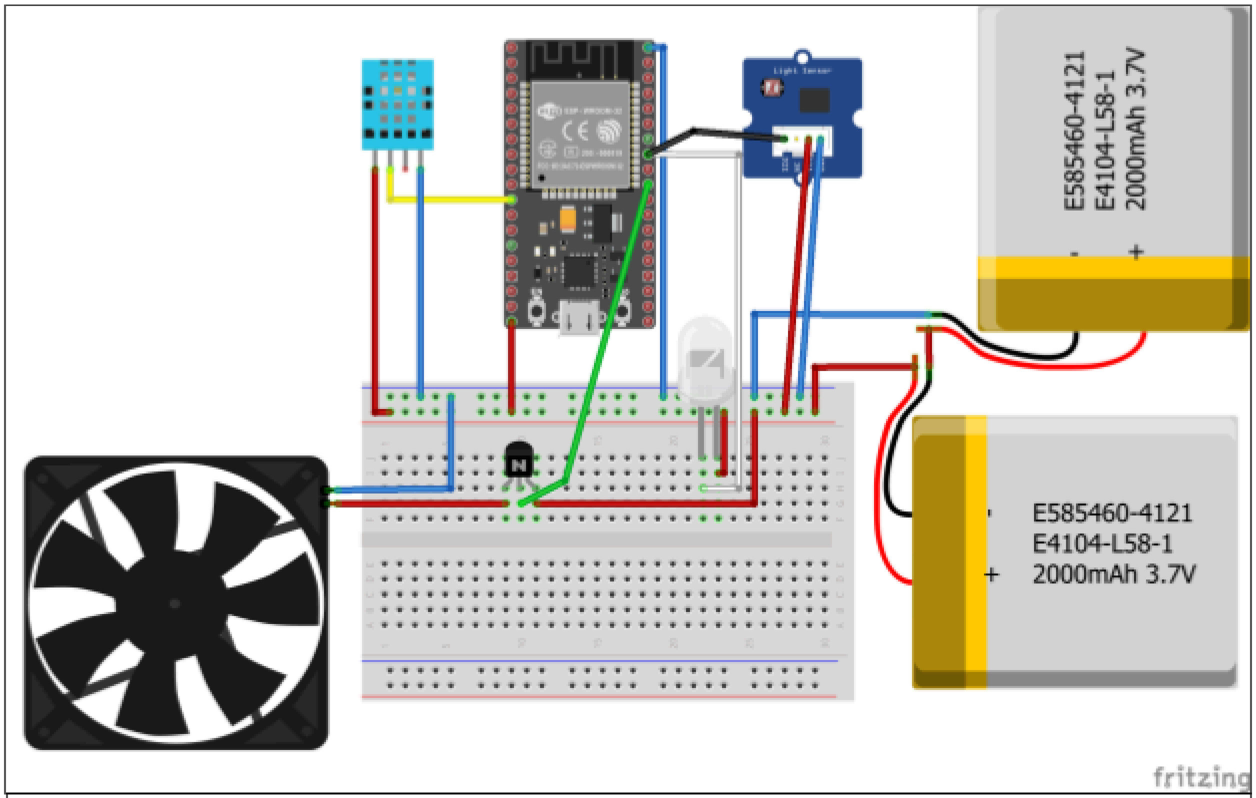

Fig 15. Sensor Station Schematic

Table V
Sensor station Pinout

ESP	DHT	Transistor	LDR	LED	Wire Color
Pin 27	Pin Data	-	-	-	Yellow
Pin 5	-	Pin Base	-	-	Green
Pin 19	-	-	Pin SIG	Anode	Black

Peripherals used to create it:

- ESP32-Wroom-32
- 2x li-ion batteries 3.7V
- DHT11 temperature sensor
- LDR Light Sensor Module
- LED
- 2N2222 Transistor
- 5-12V fan

Prompt to generate it: "Develop code for a sensor station that monitors temperature using a DHT11 sensor and light levels using an LDR sensor. If the temperature exceeds a threshold or the light level falls below a threshold, activate a fan connected through a transistor. Send sensor data to a mobile app for real-time monitoring."

