simge, sembol, grafik içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**RECEP TAYYIP ERDOGAN UNIVERSITY**

**FACULTY OF ENGINEERING AND ARCHITECTURE**

**COMPUTER ENGINEERING DEPARTMENT**

**CEN322 Internet of Things Term Project**

**2025-2026 Spring Semester**

**Homework No: Final**

**Due Date: 18.05.2025**

**Student Name Surname**

Rıdvan Şevik

**Student No**

211401051

**Instructor**

Asst. Prof. YILDIRAN YILMAZ

Contents

[1. Introduction 3](#_Toc198455974)

[2. Project Description and Objectives 3](#_Toc198455975)

[3. Methods and Materials or Requirements 5](#_Toc198455976)

[3.2 Circuit Design and Diagram 8](#_Toc198455977)

[4. Arduino Codes and Explanations 9](#_Toc198455978)

[4.1 .ino File’s Codes and Explanation 9](#_Toc198455979)

[4.2 .py File’s Codes and Explanation 28](#_Toc198455980)

[5. Results and Discussion 54](#_Toc198455981)

[6. Conclusion 55](#_Toc198455982)

[7. Appendices 56](#_Toc198455983)

[8. References 58](#_Toc198455984)

# 1. Introduction

This project explains how to build and run an interactive voice assistant. The main goal is to let users give voice commands by starting with a special word like "hello." The system, built around an ESP32 device, listens to the command, sends it to a smart AI (Large Language Model), and then speaks the answer back to the user.

The setup includes an ESP32 microcontroller (such as the ESP32-S3-DevKitM-1), which handles the sound input and output. When a hotword is heard by a program on the server, the ESP32 records the user's voice. This voice data is sent over the network using UDP to a Python server.

On the server, Google’s Speech-to-Text service turns the voice into written text. This text is then sent to an AI model through the Groq API. The AI reads the message and creates a written reply.

To speak the reply back to the user, the server uses Google Text-to-Speech (gTTS) and Pydub to turn the text into audio. The audio is sent back to the ESP32, which plays it out loud.

This project brings together a smart device (ESP32), powerful cloud services for voice recognition and understanding, and a server to manage everything. It shows how IoT and AI can work together to build a voice-controlled assistant that interacts with people in a smart way.

# 2. Project Description and Objectives

This project focuses on building a smart voice assistant that lets users talk directly to an AI (Large Language Model) using just their voice. It uses an **ESP32-S3-DevKitM-1** microcontroller to listen to the user’s voice and play back spoken answers. A Python server controls the whole system and connects to several cloud-based AI tools.

The conversation starts when the user says a specific hotword like **“hello.”** The ESP32 constantly sends audio to the server, which listens for that hotword. When the hotword is heard, the ESP32 records what the user says next and sends that voice data to the server.

On the server, **Google Cloud Speech-to-Text** turns the speech into written text. This text is then sent to a **Large Language Model** (through the **Groq API**), which reads the message and creates a response. The response is then converted into spoken audio using **Google Text-to-Speech (gTTS)** and the **Pydub** library.

Finally, the audio is sent back to the ESP32, which plays it out loud for the user. The system creates a complete hands-free experience, where the user can speak and hear answers from an AI.

**Project Goals**

* **Hotword Detection:** Build a system that always listens to the ESP32’s audio stream and starts recording only when it hears a set word like *“hello.”*
* **Speech-to-Text:** Accurately turn the user's voice into text using **Google Cloud Speech-to-Text**, with the Python server managing this process.
* **Understanding and Reply:** Use a **Large Language Model** (via the Groq API) to understand what the user is asking and generate a smart, helpful response.
* **Text-to-Speech:** Turn the AI's text response into clear, human-like speech using **gTTS** and **Pydub**, so the user can hear the reply.
* **Two-Way Audio Streaming:**
  + Stream the ESP32’s microphone audio to the server quickly and smoothly using **UDP**.
  + Stream the reply audio back from the server to the ESP32 for real-time playback.
* **ESP32 Audio Features:** Write firmware for the ESP32 to handle both recording audio from a microphone and playing it through a speaker.
* **System Integration:** Make all parts (ESP32, Python server, Google STT, Groq API, and TTS) work well together to build a reliable and interactive voice assistant.

# 3. Methods and Materials or Requirements

**Materials and Components Used:**

 **ESP32-S3-DevKitM-1U** – A microcontroller that collects gas sensor data and handles audio input/output over Wi-Fi using UDP.

 **MQ-2 Gas Sensor** – Detects gases like LPG, methane, and smoke. It gives an analog signal based on how much gas is in the air.

 **INMP441 I²S Microphone** – A digital microphone that captures surrounding sounds like voice commands.

 **PAM8403 Audio Amplifier** – A small amplifier that boosts audio signals to drive a speaker.

 **PCM5102A I²S DAC** – Converts digital audio into analog signals for clearer playback.

 **8Ω, 2W Speaker** – Outputs the final audio response to the user.

 **Breadboard & Jumper Wires** – Used to connect all the components without soldering.

 **Active Buzzer** – Makes a sound when gas levels go above a safe limit.

 **10K Ohm Resistors (x2)** – Used to reduce the MQ-2’s 5V signal to a level the ESP32 can safely read.

 **USB Cable** – Powers and programs the ESP32-S3.

**Methods and Implementation:**

**1. Circuit Setup**

* **Gas Sensor Connection:**  
  Connect the MQ-2’s output through two resistors to reduce voltage, then link it to the ESP32’s analog input. Connect the buzzer to a GPIO pin that can produce sound using PWM.
* **Audio System Wiring:**  
  Connect the INMP441 microphone’s I²S pins (SD, SCK, WS) to the ESP32. Connect the PCM5102A DAC to the ESP32’s I²S output. Then send the DAC’s analog output to the PAM8403 amplifier, which drives the speaker.

**2. Sensor and Audio Calibration**

* **Gas Sensor:**  
  Heat the MQ-2 sensor as recommended. In clean air, record the baseline analog values. Set a safe threshold to trigger alerts.
* **Audio Testing:**  
  Record test audio using the INMP441, send it through the DAC, and check if it plays clearly through the speaker.

**3. ESP32 Firmware Tasks**

* **Gas Monitoring:**  
  Continuously read the MQ-2 data. If it goes over the threshold, trigger the buzzer with a warning sound.
* **Audio Communication:**  
  Set up the ESP32 to use I²S for both recording (INMP441) and playback (PCM5102A). Capture voice, convert it to small UDP packets, and send them over Wi-Fi to a PC server. Also, receive UDP audio from the server and play it back.

**4. Python Server Tasks (on PC)**

* **Audio over UDP:**  
  Receive audio packets from the ESP32 and rebuild them into a full audio stream.
* **Speech Recognition:**  
  Use a speech-to-text engine like Whisper or Vosk to turn the voice into text.
* **LLM Integration:**  
  Send the text to a large language model (LLM) via the Groq API to understand the request and create a response.
* **Text-to-Speech:**  
  Convert the LLM’s response text into speech using gTTS or a similar library.
* **Return Audio:**  
  Send the new audio back to the ESP32 as UDP packets so it can be played.

**5. Testing and Validation**

* **Gas Safety Test:**  
  Simulate gas leaks using something like isopropyl alcohol, and make sure the buzzer activates when it should.
* **Voice Assistant Test:**  
  Speak into the microphone, check that the server recognizes and transcribes your words, gets a good reply from the LLM, and plays it back clearly.
* **Network Stability:**  
  Test the system under weak Wi-Fi or busy networks. Add basic error checks or retransmission if needed for better performance.

## 3.2 Circuit Design and Diagram

A diagram of a circuit board

AI-generated content may be incorrect.

# 4. Arduino Codes and Explanations

## 4.1 .ino File’s Codes and Explanation

// --- LIBRARIES ---

#include <WiFi.h>

#include <WiFiUdp.h>

#include "driver/i2s.h"      // ESP-IDF I2S Driver

#include <Adafruit\_NeoPixel.h> // Include Adafruit NeoPixel library

#define BLYNK\_PRINT Serial     // Redirect debug output for Blynk to Serial Port

#include "secrets.h"           // File containing secret information (WiFi passwords, Blynk token, Telegram token, etc.)

#include <BlynkSimpleEsp32.h>

#include <HTTPClient.h>

// --- Debug Level ---

#define DEBUG\_LEVEL 2 // You can set it to 0, 1, or 2 according to your needs

#if DEBUG\_LEVEL > 0

#define DEBUG\_PRINT(level, format, ...) if (DEBUG\_LEVEL >= level) { Serial.printf(PSTR("[%s] " format), millisToTimeStr(), ##\_\_VA\_ARGS\_\_); }

#else

#define DEBUG\_PRINT(level, format, ...)

