Cactus Personal Desk-Assistant

1st Gian Mario Marongiu
gianmario.marongiu@studio.unibo.it

2nd Mauro Dore
mauro.dore@studio.unibo.it

I. Introduction

The Cactus Personal Desk Assistant is an AI-driven virtual assistant integrating a physical IoT device and a Telegram bot. It provides a multi-platform experience, allowing interaction via predefined commands or natural language input, both vocally and in writing. The assistant supports reminders, timers, environmental data retrieval, and personalized settings, such as a custom LLM initialization prompt and a user-defined username. Due to hardware constraints, API calls handle advanced natural language processing (NLP) and machine learning tasks.

A button press initiates voice recording, sent via HTTP to the Python client for speech-to-text conversion and parsing. The LLM then classifies the request to determine if it requires an action, such as setting a reminder, or an informational response. The Telegram bot functions as a stateless chatbot, enabling users to manage reminders, request sensor data plots, and configure settings.

Users can create, view, and track reminders and timers via voice or text input. The system automatically schedules notifications and allows users to inquire about existing tasks through direct interaction or predefined Telegram commands. Deletion of reminders and timers is only possible through a specific Telegram command, which presents a selection interface for choosing the target entry.

Speech processing is handled via Deepgram-based speech recognition [2], while natural language understanding is managed by the Gemini 1.5 LLM [3]. The assistant dynamically includes user data in prompts based on context; for example, it excludes user data when setting a reminder but retrieves stored preferences when explicitly requested. This adaptive prompting ensures efficient interactions.

Through integration with an ESP32 device, the system collects real-time sensor data, such as temperature and humidity, storing it in an InfluxDB database. Users can request historical data plots via Telegram, selecting from predefined time frames (1 day, 1 week, 15 days, or 1 month). The assistant is always informed of current temperature and humidity, providing real-time updates upon request.

Furthermore, the assistant is equipped of a 3D-printed custom enclosure. The structure was created from an open-source cactus model. The interior was hollowed out in Blender to fit the hardware and sensors, with openings at the top to improve microphone reception. The structure was printed with a Creality Ender 3 and assembled to enhance both aesthetics and protection, giving the assistant its name: Cactus.

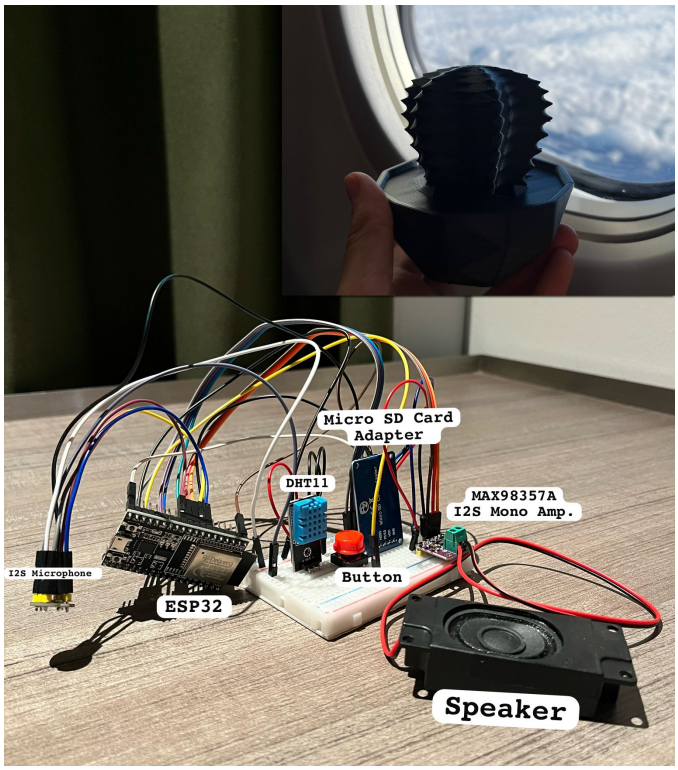

Fig. 1. Hardware Picture

II. Project's Architecture

In this section we describe the architecture of both software and hardware, highlighting the technical details of the system.

