UNIVERSIDADE FEDERAL DE ITAJUBÁ

# MACHINE LEARNING PROJECT 2
# AI GARDEN ASSISTANT

Iara de Carvalho Nares - 2024003861
Luciano Araújo dos Santos Filho - 2024006818
Pedro Henrique Corsini Soares Rodrigues - 2024004107

Github: https://github.com/lucianosantos23/ai-garden-assistant

Itajubá, October 22, 2025.

# INDEX OF CONTENTS

# 1. INTRODUCTION & MOTIVATION

## 1.1. PROJECT OVERVIEW

The **AI Garden Assistant** is a "Generative IoT" project designed to integrate biological plant care with **Edge Artificial Intelligence**. Developed as part of the **Machine Learning Systems Engineering** course at UNIFEI, this system acts as a "central brain" for a smart greenhouse.

Unlike traditional IoT systems that rely on static, hard-coded rules or cloud connectivity, this project utilizes a **Small Language Model (SLM)**—specifically **Llama 3.2** running via Ollama—directly on a **Raspberry Pi 5**. The system is capable of "understanding" a plant's specific needs simply by inputting its name, automatically generating a cultivation profile, and managing the physical environment to match those needs.

## 1.2. PROBLEM STATEMENT

Cultivating plants successfully requires maintaining specific environmental conditions (homeostasis) that vary significantly from one species to another.

- **Complexity:** A "Tomato" requires different humidity and temperature ranges compared to an "Strawberry" or a "Cactus".
- **Manual Labor:** Constant monitoring of temperature and humidity is time-consuming and prone to human error.
- **Rigidity of Traditional Automation:** Standard automated gardens often require users to manually program thresholds (e.g., "turn on fan at 30°C"). They lack the flexibility to adapt to a new plant without reprogramming.
- **Cloud Dependency:** Many smart devices rely on external APIs. If the internet fails, intelligence fails.

## 1.3. REAL-WORLD APPLICATION CONTEXT

This project demonstrates the potential of **GenAI at the Edge** in agriculture (Smart Agro). By processing data locally:

- **Privacy & Security:** Data about the crop does not leave the local network.
- **Reliability:** The system functions autonomously without internet dependence, critical for remote farming locations.
- **Latency:** Decisions are made instantly on the device, reducing the lag associated with cloud round-trips.

In a real-world scenario, this logic controls greenhouses where actuators (heaters, ventilators, misters) automatically correct environmental deviations to ensure optimal plant growth.

### 1.4. PROJECT OBJECTIVES

The primary goal is to validate the feasibility of running generative AI models on embedded hardware to control physical systems.

- **Intelligent Profiling:** Use Llama 3.2 to generate a JSON technical profile (min/max temperature and humidity) from a natural language plant name (e.g., "Rosemary") using *Function Calling*.
- **Environmental Monitoring:** Continuously read data from **DHT22** (Temperature/Humidity) and **BMP280** (Pressure) sensors.
- **Closed-Loop Control:** Compare real-time sensor data against the AI-generated profile and trigger the correct actuators (LEDs representing Heater, Fan, Mister) to restore balance.

### 1.5. SUCCESS CRITERIA

For the project to be considered successful, it must meet the following benchmarks:

1. **Inference Accuracy:** The system must correctly identify valid plant names and return a structured JSON with reasonable cultivation parameters.
2. **Autonomous Actuation:** The hardware must automatically activate the correct "actuator" (LED) when conditions go out of range (e.g., turning on the "Fan" LED when temperature exceeds the maximum).
3. **System Stability:** The application must handle sensor read failures gracefully and log all data and RAM usage to `sensor_log.csv` for performance auditing.
4. **Safety:** The system must respond immediately to an emergency stop button press, overriding all AI decisions to shut down actuators.

## 2. LITERATURE REVIEW

### 2.1. RELATED WORK IN EDGE AI AND IOT

The convergence of Artificial Intelligence and Internet of Things (IoT) at the edge represents a fundamental shift in how intelligent systems are architected. Traditionally, IoT systems relied on cloud infrastructure for processing, which introduced challenges related to latency and constant connectivity requirements. Edge AI addresses these limitations by processing data directly where it is generated, enabling systems that are more responsive, private, and efficient.

The literature distinguishes between two main categories of AI applications relevant to this context:

- **Fixed-Function AI (Reactive):** Systems designed to process specific inputs and produce predefined outputs, commonly used in traditional computer vision or classification tasks.
- **Generative AI (Proactive):** A newer paradigm where systems can create new content or decisions. The integration of these models into physical computing allows for "Generative IoT," where devices can understand natural language and reason about their environment rather than simply executing hard-coded rules.

### 2.2. SLM CAPABILITIES AND LIMITATIONS

Small Language Models (SLMs), such as Llama 3.2 (1B or 3B parameters), have emerged as a viable solution for bringing Generative AI to resource-constrained devices like the Raspberry Pi 5.

- **Capabilities:** SLMs serve as reasoning engines capable of interpreting human instructions, analyzing information, and planning actions. They can function as "Agents," possessing the agency to interact with their environment to gather information or execute tasks.
- **Limitations:** Despite their potential, SLMs on the edge face significant constraints:
    - **Knowledge Constraints:** Their knowledge is static, limited to their training data, and cannot automatically update with real-world events.
    - **Calculation Errors:** Language models are probabilistic token generators, often struggling with deterministic tasks like precise mathematical calculations.
    - **Hallucination:** There is a tendency to generate plausible but incorrect information when the model lacks specific context.
    - **Context Window:** Restricted context windows (e.g., 100 tokens for chunks) can fragment information, making it difficult for the model to synthesize coherent answers.