#endif

// Helper function for timestamp

const char\* millisToTimeStr() {

    static char time\_str[15];

    unsigned long ms = millis();

    unsigned long s = ms / 1000;

    unsigned long m = s / 60;

    unsigned long h = m / 60;

    ms %= 1000;

    s %= 60;

    m %= 60;

    sprintf(time\_str, "%02lu:%02lu:%02lu.%03lu", h, m, s, ms);

    return time\_str;

}

// --- Wi-Fi Information (will come from secrets.h file as WIFI\_SSID and WIFI\_PASSWORD) ---

const char\* ssid = WIFI\_SSID;

const char\* password = WIFI\_PASSWORD;

const int WIFI\_CONNECT\_TIMEOUT\_MS = 20000; // 20 seconds

// --- Blynk Settings (will come from secrets.h file as BLYNK\_AUTH\_TOKEN) ---

char blynk\_auth[] = BLYNK\_AUTH\_TOKEN;

// --- Telegram Settings (will come from secrets.h file as BOT\_TOKEN and CHAT\_ID) ---

String botToken = BOT\_TOKEN;

String chatID = CHAT\_ID;

// --- UDP Settings ---

const char\* udpAddress = "192.168.149.17"; // YOUR PYTHON SERVER'S IP ADDRESS

const int udpTargetPort = 12345;          // Port where the Python server listens for AUDIO

const int udpLocalPort = 12345;           // Port where ESP32 will listen for AUDIO

const int udpGasTargetPort = 12346;       // NEW: Port where the Python server listens for GAS DATA

WiFiUDP udp;

// --- I2S General Settings ---

#define I2S\_SAMPLE\_RATE 16000

#define I2S\_BITS\_PER\_SAMPLE I2S\_BITS\_PER\_SAMPLE\_16BIT

// --- I2S Pins and Settings for Microphone Input (I2S\_NUM\_0) ---

#define I2S\_MIC\_WS          15  // LRCK

#define I2S\_MIC\_SD          14  // DOUT (Data IN for ESP32)

#define I2S\_MIC\_SCK         13  // BCLK

#define I2S\_MIC\_CHANNEL\_FMT I2S\_CHANNEL\_FMT\_ONLY\_LEFT // For mono microphone

#define I2S\_MIC\_DMA\_BUF\_COUNT 4

#define I2S\_MIC\_DMA\_BUF\_LEN   256

#define I2S\_MIC\_READ\_LEN      1024 // Buffer size to be read from microphone at once (bytes)

char i2s\_mic\_read\_buffer[I2S\_MIC\_READ\_LEN];

// --- I2S Pins and Settings for Speaker Output (I2S\_NUM\_1) ---

#define I2S\_SPEAKER\_WS      10  // GPIO you connected to PCM5102A LRCK pin

#define I2S\_SPEAKER\_SD      12  // GPIO you connected to PCM5102A DIN pin (Data OUT for ESP32)

#define I2S\_SPEAKER\_SCK     8   // GPIO you connected to PCM5102A BCK pin

#define I2S\_SPEAKER\_CHANNEL\_FMT I2S\_CHANNEL\_FMT\_ONLY\_LEFT // We assume mono audio comes from the server

#define I2S\_SPEAKER\_DMA\_BUF\_COUNT 10

#define I2S\_SPEAKER\_DMA\_BUF\_LEN   512

#define INCOMING\_AUDIO\_BUFFER\_SIZE 2048 // Buffer size for incoming audio

char incomingAudioBuffer[INCOMING\_AUDIO\_BUFFER\_SIZE];

// --- Status LED (NeoPixel Settings) ---

#define NEOPIXEL\_PIN 48      // Internal RGB LED pin for ESP32-S3-DevKitM-1U

#define NEOPIXEL\_COUNT 1     // Usually, the internal LED is a single unit

Adafruit\_NeoPixel statusPixel(NEOPIXEL\_COUNT, NEOPIXEL\_PIN, NEO\_GRB + NEO\_KHZ800);

uint32\_t color\_off, color\_wifi\_connecting, color\_wifi\_connected, color\_audio\_playing, color\_mic\_active, color\_gas\_alert;

// --- Audio Playback Status and Timeout ---

bool isPlayingAudio = false;

unsigned long lastAudioPacketTime = 0;

const unsigned long audioTimeoutMs = 500;      // Max waiting time between audio packets

const unsigned long i2sWriteTimeoutMs = 100;   // Max waiting time for I2S write operation

// --- MQ2 Gas Sensor and Buzzer Pins ---

#define MQ2\_PIN 4      // Must be a pin capable of analog reading (e.g., GPIO4, ADC1\_CH3)

#define BUZZER\_PIN 18  // Buzzer pin. Make sure it does not conflict with I2S pins.

int threshold = 1000;          // Initial threshold value (can be changed via V1 on Blynk)

bool gas\_alert\_active = false; // Track whether the gas alarm is active

// --- Timing Variables ---

unsigned long lastMq2Check = 0;

const unsigned long mq2CheckInterval = 750; // MQ2 check interval in ms

// NEW: Timing for sending gas data

unsigned long lastGasSendTime = 0;

const unsigned long gasSendInterval = 2000; // Gas data sending interval in ms (2 seconds)

// --- I2S Microphone Setup ---

void setupI2SMic() {

    DEBUG\_PRINT(1, "I2S\_MIC: Setting up I2S Microphone (I2S\_NUM\_0)...\n");

    const i2s\_config\_t i2s\_mic\_config = {

        .mode = (i2s\_mode\_t)(I2S\_MODE\_MASTER | I2S\_MODE\_RX),

        .sample\_rate = I2S\_SAMPLE\_RATE,

        .bits\_per\_sample = I2S\_BITS\_PER\_SAMPLE,

        .channel\_format = I2S\_MIC\_CHANNEL\_FMT,

        .communication\_format = I2S\_COMM\_FORMAT\_STAND\_I2S,

        .intr\_alloc\_flags = ESP\_INTR\_FLAG\_LEVEL1,

        .dma\_buf\_count = I2S\_MIC\_DMA\_BUF\_COUNT,

        .dma\_buf\_len = I2S\_MIC\_DMA\_BUF\_LEN,

        .use\_apll = false,

        .tx\_desc\_auto\_clear = false,

        .fixed\_mclk = 0

    };

    esp\_err\_t err\_install = i2s\_driver\_install(I2S\_NUM\_0, &i2s\_mic\_config, 0, NULL);

    if (err\_install != ESP\_OK) {

        Serial.printf(PSTR("[%s] I2S\_MIC: ERROR: Failed to install I2S\_NUM\_0 driver! Error Code: %d (%s)\n"), millisToTimeStr(), err\_install, esp\_err\_to\_name(err\_install));

        return;

    }

    DEBUG\_PRINT(1, "I2S\_MIC: I2S\_NUM\_0 driver installed successfully.\n");

    const i2s\_pin\_config\_t mic\_pin\_config = {

        .bck\_io\_num = I2S\_MIC\_SCK,

        .ws\_io\_num = I2S\_MIC\_WS,

        .data\_out\_num = I2S\_PIN\_NO\_CHANGE,

        .data\_in\_num = I2S\_MIC\_SD

    };

    esp\_err\_t err\_pin = i2s\_set\_pin(I2S\_NUM\_0, &mic\_pin\_config);

    if (err\_pin != ESP\_OK) {

        Serial.printf(PSTR("[%s] I2S\_MIC: ERROR: Failed to set I2S\_NUM\_0 pins! Error Code: %d (%s)\n"), millisToTimeStr(), err\_pin, esp\_err\_to\_name(err\_pin));

    } else {

        DEBUG\_PRINT(1, "I2S\_MIC: I2S\_NUM\_0 pins set successfully.\n");

    }

    DEBUG\_PRINT(1, "I2S\_MIC: I2S Microphone (I2S\_NUM\_0) setup completed.\n");

}

// --- I2S Speaker Setup ---

void setupI2SSpeaker() {

    DEBUG\_PRINT(1, "I2S\_SPK: Setting up I2S Speaker (I2S\_NUM\_1)...\n");

    const i2s\_config\_t i2s\_speaker\_config = {

        .mode = (i2s\_mode\_t)(I2S\_MODE\_MASTER | I2S\_MODE\_TX),

        .sample\_rate = I2S\_SAMPLE\_RATE,

        .bits\_per\_sample = I2S\_BITS\_PER\_SAMPLE,

        .channel\_format = I2S\_SPEAKER\_CHANNEL\_FMT,

        .communication\_format = I2S\_COMM\_FORMAT\_STAND\_I2S,

        .intr\_alloc\_flags = ESP\_INTR\_FLAG\_LEVEL1,

        .dma\_buf\_count = I2S\_SPEAKER\_DMA\_BUF\_COUNT,

        .dma\_buf\_len = I2S\_SPEAKER\_DMA\_BUF\_LEN,

        .use\_apll = false,

        .tx\_desc\_auto\_clear = true,

        .fixed\_mclk = 0

    };

    esp\_err\_t err\_install = i2s\_driver\_install(I2S\_NUM\_1, &i2s\_speaker\_config, 0, NULL);

    if (err\_install != ESP\_OK) {

        Serial.printf(PSTR("[%s] I2S\_SPK: ERROR: Failed to install I2S\_NUM\_1 driver! Error Code: %d (%s)\n"), millisToTimeStr(), err\_install, esp\_err\_to\_name(err\_install));

        return;