A. Software

From a high-level perspective, the software system is depicted in Figure 2. The Assistant Manager class (2 in Fig. 2) orchestrates the execution of the entire code. Specifically, the ESP32 acts as a server, generating data upon request (1 in Fig. 2). The Python client periodically requests sensor data (4 in Fig. 2) and distinguishes between user input (5a in Fig. 2) and temperature-humidity samples (5b in Fig. 2).

Audio and textual input require further processing to determine whether the user's request requires an action, whereas temperature-humidity samples are directly sent to the Cactus class (7 in Fig. 2), which uploads them to InfluxDB (9 in Fig. 2). The Telegram bot loop (3 in Fig. 2) continuously monitors the Telegram chat, waiting for user input, which follows the

same processing path as verbal commands received from the physical system.

If an action is required (e.g., setting a timer, deleting a reminder, setting a username), a dedicated function (6 in Fig. 2) executes the action, and its output is stored in a local JSON file (9 in Fig. 2). If no action is required, the user input is passed to the Cactus class, which generates a response by calling the Gemini-flash-1.5 API. Gemini is also informed about the system's capabilities and the user's stored information, enabling it to provide high-level guidance.

Finally, the system outputs a response to the user's initial request (8 in Fig. 2) in one of three forms:

- Confirmation (e.g., after setting a timer, a predefined message is displayed).
- LLM-generated response (e.g., answering an open-ended question).
- Image output (e.g., plotting sensor data upon request).

The software runs on four main asynchronous tasks:

1) **ESP32 Code** This task runs in an infinite loop, handling low-level operations on the ESP32.
2) **Telegram Bot Loop** This task waits for user input, either via predefined commands or general messages. The available commands are:
   - `init_prompt` – Set a new initialization prompt for the LLM.
   - `show_init` – Display the current assistant initialization prompt.
   - `username` – Set a new username.
   - `show_username` – Display the current username.
   - `show_reminders` – Show active reminders.
   - `show_timers` – Show active timers.
   - `delete_reminder` – Delete an active reminder.
   - `delete_timer` – Delete an active timer.
   - `plot_temperature` – Display a temperature plot from the sensor data. The user can choose a summary of the last 1, 7, 15, or 30 days.
   - `plot_humidity` – Display a humidity plot from the sensor data, with the same time range options as temperature.

   Alternatively, users can send free-form messages, leveraging the system's LLM capabilities to ask about any topic.
3) **Timer and Reminder Check** Every second, the system checks active reminders and timers. When a notification is due, it is sent via the Telegram bot (if available) and announced through the Cactus speaker.
4) **Microphone Data Check** At regular intervals, the Python client verifies if new audio recordings are available on the ESP32's SD card. If so, the audio is transcribed into text using the DeepGram API.
5) **Sensor Data Management** The Python client periodically retrieves sensor data from the physical system and uploads it to InfluxDB for remote storage and analysis.