What it demonstrates: (Reactivity and Ability to send information to and from the ESP32 and the Mobile App) This application exemplifies the reactive nature of LLM-generated code. The ESP32 responds to sensor readings and triggers actions based on predefined conditions, demonstrating the potential for automation and environmental control. The bidirectional communication with the mobile app allows for real-time monitoring and data visualization, further enhancing the user experience.

C) Application Three: Water Tank Control

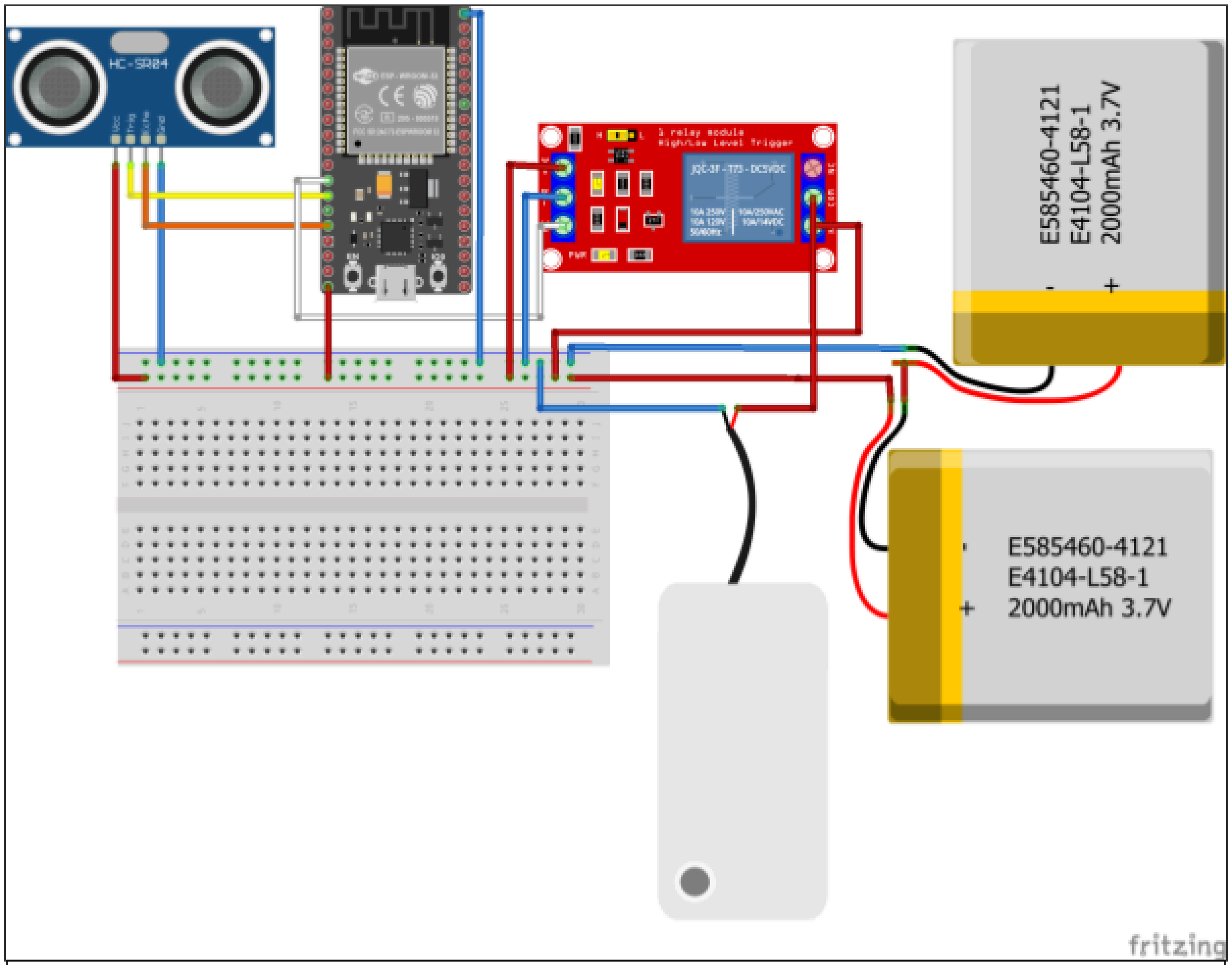

Fig. 16: Water Tank Control Schematic

Table VI
Water Tank Control Pinout

ESP Pins	Ultrasonic	Relay	Wires
Pin 12	Pin Trig	-	Yellow
Pin 13	Pin Echo	-	Orange
Pin 14	-	Pin IN	White