    }

    DEBUG\_PRINT(1, "I2S\_SPK: I2S\_NUM\_1 driver installed successfully.\n");

    const i2s\_pin\_config\_t speaker\_pin\_config = {

        .bck\_io\_num = I2S\_SPEAKER\_SCK,

        .ws\_io\_num = I2S\_SPEAKER\_WS,

        .data\_out\_num = I2S\_SPEAKER\_SD,

        .data\_in\_num = I2S\_PIN\_NO\_CHANGE

    };

    esp\_err\_t err\_pin = i2s\_set\_pin(I2S\_NUM\_1, &speaker\_pin\_config);

    if (err\_pin != ESP\_OK) {

        Serial.printf(PSTR("[%s] I2S\_SPK: ERROR: Failed to set I2S\_NUM\_1 pins! Error Code: %d (%s)\n"), millisToTimeStr(), err\_pin, esp\_err\_to\_name(err\_pin));

    } else {

        DEBUG\_PRINT(1, "I2S\_SPK: I2S\_NUM\_1 pins set successfully.\n");

    }

    DEBUG\_PRINT(1, "I2S\_SPK: I2S Speaker (I2S\_NUM\_1) setup completed.\n");

}

// --- Telegram Message Sending Function ---

void sendTelegramMessage(String message) {

    if (WiFi.status() != WL\_CONNECTED) {

        DEBUG\_PRINT(1, "TELEGRAM: WiFi not connected, message cannot be sent: %s\n", message.c\_str());

        return;

    }

    DEBUG\_PRINT(1, "TELEGRAM: Sending message: %s\n", message.c\_str());

    HTTPClient http;

    // URL encode message for safety if it contains special characters

    String urlEncodedMessage = "";

    for (char c : message) {

        if (isalnum(c) || c == '-' || c == '\_' || c == '.' || c == '~' || c == ' ') {

            urlEncodedMessage += (c == ' ' ? "%20" : String(c));

        } else {

            char buff[4];

            sprintf(buff, "%%%02X", (unsigned char)c);

            urlEncodedMessage += buff;

        }

    }

    String url = "https://api.telegram.org/bot" + botToken + "/sendMessage?chat\_id=" + chatID + "&text=" + urlEncodedMessage;

    DEBUG\_PRINT(2, "TELEGRAM: URL: %s\n", url.c\_str());

    http.begin(url); // HTTPS by default if URL starts with https://

    // For ESP32, if you face issues with HTTPS, you might need to provide a root CA certificate

    // http.begin(client, url); // where client is a WiFiClientSecure object

    int httpResponseCode = http.GET();

    if (httpResponseCode > 0) {

        DEBUG\_PRINT(2, "TELEGRAM: Message sent, HTTP Response Code: %d\n", httpResponseCode);

        String responsePayload = http.getString();

        DEBUG\_PRINT(2, "TELEGRAM: Response Content: %s\n", responsePayload.c\_str());

    } else {

        DEBUG\_PRINT(1, "TELEGRAM: ERROR: Message could not be sent, HTTP Error Code: %d, Error: %s\n", httpResponseCode, http.errorToString(httpResponseCode).c\_str());

    }

    http.end();

}

// --- Get Blynk Threshold Value (from V1 virtual pin) ---

BLYNK\_WRITE(V1) {

    int newThreshold = param.asInt();

    DEBUG\_PRINT(1, "BLYNK: New gas threshold value received from V1: %d\n", newThreshold);

    if (newThreshold > 0 && newThreshold < 4096) { // Check if it's in a reasonable range

        threshold = newThreshold;

        DEBUG\_PRINT(1, "BLYNK: Gas threshold value updated to %d.\n", threshold);

    } else {

        DEBUG\_PRINT(1, "BLYNK: Received threshold value (%d) is invalid, not updated.\n", newThreshold);

    }

}

// --- MAIN SETUP FUNCTION ---

void setup() {

    Serial.begin(115200);

    unsigned long setup\_start\_time = millis();

    for(int i=0; i<10 && !Serial; i++) delay(100);

    DEBUG\_PRINT(1, "SETUP: ESP32 Initializing...\n");

    statusPixel.begin();

    statusPixel.setBrightness(20);

    color\_off               = statusPixel.Color(0, 0, 0);

    color\_wifi\_connecting   = statusPixel.Color(0, 0, 25);

    color\_wifi\_connected    = statusPixel.Color(0, 25, 0);

    color\_audio\_playing     = statusPixel.Color(25, 0, 25);

    color\_mic\_active        = statusPixel.Color(25, 25, 0);

    color\_gas\_alert         = statusPixel.Color(50, 0, 0);

    statusPixel.setPixelColor(0, color\_off);

    statusPixel.show();

    DEBUG\_PRINT(1, "SETUP: Setting MQ2 pin (GPIO%d) as INPUT, Buzzer pin (GPIO%d) as OUTPUT.\n", MQ2\_PIN, BUZZER\_PIN);

    pinMode(MQ2\_PIN, INPUT);

    pinMode(BUZZER\_PIN, OUTPUT);

    digitalWrite(BUZZER\_PIN, HIGH); // Buzzer initially off (assuming HIGH = off)

    DEBUG\_PRINT(1, "WIFI: Connecting to Wi-Fi Network: %s\n", ssid);

    WiFi.mode(WIFI\_STA);

    WiFi.begin(ssid, password);

    unsigned long wifi\_start\_time = millis();

    bool wifi\_led\_blink\_state = false;

    while (WiFi.status() != WL\_CONNECTED) {

        wifi\_led\_blink\_state = !wifi\_led\_blink\_state;

        statusPixel.setPixelColor(0, wifi\_led\_blink\_state ? color\_wifi\_connecting : color\_off);

        statusPixel.show();

        delay(300);

        Serial.print(".");

        if (millis() - wifi\_start\_time > WIFI\_CONNECT\_TIMEOUT\_MS) {

            Serial.printf(PSTR("\n[%s] WIFI: ERROR: Wi-Fi connection timed out! (%lu ms). Restarting...\n"), millisToTimeStr(), WIFI\_CONNECT\_TIMEOUT\_MS);

            ESP.restart();

        }

    }

    Serial.printf(PSTR("\n[%s] WIFI: Wi-Fi Connected!\n"), millisToTimeStr());

    Serial.printf(PSTR("[%s] WIFI: IP Address: %s\n"), millisToTimeStr(), WiFi.localIP().toString().c\_str());

    statusPixel.setPixelColor(0, color\_wifi\_connected);

    statusPixel.show();

    DEBUG\_PRINT(1, "BLYNK: Connecting to Blynk (Token: %sxxxx)\n", String(blynk\_auth).substring(0,4).c\_str() ); // Don't log the entire token

    Blynk.config(blynk\_auth);

    unsigned long blynk\_connect\_start = millis();

    bool blynk\_connected\_successfully = false;

    while(millis() - blynk\_connect\_start < 10000){

        if (Blynk.connect(500)) {

            DEBUG\_PRINT(1, "BLYNK: Blynk connected successfully.\n");

            blynk\_connected\_successfully = true;

            break;

        }

        // DEBUG\_PRINT(2, "BLYNK: Connection attempt ongoing...\n"); // Could be very frequent logging

    }

    if (!blynk\_connected\_successfully) {

        DEBUG\_PRINT(1, "BLYNK: WARNING: Could not connect to Blynk within 10 seconds during setup. Will keep trying in loop.\n");

    }

    DEBUG\_PRINT(1, "UDP: Starting UDP listener (for audio) on port %d...\n", udpLocalPort);

    if (udp.begin(udpLocalPort)) { // This is to listen for incoming AUDIO packets

        DEBUG\_PRINT(1, "UDP: UDP listener started successfully on port %d.\n", udpLocalPort);

    } else {

        Serial.printf(PSTR("[%s] UDP: ERROR: Failed to start UDP listener!\n"), millisToTimeStr());

    }

    setupI2SMic();

    setupI2SSpeaker();

    DEBUG\_PRINT(1, "I2S\_SPK: Initially stopping I2S\_NUM\_1 (speaker) and clearing buffer.\n");

    esp\_err\_t err\_zero = i2s\_zero\_dma\_buffer(I2S\_NUM\_1);

    if (err\_zero != ESP\_OK) DEBUG\_PRINT(1, "I2S\_SPK: WARNING: Initial i2s\_zero\_dma\_buffer error: %s\n", esp\_err\_to\_name(err\_zero));

    esp\_err\_t err\_stop\_init = i2s\_stop(I2S\_NUM\_1);

    if (err\_stop\_init != ESP\_OK) Serial.printf(PSTR("[%s] I2S\_SPK: ERROR: I2S\_NUM\_1 could not be stopped initially! Error: %s\n"), millisToTimeStr(), esp\_err\_to\_name(err\_stop\_init));

    else DEBUG\_PRINT(1, "I2S\_SPK: I2S\_NUM\_1 (speaker) stopped initially.\n");

    DEBUG\_PRINT(1, "SETUP: Setup took %lu ms. Starting loop...\n", millis() - setup\_start\_time);

}

// --- MAIN LOOP FUNCTION ---

void loop() {

    unsigned long currentTime = millis();

    if (WiFi.status() == WL\_CONNECTED) {

        if (!Blynk.connected()) {

            DEBUG\_PRINT(1, "LOOP: Blynk connection lost! Attempting to reconnect...\n");

            // Blynk.connect() can be blocking. Blynk.run() already tries.

