UNIVERSITY OF PISA

Department of Information Engineering

Master's degree in Artificial Intelligence and Data Engineering

Industrial Application Project

VisionChat

Real-Time Vision and Voice Assistant using Raspberry PI 3B+

Work Group:
Andrea Bochicchio
Ivan Brillo
Filippo Gambelli

ACADEMIC YEAR 2025/2026

Contents

Abstract

This project presents the design and implementation of a prototype system based on a Raspberry Pi 3 B+ equipped with a camera and a microphone, capable of performing real-time object detection and natural language interaction. The main objective is to demonstrate that an interactive and intelligent vision-based system can operate effectively on hardware with limited computational resources.
The system allows the user to interact using natural language to ask questions about the current camera view, such as which objects are detected, their positions or whether a specific object is present. In addition to reactive behavior, the system also supports proactive interaction through a user profiling mechanism. For example, users can request to be notified whenever a particular object appears in the camera's field of view or when motion is detected in video and the system will automatically provide such notifications. All interactions are handled through natural language, enabled by integrated Speech-to-Text (STT) and Text-to-Speech (TTS) components. Particular attention was given to optimizing the object detection and interaction pipeline in order to meet the computational constraints of the Raspberry Pi.
The results show that systems combining real-time object detection and natural language interaction can be successfully deployed on low-cost and resource-constrained hardware. This approach has potential applications in several domains, including autonomous driving, where similar systems could be used to explain the vehicle's perception and decisions to passengers, improving transparency and user comfort.
The source code of the project is publicly available on GitHub[1].

[1] `https://github.com/andreabochicchio02/VisionChat`

1 Introduction

The aim of this document is to present in detail the design, implementation and evaluation of an interactive vision-based system. The system is built on a **Raspberry Pi 3 B+** equipped with a **Raspberry Pi Camera Module 2**, a **microphone** for capturing spoken natural-language input and **headphones** for audio output (which can be easily replaced with external speakers) and is capable of performing real-time object detection combined with spoken natural-language interaction.
The document describes what the system does, how its architecture is organized, the main technical challenges encountered during development and the solutions adopted to address them, as well as the performance achieved by the final prototype. Particular emphasis is placed on the constraints imposed by limited computational resources and on the design choices required to ensure satisfactory responsiveness and reliability.

1.1 System description and highlights

The developed prototype integrates computer vision, speech processing and natural-language interaction on embedded hardware. Using the Raspberry Pi Camera Module 3, the system continuously acquires images and performs real-time object detection.
Users can interact with the system through spoken natural language, asking questions such as which objects are currently visible, where an object is located in the scene, or whether a specific object is present. In addition to reactive interaction, the system also supports proactive behaviour through a user profiling mechanism. In this context, user profiling refers to the automatic acquisition and updating of user-related information that enables the system to adapt its behaviour and anticipate user needs. The profile is built dynamically from the user's spoken interactions rather than through a rigid, predefined form to be completed beforehand.
In particular, the system supports two forms of user-defined personalization. First, a user may request to be notified whenever a specific object appears in the camera's field of view; in this case, the system stores this preference and automatically generates a spoken notification when the event occurs. Second, a user may request to be notified whenever an object moves in the scene; in this case, the system monitors motion and likewise generates a spoken notification whenever movement is detected
All communication takes place through speech, enabled by Speech-to-Text (STT) and Text-to-Speech (TTS) modules. The overall architecture is designed to balance real-time performance, accuracy and resource consumption, allowing multiple components to operate concurrently on a low-cost embedded platform.

1.2 Problem formulation and critical issues

The main challenge of this project lies in implementing an interactive vision-and-language system on hardware with limited computational capabilities. Several critical issues had to be addressed:

Object detection model selection: A key problem was selecting an object detection model that provides a sufficiently large number of detectable classes and good accuracy, while remaining lightweight enough to perform real-time inference on the Raspberry Pi. This required careful evaluation of model complexity, inference speed and memory usage.