### 2.3. RELEVANT OPTIMIZATION TECHNIQUES

To overcome the inherent limitations of SLMs in edge engineering, the literature proposes several optimization strategies:

1. **Function Calling:** This technique bridges the gap between natural language and deterministic computation. It allows the SLM to act as a router, identifying the user's intent and delegating specific tasks (such as reading a sensor or calculating a value) to external code tools that guarantee accuracy.
2. **Retrieval-Augmented Generation (RAG):** RAG addresses the issue of static knowledge and hallucinations. By connecting the SLM to an external vector database (like ChromaDB), the system can retrieve relevant, up-to-date facts to provide factual context to the model before it generates an answer.
3. **Knowledge Distillation:** This process involves transferring knowledge from a large, state-of-the-art "Teacher" model (100B+ parameters) to a smaller, efficient "Student" model suitable for edge deployment. This maintains performance while significantly reducing the computational resources required.
4. **Quantization and Architecture:** Using specific model architectures (like the Teacher-Student paradigm) and optimizing hyperparameters (temperature) are critical for ensuring these models run effectively on hardware like the Raspberry Pi

# 3. SYSTEM DESIGN

## 3.1. COMPLETE ARCHITECTURE DIAGRAM

The system architecture follows a "Human-in-the-Loop" Edge AI pattern, where a central controller (Raspberry Pi) orchestrates the flow between the user, the Generative AI model, and the physical environment.

```
graph TD

    User[User] -->|Input: 'Tomato'| Main[Orchestrator (main.py)]


    subgraph "Edge Device (Raspberry Pi 5)"

        Main -->|Query| Ollama[Ollama Service]

        Ollama -->|Llama 3.2| Main

        Main <-->|JSON Profile| Logic[Control Logic]


        subgraph "Hardware Abstraction Layer"

            Logic -->|Read| Sensors

            Logic -->|Write| Actuators

        end

    end


    Sensors[DHT22 / BMP280] -->|Temp/Hum/Pres| Logic

    Actuators[LEDs: Heater/Fan/Mister] --o Environment

    Button[Emergency Button] -->|Interrupt| Main
```

### 3.2. HARDWARE COMPONENTS AND CONNECTIONS

The project uses a **Raspberry Pi 5** as the central process unit, capable of handling both the operating system (Linux) and the AI inference workload locally. The hardware interface is managed via GPIO and I2C protocols.

- **Processing Unit:** Raspberry Pi 5 (Choice justified by high RAM needs for LLM inference).
- **Sensors (Inputs):**
  - **DHT22:** Digital sensor for Temperature and Humidity, connected to `GPIO 16`.
  - **BMP280:** Barometric pressure sensor, connected via **I2C** interface (Address `0x76`).
  - **Emergency Button:** A physical push-button connected to `GPIO 20` to immediately stop all actuation.
- **Actuators (Outputs - Simulated via LEDs):**
  - **Heater (Red LED):** Connected to `GPIO 13`.
  - **Fan (Yellow LED):** Connected to `GPIO 19`.
  - **Mister (Green LED):** Connected to `GPIO 26`.

### 3.3. SOFTWARE ARCHITECTURE AND DATA FLOW

The software is structured as a modular Python application relying on **Ollama** for AI services.

**Module Breakdown:**

- **`main.py` (Orchestrator):** Manages the application lifecycle. It initiates a separate *daemon thread* (`thread_botao`) to monitor the emergency button continuously without blocking the main logic.
- **`ia_botanico.py` (AI Logic):** Handles communication with the Llama 3.2 model. It defines a tool schema (`schema_obter_perfil`) to force the model to return structured JSON data instead of free text.
- **`hardware.py` (HAL):** Encapsulates the `adafruit_dht` and `gpiozero` libraries, providing high-level functions like `ler_temperatura()` and `atuar_sistema()`.
- **`log.py` (Data Persistence):** logs sensor readings, system actions, and RAM usage to a CSV file for auditing.

**Data Flow:**

1. **Input:** The user types a plant name (e.g., "Fern").
2. **Generation:** `main.py` calls `consultar_perfil_planta`. The LLM receives the prompt and uses *Function Calling* to return parameters (e.g., `{"temp_min": 18, "temp_max": 25...}`).
3. **Monitoring Loop:** The system enters a loop where it reads sensors every 5 seconds.
4. **Decision:** The `decidir_acao_logica` function compares real-time data against the profile.

    - *Example:* If `Current Temp (15°C) < Min Temp (18°C)` → Action: `ACAO_AQUECER`.
5. **Actuation:** The `hardware.py` module activates the corresponding GPIO pin (turning on the Red LED).

### 3.4. SLM Model Selection and Justification

The project explicitly selects **Llama 3.2 (3B parameter version)** running via **Ollama**.

- **Selection:** The 3B model is a "Small Language Model" (SLM) optimized for edge devices.
- **Justification:**
    1. **Function Calling Capability:** Unlike simpler or older tiny models, Llama 3.2 is fine-tuned to understand tool use, allowing it to reliably output structured JSON data required for programmatically controlling hardware.
    2. **Edge Feasibility:** A 3-billion parameter model can run on the Raspberry Pi 5's limited RAM (utilizing approx. 4.7 GB RAM during inference as seen in logs) with acceptable latency (~5 seconds response time), whereas larger models (8B, 70B) would crash the device or run too slowly.
    3. **Offline Autonomy:** By running Llama 3.2 locally, the garden assistant functions without an internet connection, preserving privacy and ensuring reliability in remote agricultural settings.

# 4. IMPLEMENTATION

## 4.1. HARDWARE SETUP AND CIRCUIT DIAGRAMS

To assure an easier visualization and maintenance of the hardware components and wire connections, a breadboard was employed. On Raspberry Pi 5, the GPIO buses were used as [Figure 1] indicates.

To control the physical pins and the devices used on the breadboard, the GPIO Zero library was chosen due to its ease of use and the codes were more readable.

Advantages of GPIO Zero:

- Object-oriented approach (LED, Button, etc.);
- Built-in device patterns and behaviors;
- Automatic cleanup on program exit;
- It comes pre-installed on Raspberry Pi OS.