        }

        Blynk.run();

    } else {

        DEBUG\_PRINT(1, "LOOP: WiFi connection lost! Attempting to reconnect...\n");

        statusPixel.setPixelColor(0, (currentTime / 300 % 2 == 0) ? color\_wifi\_connecting : color\_off);

        statusPixel.show();

        delay(1000);

        return;

    }

    // 1. Check for Incoming Audio Packets from UDP (for Speaker) - listens on udpLocalPort

    int packetSize = udp.parsePacket();

    if (packetSize > 0) {

        IPAddress remoteIp = udp.remoteIP();

        int remotePort = udp.remotePort(); // Python server's source port (usually random)

        // Process only if it comes from the expected server and audio port (for security)

        // But we are keeping it simple for now.

        DEBUG\_PRINT(2, "UDP\_RX\_AUDIO: Received %d byte UDP packet (for audio) from %s:%d.\n", packetSize, remoteIp.toString().c\_str(), remotePort);

        int len = udp.read(incomingAudioBuffer, min(packetSize, INCOMING\_AUDIO\_BUFFER\_SIZE));

        if (len > 0) {

            if (!isPlayingAudio) {

                DEBUG\_PRINT(1, "I2S\_SPK: Audio playback starting. Initializing I2S\_NUM\_1...\n");

                esp\_err\_t err\_zero\_bfr\_play = i2s\_zero\_dma\_buffer(I2S\_NUM\_1);

                if (err\_zero\_bfr\_play != ESP\_OK) DEBUG\_PRINT(1, "I2S\_SPK: WARNING: i2s\_zero\_dma\_buffer error before playback: %s\n", esp\_err\_to\_name(err\_zero\_bfr\_play));

                esp\_err\_t err\_start = i2s\_start(I2S\_NUM\_1);

                if (err\_start != ESP\_OK) {

                    Serial.printf(PSTR("[%s] I2S\_SPK: ERROR: Failed to start I2S\_NUM\_1! Error: %s\n"), millisToTimeStr(), esp\_err\_to\_name(err\_start));

                } else {

                    DEBUG\_PRINT(2, "I2S\_SPK: I2S\_NUM\_1 started successfully.\n");

                    isPlayingAudio = true;

                }

            }

            if(isPlayingAudio) {

                lastAudioPacketTime = currentTime;

                size\_t bytes\_written;

                // DEBUG\_PRINT(2, "I2S\_SPK: Writing %d bytes of received audio data to I2S\_NUM\_1...\n", len); // Very frequent log

                esp\_err\_t err\_write = i2s\_write(I2S\_NUM\_1, incomingAudioBuffer, len, &bytes\_written, pdMS\_TO\_TICKS(i2sWriteTimeoutMs));

                if (err\_write != ESP\_OK) {

                    Serial.printf(PSTR("[%s] I2S\_SPK: ERROR: Error writing to I2S\_NUM\_1! Error: %s\n"), millisToTimeStr(), esp\_err\_to\_name(err\_write));

                }

                if (bytes\_written < len) {

                    DEBUG\_PRINT(1, "I2S\_SPK: WARNING: Not all audio data could be written! Requested: %d, Written: %d (Timeout: %lu ms)\n", len, bytes\_written, i2sWriteTimeoutMs);

                }

            }

        } else {

            DEBUG\_PRINT(1, "UDP\_RX\_AUDIO: WARNING: UDP packet received but readable data length is 0.\n");

        }

    }

    // 2. Check Audio Playback Status and Timeout

    if (isPlayingAudio && (currentTime - lastAudioPacketTime > audioTimeoutMs)) {

        DEBUG\_PRINT(1, "I2S\_SPK: Audio stream timed out (%lu ms). Stopping I2S\_NUM\_1.\n", audioTimeoutMs);

        isPlayingAudio = false;

        esp\_err\_t err\_zero\_after\_play = i2s\_zero\_dma\_buffer(I2S\_NUM\_1);

        if (err\_zero\_after\_play != ESP\_OK) DEBUG\_PRINT(1, "I2S\_SPK: WARNING: i2s\_zero\_dma\_buffer error after timeout: %s\n", esp\_err\_to\_name(err\_zero\_after\_play));

        esp\_err\_t err\_stop = i2s\_stop(I2S\_NUM\_1);

        if (err\_stop != ESP\_OK) {

            Serial.printf(PSTR("[%s] I2S\_SPK: ERROR: Failed to stop I2S\_NUM\_1! Error: %s\n"), millisToTimeStr(), esp\_err\_to\_name(err\_stop));

        } else {

            DEBUG\_PRINT(2, "I2S\_SPK: I2S\_NUM\_1 stopped successfully (timeout).\n");

        }

    }

    // 3. Read and Process MQ2 Gas Sensor (at specific intervals)

    if (currentTime - lastMq2Check >= mq2CheckInterval) {

        lastMq2Check = currentTime;

        int gasValue = analogRead(MQ2\_PIN); // Read instantaneous gas value

        // DEBUG\_PRINT(2, "MQ2: Gas Sensor Value: %d (Threshold: %d)\n", gasValue, threshold); // Can be very frequent log, exists in gasSend

        if (Blynk.connected()) {

            Blynk.virtualWrite(V0, gasValue);

        }

        bool previous\_gas\_alert\_active = gas\_alert\_active;

        if (gasValue > threshold) {

            gas\_alert\_active = true;

            if (!previous\_gas\_alert\_active) {

                DEBUG\_PRINT(1, "MQ2: DANGER! Gas Detected! Value: %d\n", gasValue);

                if (Blynk.connected()) Blynk.logEvent("gas\_warning", "Danger! Gas Detected! Sensor Value: " + String(gasValue));

                sendTelegramMessage("DANGER! Gas Detected on ESP32! Value: " + String(gasValue));

            }

            digitalWrite(BUZZER\_PIN, LOW);

        } else {

            gas\_alert\_active = false;

            if (previous\_gas\_alert\_active) {

                DEBUG\_PRINT(1, "MQ2: Gas level returned to normal. Value: %d\n", gasValue);

            }

            digitalWrite(BUZZER\_PIN, HIGH);

        }

    }

    // 4. NEW: Send Gas Data to Python Server (at specific intervals)

    if (WiFi.status() == WL\_CONNECTED && (currentTime - lastGasSendTime >= gasSendInterval)) {

        lastGasSendTime = currentTime;

        int currentGasValue = analogRead(MQ2\_PIN); // Read gas data again just before sending

        String gasDataPayload = "gas:" + String(currentGasValue) + ",alert:" + String(gas\_alert\_active ? "true" : "false");

        // Send to Python server's GAS listening port using udp object

        udp.beginPacket(udpAddress, udpGasTargetPort);

        udp.print(gasDataPayload);

        if (udp.endPacket()) {

            DEBUG\_PRINT(2, "UDP\_TX\_GAS: Gas data sent: \"%s\" -> %s:%d\n", gasDataPayload.c\_str(), udpAddress, udpGasTargetPort);

        } else {

            DEBUG\_PRINT(1, "UDP\_TX\_GAS: ERROR: Gas data UDP packet could not be sent!\n");

        }

    }

    // 5. Update LED Status and (if not playing audio) Send Microphone Data

    uint32\_t current\_led\_color = color\_off;

    if (gas\_alert\_active) {

        current\_led\_color = ((currentTime / 200) % 2 == 0) ? color\_gas\_alert : color\_off;

    } else if (isPlayingAudio) {

        current\_led\_color = ((currentTime / 150) % 2 == 0) ? color\_audio\_playing : color\_off;

    } else {

        if(WiFi.status() == WL\_CONNECTED) {

            current\_led\_color = color\_mic\_active;

        } else {

            current\_led\_color = ((currentTime / 500) % 2 == 0) ? color\_wifi\_connecting : color\_off;

        }

        // When not playing audio and no gas alarm, send microphone data to Python server's AUDIO port

        size\_t bytesRead;

        esp\_err\_t err\_read = i2s\_read(I2S\_NUM\_0, i2s\_mic\_read\_buffer, I2S\_MIC\_READ\_LEN, &bytesRead, pdMS\_TO\_TICKS(10)); // Short timeout

        if (err\_read == ESP\_OK && bytesRead > 0) {

            // DEBUG\_PRINT(2, "I2S\_MIC: %d bytes of microphone data read. Sending via UDP (audio)...\n", bytesRead); // Very frequent log

            udp.beginPacket(udpAddress, udpTargetPort); // Python server's AUDIO listening port

            size\_t sentBytes = udp.write((uint8\_t\*)i2s\_mic\_read\_buffer, bytesRead);

            if (!udp.endPacket()) {

                DEBUG\_PRINT(1, "UDP\_TX\_AUDIO: WARNING: UDP audio packet sending error (endPacket)!\n");

            } else if (sentBytes < bytesRead) {

                DEBUG\_PRINT(1, "UDP\_TX\_AUDIO: WARNING: UDP packet sent but not all bytes could be written! Read: %d, Sent: %d\n", bytesRead, sentBytes);

            }

        } else if (err\_read != ESP\_OK && err\_read != ESP\_ERR\_TIMEOUT) {

            Serial.printf(PSTR("[%s] I2S\_MIC: ERROR: Error reading from I2S\_NUM\_0! Error: %s (%d)\n"), millisToTimeStr(), esp\_err\_to\_name(err\_read), err\_read);

        }

    }

    if (statusPixel.getPixelColor(0) != current\_led\_color) {

        statusPixel.setPixelColor(0, current\_led\_color);

    }

    statusPixel.show();

    // delay(1); // To give CPU a breather in very fast loops, usually unnecessary.