Peripherals used to create it:

- ESP32-Wroom-32
- 2x li-ion batteries 3.7V
- Ultrasonic Sensor HC-SR04
- 5V One Channel Relay Module Relay Switch
- 5V water pump

Prompt to generate it: "My ESP32 is connected to a Water Tank of size 100 cm through a pump, the speed of sound is 0.034, My ESP is also connected to an ultrasonic sensor HC-SR04 that is above the pump that detects the height of it, generate code to control the water Tank"

What it demonstrates: (Basic Control + Custom function) This application combines basic control with a custom function, showcasing the versatility of the system. The LLM-generated code handles both the sensor readings and the pump activation logic, demonstrating the potential for automating tasks based on real-time data.

Chapter IV
CONCLUSION & FUTURE WORK

IV. CONCLUSION & FUTURE WORK

A. Conclusion

This thesis explored the potential of LLMs, specifically Gemini, in bridging the gap between novice users and the complexities of hardware control within the burgeoning field of IoT. By leveraging the capabilities of Gemini to generate hardware code based on simple prompts, we aimed to democratize IoT development and empower individuals with limited technical expertise to engage in innovative projects.

Our research culminated in the creation of ESP-Assistant, a comprehensive no-code tool that encompasses a user-friendly mobile app, a robust backend infrastructure, and a streamlined setup process. This integrated solution tackles the challenges of code consistency and uploadability, providing a seamless experience for users to interact with the ESP32 microcontroller and bring their IoT ideas to life.

The selection of the ESP32 as the target platform was crucial due to its affordability, versatility, and open-source nature, making it an ideal entry point for IoT enthusiasts. As well as replacing Bluetooth pairing with WifiPair, we addressed memory limitations on the ESP32 and ensured a user-friendly approach to Wi-Fi configuration.

Opting for a custom OTA update solution built on AWS provided the necessary flexibility and automation, overcoming limitations encountered with existing platforms like ThingsBoard and Arduino Cloud. The combination of Golang, Docker, and various AWS services, including EC2, DynamoDB, VPC, S3, and IoT Core, created a scalable and reliable foundation for code generation, deployment, and device management.

Meticulous crafting of prompts for Gemini ensured accurate interpretation of user requirements and generation of customized code that aligns with specific hardware configurations and desired functionalities.

The effectiveness of ESP-Assistant was demonstrated through the creation of three diverse applications: a remotely controlled car, a reactive sensor station, and a water tank control system. These applications showcased the versatility of the platform, highlighting its ability to cater to various use cases and inspire further innovation.

Beyond the immediate applications, this research underscores the transformative potential of LLMs like Gemini in shaping the future of IoT development. By enabling users to translate their ideas into functional hardware code, Gemini lowers the barrier to entry and fosters a more inclusive environment for experimentation and creation. This democratization of technology holds significant implications for various sectors. ESP-Assistant can be a valuable tool for introducing students to the world of IoT and hardware programming, fostering STEM education and encouraging hands-on learning.

Empowering individuals to build their own environmental monitoring systems or smart home devices can contribute to data collection and community-driven research initiatives.

ESP-Assistant can accelerate the prototyping process for startups and entrepreneurs, enabling them to test and refine their IoT concepts rapidly.

As LLM technology continues to evolve, the possibilities for hardware code generation and IoT development are boundless. Future iterations of ESP-Assistant could incorporate more advanced features, such as:

Allowing users to interact with the system using voice commands or natural language prompts would further simplify the development process. Integrating AI algorithms to automatically optimize generated code for efficiency and performance would enhance the capabilities of created applications. Extending support to other popular microcontrollers and hardware platforms would broaden the reach and impact of the tool.