STT and TTS model selection: Another major challenge concerned the choice of models for Speech-to-Text and Text-to-Speech. These models had to be computationally efficient to run on the same

device as the object detection model, while still providing acceptable transcription accuracy and natural-sounding speech output.

Natural-language conversation management: Enabling natural-language interaction required a mechanism capable of understanding a wide range of user queries and generating coherent responses. For this reason, a Large Language Model (LLM) was chosen due to its expressive power and flexibility. However, the high computational cost of LLMs made it impractical to run them locally on the Raspberry Pi. As a solution, a cloud-based approach was adopted, where the LLM runs on an external PC and communicates with the embedded system via API calls.

2 System Description

2.1 High-level components

At a high level, the system is composed of three main actors: the **user**, the **embedded device** (a Raspberry Pi 3 B+ equipped with a Raspberry Pi Camera Module 3) and a **cloud host** (for the purposes of this project, the cloud is simulated by a PC running the LLM).

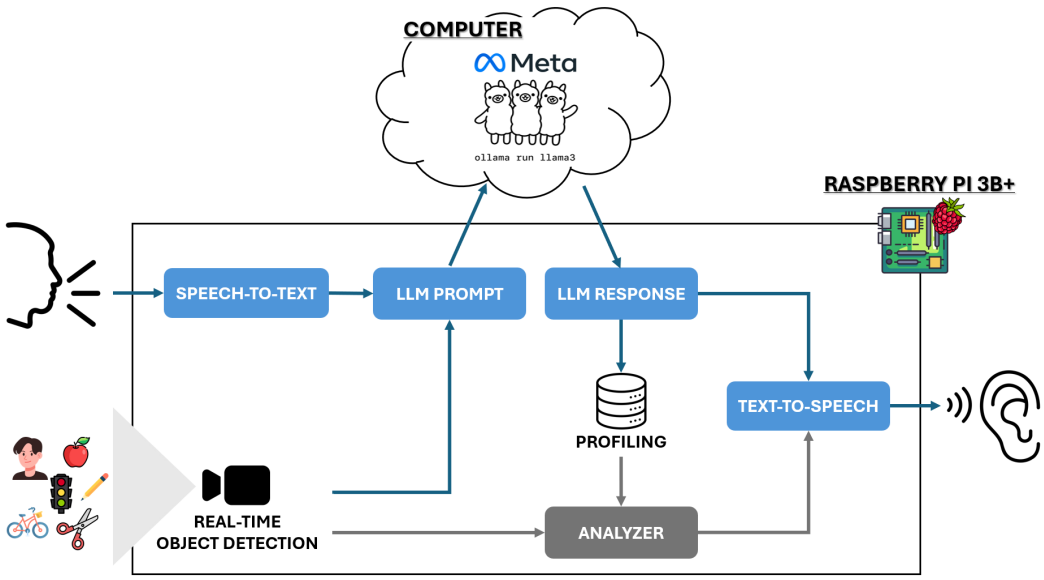

Figure 2.1: System overview

2.2 Main modules and data flow

The Raspberry Pi runs the main runtime components and communicates with the cloud LLM when needed. The primary modules inside the Raspberry Pi are:

- **Camera and Capture module:** acquires frames from the Raspberry Pi Camera Module V2 (v2.1). The camera is initialized and configured using the Picamera2 library. It captures video frames at a resolution defined by the system's parameter. The module handles camera start-up, configuration and frame streaming.
- **Object Detection module:** performs real-time object detection on sampled frames . For this project we use **SSD MobileNet V3 Large** object detection model, which is a lightweight deep learning model designed for efficient real-time inference on resource-constrained devices.
- **STT (Speech-to-Text) module:** listens to the user, detects speech endpoints (start/stop) and produces a transcription of the spoken input. The system supports two languages, **English** and **Italian**, which can be selected by the user through system configuration. Depending on the selected language, a different Vosk acoustic model is loaded at runtime. For English, the system uses the lightweight model `vosk-model-small-en-us-0.15`, while for Italian it uses `vosk-model-small-it-0.22`.
- **Prompt Builder / Fusion module:** combines the current scene description (object detection outputs) with the user transcription into a structured prompt and operates in **two logical stages**.