}

## 4.2 .py File’s Codes and Explanation

import socket

import os

import threading

import sys

import io

import time

import json

import logging

# ... (Your existing imports will remain here) ...

# Required libraries for TTS and FFmpeg check

try:

from gtts import gTTS

from pydub import AudioSegment

# pydub needs FFmpeg or Libav.

try:

AudioSegment.converter = AudioSegment.ffmpeg or AudioSegment.avconv

if not AudioSegment.converter:

# You can define a path manually, e.g.:

# os.environ["PATH"] += os.pathsep + "C:/ffmpeg/bin" # Windows

# AudioSegment.converter = "C:/ffmpeg/bin/ffmpeg.exe" # Windows

pass # For now, we trust pydub to find it on its own.

except Exception as e\_pydub\_converter:

print(f"There might be an issue finding Pydub converter (ffmpeg/avconv): {e\_pydub\_converter}")

except ImportError:

print("ERROR: Please install 'gtts' and 'pydub' libraries. (pip install gtts pydub)")

print("Additionally, for pydub to work, FFmpeg or Libav must be installed on your system and in the PATH.")

exit(1)

# Google Cloud libraries

try:

from google.cloud import speech

from google.api\_core.client\_options import ClientOptions

except ImportError:

print("ERROR: Please install the 'google-cloud-speech' library. (pip install google-cloud-speech)")

exit(1)

# requests library (for Groq API)

try:

import requests

except ImportError:

print("ERROR: Please install the 'requests' library. (pip install requests)")

exit(1)

# --- Logging Settings ---

LOG\_LEVEL = logging.INFO # logging.DEBUG for more detailed logs

logging.basicConfig(

level=LOG\_LEVEL,

format='%(asctime)s - %(levelname)s - [%(filename)s:%(lineno)d] - %(message)s',

datefmt='%Y-%m-%d %H:%M:%S'

)

logger = logging.getLogger(\_\_name\_\_)

# --- GLOBAL SETTINGS ---

# API Keys (From Environment Variables)

GOOGLE\_API\_KEY = os.environ.get("GOOGLE\_API\_KEY")

GROQ\_API\_KEY = os.environ.get("GROQ\_API\_KEY")

# UDP Settings

UDP\_IP = "0.0.0.0" # Listen on all interfaces

UDP\_PORT = 12345 # Port for audio from ESP32

UDP\_PORT\_GAS = 12346 # NEW: Port for gas data from ESP32

SOCKET\_BUFFER\_SIZE = 4096

ESP32\_AUDIO\_CHUNK\_SIZE = 1400

ESP32\_AUDIO\_PACKET\_DELAY = 0.040

# Audio Settings

SAMPLE\_RATE = 16000

VOLUME\_REDUCTION\_DB = 0.0

MAX\_LISTEN\_SECONDS = 7.0

# Hotword and Language Settings

HOTWORD = "hello"

HOTWORD\_LANGUAGE\_CODE = "en-US"

STT\_LANGUAGE\_CODE = "en-US"

TTS\_LANGUAGE\_CODE = "en"

# Groq API Settings

GROQ\_MODEL = "llama3-70b-8192"

GROQ\_TEMPERATURE = 0.7

GROQ\_MAX\_TOKENS = 1024

GROQ\_TIMEOUT\_SECONDS = 30

# Conversation History

MAX\_CONVERSATION\_HISTORY\_PAIRS = 5

conversation\_history = [

{

"role": "system",

"content": f"You are a helpful AI assistant. Your responses should be in {STT\_LANGUAGE\_CODE.split('-')[0]}."

" Keep your answers concise and short"

}

]

# --- Global Variables ---

speech\_client = None

udp\_socket = None

udp\_socket\_gas = None # NEW: Socket for gas data

last\_known\_client\_address = None

stop\_audio\_event = threading.Event()

main\_loop\_stop\_event = threading.Event()

gas\_listener\_stop\_event = threading.Event() # NEW: Stop event for gas listener

# NEW: Global variables for gas data

latest\_gas\_data = {

"value": None,

"alert": None,

"timestamp": 0

}

gas\_data\_lock = threading.Lock()

GAS\_DATA\_STALE\_SECONDS = 30 # Gas data will not be used if older than 30 seconds

def initialize\_clients\_and\_socket():

global speech\_client, udp\_socket, udp\_socket\_gas # udp\_socket\_gas added

if not GOOGLE\_API\_KEY:

logger.critical("ERROR: GOOGLE\_API\_KEY environment variable is not set!")

return False

if not GROQ\_API\_KEY:

logger.critical("ERROR: GROQ\_API\_KEY environment variable is not set!")

return False

try:

logger.info("Initializing Google Speech client...")

client\_options = ClientOptions(api\_key=GOOGLE\_API\_KEY)

speech\_client = speech.SpeechClient(client\_options=client\_options)

logger.info("Google Speech client initialized successfully.")

except Exception as e:

logger.critical(f"Failed to initialize Google Speech client. Error: {e}", exc\_info=True)

return False

try:

logger.info(f"Binding main UDP socket to {UDP\_IP}:{UDP\_PORT} (for audio)...")

udp\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

udp\_socket.bind((UDP\_IP, UDP\_PORT))

logger.info(f"Main UDP socket successfully bound to {UDP\_IP}:{UDP\_PORT}.")

except socket.error as e:

logger.critical(f"Error binding main UDP socket: {e}", exc\_info=True)

return False

# NEW: UDP socket for gas data

try:

logger.info(f"Binding gas data UDP socket to {UDP\_IP}:{UDP\_PORT\_GAS}...")

udp\_socket\_gas = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

udp\_socket\_gas.bind((UDP\_IP, UDP\_PORT\_GAS))

logger.info(f"Gas data UDP socket successfully bound to {UDP\_IP}:{UDP\_PORT\_GAS}.")

except socket.error as e:

logger.critical(f"Error binding gas data UDP socket: {e}", exc\_info=True)

# Since the main socket was successful, we might not return False here, but gas data cannot be received.

# For now, let's allow it to continue; it can work without gas data.

udp\_socket\_gas = None # Leave as None in case of error

return True

# NEW: Function to listen for gas data

def listen\_for\_gas\_data():

global latest\_gas\_data, gas\_data\_lock, udp\_socket\_gas

if not udp\_socket\_gas:

logger.warning("Gas data socket could not be initialized, so gas data cannot be listened for.")

return

logger.info(f"Waiting for gas data from ESP32 on UDP port {UDP\_PORT\_GAS}...")

while not gas\_listener\_stop\_event.is\_set():

try:

udp\_socket\_gas.settimeout(1.0) # Timeout to prevent blocking

data, addr = udp\_socket\_gas.recvfrom(SOCKET\_BUFFER\_SIZE) # Buffer size can be small

if data:

data\_str = data.decode('utf-8').strip()

logger.debug(f"Gas data received ({addr}): {data\_str}")

# Expected format: "gas:VALUE,alert:STATUS" (e.g., "gas:450,alert:false")

parts = data\_str.split(',')

gas\_val\_str = None

alert\_str = None

if len(parts) == 2:

if parts[0].startswith("gas:"):

gas\_val\_str = parts[0][len("gas:"):]

if parts[1].startswith("alert:"):

alert\_str = parts[1][len("alert:"):]

if gas\_val\_str is not None and alert\_str is not None:

try:

gas\_value = int(gas\_val\_str)

is\_alert = alert\_str.lower() == "true"

with gas\_data\_lock:

latest\_gas\_data["value"] = gas\_value

latest\_gas\_data["alert"] = is\_alert

latest\_gas\_data["timestamp"] = time.time()

logger.debug(f"Gas data updated: Value={gas\_value}, Alert={is\_alert}")

except ValueError:

logger.warning(f"Received gas data format invalid (not numeric): {data\_str}")

else:

logger.warning(f"Received gas data format invalid: {data\_str}")

except socket.timeout:

continue # Timeout is normal, continue listening

except Exception as e:

if not gas\_listener\_stop\_event.is\_set(): # Log error only if not stopped

logger.error(f"Error while listening for gas data: {e}", exc\_info=True)

time.sleep(0.5) # Short wait in case of error

logger.info("Gas data listening thread terminated.")

def generate\_audio\_requests(current\_stop\_event):

global last\_known\_client\_address, udp\_socket

while not current\_stop\_event.is\_set():

try:

udp\_socket.settimeout(0.1)

data, addr = udp\_socket.recvfrom(SOCKET\_BUFFER\_SIZE)

if not current\_stop\_event.is\_set() and data:

if last\_known\_client\_address != addr:

logger.info(f"New ESP32 address detected (for audio): {addr}")

last\_known\_client\_address = addr

yield speech.StreamingRecognizeRequest(audio\_content=data)

except socket.timeout:

continue

except Exception as e:

if not current\_stop\_event.is\_set():

logger.error(f"Socket error in generate\_audio\_requests: {e}", exc\_info=True)

break

logger.debug(f"generate\_audio\_requests terminated (event set: {current\_stop\_event.is\_set()}).")

def get\_groq\_response(user\_text):

global conversation\_history, latest\_gas\_data, gas\_data\_lock

logger.info(f"Sending to Groq API (original): \"{user\_text}\"")

user\_text\_for\_api = user\_text

with gas\_data\_lock:

gas\_value = latest\_gas\_data["value"]

is\_alert = latest\_gas\_data["alert"]

data\_timestamp = latest\_gas\_data["timestamp"]