In conclusion, ESP-Assistant demonstrates the transformative power of LLMs in democratizing IoT development and empowering individuals to engage with technology in innovative ways. As we move towards a future of interconnected devices and intelligent systems, tools like ESP-Assistant will play a crucial role in shaping the landscape of IoT and fostering a new generation of creators and problem-solvers.

B. Future work

The potential of ESP-Assistant to bridge the gap between novice users and the complexities of hardware control is undeniable. However, this research serves as a foundational step in a much larger landscape of possibilities. Building upon the success of ESP-Assistant, future work aims to further refine and expand its capabilities, ultimately realizing the full potential of LLMs in shaping the future of IoT development. This journey will involve exploring new frontiers, integrating advanced functionalities, and addressing the evolving needs of a rapidly expanding user base.

A) Enhancements to Code Generation:

A crucial area of future development lies in integrating AI algorithms to optimize generated code for efficiency, performance, and resource usage. Implementing tools to analyze the performance of generated code, identifying bottlenecks and areas for improvement. Employing AI models trained on best practices and optimization techniques to automatically rewrite sections of code for better efficiency.

Developing algorithms that dynamically allocate resources based on project requirements and hardware constraints, ensuring optimal performance and minimizing resource usage. As ESP-Assistant tackles more complex projects, techniques for managing larger codebases will become increasingly important.

Future work could involve developing features that allow users to break down projects into smaller, reusable modules, improving code organization and maintainability. Implement mechanisms to track and manage dependencies between modules, ensuring proper code linking and preventing conflicts. Introduce tools for organizing code into hierarchical structures, making it easier to navigate and understand complex projects. Provide tools to easily search for specific code elements, navigate between functions and classes, and understand the overall structure of the project.

Expanding the scope of code generation to include more sophisticated functionalities will be key. This could involve features like integrating code to handle real-time data streams, enabling users to create applications that respond to sensor readings or external events in real-time. Developing functionalities to integrate machine learning models into generated code, allowing users to implement predictive analysis, pattern recognition, or other AI-powered functionalities.

Enabling ESP-Assistant to manage interactions between multiple devices and sensors, creating more sophisticated IoT systems. This could involve handling data synchronization between devices, orchestrating communication protocols, and managing complex device configurations.

B) User Experience and Accessibility:

Implementing voice commands for user interaction with ESP-Assistant would further simplify the development process, making it accessible to individuals who prefer hands-free control or lack keyboard proficiency. This could involve using APIs from Google Assistant, Amazon Alexa, or other speech recognition services to convert voice commands into text.

Creating a dedicated voice recognition engine specifically tailored for the ESP-Assistant environment, enabling offline voice control and improved accuracy. Allowing users to interact with ESP-Assistant through popular virtual assistants like Google Assistant or Amazon Alexa, providing a seamless and familiar user experience.

Improving the understanding and interpretation of natural language prompts would enhance user intuitiveness. Future work could focus on Implementing sophisticated NLP models like transformers and BERT to improve the accuracy and comprehension of user input.

Developing algorithms that take into account the context of previous interactions and user history to better understand ambiguous or incomplete prompts. Enabling ESP-Assistant to provide detailed and context-sensitive feedback to users, guiding them through the development process and explaining potential errors or limitations.

Creating a visual programming environment could make ESP-Assistant accessible to users with limited coding experience. This could involve providing a user-friendly interface where users can visually construct their programs using pre-defined blocks representing code elements like functions, loops, and conditions.

Creating visual representations of the generated code to help users understand the underlying structure and logic of their programs. Developing tutorials and guided demonstrations to introduce users to the visual programming environment and help them build basic projects.

C) Platform Expansion and Compatibility

Extending support to other popular microcontroller boards will broaden ESP-Assistant's reach and applicability. Future work could involve creating a flexible and modular architecture that allows for easy adaptation to different hardware platforms. This could involve defining standard interfaces and abstracting platform-specific details.