First, a lightweight classification prompt is generated to determine the intent of the user request, in particular whether the user is asking to enable a notification (e.g., when a specific object appears or when motion is detected) or whether the request is a general question about the scene. If the request corresponds to a notification setup, the module extracts and stores the relevant parameters and no further LLM reasoning about the scene is required. If, instead, the request is identified as a general query, the module constructs a second, full prompt that fuses the user transcription with the current object detection results. This final structured prompt is carefully formatted so that the LLM can answer questions about what the camera sees, generate natural-language responses

- **Profiling / Storage module:** stores user preferences and subscription rules (for example, "notify me whenever a bicycle appears"). User preferences are expressed through natural language and are automatically extracted using the LLM. In particular, a structured prompt is provided to the LLM in order to obtain a JSON-formatted output containing, for example, the list of objects for which the user wishes to receive notifications when they are detected by the camera. These extracted preferences are then stored locally on the Raspberry Pi and are made available to the analyzer module so that they can be used to trigger proactive notifications based on the current scene.

- **Analyzer:** continuously monitors the output of the Object Detection module and compares the list of currently detected objects with the user preferences stored in the Profiling/Storage module. When a user-defined condition is satisfied (for example, when the user has requested to be notified whenever a bicycle appears and a bicycle is detected in the current scene), the analyzer automatically triggers a notification, which is forwarded to the TTS module so that the user is informed through spoken feedback

- **TTS (Text-to-Speech) module:** converts the reply text into audio output and plays it to the user. The system supports both English and Italian and the language of the synthesized speech is selected according to the user's configuration, ensuring consistency with the chosen STT language. For speech synthesis, the system uses pico2wave, a lightweight TTS engine based on SVOX Pico, which provides compact, low-latency speech synthesis suitable for embedded and resource-constrained platforms.

2.3 Detailed interaction sequence

The typical run-time sequence for a user query is:

1. The user starts an interaction by clicking a button on the user interface, which activates the microphone and allows the user to speak in natural language. While the user is speaking, speech is transcribed into text in real time, without waiting for the end of the speech. The STT module automatically detects when the user has finished speaking using speech endpoint detection.

2. After the transcription is available, the system builds a first lightweight prompt and sends it to the LLM to determine the intent of the user request. In particular, this first prompt is used to decide whether the user is requesting to set a preference or notification (for example, "notify me when there is motion") or whether the request is a general question about the scene. This first prompt is sent to the LLM and the response is returned as a structured JSON object (see Prompt and Response Formats section).

3. If the request is identified as a general question, the system builds a second, full prompt. This prompt includes the list of objects detected in the most recent camera frame, the position of each object in the scene, the user's transcribed message and additional instructions for generating a correct and helpful answer. This full prompt is then sent to the LLM to generate the final response

4. The LLM response is streamed back to the user interface and simultaneously converted into audio by the TTS module, allowing the user to both read and hear the system's reply in real time.