# Enrich user command to send gas data to LLM

if gas\_value is not None and (time.time() - data\_timestamp) < GAS\_DATA\_STALE\_SECONDS:

status\_text = "ALERT" if is\_alert else "Normal"

gas\_info = f"Current gas sensor reading: {gas\_value} (Status: {status\_text})."

user\_text\_for\_api = f"{gas\_info} if the user asks something about this, answer it, but if the user doesn't ask something about this you don't have to say it, the user's query is: {user\_text}"

logger.info(f"Using gas data (age: {time.time() - data\_timestamp:.1f}s). Enriched text to be sent to API: \"{user\_text\_for\_api}\"")

elif gas\_value is not None:

logger.info(f"Gas data is stale (age: {time.time() - data\_timestamp:.1f}s), will not be added to LLM prompt.")

else:

logger.info("No available gas data, will not be added to LLM prompt.")

current\_conversation\_for\_api = [conversation\_history[0]] # System message

user\_assistant\_pairs = []

for i in range(1, len(conversation\_history), 2):

if i + 1 < len(conversation\_history):

user\_assistant\_pairs.append(

(conversation\_history[i], conversation\_history[i+1])

)

for user\_msg, assistant\_msg in user\_assistant\_pairs[-MAX\_CONVERSATION\_HISTORY\_PAIRS:]:

current\_conversation\_for\_api.append(user\_msg)

current\_conversation\_for\_api.append(assistant\_msg)

# The last user message to be sent to the API is the enriched one

current\_conversation\_for\_api.append({"role": "user", "content": user\_text\_for\_api})

logger.debug(f"Messages to be sent to Groq ({len(current\_conversation\_for\_api)} count): {json.dumps(current\_conversation\_for\_api, indent=2)}")

payload = {

"model": GROQ\_MODEL,

"messages": current\_conversation\_for\_api, # Use the enriched conversation

"temperature": GROQ\_TEMPERATURE,

"max\_tokens": GROQ\_MAX\_TOKENS,

"stream": False

}

headers = {

"Authorization": f"Bearer {GROQ\_API\_KEY}",

"Content-Type": "application/json"

}

api\_url = "https://api.groq.com/openai/v1/chat/completions"

try:

response = requests.post(api\_url, headers=headers, json=payload, timeout=GROQ\_TIMEOUT\_SECONDS)

response.raise\_for\_status()

response\_data = response.json()

assistant\_response\_text = response\_data["choices"][0]["message"]["content"]

logger.info(f"Response received from Groq API: \"{assistant\_response\_text}\"")

# Save the original user text to conversation history

conversation\_history.append({"role": "user", "content": user\_text})

conversation\_history.append({"role": "assistant", "content": assistant\_response\_text})

while (len(conversation\_history) -1) / 2 > MAX\_CONVERSATION\_HISTORY\_PAIRS:

conversation\_history.pop(1)

conversation\_history.pop(1)

logger.debug(f"Updated conversation history ({len(conversation\_history)} count): {json.dumps(conversation\_history, indent=2)}")

return assistant\_response\_text

except requests.exceptions.Timeout:

logger.error(f"Groq API request timed out ({GROQ\_TIMEOUT\_SECONDS} seconds).")

return "Sorry, I couldn't reach my brain in time."

# ... (other except blocks will remain the same) ...

except requests.exceptions.HTTPError as e:

logger.error(f"Groq API HTTP Error: {e.response.status\_code} - {e.response.text}", exc\_info=True)

return "Sorry, there was an issue with my connection to the thinking cap."

except requests.exceptions.RequestException as e:

logger.error(f"Groq API Connection Error: {e}", exc\_info=True)

return "Sorry, I'm having trouble connecting right now."

except Exception as e:

logger.error(f"Unexpected error while processing response from Groq API: {e}", exc\_info=True)

return "Sorry, a little hiccup happened while I was thinking."

def text\_to\_pcm\_audio(text\_to\_speak, lang=TTS\_LANGUAGE\_CODE, sample\_rate=SAMPLE\_RATE, volume\_reduction\_db=VOLUME\_REDUCTION\_DB):

# ... (This function will remain the same) ...

tts\_conversion\_start\_time = time.time()

logger.info(f"Text for TTS: \"{text\_to\_speak}\" (Language: {lang}, Volume Reduction: {volume\_reduction\_db}dB)")

try:

tts = gTTS(text=text\_to\_speak, lang=lang, slow=False)

mp3\_fp = io.BytesIO()

tts.write\_to\_fp(mp3\_fp)

mp3\_fp.seek(0)

audio\_segment = AudioSegment.from\_file(mp3\_fp, format="mp3")

if volume\_reduction\_db > 0:

logger.debug(f"Reducing volume by {volume\_reduction\_db} dB.")

audio\_segment = audio\_segment - volume\_reduction\_db

elif volume\_reduction\_db < 0:

logger.debug(f"Increasing volume by {-volume\_reduction\_db} dB.")

audio\_segment = audio\_segment + (-volume\_reduction\_db)

audio\_segment = audio\_segment.set\_channels(1)

audio\_segment = audio\_segment.set\_frame\_rate(sample\_rate)

audio\_segment = audio\_segment.set\_sample\_width(2) # 16-bit

pcm\_data = audio\_segment.raw\_data

duration\_seconds = len(pcm\_data) / (sample\_rate \* 1 \* 2.0)

processing\_time = time.time() - tts\_conversion\_start\_time

logger.info(

f"PCM audio data created: {len(pcm\_data)} bytes, "

f"Audio Duration: {duration\_seconds:.2f}s, {sample\_rate}Hz, 16-bit Mono. "

f"TTS+PCM PROCESSING TIME: {processing\_time:.2f} SECONDS"

)

return pcm\_data

except Exception as e:

if "Language not supported" in str(e) or "failed to read" in str(e).lower():

logger.error(f"TTS Error: Language '{lang}' may not be supported by gTTS or there is an issue with FFmpeg/Libav. Details: {e}", exc\_info=True)

if "FFMPEG\_PATH" not in os.environ and AudioSegment.converter is None:

logger.error("It seems FFmpeg/Libav is not installed and in PATH, or pydub could not find it.")

else:

logger.error(f"TTS or audio conversion error: {e}", exc\_info=True)

return None

def send\_audio\_to\_esp32(pcm\_audio\_to\_send):

# ... (This function will remain the same) ...

if not pcm\_audio\_to\_send:

logger.warning("No PCM audio data to send.")

return

if not last\_known\_client\_address:

logger.warning("ESP32 address is unknown, audio cannot be sent.")

return

logger.info(f"--> Sending assistant's voice response ({len(pcm\_audio\_to\_send)} bytes PCM) to ESP32 ({last\_known\_client\_address})...")

total\_chunks = (len(pcm\_audio\_to\_send) + ESP32\_AUDIO\_CHUNK\_SIZE - 1) // ESP32\_AUDIO\_CHUNK\_SIZE

logger.info(f" Will be sent in a total of {total\_chunks} packets.")

for i in range(0, len(pcm\_audio\_to\_send), ESP32\_AUDIO\_CHUNK\_SIZE):

chunk = pcm\_audio\_to\_send[i:i + ESP32\_AUDIO\_CHUNK\_SIZE]

try:

udp\_socket.sendto(chunk, last\_known\_client\_address)

if ESP32\_AUDIO\_PACKET\_DELAY > 0:

time.sleep(ESP32\_AUDIO\_PACKET\_DELAY)

except socket.error as e:

logger.error(f"Socket error while sending audio packet to ESP32 ({last\_known\_client\_address}): {e}", exc\_info=True)

break

logger.info(f" All audio packets sent to ESP32.")

def process\_transcription\_and\_respond(final\_transcript\_text):

# ... (This function will remain the same, the call to get\_groq\_response will use the enriched data) ...

if not final\_transcript\_text:

logger.info("Empty transcript received, no action will be taken.")

return

logger.info(f"Final transcript to process: \"{final\_transcript\_text}\"")

groq\_assistant\_text\_response = get\_groq\_response(final\_transcript\_text) # Logic within get\_groq\_response will add gas data

if groq\_assistant\_text\_response:

logger.info(f"Assistant (Text): {groq\_assistant\_text\_response}")

pcm\_audio\_to\_send = text\_to\_pcm\_audio(

groq\_assistant\_text\_response,

lang=TTS\_LANGUAGE\_CODE,

sample\_rate=SAMPLE\_RATE,

volume\_reduction\_db=VOLUME\_REDUCTION\_DB

)

send\_audio\_to\_esp32(pcm\_audio\_to\_send)

def listen\_for\_hotword(current\_stop\_event):

# ... (This function will remain the same) ...

global speech\_client, last\_known\_client\_address

logger.info(f"\nWaiting for '{HOTWORD}' command ({HOTWORD\_LANGUAGE\_CODE})...")

if not last\_known\_client\_address: # At least one audio packet (or gas packet) must come from ESP32 before hotword

logger.warning("ESP32 address is unknown for hotword listening. Please wait for ESP32 to send audio or gas data.")