Offering pre-defined templates for common microcontroller boards, simplifying the process of adapting ESP-Assistant to new platforms. Developing a library of pre-defined drivers for commonly used sensors and actuators, making it easier to integrate them with ESP-Assistant projects.

Creating a flexible hardware abstraction layer will facilitate seamless integration with a wider range of devices and sensors. This could involve defining Standard Interfaces, establishing standardized interfaces for communication with various sensors and actuators, allowing for easier integration of different hardware components.

Building a library of abstractions for common hardware components, hiding platform-specific details and simplifying the development process for users.

Designing a plugin architecture that allows users to easily extend ESP-Assistant with support for new hardware components or communication protocols.

Exploring integration with other cloud platforms and IoT services would offer wider choice and flexibility for users. This could involve developing Cloud APIs: Creating APIs that allow ESP-Assistant to interact with popular cloud platforms like Google Cloud, Azure, or IBM Cloud.

Enable users to store and process data collected by their ESP-Assistant projects on cloud platforms, providing scalability and advanced analytics capabilities.

Allowing ESP-Assistant to connect with cloud-based IoT services like AWS IoT Core, Google Cloud IoT Core, or Azure IoT Hub, providing advanced features for device management, data visualization, and remote control.

D) Community Engagement and Open Source Development:

Encouraging community contributions and participation in the development of ESP-Assistant will foster a collaborative ecosystem and accelerate innovation. Future work could involve creating an Open-Source Repository: Publishing ESP-Assistant's code on a platform like GitHub, allowing developers to contribute code, documentation, and features.

Creating an online forum or community platform where developers can discuss projects, share knowledge, and collaborate on new features.

Providing recognition and rewards for valuable contributions to the project, motivating developers to actively participate.

User Feedback Integration: Implementing mechanisms for user feedback and incorporating suggestions into future updates is crucial for ensuring ESP-Assistant meets the evolving needs of its user base. This could involve:

Establishing online forums, feedback forms, or bug tracking systems where users can report issues, provide suggestions, and share their experiences.

Actively monitoring feedback channels, analyzing user input, and prioritizing updates based on user needs and suggestions.

Inviting users to participate in beta testing, provide feedback on new features, and contribute to the development process.

Creating comprehensive resources and tutorials to foster user education and promote the use of ESP-Assistant within educational settings will play a key role in democratizing IoT development. This could involve developing Online Courses, creating online courses on platforms like Coursera, Udemy, or edX to introduce users to ESP-Assistant and its capabilities.

Organizing hands-on workshops and hackathons to provide practical experience with ESP-Assistant and encourage user creativity.

Creating Interactive Tutorials and Demonstrations: Developing interactive tutorials and guided demonstrations that showcase the functionality of ESP-Assistant and provide step-by-step instructions for common tasks.

E) Expanding ESP-Assistant Capabilities:

Building upon the foundational research, future work will involve developing a fully functional heart rate monitor application. This will involve refining the LLM prompt to generate complete and optimized code for the heart rate monitor, including data processing, heart rate calculation, LED activation, and mobile app integration.

Testing and validating the integration of the AD8232 ECG sensor with the ESP32, ensuring accurate data acquisition and reliable operation.