2.3.1 Prompt and Response Formats

```
USER MESSAGE: {<user_text>}

TASK: Classify whether the user is asking for a future notification when objects appear or when there
is movement.

DECISION RULES (FOLLOW ALL):
1. Set "is_alert_request" to true only if the user explicitly asks for a future notification.
2. The message must contain clear notification verbs such as: notify me, alert me, tell me when, let me
know.
3. Questions about the current scene (for example, what do you see? or what is there?) are not alert
requests.
4. If there is any doubt, set "is_alert_request" to false.
5. Do not guess or invent objects.
6. Extract target objects only if they are explicitly mentioned in the user message.
7. Set "is_motion_request" to true if the user asks for movement or motion detection.

OUTPUT FORMAT (JSON ONLY):
{
    "is_alert_request": true or false,
    "is_motion_request": true or false,
    "target_objects": ["object1", "object2"]
}

EXAMPLES:
User: "What can you see?"
Output: { "is_alert_request": false, "is_motion_request": false, "target_objects": [] }

User: "Notify me when you see a dog"}
Output: { "is_alert_request": true, "is_motion_request": false, "target_objects": ["dog"] }

User: "Alert me if a person appears"}
Output: { "is_alert_request": true, "is_motion_request": false, "target_objects": ["person"] }

User: "Do you see a cat?"}
Output: { "is_alert_request": false, "is_motion_request": false, "target_objects": [] }

User: "Notify me when there's movement"
Output: { "is_alert_request": true, "is_motion_request": true, "target_objects": [] }
```

Figure 2.2: Classification Prompt

```
HISTORY LABEL: {<conversation history>}

OBJECTS LABEL: {<currently visible object with position>}

USER MESSAGE LABEL: {<user_text}

INSTRUCTION: You must converse with the user while observing objects. Follow these rules strictly:
1. Always provide short and direct answers.
2. Answer general or common knowledge questions using only your general knowledge, without referring to
visible objects or the visual context.
3. If the user asks questions such as: What are you seeing? Which objects are present in front of you?
What objects are around you? refer exclusively to the objects listed above under CURRENTLY VISIBLE
OBJECTS.
4. Never mention visible objects if the question does not directly concern them.
```

Figure 2.3: Conversation Prompt

2.4 Why this distribution of computation?

We distribute computation between the Raspberry Pi and the cloud PC to balance responsiveness, autonomy and language flexibility:

- **On the Raspberry Pi:** core perception and interaction components are executed locally, including vision inference (SSD MobileNet V3), Speech-to-Text (Vosk small), Text-to-Speech and the Analyzer module. This design choice was driven by the need to evaluate whether a complete perception–interaction loop could realistically run on hardware with limited computational resources. Several configurations were tested to assess CPU load, memory usage and latency, confirming that lightweight models and careful scheduling allow the system to remain responsive without saturating the device. Running these components locally reduces dependency on network availability, minimizes interaction latency and demonstrates the feasibility of deploying intelligent, interactive systems directly on low-cost embedded platforms.
- **On the cloud PC:** the LLM (Llama 3.2) runs remotely using Ollama to provide robust and flexible natural-language understanding and generation. Ollama makes it straightforward to serve LLMs locally on a PC and expose a simple API that the Raspberry Pi can call. This hybrid approach lets us use a powerful LLM without attempting to run it directly on the Raspberry Pi.