# Gas data listening thread might update the address, so a short wait can be added

# However, main address detection is done at the beginning of main\_loop.

# time.sleep(1)

# if not last\_known\_client\_address:

# logger.error("ESP32 address still not detected. Hotword listening cannot be started.")

# return False

pass # Address detection in main\_loop is more important

stt\_config\_hotword = speech.RecognitionConfig(

encoding=speech.RecognitionConfig.AudioEncoding.LINEAR16,

sample\_rate\_hertz=SAMPLE\_RATE,

language\_code=HOTWORD\_LANGUAGE\_CODE,

)

streaming\_stt\_config\_hotword = speech.StreamingRecognitionConfig(

config=stt\_config\_hotword,

interim\_results=True

)

audio\_stream\_generator = generate\_audio\_requests(current\_stop\_event)

try:

responses = speech\_client.streaming\_recognize(streaming\_stt\_config\_hotword, audio\_stream\_generator)

for response in responses:

if current\_stop\_event.is\_set():

logger.info("Hotword listening stopped externally.")

break

if not response.results: continue

result = response.results[0]

if not result.alternatives: continue

transcript\_segment = result.alternatives[0].transcript.strip().lower()

if HOTWORD in transcript\_segment:

logger.info(f"'{HOTWORD}' command detected!")

sys.stdout.write('\r' + ' ' \* 80 + '\r')

sys.stdout.flush()

print(f"'{HOTWORD}' command detected!")

return True

except Exception as e\_stream:

if "rst stream" in str(e\_stream).lower() or "service unavailable" in str(e\_stream).lower() or "internal" in str(e\_stream).lower():

logger.warning(f"Google STT connection/service issue during hotword listening: {e\_stream}. Will retry.")

elif "inactive" in str(e\_stream).lower() or "client\_stream\_inactive" in str(e\_stream).lower():

logger.debug(f"STT stream became inactive during hotword listening (could be silence): {e\_stream}")

elif "maximum allowed stream duration" in str(e\_stream).lower():

logger.info("Hotword listening STT exceeded maximum stream duration, will restart.")

else:

logger.error(f"Unexpected error during hotword listening: {e\_stream}", exc\_info=LOG\_LEVEL==logging.DEBUG)

finally:

pass

return False

def listen\_for\_command\_and\_respond():

global stop\_audio\_event, speech\_client, last\_known\_client\_address

if not last\_known\_client\_address:

logger.warning("ESP32 address is unknown for command listening.")

return

logger.info(f"\nListening (main command)... ({STT\_LANGUAGE\_CODE}). Maximum {MAX\_LISTEN\_SECONDS} seconds.")

stop\_audio\_event.clear()

start\_time = time.time()

stt\_config\_command = speech.RecognitionConfig(

encoding=speech.RecognitionConfig.AudioEncoding.LINEAR16,

sample\_rate\_hertz=SAMPLE\_RATE,

language\_code=STT\_LANGUAGE\_CODE,

)

streaming\_stt\_config\_command = speech.StreamingRecognitionConfig(

config=stt\_config\_command,

interim\_results=True

)

prompt\_text = "I'm listening." if TTS\_LANGUAGE\_CODE == "en" else "Listening."

pcm\_prompt\_sound = text\_to\_pcm\_audio(prompt\_text, lang=TTS\_LANGUAGE\_CODE)

if pcm\_prompt\_sound:

send\_audio\_to\_esp32(pcm\_prompt\_sound)

audio\_stream\_generator\_command = generate\_audio\_requests(stop\_audio\_event)

final\_transcript\_text = ""

processed\_final\_for\_session = False

try:

responses = speech\_client.streaming\_recognize(streaming\_stt\_config\_command, audio\_stream\_generator\_command)

for response in responses:

if time.time() - start\_time > MAX\_LISTEN\_SECONDS:

if not stop\_audio\_event.is\_set():

logger.info(f"Command listening time expired ({MAX\_LISTEN\_SECONDS} seconds).")

stop\_audio\_event.set()

if stop\_audio\_event.is\_set() and not processed\_final\_for\_session and final\_transcript\_text.strip():

logger.info("Command listening time finished/stopped, processing current final transcript...")

process\_transcription\_and\_respond(final\_transcript\_text.strip())

processed\_final\_for\_session = True

break

if stop\_audio\_event.is\_set() and processed\_final\_for\_session:

break

if not response.results: continue

result = response.results[0]

if not result.alternatives: continue

transcript\_segment = result.alternatives[0].transcript.strip()

if result.is\_final:

logger.debug(f"Command STT (Final Segment): \"{transcript\_segment}\"")

final\_transcript\_text += transcript\_segment + " "

sys.stdout.write('\r' + ' ' \* 80 + '\r')

sys.stdout.flush()

print(f"User (Command Completed): {final\_transcript\_text.strip()}")

if not processed\_final\_for\_session:

process\_transcription\_and\_respond(final\_transcript\_text.strip())

processed\_final\_for\_session = True

if not stop\_audio\_event.is\_set():

logger.info("Final command transcript received, stopping listening.")

stop\_audio\_event.set()

break

else:

current\_display\_text = final\_transcript\_text + transcript\_segment

sys.stdout.write('\r' + ' ' \* 80 + '\r')

sys.stdout.write(f"Recognizing (Command): {current\_display\_text}")

sys.stdout.flush()

if not processed\_final\_for\_session and final\_transcript\_text.strip():

logger.info("Unprocessed transcript found at the end of command listening loop, processing...")

sys.stdout.write('\r' + ' ' \* 80 + '\r'); sys.stdout.flush()

print(f"User (Command Partial/Timeout): {final\_transcript\_text.strip()}")

process\_transcription\_and\_respond(final\_transcript\_text.strip())

elif not final\_transcript\_text.strip() and not processed\_final\_for\_session:

logger.info("No significant speech detected in this command listening session.")

sys.stdout.write('\r' + ' ' \* 80 + '\r'); sys.stdout.flush()

no\_understand\_text = "Sorry, I didn't catch that." if TTS\_LANGUAGE\_CODE == "en" else "Sorry, I couldn't understand."

pcm\_error\_sound = text\_to\_pcm\_audio(no\_understand\_text, lang=TTS\_LANGUAGE\_CODE)

send\_audio\_to\_esp32(pcm\_error\_sound)

except Exception as e\_stream:

# ... (error handling will remain the same) ...

if "maximum allowed stream duration" in str(e\_stream).lower():

logger.warning(f"Google STT exceeded maximum stream duration (command listening): {e\_stream}.")

if not processed\_final\_for\_session and final\_transcript\_text.strip():

logger.info("Maximum stream duration exceeded, processing current command transcript...")

process\_transcription\_and\_respond(final\_transcript\_text.strip())

elif "rst stream" in str(e\_stream).lower() or "service unavailable" in str(e\_stream).lower() or "internal" in str(e\_stream).lower():

logger.warning(f"Google STT connection/service issue during command listening: {e\_stream}.")

elif "inactive" in str(e\_stream).lower() or "client\_stream\_inactive" in str(e\_stream).lower():

logger.debug(f"STT stream became inactive during command listening (could be silence): {e\_stream}")

else:

logger.error(f"Error during command listening and transcription: {e\_stream}", exc\_info=LOG\_LEVEL==logging.DEBUG)

finally:

if not stop\_audio\_event.is\_set():

stop\_audio\_event.set()

sys.stdout.write('\r' + ' ' \* 80 + '\r')

sys.stdout.flush()

def main\_loop():

global last\_known\_client\_address, main\_loop\_stop\_event, stop\_audio\_event

# We use generate\_audio\_requests for address detection,

# assuming ESP32 can send both audio and gas data.

# Even if gas data comes from a different port, the address is learned when the first packet comes from the audio port.

if not last\_known\_client\_address:

logger.info("Program starting. Waiting ~5 seconds to detect ESP32 address (from audio or gas packet)...")

# This part relies on generate\_audio\_requests receiving the first packets.

# last\_known\_client\_address will be updated via udp\_socket.

# The gas thread will also be running on its own.

# For simplicity, let's wait for the first audio packet or gas packet.

# Address detection for gas data can be done indirectly by the gas thread

# but setting last\_known\_client\_address is important for voice communication.

# A simple wait for the first packet to arrive from ESP32.

# generate\_audio\_requests sets last\_known\_client\_address when it receives the first audio packet.

# The gas data thread also runs on its own and can provide address information in its logs.

detection\_start\_time = time.time()

initial\_wait\_completed = False

while not last\_known\_client\_address and (time.time() - detection\_start\_time < 7): # Wait 7 seconds

logger.debug("Waiting for the first audio/data packet from ESP32 (for address detection)...")

time.sleep(0.5) # Allow time for generate\_audio\_requests to run

# During this wait, generate\_audio\_requests doesn't need to be running in a thread,

# it performs address detection when called within listen\_for\_hotword.

# For now, let's leave this part passive; the address will be learned when listen\_for\_hotword is first called.

initial\_wait\_completed = True # Continue even if there's no address after this loop

if not last\_known\_client\_address and initial\_wait\_completed:

logger.warning("ESP32 address could not be detected initially (from audio port). Hotword listening will still be attempted.")

elif last\_known\_client\_address:

logger.info(f"ESP32 audio address detected initially: {last\_known\_client\_address}")

try:

while not main\_loop\_stop\_event.is\_set():

if not last\_known\_client\_address: # Check in every loop, maybe ESP32 connects later

logger.info("ESP32 address (for audio) is not yet known. Waiting for packet...")