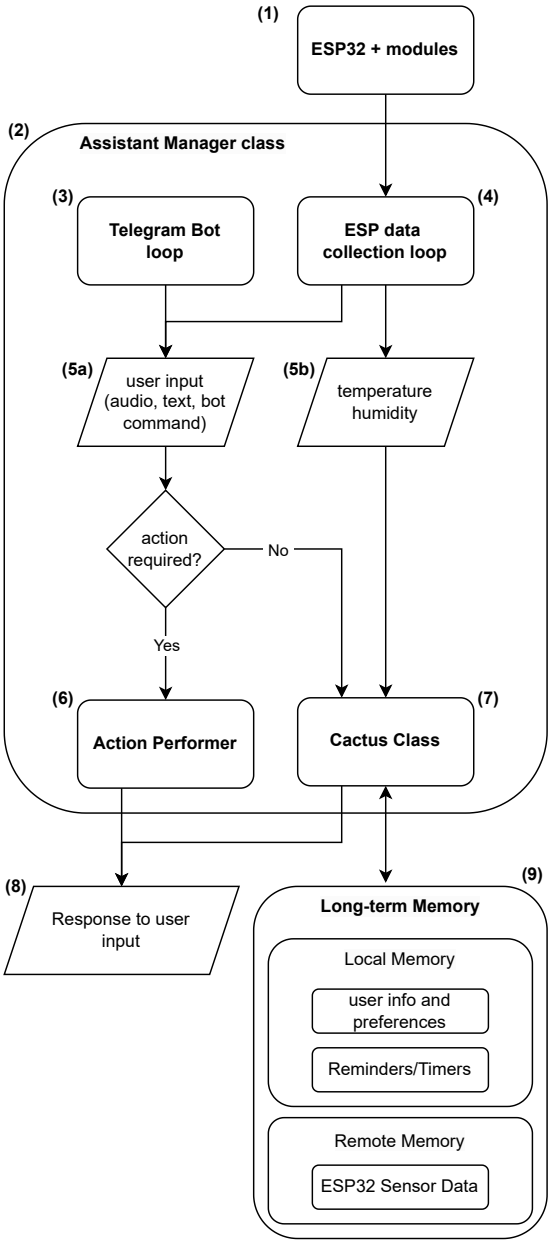

Fig. 2. Assistant software high-level scheme

B. Hardware

This project utilizes an ESP32 microcontroller to perform audio recording, read data from a DHT11 temperature and humidity sensor. The project also handles the storage of audio files on a microSD card and supports Text-to-Speech (TTS) by connecting to Google's TTS service. The system consists of the following main components:

1) **ESP32 WROOM 32** The main microcontroller of the system, based on the ESP32 chip. It provides Wi-Fi connectivity, controls I2S audio recording, and interfaces with the DHT11 sensor and SD card.
2) **MAX98357A (I2S Mono Amp.)** A mono audio amplifier that uses the I2S protocol for digital communication. This component is connected to:
   - A GND pin for ground
   - 3.3V power supply
   - LRC (Left/Right Clock), DIN, and BCLK (Bit Clock) pins for I2S communication
   - GAIN pin for gain control (dB)
   - Connected to a speaker for audio output
3) **I2S Microphone** A digital microphone that uses the I2S protocol for communication. Its connections include:
   - VDD for power supply
   - GND for ground
   - SD (Serial Data): Transmission of digitized audio data
   - SCK (Serial Clock): Clock for the I2S bus
   - WS (Word Select)
   - L/R (Left/Right Select): for channel selection (same as WS for I2S connections)
4) **DHT11 Sensor** A temperature and humidity sensor that requires:
   - GND pin
   - Output pin for data communication

   The sensor data is retrieved using the `dht.readHumidity()` and `dht.readTemperature()` functions of the `<DHT.h>` library.
5) **Micro SD Card Adapter** A module for reading/writing Micro SD cards, which uses the SPI interface with the following connections:
   - CS (Chip Select)
   - SCK (Clock): Clock for the SPI bus
   - MOSI (Master Out Slave In): Data line from ESP32 to SD card
   - MISO (Master In Slave Out): Data line from SD card to ESP32
   - 5V power supply (also compatible with 3.3V)
   - GND for ground

   The SD card stores recorded audio files.
6) **Button** A simple button connected with:
   - OUT pin for signal
   - GND for ground

   Pressing the button triggers audio recording through the I2S interface, which is saved as a WAV file on the SD card. The recording stops once the button is released and the system checks if the audio file is valid by verifying its duration and size. A short delay is added at the end of each iteration to prevent excessive polling.

The ESP32 runs an HTTP server, handling requests through defined endpoints:

- '/microphone': Serves recorded audio files in WAV format.
- '/sensor': Returns temperature and humidity data in JSON format.
- '/message_speak': Accepts a POST request with a message to be spoken using Google TTS.

In fact, the ESP32 main loop continuously handles incoming HTTP requests from clients via the web server. We used an open-source library based on `i2s_std.h` to facilitate audio recording [1], ensuring seamless integration with hardware using the latest drivers.

III. Project's Implementation

In this section, we describe the implementation choices we made and the motivations behind the system's design.

A. Models and APIs

All the APIs we selected were free (an important selection criterion), although they do have certain limitations. However, these restrictions do not compromise the quality of the assistant's performance in moderate usage.

In particular, we chose Google TTS for its versatility and compatibility with the ESP32. Additionally, it allows customization of parameters such as volume, preference for male or female voices, and pronunciation (which varies by language). While we are not setting these parameters at the moment, we plan to implement them in a future version of the project.