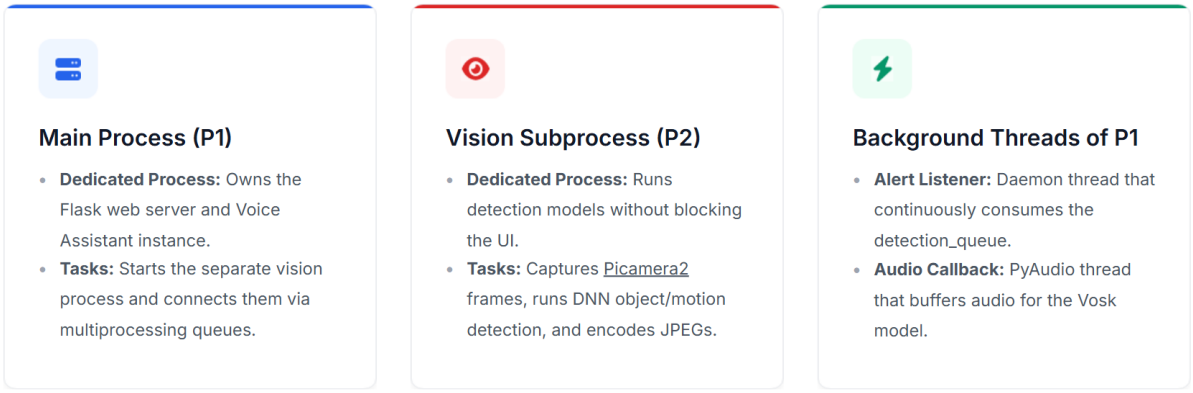

Figure 2.4: System architecture diagram illustrating the separation of concerns between the Main Process (P1), the Vision Subprocess (P2) and associated background threads for audio and alert handling.

2.5 Models and important technical choices

Here we summarise the main models and the reasons behind their selection:

Object detection model: `ssd_mobilenet_v3_large_coco_2020_01_14`

For the object detection module, the system employs the SSD MobileNet V3 Large model [5], a lightweight deep learning architecture specifically designed for efficient real-time inference on resource-constrained devices. The model combines the Single Shot Detector (SSD) framework with the MobileNet V3 Large backbone, enabling single-pass object detection with a good balance between computational efficiency and detection accuracy.

The model is trained on the MS COCO (Common Objects in Context) dataset, a large-scale and widely adopted benchmark for object detection. The COCO dataset contains over 300,000 images and annotations for 80 everyday object categories, spanning a wide range of object types, sizes and real-world scenarios. This makes the model well suited for detecting common objects encountered in indoor and outdoor environments.

Thanks to its lightweight design and optimized inference pipeline, SSD MobileNet V3 Large can run on CPU-only embedded platforms such as the Raspberry Pi, provided that appropriate input resolution, frame sampling and inference frequency are carefully configured. This makes it an effective trade-off between model complexity, runtime performance and practical usability in embedded vision systems.

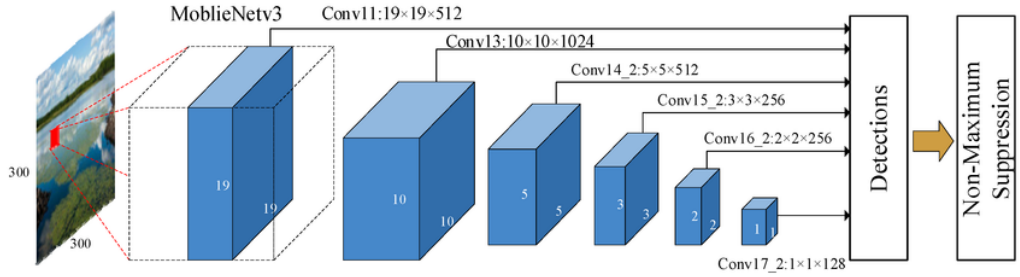

Figure 2.5: SSD MobileNet V3 Large Architecture

STT model: `Vosk small models (English and Italian)`

Depending on the selected language, a different Vosk acoustic model is loaded dynamically at runtime. For English, the system uses the lightweight model `vosk-model-small-en-us-0.15`, while for Italian it uses `vosk-model-small-it-0.22`.

Vosk models are based on the Kaldi speech recognition toolkit and rely on compact deep neural network acoustic models combined with statistical language models and pronunciation dictionaries. These models are specifically designed for offline, real-time speech recognition on resource-constrained devices. They provide a small disk footprint, low memory usage and low-latency decoding, while still offering acceptable transcription accuracy for conversational speech.

TTS model: `pico2wave (SVOX Pico)`

The system supports both English and Italian speech synthesis, with the output language automatically selected according to the user's configuration to ensure consistency with the chosen STT language. For text-to-speech synthesis, the system uses pico2wave, a lightweight TTS engine based on the SVOX Pico speech synthesis technology.

SVOX Pico is designed for embedded and resource-constrained platforms and is widely used in low-power devices due to its compact size, low computational requirements and low-latency speech generation. While it does not provide high-fidelity or highly natural prosody compared to modern neural TTS systems, it produces intelligible and fast speech output, making it well suited for real-time feedback in embedded interactive systems.