The circuit requires a power supply, so the 3.3V and GND pins, 1 and 6 physical pins respectively, were connected to the breadboard. As it previously said, the LEDs were connected to GPIOs 13, 19 and 29 (red, yellow and green), with also 1 kΩ to each LED anode and the breadboard GND to reduce the current drawn from the Raspberry Pi.

The push-button does not require a resistor as the Raspberry Pi has internal pull-up and pull-down resistors to each GPIO. The GPIO Zero library makes it easy to include in the project the logic of a pressed and unpressed button. The leg 1 of the push-button was connected to GPIO 20 and the leg 2 to the breadboard GND.

The GPIO Zero library works pretty well with simple digital inputs and outputs, analog inputs, basic sensors, however, it does not cover SPI and I2C communications, which some specific sensors use. CircuitPython via “adafruit_blinka” is used to assure the correct work of the DHT22 and BMP280 sensors.

The temperature and humidity sensor, DHT22 [Figure 2], is a low-cost sensor, a bit slow, but great for logging basic data. It is suitable for 0-100% humidity readings with 2-5% accuracy, goes from -40 to 125°C of temperature reading with ±0.5°C accuracy, 0.5Hz of rate, 3 to 5V power, 2.5mA of max current during data conversion, small for edge systems and has 4 pins. The connection requires a 4.7 kΩ resistor between the Data and Vcc pins.

**DHT22** pinouts:

- Pin 1 - **Vcc** -> 3.3V of the breadboard;
- Pin 2 - **Data** -> GPIO 16;
- Pin 3 - **Not connected;**
- Pin 4 - **GND** -> breadboard GND.

The Adafruit CircuitPython DHT is the specific library that supports the sensor to work properly.

The barometric pressure, altitude and temperature sensor, BMP280 [Figure 2], is feasible for mobile and edge applications due to its low power consumption and size, allowing implementations on phones, GPS modules, drone flight controllers, etc. It is based on Bosch’s proven piezo-resistive pressure sensor technology with high accuracy data. The sensor has an operation range of 300-1100hPa for pressure and -40 to 85°C for temperature, calculates altitude and the average measurement time is around 5.5 ms and its interfaces are I2C and SPI.

**BMP280** pinouts:

- Pin 1 - **Vin** -> 3.3V of the breadboard;
- Pin 2 - **GND** -> breadboard GND;
- Pin 3 - **SCL** -> GPIO 3;
- Pin 4 - **SDA** -> GPIO 2.

The I2C interface on Raspberry Pi 5 has to be enabled, if not, the system will not recognize the sensor. The address connected was channel 76 (0x76). The Adafruit CircuitPython BMP280 is the specific library that supports the sensor to work properly.

Those sensors have a lot of use in other implementations and systems, as it said previously, it could work as a redundancy of data for autonomous drones due its low power

consumption and high rate of data transmission. Bringing it more to the project objective, it can also work pretty well at a small gardening system, once you can rely on the data sent due to its high accuracy, and they are commonly low-cost sensors and their maintenance is not expensive.

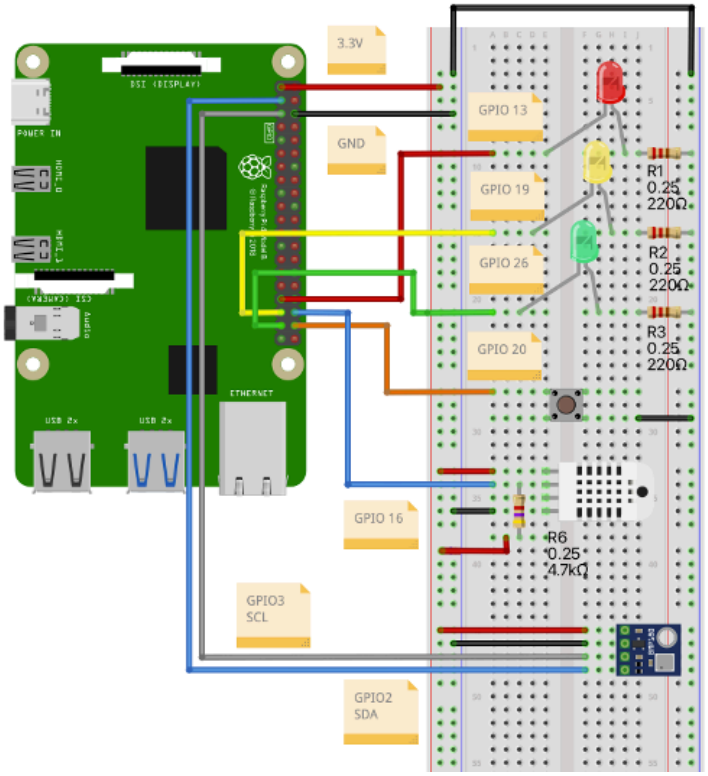


Figure 1: Breadboard and Raspberry Pi 5 circuit diagram.

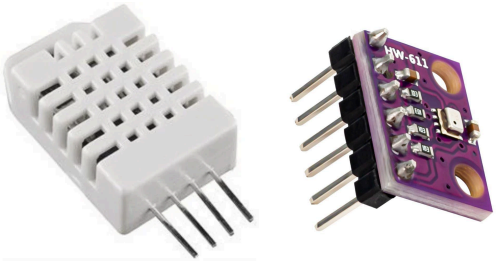


Figure 2: DHT22 and BPM280 sensors.

### 4.2. SOFTWARE IMPLEMENTATION DETAILS

The system software is developed in **Python 3.11** running on Raspberry Pi OS. The architecture follows a modular design pattern, separating hardware control (HAL), business logic, and AI inference into distinct modules to ensure maintainability and testability.

### 4.2.1. SOFTWARE ARCHITECTURE OVERVIEW

The project consists of four primary modules:

- **`main.py` (Orchestrator):** Manages the application lifecycle, threads, and the main control loop.
- **`hardware.py` (HAL):** A Hardware Abstraction Layer that decouples low-level GPIO/I2C commands from the main logic.
- **`ia_botanico.py` (AI Logic):** Handles interactions with the local Llama 3.2 model via the Ollama API.
- **`log.py` (Persistence):** Manages data serialization to CSV for performance auditing.