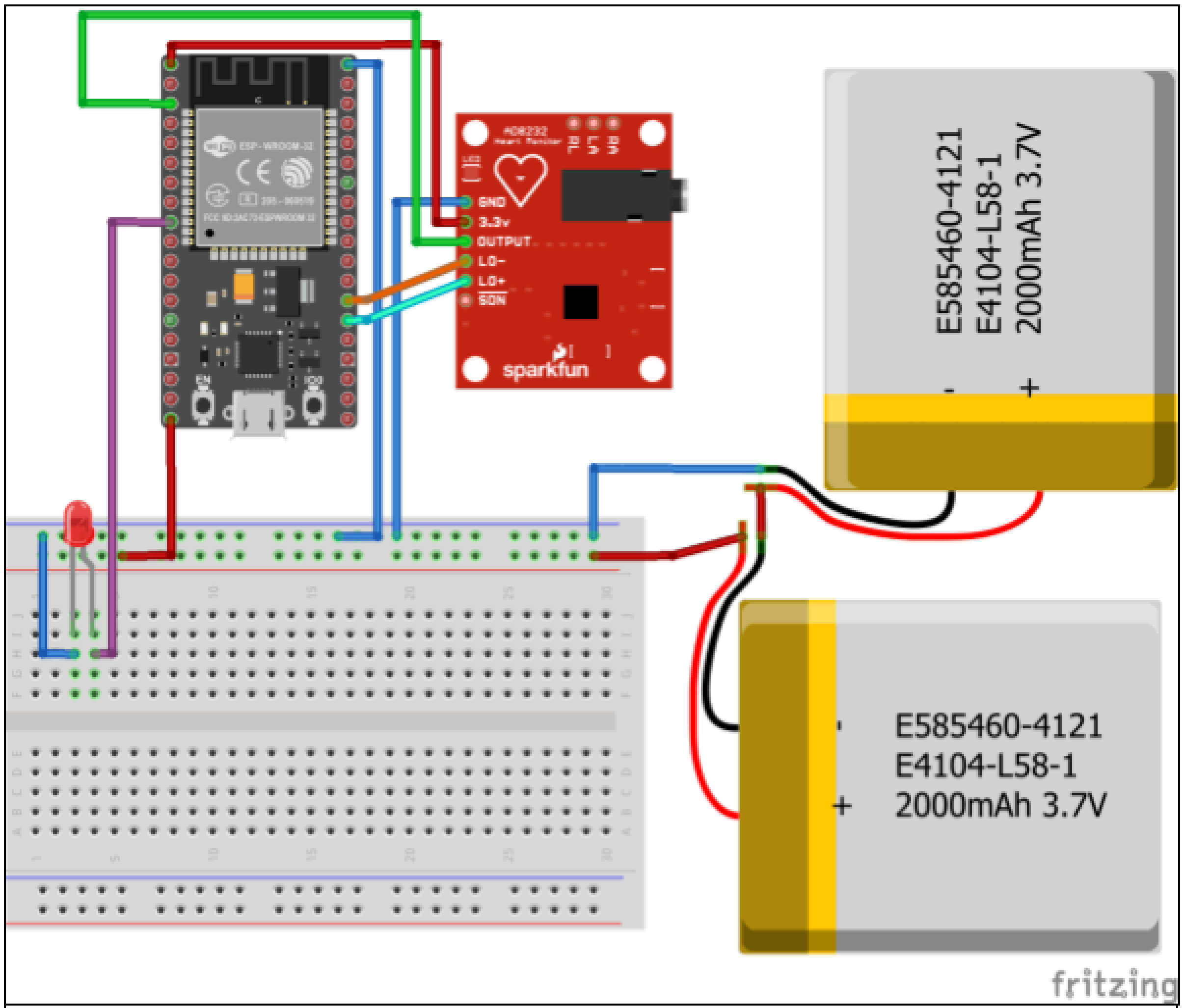

Fig 17. Heart Rate Monitor Schematic

Table VII
Heart Rate Monitor Pinout

ESP32 Pins	Heart Rate Pins	LED	Wire Color
Pin 36	Pint OUTPUT	-	Green
Pin 4	Pin LO-	-	Orange
Pin 2	Pin LO+	-	Cyan
Pin 25	-	Anode	Purple

Peripherals will be used to create it:

- ESP32-Wroom-32
- 2x li-ion batteries 3.7V
- LED
- ECG AD8232 Heart Rate Sensor

Prompt to generate it: "Create code for a heart rate monitor using an AD8232 ECG sensor. Read the sensor data and calculate the heart rate. If the heart rate exceeds a certain threshold, activate an LED as a warning indicator. Display the heart rate data on a mobile app for real-time monitoring."

What it demonstrates: (Sensor usage) This application highlights the integration of specialized sensors with the ESP32 and LLM-generated code. By utilizing the AD8232 ECG sensor, the system can capture and process complex physiological data, demonstrating its potential for healthcare applications and personal health monitoring. However this app was only generated, not actually made.