LLM: `Llama 3.2 3B`

The system integrates the Llama 3.2 large language model [4], which is hosted on a cloud-connected PC and served through Ollama. Llama 3.2 provides strong instruction-following capabilities and support for structured text outputs, enabling advanced reasoning, dialogue management and high-level decision-making within the system.

Ollama simplifies the deployment and management of large language models on local machines and exposes the model through a REST API. This allows the Raspberry Pi to offload computationally intensive language processing tasks to the cloud PC while maintaining a lightweight client-side implementation. This architecture enables the use of a powerful LLM without exceeding the hardware limitations of the embedded platform, while still supporting low-latency communication and scalable system design.

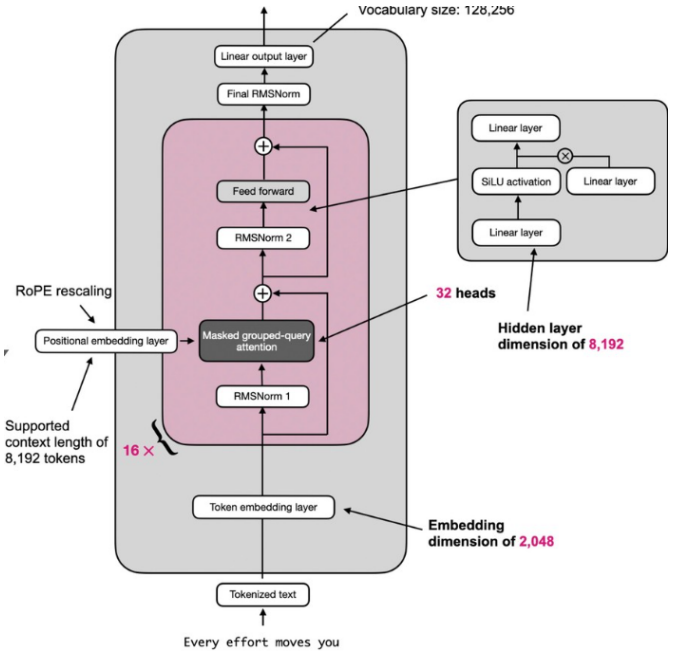

Figure 2.6: Llama 3.2 Architecture

2.6 Performance trade-offs and issues encountered

Object detection model We initially considered **YOLO-based models**, including lightweight and quantized variants, due to their high detection accuracy. However, practical tests and literature reports showed that these models are often too heavy for a Raspberry Pi 3 B+, achieving unviable frame rates (2.7) and causing very high CPU usage. Such load risked system instability and left little room for the STT, TTS and control logic.

We therefore chose the SSD MobileNet V3 Large model instead. Unlike YOLO, which prioritizes accuracy but demands more computational resources, SSD with a MobileNet backbone offers a better trade-off for embedded devices: lower CPU usage, stable frame rates and sufficient accuracy for common objects. This choice ensured real-time performance and system reliability on the Pi while keeping headroom for other modules.