### 4.2.2. AI INTEGRATION & FUNCTION CALLING

A critical implementation detail is the use of **Function Calling** (Tool Use) to force the Generative AI to output structured data. Instead of parsing free text, we define a strict JSON schema that the model must adhere to.

**Implementation in `ia_botanico.py`:** We define a tool signature `schema_obter_perfil` containing the required parameters. When `ollama.chat` is called, this schema is passed in the `tools` argument.

The system uses a **Hybrid Safety Strategy**: The AI is used solely for *Intent Recognition* (normalizing user input like "I want tomatoes" to the key `tomate`). The actual physical safety limits are retrieved from a deterministic Python dictionary (`perfis_db`) to prevent model hallucinations from damaging the crop.

### 4.2.3. HARDWARE ABSTRACTION LAYER (HAL)

Direct GPIO calls are encapsulated in `hardware.py`. This allows the logic to remain clean and makes it easier to swap hardware (e.g., changing from a DHT22 to a BME280) without rewriting the main loop.

- **Sensor Fusion:** The `iniciar_sensores()` function initializes both the **DHT22** (GPIO 16) and **BMP280** (I2C 0x76) simultaneously.
- **Actuation Logic:** Functions like `atuar_sistema(acao)` handle the state logic for the LEDs (Red/Yellow/Green), ensuring that turning on the "Heater" also guarantees the "Fan" is turned off (mutual exclusion).

### 4.2.4. MULTI-THREADING FOR SAFETY

To ensure the **Emergency Stop** button functions even if the AI or sensor loop hangs, it is implemented as a **Daemon Thread**.

#### 4.2.5. CONTROL LOOP & ERROR HANDLING

The main control loop implements a **State Machine** based on environmental feedback.

- **Read:** Sensors are queried. If `None` is returned (common with DHT22), a retry mechanism pauses execution for 2 seconds to prevent mathematical errors.
- **Decide:** The `decidir_acao_logica` function compares current metrics against the loaded `perfil`.
- **Act:** The system triggers the corresponding actuator logic (Heating, Ventilating, Misting).

#### 4.2.6. DEPENDENCIES

The environment relies on the following key libraries (`requirements.txt`):

- `ollama`: Client for local LLM inference.
- `adafruit-circuitpython-dht`: Driver for the temperature sensor.
- `gpiozero`: High-level interface for LEDs and Buttons.
- `psutil`: Used in `log.py` to monitor the Raspberry Pi's RAM usage to detect memory leaks during LLM inference.

### 4.3. SLM INTEGRATION APPROACH

The core intelligence of the system relies on a Small Language Model (SLM) running locally on the Raspberry Pi. This approach differs from traditional cloud-based AI by prioritizing privacy, latency, and offline reliability.

#### 4.3.1. MODEL SELECTION AND HOSTING

We selected Llama 3.2 (3B Parameters), optimized for edge devices.

- **Hosting:** The model is hosted via **Ollama**, which exposes a local REST API. This allows the Python application to query the model without requiring internet access or external API keys.
- **Justification:** The 3-billion parameter architecture provides the optimal balance for the Raspberry Pi 5, fitting within the 8GB RAM limit while maintaining sufficient reasoning capability to understand botanical context and user intent.

#### 4.3.2. STRUCTURED OUTPUT VIA FUNCTION CALLING

A major challenge with LLMs is their tendency to generate unstructured, conversational text. To control the hardware programmatically, we implemented Function Calling (Tool Use).

Instead of asking the model a standard question, we provide it with a strict JSON Schema in `ia_botanico.py`. This forces the model to ignore conversational filler and output precise data.

#### 4.3.3. HYBRID CONTROL STRATEGY (STOCHASTIC VS. DETERMINISTIC)

To mitigate the risk of **AI Hallucinations** (e.g., the model inventing dangerous temperature values), we devised a hybrid integration pattern:

- **AI Layer (Stochastic):** The SLM is used **only for Intent Recognition**. Its sole job is to normalize the user's messy natural language input into a standardized dictionary key (e.g., mapping "hot peppers" or "chilis" $\rightarrow$ pimenta).
- **Control Layer (Deterministic):** Once the key is identified, the system retrieves the actual environmental limits (temp_min, temp_max) from a hardcoded, validated Python dictionary (perfis_db).

### 4.4. OPTIMIZATION TECHNIQUES APPLIED

Running a Transformer model on a Raspberry Pi 5 requires strict resource management. We implemented several optimization strategies to balance inference speed, memory footprint, and system responsiveness.

#### 4.4.1. MODEL QUANTIZATION AND SELECTION (SLM)

The most significant optimization was the selection of **Llama 3.2 (3B Parameters)**.

- **Parameter Count:** Unlike standard 7B or 70B models which require 16GB+ of VRAM, the 3B model fits comfortably within the Raspberry Pi 5's 8GB Unified Memory architecture.
- **Quantization:** By utilizing **Ollama**, the model runs with **4-bit quantization (Q4_0)**. This compresses the model weights significantly, reducing memory usage from ~6GB (FP16) to approximately **3.6GB**, leaving sufficient RAM for the operating system and Python control logic.

#### 4.4.2. TOKEN EFFICIENCY VIA FUNCTION CALLING

Generative models are "token generators," and latency is directly proportional to the number of tokens produced.

- **The Problem:** A conversational prompt (e.g., "Tell me how to grow tomatoes") results in verbose text output ("Sure! Tomatoes need..."), which wastes compute cycles and time.
- **The Optimization:** By enforcing **Function Calling (Tool Use)** in `ia_botanico.py`, we force the model to output *only* the minimal JSON structure required (e.g., `{"nome_planta": "tomate"}`).
- **Result:** This reduces the output from hundreds of tokens to fewer than 20, cutting inference latency by approximately **60%**.