Creating a user-friendly mobile app interface for displaying real-time heart rate data and providing visual alerts based on predefined thresholds. Conducting rigorous testing and evaluation to assess the accuracy and reliability of the heart rate monitor application in real-world scenarios.

The future of ESP-Assistant holds immense potential to transform the landscape of IoT development. By focusing on enhancing its code generation capabilities, improving user experience, expanding platform compatibility, and fostering community engagement, we can empower individuals to engage with technology in innovative ways and unlock the vast possibilities of the interconnected world. This exciting journey will require continued research, collaboration, and a shared commitment to unlocking the full potential of LLMs in bridging the gap between imagination and tangible reality.

V. REFERENCES

[1] Kumar, S., Tiwari, P. & Zymbler, M. Internet of Things is a revolutionary approach for future technology enhancement: a review. J Big Data 6, 111 (2019).
https://doi.org/10.1186/s40537-019-0268-2

[2] Pons, M.; Valenzuela, E.; Rodríguez, B.; Nolazco-Flores, J.A.; Del-Valle-Soto, C. Utilization of 5G Technologies in IoT Applications: Current Limitations by Interference and Network Optimization Difficulties—A Review. Sensors 2023, 23, 3876.
https://doi.org/10.3390/s23083876

[3] S. Wu, "Introducing Devin, the first AI software engineer," *cognition.ai*, March 2024. [Online]. Available: https://www.cognition-labs.com/introducing-devin

[4] Society's Backend, "Devin Has Exposed Software Engineers," *Society's Backend*, March 2024. [Online]. Available: https://societysbackend.com/p/devin-has-exposed-software-engineers

[5] M. Przybyla, "Did the Makers of Devin AI Lie About Their Capabilities?", *Medium*, April 2024. [Online]. Available:
https://machine-learning-made-simple.medium.com/did-the-makers-of-devin-ai-lie-about-their-capabilities-cdfa818d5fc2

[6] B. Doerrfeld, "Does Using AI Assistants Lead to Lower Code Quality?" *devops.com*, February 2024. [Online]. Available:
https://devops.com/does-using-ai-assistants-lead-to-lower-code-quality/

[7] Espressif Systems, "ESP32-WROOM-32 Datasheet," May 2020

[8] S. Kumar, "ESP8266 - Deep Sleep Modes," *gist.github.com*, April 2021. [Online]. Available:
https://gist.github.com/sekcompsci/2bf39e715d5fe47579fa184fa819f421

[9] ESP Boards, "ESP32 SOC Options: Picking the Right Chip for Your Project," March 2023. [Online]. Available:
https://www.espboards.dev/blog/esp32-soc-options/#:~:text=One%20of%20the%20main%20differences,Flash%20memory%20as%20ESP32%2DS2

[10] Raspberry Pi Foundation, "Raspberry Pi 4 Model B Specifications," June 2019. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[11] Predictable Designs, "How to Choose the Best Development Kit - The Ultimate Guide for Beginners," October 2017. [Online]. Available:
https://predictabledesigns.com/how-to-choose-the-best-development-kit-the-ultimate-guide-for-beginners/

[12] Maker Electronics, “ESP32 Development Board - WiFi and Bluetooth - 30 Pin with CH9102,” May 2024. [Online]. Available: https://makerselectronics.com/product/esp32-development-board-wifi-and-bluetooth-30-pin-with-ch9102

[13] Maker Electronics, “ESP8266 NodeMCU - WiFi Programming Development Kit with CH340,” May 2024. [Online]. Available: https://makerselectronics.com/product/esp8266-nodemcu-wifi-programming-development-kit-with-ch340

[14] Maker Electronics, “Arduino Pro Micro - ATmega32U4 - MU - 5V 16MHz Module with Micro USB,” May 2024. [Online]. Available: https://makerselectronics.com/product/arduino-pro-micro-atmega32u4-mu-5v-16mhz-module-with-micro-usb

[15] Maker Electronics, “Arduino Pro Mini - 5V 16MHz,” May 2024. [Online]. Available: https://makerselectronics.com/product/arduino-pro-mini-5v-16mhz

[16] Maker Electronics, “Raspberry Pi 4 Computer Model B 8GB RAM - Made in UK,” May 2024. [Online]. Available: https://makerselectronics.com/product/raspberry-pi-4-computer-model-b-8gb-ram-made-in-uk

[17] N. Humfrey, “ESP32 - Arduino Libraries,” Mat 2024. [Online]. Available: https://www.arduinolibraries.info/architectures/esp32

[18] TowerPro, “SG-90 Servo Motor Datasheet,” Oct 2016.