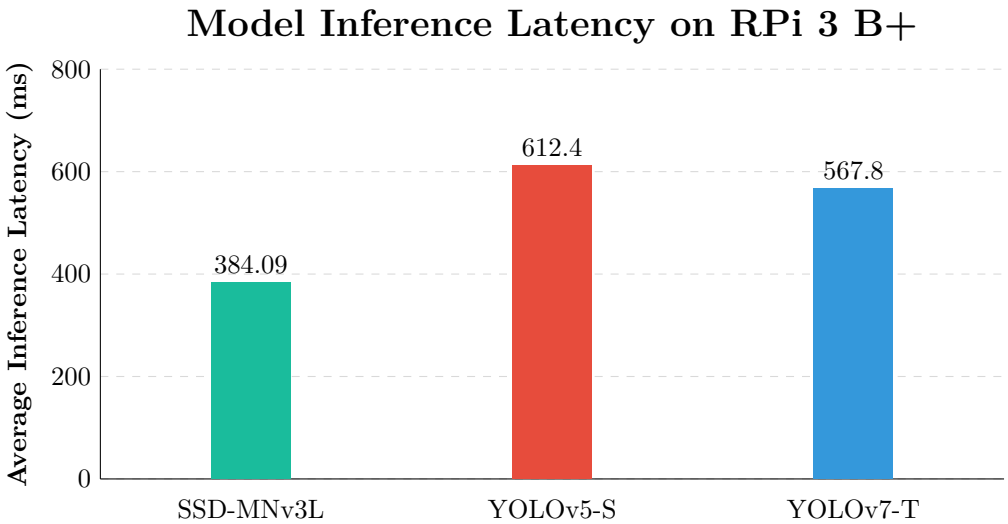

Figure 2.7: Average inference latency comparison in Raspberry Pi 3 Model B+. SSD MobileNet V3 Large (5.4M parameters), YOLOv5 Small (7.5M), YOLOv7 Tiny (6.24M).

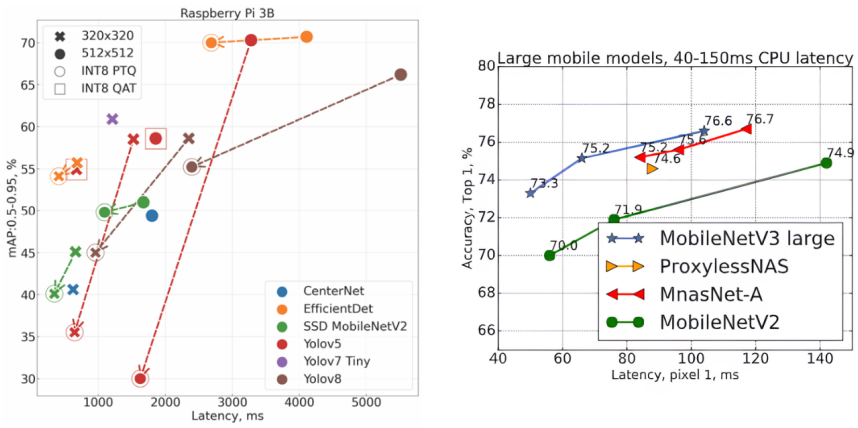

Figure 2.8: Comparative benchmarks for model selection. Left: Performance of object detection models on Raspberry Pi 3B, plotting mAP against latency to illustrate trade-offs on low-power devices [1]. Right: Accuracy vs. CPU Latency comparison, demonstrating that MobileNetV3 Large (blue stars) achieves higher accuracy than MobileNetV2 (green circles) at comparable speeds [2].

Our selection of SSD MobileNetV3 is grounded also in comparative performance metrics and architectural improvements over its predecessors present in the literature 2.8.

Large Language Model (LLM) Due to hardware constraints, we needed to run the LLM on a PC without high-end GPUs, which limited our options to models capable of running efficiently on modest GPUs or even CPU-only. We selected **Llama 3.2:3B** for the system. Larger models, such as Llama 3.1:8B, do not significantly improve response quality but noticeably increase latency. Conversely, specialized reasoning models like **DeepSeek-R1:1.5B** are too small to provide reliable answers, while larger reasoning models introduce unacceptable delays.
Using Llama 3.2:3B provided a good balance between response accuracy, instruction-following capabilities and acceptable latency, making it suitable for real-time interaction in our setup.