# We can temporarily call generate\_audio\_requests to try to get a packet.

temp\_stop\_event = threading.Event()

audio\_gen\_for\_addr = generate\_audio\_requests(temp\_stop\_event)

try:

next(audio\_gen\_for\_addr) # Try to get a packet

except StopIteration:

pass # Nothing came

except Exception as e\_addr:

logger.debug(f"Error in temporary listening for address detection: {e\_addr}")

finally:

temp\_stop\_event.set()

if not last\_known\_client\_address:

time.sleep(1) # If no address, wait a bit and continue loop

continue # Skip hotword listening

hotword\_detected = listen\_for\_hotword(main\_loop\_stop\_event)

if main\_loop\_stop\_event.is\_set():

break

if hotword\_detected:

listen\_for\_command\_and\_respond()

if main\_loop\_stop\_event.is\_set():

break

logger.info(f"\nWaiting for '{HOTWORD}' command again...")

else:

if main\_loop\_stop\_event.is\_set(): break

pass

except KeyboardInterrupt:

logger.info("\nProgram terminating (Ctrl+C detected)...")

except Exception as e:

logger.critical(f"An unexpected general error occurred in the main loop: {e}", exc\_info=True)

finally:

logger.info("Main loop terminating, setting all events.")

main\_loop\_stop\_event.set()

stop\_audio\_event.set()

gas\_listener\_stop\_event.set() # NEW: Stop the gas listener

if udp\_socket:

logger.info("Closing main UDP socket...")

udp\_socket.close()

logger.info("Main UDP socket closed.")

if udp\_socket\_gas: # NEW

logger.info("Closing gas data UDP socket...")

udp\_socket\_gas.close()

logger.info("Gas data UDP socket closed.")

if \_\_name\_\_ == "\_\_main\_\_":

logger.info(f"===== Voice Assistant Server Starting (Hotword: '{HOTWORD}') =====")

if not AudioSegment.converter:

logger.warning("FFmpeg/Libav path for Pydub could not be found automatically.")

# ... (existing FFmpeg warnings) ...

else:

logger.info(f"Converter found for Pydub: {AudioSegment.converter}")

if initialize\_clients\_and\_socket():

# NEW: Start the gas listening thread

gas\_thread = None

if udp\_socket\_gas: # Start the thread only if the socket was created successfully

gas\_thread = threading.Thread(target=listen\_for\_gas\_data, daemon=True)

gas\_thread.start()

logger.info("Gas data listening thread started.")

else:

logger.warning("Gas data listening thread could not be started because the gas data socket could not be established.")

main\_loop() # Start the main loop

if gas\_thread and gas\_thread.is\_alive():

logger.info("Waiting for gas listening thread to terminate...")

gas\_listener\_stop\_event.set() # Make sure to set the event

gas\_thread.join(timeout=2) # Wait for a maximum of 2 seconds

if gas\_thread.is\_alive():

logger.warning("Gas listening thread did not terminate in time.")

logger.info(f"===== Voice Assistant Server Stopped =====")

# 5. Results and Discussion

In this project, we built a smart system that combines gas detection with a voice assistant using the ESP32-S3. Both parts of the project were tested separately and worked well. After testing, they were combined into one complete system.

**Gas Detection System:**

The MQ-2 gas sensor was able to notice gas in the environment. When we brought a lighter (without flame) near the sensor, the values from the sensor increased. If the reading passed the set limit (around 1000), the system:

* Turned on a buzzer right away,
* Sent a warning message through Telegram,
* Showed an alert on the Blynk mobile dashboard.

When the gas level went back to normal, the buzzer turned off automatically. The warning limit was set by testing different values until the right one was found. The system reacted quickly and worked well, as long as the Wi-Fi connection was stable.

**Voice Assistant System:**

The ESP32-S3-based voice assistant was able to complete the full voice interaction process. It did the following steps successfully:

* **Hotword Detection**: The system started listening when the user said “hello”.
* **Speech Recognition**: It sent the recorded voice to Google’s Speech-to-Text service and got the correct text version.
* **LLM Response**: The text was sent to a large language model (via the Groq API), which created a reply.
* **Text-to-Speech**: The reply was turned into audio using gTTS and played back to the user through a speaker.
* **Audio Streaming**: The microphone and speaker used UDP to send and receive audio data with low delay.

The microphone worked well, although background noise sometimes made it harder to get clear audio. Overall, the system responded smoothly and gave voice replies quickly.

**Final Integration and Overall Result**

By combining both the gas sensor and voice assistant, we created a smart and affordable system that can both keep users safe and allow them to talk to it using voice. Both parts worked at the same time without problems. The project shows how simple hardware and cloud AI tools can work together to make a helpful, voice-controlled IoT device.

# 6. Conclusion

This project successfully brought together two useful features into one smart system: a gas warning system and a voice-controlled assistant. Using the ESP32-S3 board, we built a device that can both detect dangerous gases like smoke or LPG and also understand and reply to voice commands.

Some important results of the project include:

* The gas sensor quickly reacted to unsafe gas levels and gave alerts through sound, Telegram, and the Blynk app.
* The voice assistant could start listening with a hotword, understand what the user said, and reply with spoken answers.
* Audio was sent and received with little delay over Wi-Fi using UDP.
* The system is affordable and practical, showing how well simple hardware and cloud services can work together.

This prototype could be useful in homes for safety, smart control, or to help people with accessibility needs. In the future, we could add features like a mobile app, LCD screen for status display, saving data to the cloud, or turning on fans when gas is detected.

In short, the project shows how modern AI tools and IoT devices can be combined to create smart systems that are helpful, responsive, and easy to use.

# 7. Appendices

This section contains supplementary materials related to the project, including code files, circuit diagrams, and usage instructions.

**A. Arduino Code Files**

The main Arduino sketch contains the firmware for the ESP32-S3 board. It includes the following features:

* Reading gas values from the MQ-2 sensor
* Activating the buzzer when dangerous gas levels are detected
* Sending alerts through the Telegram Bot API
* Updating gas status on the Blynk IoT platform
* Using NeoPixel LEDs for visual status indication
* Handling audio input/output using I2S communication

**File:** [gas\_detector\_mq2.ino](https://drive.google.com/file/d/1XL3yHXZtlOwCKQOMCKDf27U47ERZ8N_c/view)

**B. Python Code Files**

This Python script runs on a computer/server and handles the voice assistant logic. Its main functions are:

* Capturing audio data sent from the ESP32 microphone
* Converting speech to text using Google Cloud Speech-to-Text
* Sending the transcribed query to the Groq API for AI processing
* Receiving the AI-generated response and converting it to speech using gTTS
* Playing the audio reply locally and optionally sending it back to the ESP32
* Handling communication errors, timing issues, and API responses

**File:** [proje.py](https://drive.google.com/file/d/1DyPm0XMJh_Q_Iu7bo9GbTWiqHeIbH7Bl/view)

**B. Circuit Diagram File**

A detailed circuit diagram is provided, showing how all hardware components are connected. These include:

* MQ-2 gas sensor
* ESP32-S3-DevKitM-1U microcontroller
* INMP441 I2S microphone
* PCM5102A I2S DAC
* PAM8403 audio amplifier
* 8Ω speaker
* Active buzzer
* Breadboard and jumper wires

This diagram helps in replicating the setup accurately for prototyping or development purposes.

**File:** [circuit.pdf](https://drive.google.com/file/d/1L6mFoqQuhrpl6nMi77s2ka5fu7zcBFix/view)

**C. README File**

The README.md file provides step-by-step guidance for setting up and using the project. It includes:

* **Required Libraries and Tools:**  
  Lists all necessary Arduino and Python libraries (e.g., WiFi, ESPAsyncUDP, I2S, gTTS, speech\_recognition, socket, requests).
* **Hardware Wiring Instructions:**  
  Diagrams and pin mappings for connecting the MQ-2 sensor, INMP441 microphone, PCM5102A DAC, PAM8403 amplifier, buzzer, and speaker to the ESP32-S3.
* **System Setup Steps:**  
  Instructions for flashing the Arduino sketch, installing Python dependencies, and running the voice assistant server on a PC.
* **Wi-Fi & Network Configuration:**  
  Details on entering your Wi-Fi credentials, setting static IP (if needed), and configuring UDP ports for audio communication.
* **Voice Assistant Usage:**  
  How to trigger the assistant with a hotword, speak commands, and hear back spoken responses.
* **Gas Detection Usage:**  
  Explanation of gas threshold adjustment, buzzer alert logic, and how to simulate gas presence for testing.

**File:** [README.md](https://drive.google.com/file/d/1T6QuCqnhYbBZd6YW6pfgLF4rg-LIcKNh/view?usp=drive_open)

# 8. References

[1] Espressif Systems, \*ESP32-S3-DevKitM-1 Development Board Documentation\*, Release Master, March 2025. [Online]. Available: <https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32s3/esp-dev-kits-en-master-esp32s3.pdf>

[2] Henan Hanwei Electronics Co., Ltd., \*MQ-2 Semiconductor Sensor for Combustible Gas – Technical Data\*, [[PDF Document].](https://www.pololu.com/file/0j309/mq2.pdf)