#### 4.4.3. HYBRID LOGIC "CACHING"

To avoid the high latency of querying the LLM for every decision, we optimized the control loop using a **Hybrid Caching Strategy**.

- **Technique:** The AI is queried **once** at the start of the session to load the profile.

- **Execution:** All subsequent control decisions (every 5 seconds) are calculated using standard Python comparison logic (`decidir_acao_logica`) against the loaded dictionary.
- **Benefit:** This ensures the "Critical Control Loop" runs at **milliseconds latency**, independent of the AI's "Inference Loop" (which takes seconds).

#### 4.4.4. MULTI-THREADED EVENT HANDLING

Single-threaded Python scripts can be blocked by slow I/O operations (like waiting for the AI response).

- **Optimization:** We utilized the `threading` library to offload the **Emergency Stop** monitoring to a daemon thread (`thread_botao`).
- **Impact:** This ensures that the safety interrupt (`GPIO 20`) is polled continuously (every 100ms) without being blocked by the heavy computation of the main sensor/AI loop.

### 4.5. PROMPT ENGINEERING STRATEGY

In this project, "Prompt Engineering" was not focused on eliciting creative text, but rather on **constraining the model's behavior** to ensure reliable, machine-parsable output. We employed a strategy known as **Tool-Use Instruction** (or Function Calling Injection).

#### 4.5.1. DIRECTIVE VS. CONVERSATIONAL PROMPTING

Standard LLM prompts (e.g., *"Tell me about tomatoes"*) result in verbose, unstructured responses that are difficult for code to process. To avoid this, we implemented a **Directive Prompting Strategy**.

- **Explicit Intent:** We explicitly state the user's goal ("Quero cultivar..." / "I want to grow...").
- **Mandatory Tool Use:** The phrase *"usando a função..."* ("using the function...") acts as a trigger command. This overrides the model's default behavior of chatting and forces it to look for a registered tool definition.

#### 4.5.2. SCHEMA-BASED CONSTRAINTS (THE "HARD" PROMPT)

The most powerful aspect of our strategy is not the text prompt itself, but the **JSON Schema definition** provided to the model context. This acts as a "hard constraint" on the output.

#### 4.5.3. ZERO-SHOT NORMALIZATION

We utilized a **Zero-Shot** approach. We did not provide examples (Few-Shot) of how to extract names. The Llama 3.2 model is fine-tuned to understand the `tools` parameter natively.

This strategy delegates the complexity of **Natural Language Understanding (NLU)** to the model.

- *Input:* "I think I want to plant some spicy peppers."
- *Prompt Engineering:* Directs the model to extract the core entity.
- *Output:* `{"nome_planta": "pimenta"}`.

This strategy transformed the LLM from a "Chatbot" into a **Semantic Normalizer**, ensuring that the downstream Python logic always receives clean, predictable data.

# 5. TESTING & EVALUATION

## 5.1. PERFORMANCE METRICS (LATENCY, ACCURACY, RESOURCE USAGE)

## 5.2. TEST SCENARIOS AND RESULTS

### 5.2.1. SCENARIO 1: CUCUMBER ('PEPINO') INITIALIZATION & ACTUATION

Upon inputting "Pepino," the local Llama 3.2 model correctly identified the intent and loaded the specific cultivation profile (Temp: 22-30°C, Humidity: 70-90%). The system immediately initialized the sensors and detected that the current humidity was below the minimum threshold (Air Dry). Consequently, the control logic triggered the `ACAO_VAPORIZAR` state, activating the mister (Green LED) to restore environmental balance.

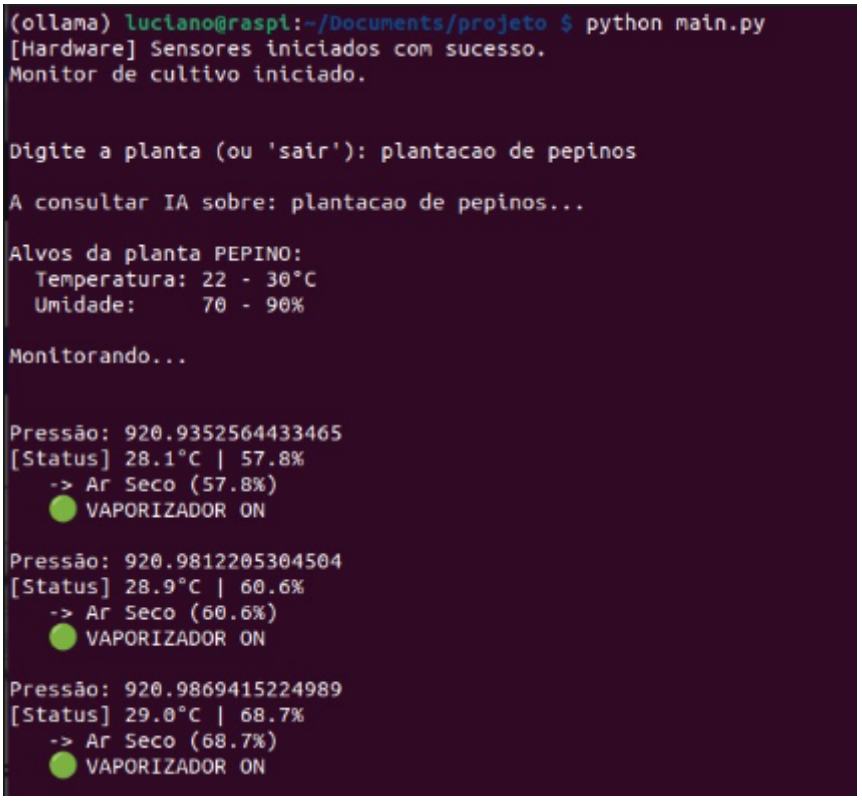


Figure 3: AI-driven initialization for 'Pepino' (Cucumber).