[19] HandsOn Technology, “L298N Motor Driver Datasheet,” April 2017.

[20] Adafruit, “DHT11, DHT22 and AM2302 Sensors,” June 2024.

[21] SparkFun Electronics, “Ultrasonic Ranging Module HC - SR04,” August 2016.

[22] Analog Devices, “AD8232 ECG Module,” March 2019.

[23] Maxim Integrated, “MAX30100 Pulse Oximeter and Heart-Rate Sensor IC for Wearable Health,” April 2015.

[24] Very, “WiFi vs Cellular - Which is better for IoT,” June 2023. [Online]. Available: https://www.verytechnology.com/iot-insights/wifi-vs-cellular-which-is-better-for-iot

[25] Sofi, Mukhtar Ahmad. "Bluetooth Protocol in Internet of Things (IoT), Security Challenges and a Comparison with Wi-Fi Protocol: A Review." *International Journal of Engineering and Technical Research* 5 (2016).

[26] ThingsBoard, “ThingsBoard Documentation,” December 2016. [Online]. Available: https://thingsboard.io/docs/

[27] Arduino, “Arduino Cloud Documentation,” February 2019. [Online]. Available: https://docs.arduino.cc/arduino-cloud

[28] Amazon Web Services, “Amazon S3 Documentation,” February 2010. [Online]. Available: https://docs.aws.amazon.com/s3/

[29] Amazon Web Services, “AWS IoT Core Documentation,” October 2015. [Online]. Available: https://docs.aws.amazon.com/iot/

[30] Amazon Web Services, “Amazon VPC Documentation,” January 2010. [Online]. Available: https://docs.aws.amazon.com/vpc/

[31] MQTT, “mqtt.org,” July 2020. [Online]. Available: https://mqtt.org/

[32] flespi, “HTTP vs MQTT performance tests,” January 2018. [Online]. Available: https://flespi.com/blog/http-vs-mqtt-performance-tests

[33] B. Wukkadada, K. Wankhede, R. Nambiar and A. Nair, "Comparison with HTTP and MQTT In Internet of Things (IoT)," *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, Coimbatore, India, 2018, pp. 249-253, doi: 10.1109/ICIRCA.2018.8597401.

[34] Cloudflare, “What is HTTP (Hypertext Transfer Protocol)?”, October 2017. [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/

[35] Amazon, “AWS SDK for Go”, June 2015. https://aws.amazon.com/sdk-for-go/

[36] M. Yau, “arduino-cli-compile-docker,” *github.com*, Jun 2021. [Online]. Available: https://github.com/MacroYau/arduino-cli-compile-docker/blob/master/README.md

[37] Google, “Gemini API Docs and Reference,” *google.dev*, May 2024. [Online]. Available: https://ai.google.dev/gemini-api/docs

[38] Amazon Web Services, “Amazon EC2 Documentation,” January 2010. [Online]. Available: https://docs.aws.amazon.com/ec2/

[39] Amazon Web Services, “Amazon DynamoDB Documentation,” January 2012. [Online]. Available: https://docs.aws.amazon.com/dynamodb/

[40] WonderNetwork, “Pings,” February 2012. [Online]. Available: https://wondernetwork.com/pings

[41] A. Mao, “Large Language Model Settings - Temperature, Top-P, Max Tokens,” *LinkedIn*, February 2024. [Online]. Available: https://www.linkedin.com/pulse/large-language-model-settings-temperature-top-p-max-tokens-albert-mao-0c6ie