We selected DeepGram for speech-to-text conversion due to its versatility in understanding both English and Italian. Other options were tested for this purpose, including an attempt to integrate a local version of wav2vec2-base-960h [4] by Meta. However, it was unable to process the audio from the microphone due to background noise.

Finally, we chose the Gemini API [3] because of its large context window (1M tokens), which allows us to provide the model with extensive information about its functionalities in each call.

B. User Input Classification

The system employs a sophisticated approach to natural language understanding, leveraging the Gemini LLM for both input classification and temporal information extraction. Rather than relying on simple keyword matching, the system uses carefully crafted prompts to classify user inputs into appropriate categories (e.g., setting reminders, requesting system information, or general queries). The LLM returns specific keywords that enable the system to determine the

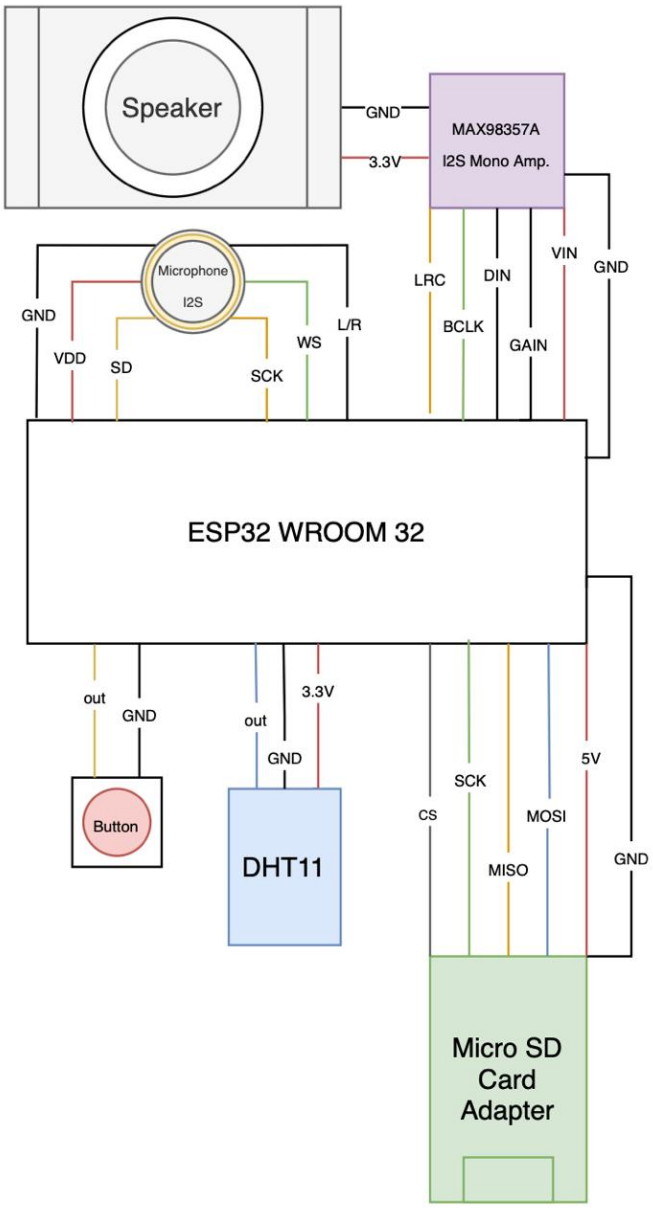

Fig. 3. Hardware scheme

appropriate action for each natural language request. This approach provides significantly more flexibility than traditional keyword-based systems while maintaining high accuracy in request classification. Temporal information processing follows a similar pattern but includes additional formal validation in Python. The LLM is instructed to output standardized date formats that address three distinct temporal scenarios:

1) Delay-based requests (e.g., "in 10 minutes")
2) Specific dates: For explicit calendar dates (e.g., "March 10th")
3) Relative dates: For context-dependent times (e.g., "next Thursday")

Here are examples of the standardized output format for each case:

- **Delay-based format:**
  Input: `"Remind me to call John in 3 hours"`
  Output: {
  `"content": "Call John",`
  `"time_type": "delay",`
  `"time_value": "0y0m0d3h0m0s"`
  }
- **Specific date format:**
  Input: `"Set a reminder for dentist on March 10th at 9am"`
  Output: {
  `"content": "Dentist appointment",`
  `"time_type": "time",`
  `"time_value": "2025-03-10 09:00"`
  }
- **Relative time format:**
  Input: `"Wake me up at 7 AM"`
  Output: {
  `"content": "Wake up",`
  `"time_type": "relative",`
  `"time_value": "RELATIVE:TIME:07:00"`
  }

C. Memory Management

The system implements a strategic approach to memory management, optimizing data storage based on access frequency and growth patterns. Data that requires frequent access is stored locally to minimize latency and network overhead, while data that grows over time is managed through remote storage. Local storage is implemented through a JSON file containing essential system configurations and temporary data:

- `user_reminders`: an array storing active reminders
- `timers`: an array storing active timers
- `user_initialization_prompt`: string containing the LLM initialization prompt
- `user_name`: string storing the user's name
- `chat_id`: integer storing the Telegram chat identifier

While the `user_reminders` and `timers` arrays can grow over time, they are automatically maintained through a cleanup mechanism that removes entries upon expiration. When a notification is triggered, the system automatically removes the corresponding reminder or timer from local storage, ensuring efficient memory utilization. Time-series data, specifically temperature and humidity readings, are stored remotely in InfluxDB. This approach is optimal for data that continuously accumulates over time. The database implements a 30-day retention policy, automatically removing data older than this threshold. This remote storage strategy is particularly effective as this historical data is accessed less frequently, typically only when users request specific visualization plots. The system retrieves this data through customized queries only when needed, reducing the overall system load while maintaining data accessibility.

D. HTTP Protocol

The system utilizes HTTP as its communication protocol, a choice that aligns well with the project's specific requirements and architecture. While IoT applications often employ protocols like MQTT or CoAP, HTTP proved to be a suitable solution for several reasons:

- First, HTTP's robust support for large payload transfers was essential for handling the voice command feature. The protocol's native capabilities in file transfer operations simplified the management and transmission of audio recordings between the ESP32 and the Python client.
- While MQTT would have been advantageous for temperature/humidity data transfer due to its publish-subscribe model and lightweight nature compared to HTTP, it would have introduced additional complexity without significant system-wide benefits, given that the primary purpose of the code is not efficient data exchange but rather interfacing with the LLM.
- The extensive library support for both ESP32 and Python environments streamlined the implementation process, ensuring reliable integration between all system components.

IV. Results

In this report, we presented Cactus, a personal desk assistant designed to process user requests through audio input and an LLM-based reasoning pipeline. The system successfully integrates hardware and software components to provide an interactive experience.

The assistant successfully records and processes audio, demonstrating stable performance under normal conditions. During testing, latency was minimal, with response time primarily dependent on API availability. A single instance of delayed response was observed due to an external issue with Gemini's API, but this did not reoccur in subsequent tests.

The model reliably interprets user requests in most cases. The system dynamically selects the most appropriate prompt template based on the request's structure and intent, ensuring accurate responses. However, performance may degrade if a user asks for multiple unrelated pieces of information within a single request. In such cases, the assistant may struggle to structure an optimal response.

Date parsing proved to be robust, even when the user's phrasing was ambiguous. If the requested date or time is unclear, the system automatically assigns an "undefined" value and prompts the user for clarification, preventing misinterpretations.

While the current implementation provides a functional and interactive assistant, several enhancements can further improve its capabilities:

- Enhanced error handling, particularly for sensors and actuators failures.
- Additional functionalities, including customizable language and voice settings, volume control, and potentially specialized task modes, such as a learning-oriented assistant or air quality monitoring and feedback.
- A short-term memory for the assistant, in such a way that it can be able to store and retrieve the messages exchanged with the user.

Overall, Cactus demonstrates the feasibility of a compact and interactive desk assistant while highlighting opportunities for further refinement and expansion.