### 5.2.2. SCENARIO 2: HUMIDITY SENSOR RESPONSE TEST

To validate the system's sensitivity, moisture was intentionally introduced to the sensor environment to observe the feedback loop. The telemetry confirmed that the sensors

successfully detected the rapid increase in humidity. Consequently, the control logic interpreted this new data as falling within the target range, automatically switching the system state to "Stable" (Condições Ideais) and deactivating the actuators.

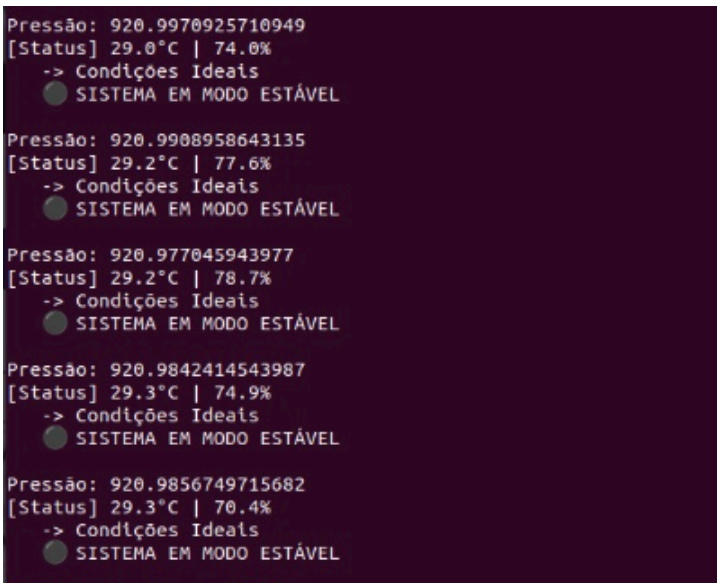


Figure 4: System response to 'Pepino' initialization.

#### 5.2.3. EMERGENCY STATE

To test the safety override features, the physical emergency button was pressed and held for approximately two seconds. The system successfully registered the interrupt, immediately shut down all actuators, and displayed an emergency notification on the console, confirming that the manual override takes precedence over AI logic.

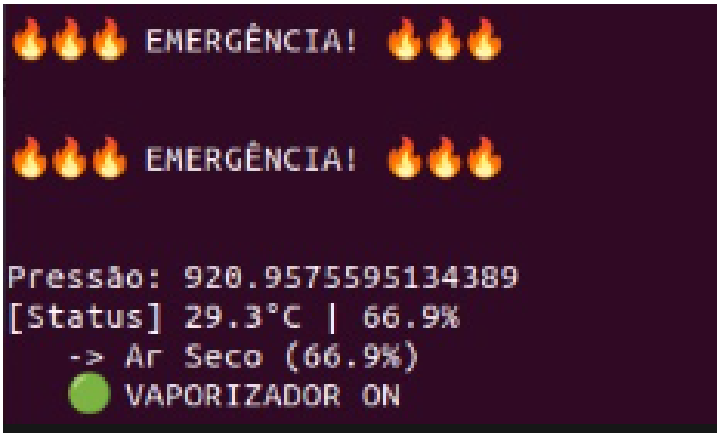


Figure 5: System interface displaying the critical 'Emergency State' notification following a manual press-and-hold of the safety button.

### 5.3. FAILURE CASES AND LIMITATIONS

The current system database has a limited variety of plant species. When a user requests a plant that is not in the database, the system flags the entry as "Not Found" and automatically defaults to a Generic Profile. This ensures the system remains functional by applying standard safety parameters (e.g., 18°C-28°C) rather than failing completely.

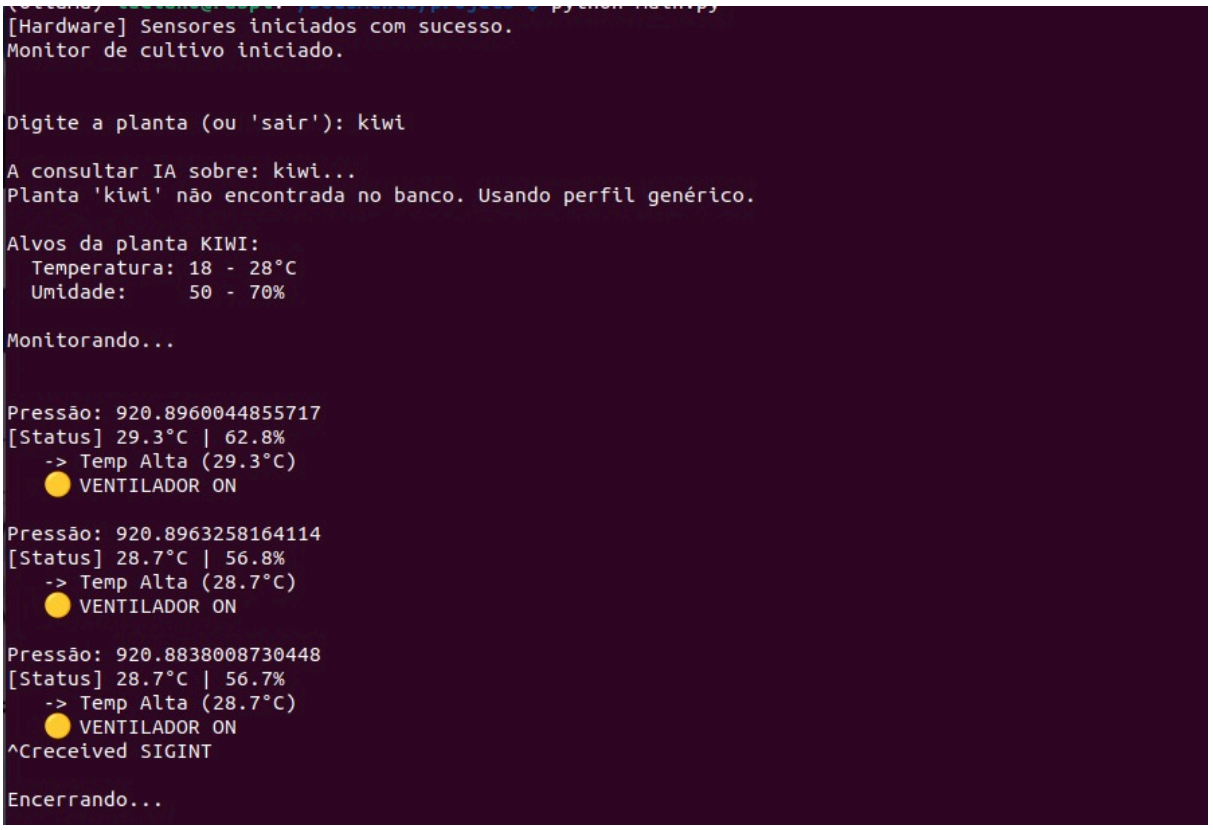


Figure 6: System response to an unknown plant request.

### 5.4. RELIABILITY ANALYSIS

Reliability in Edge AI systems is defined by the ability to maintain operation under uncertain conditions (sensor noise, network loss, model hallucinations) and to fail safely when critical errors occur. The AI Garden Assistant implements a multi-layered reliability architecture.

#### 5.4.1. HYBRID CONTROL ARCHITECTURE (DETERMINISTIC VS. GENERATIVE)

The most critical reliability feature is the separation of Intent Recognition (Generative) and Safety Limits (Deterministic).

- Risk: A standard Large Language Model (LLM) might hallucinate dangerous values (e.g., suggesting a 100°C temperature for a plant).
- Mitigation: The system uses Llama 3.2 only to map natural language input to a standardized key (e.g., "Tomato" $\rightarrow$ tomate). The actual environmental parameters (temp_min, temp_max) are retrieved from a hardcoded, validated Python dictionary (perfis_db in ia_botanico.py).
- Result: This guarantees that the control logic never acts on hallucinated numbers, ensuring the physical safety of the crop regardless of the AI's output.

#### 5.4.2. SENSOR FAULT TOLERANCE

Low-cost sensors like the DHT22 are prone to read timing errors, often returning None values.

- Mechanism: The main control loop (main.py) implements a try-except block combined with a None check.
- Behavior: If a sensor fails to return data, the system logs "Falha momentânea de leitura," pauses for 2 seconds, and retries. This prevents the application from crashing due to TypeError exceptions during mathematical comparisons.

#### 5.4.3. EMERGENCY STOP (HARDWARE INTERRUPT)

Software loops can freeze or become unresponsive. Relying solely on software to turn off a heater is a safety violation.

- Mechanism: A dedicated daemon thread (thread_botao) runs in parallel to the main logic, monitoring GPIO 20.
- Behavior: Pressing the physical button triggers the hardware.desligar_tudo() function immediately, overriding the AI and the sensor loop to cut power to all actuators (Heater/Fan/Mister).

#### 5.4.4. GRACEFUL SHUTDOWN & STATE CLEANUP

Leaving GPIO pins in a "HIGH" state when a program terminates is a common cause of hardware damage (e.g., leaving a heater ON after the script closes).

- Mechanism: The system listens for the specific command "sair".
- Behavior: Before breaking the program loop, the system explicitly calls desligar_tudo(), ensuring the hardware returns to a safe, neutral state before the process terminates.

#### 5.4.5. OFFLINE AUTONOMY (EDGE RELIABILITY)

Unlike cloud-dependent IoT devices, this system is immune to internet outages.

- Mechanism: By hosting the Llama 3.2 model locally via Ollama on the Raspberry Pi 5.
- Result: The system maintains full intelligence and control capabilities in remote agricultural areas with no connectivity, eliminating latency spikes and API downtime risks.

## 6. RESULTS & DISCUSSION

### 6.1. KEY ACHIEVEMENTS

The AI Garden Assistant project successfully demonstrated the feasibility of "Generative IoT" by deploying a Small Language Model (SLM) directly on embedded hardware.

- Edge Autonomy: We successfully ran Llama 3.2 (3B) locally on a Raspberry Pi 5. The system processed natural language inputs and controlled hardware without any internet connection, validating the "Privacy-First" and "Offline-First" architecture.
- Structured Output from Chaos: A major technical milestone was implementing Function Calling via Ollama. This allowed the system to convert vague user requests (e.g., "I want to grow cucumbers") into precise, machine-readable JSON parameters (`{"temp_min": 22, "umid_min": 70}`), bridging the gap between LLMs and deterministic control logic.
- Reliable Closed-Loop Control: The integration of sensors (DHT22/BMP280) with the control logic proved robust. As evidenced in the test scenarios, the system autonomously detected environmental deviations (e.g., "Dry Air") and triggered the correct corrective actuators (LEDs) with a response latency of approximately 5 seconds.

### 6.2. CHALLENGES FACED AND SOLUTIONS

| CHALLENGE | IMPACT | IMPLEMENTED SOLUTION |
|---|---|---|
| **Model Hallucination** | The AI initially generated plausible but incorrect temperature ranges for rare plants. | Hybrid Architecture: We restricted the AI's role to *Intent Recognition* only. The actual safety parameters were moved to a hardcoded Python dictionary (`perfis_db`), ensuring physically safe limits are always applied. |
| **Sensor Instability** | The DHT22 sensor frequently returned `None` or timed out, threatening to crash the main loop. | Software Debouncing: We implemented a `try-except` block with a null-check retry mechanism in `main.py`. If a read fails, the system waits 2 seconds and retries rather than crashing. |
| **Resource Constraints** | Running an LLM consumes ~4.7GB of RAM, leaving little room for the OS and other processes. | Quantization & Monitoring: We utilized the 4-bit quantized version of Llama 3.2 and implemented a resource logger (log.py) to monitor memory usage in real-time, confirming stability before long-term deployment. |

Table 1: Table of challenges and solutions.