References

[1] Audio external library. https://github.com/kaloprojects/KALO-ESP32-Voice-Assistant/blob/main/KALO_ESP32_Voice_Assistant/lib_audio_recording.ino.
[2] Deepgram website. https://deepgram.com/.
[3] Gemini 1.5 website. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#sundar-note.
[4] wav2vec. https://ai.meta.com/blog/wav2vec-20-learning-the-structure-of-speech-from-ra

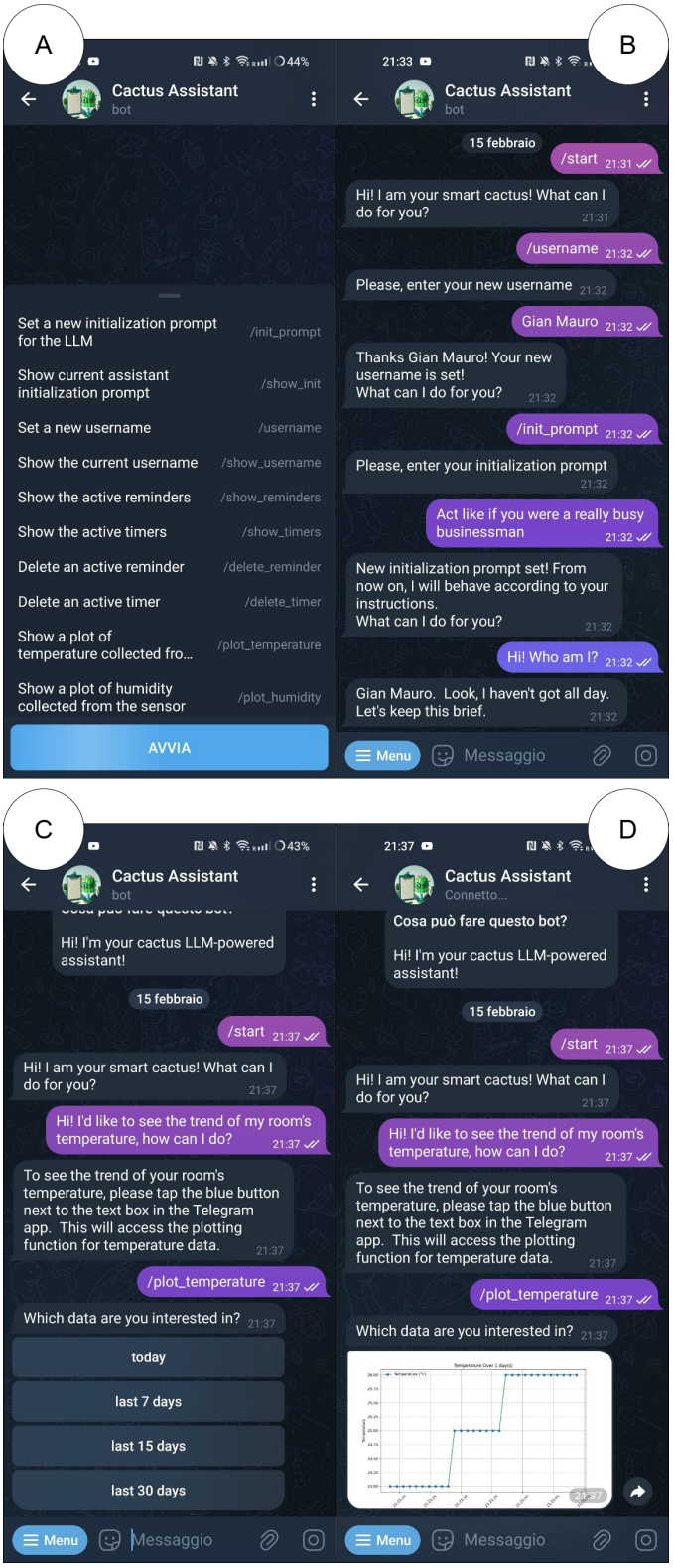

Fig. 4. Some screenshots from the assistant chat on Telegram where are shown the list of commands that is possible to call on the chat (A), a basic initial interaction where the user sets an username and an initialization prompt (B), and an interaction where the assistant shows the list of possibilities from which a user can choose whether to plot last day data or 1 week, 15 days or 1 month (C), after the selection, the options are removed and the plot dynamically computed is shown (D). In (D) the data is artificially uploaded on InfluxDB for demonstration purpose.