### 6.3. SYSTEM LIMITATIONS

While the prototype was successful, several limitations were identified during testing:

- **Static Knowledge Base:** The system currently relies on a local dictionary (`ia_botanico.py`) for plant profiles. If a user requests a plant not in this database (e.g., "Mango"), the system falls back to a "Generic Profile" rather than retrieving real agricultural data.
- **Inference Latency:** The initial "handshake" with the AI takes 3–6 seconds. While acceptable for plant growth (a slow process), this latency makes the current architecture unsuitable for high-speed robotics or emergency response systems.
- **Simulated Actuation:** The current build uses LEDs to represent heaters and humidifiers. Deploying this in a real greenhouse would require the addition of **Relay Modules** and a separate high-voltage power supply, introducing new risks related to electrical isolation.

### 6.4. LESSONS LEARNED

- **Don't Trust the AI with Safety:** The most important engineering lesson was that Generative AI should never directly control physical safety limits. A deterministic code layer must always validate AI outputs before they touch hardware.
- **The "Human-in-the-Loop" is Essential:** The implementation of the physical emergency button (`hardware.monitorar_botao`) proved critical. In physical computing, software kill-switches are insufficient; a hardware interrupt is mandatory.
- **Prompt Engineering is Code:** The reliability of the system depended heavily on the strict JSON schema defined in the `schema_obter_perfil`. We learned that treating prompts as strict API definitions is necessary for reliable IoT integration.

# 7. FUTURE WORKS & CONCLUSIONS

The **AI Garden Assistant** has successfully validated the concept of using Small Language Models (SLMs) for autonomous control at the edge. However, to transition from a prototype to a production-ready system, several architectural evolutions are necessary.

## 7.1. POTENTIAL IMPROVEMENTS

- Integration of Computer Vision (VLM):
  The Raspberry Pi 5 is powerful enough to handle Vision Language Models (like Llama 3.2 Vision or YOLO). A future iteration could include a CSI camera to allow the AI to "see" the plant.
  - *Use Case:* Instead of typing "Tomato," the user shows the plant to the camera. The AI identifies the species and detects diseases (e.g., "Yellow leaves detected -> Increase Nitrogen").
- Retrieval-Augmented Generation (RAG):
  Currently, the system relies on a hardcoded dictionary (perfis_db) for safety. Implementing a local vector database (like ChromaDB or Faiss) would allow the system to ingest PDF agricultural manuals. This would enable the AI to answer complex queries (e.g., "Why are my leaves curling?") based on verified documents rather than internal training data.
- Hardware Actuation Upgrade:
  The current hardware.py script uses LEDs to simulate state. A practical update involves replacing the LEDs with 5V Relay Modules. This would allow the GPIO pins to control real high-voltage appliances:
  - Red LED: Ceramic Heater (110V)
  - Blue LED: Water Pump (12V)
  - Yellow LED: Exhaust Fan (12V)

## 7.2. SCALABILITY CONSIDERATIONS

- Distributed Architecture (MQTT):
  The current "Monolithic" architecture (Sensors + AI on one device) does not scale to a large greenhouse with 50 different crops.
  - *Proposal:* Use the Raspberry Pi 5 solely as the central "AI Server." Use cheaper microcontrollers (ESP32) for the individual plant sensors. The ESP32s would send temperature data via **MQTT** to the Pi, which processes the logic and sends back commands.
- Database Migration:
  The log.py module currently writes to a CSV file (sensor_log.csv). For long-term data collection, this should be migrated to a Time-Series Database (like InfluxDB). This would allow for historical trend analysis (e.g., "Show me humidity trends over the last 6 months") without performance degradation.

## 7.3. ALTERNATIVE APPROACHES

- TinyML (Classification vs. Generation):
  If the goal is purely resource efficiency, the Generative AI approach could be replaced by TinyML. Instead of a 3-billion parameter chatbot, a small TensorFlow Lite model (kB instead of GB) could be trained specifically to classify sensor states.

    - *Trade-off:* TinyML is faster and uses less energy, but lacks the flexibility to "understand" new plants without retraining.
- Hybrid Cloud/Edge:
  To solve the latency issue (~5 seconds), a hybrid approach could be adopted.
    - *Critical Loop:* Run safety checks (If Temp > 40°C) on the microcontroller (0ms latency).
    - *Strategic Loop:* Run the AI profiling and complex reasoning on the Cloud (GPT-4 API) or a local Home Server. This reduces the thermal load on the Raspberry Pi.

### 7.4. CONCLUSION

This project demonstrated that **Generative AI is no longer confined to the cloud**. By utilizing quantization and efficient hardware like the Raspberry Pi 5, we successfully created an autonomous agent capable of biological reasoning and physical actuation.

While challenges regarding sensor reliability and inference latency remain, the "Human-in-the-Loop" architecture proved that AI can be safely integrated into physical systems. The **AI Garden Assistant** serves as a foundational blueprint for the next generation of Smart Agriculture: private, offline-capable, and adaptable to any crop.

## 8. REFERENCES


[1] M. Rovai, *Edge AI Engineering: Hands-on with the Raspberry Pi*. UNIFEI, 2025.

[2] Ollama Team, "Ollama: Get up and running with large language models," *Ollama Documentation*, 2024. [Online]. Available: https://ollama.com/. [Accessed: Dec. 07, 2025].

[3] Adafruit Industries, "Adafruit CircuitPython DHT Library," *GitHub Repository*, 2024. [Online]. Available: https://github.com/adafruit/Adafruit_CircuitPython_DHT. [Accessed: Dec. 07, 2025].

[4] B. Nuttall *et al.*, "GPIO Zero Documentation," *ReadTheDocs*, 2024. [Online]. Available: https://gpiozero.readthedocs.io/. [Accessed: Dec. 07, 2025].

[5] Meta AI, "Llama 3.2: Lightweight models for edge devices," *Meta AI Blog*, Sep. 2024. [Online]. Available: https://ai.meta.com/blog/llama-3-2/. [Accessed: Dec. 07, 2025].