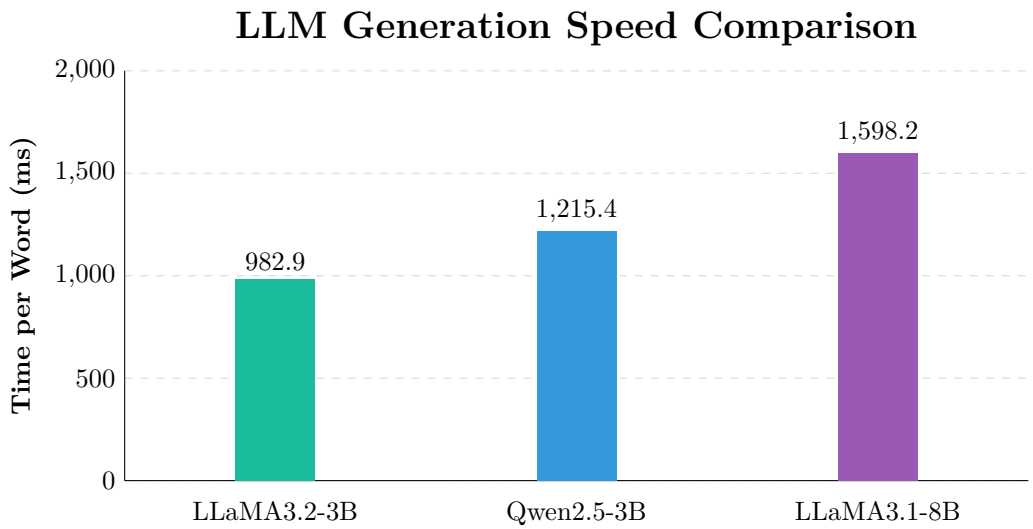

Figure 2.9: LLM time per generated word on a local system. LLaMA 3.2 3B achieves the lowest latency. Experiments conducted on an AMD Ryzen 7 8845HS CPU with 32 GB RAM and integrated GPU.

Recent literature highlights the advantages of smaller Llama 3.2 models compared to larger Llama 3.1 variants or the Qwen 2.5 architecture, as shown in Figure 2.10.

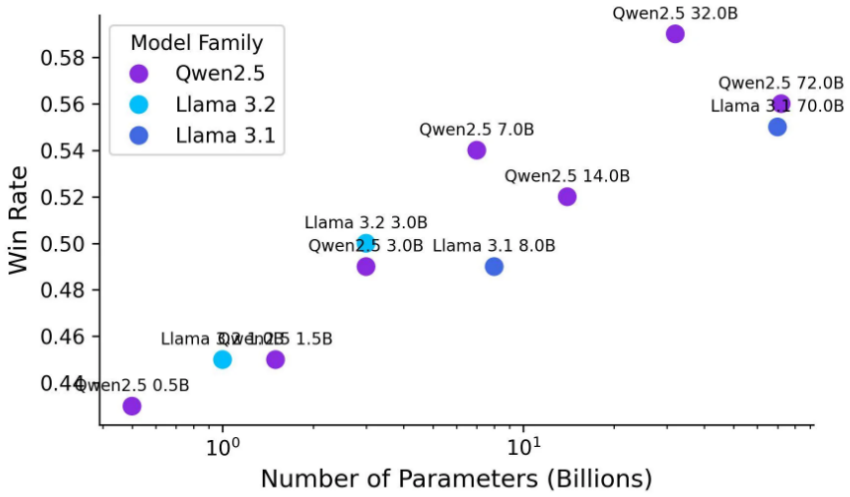

Figure 2.10: Win Rate vs GPT-4o mini: Llama 3.2, Llama 3.1 and Qwen 2.5 [3]

Speech-to-Text module We evaluated several STT models. For English, larger models such as vosk-model-en-0.22 offered higher accuracy but were too large (1.8 GB) for the Raspberry Pi, while vosk-model-en-us-0.22-lgraph produced fewer errors but introduced multi-second latency, which was unacceptable for real-time interaction. OpenAI's **Whisper-Tiny** showed good accuracy but had slightly higher latency and lacks true streaming

Text-to-Speech module Pico2wave TTS solution supports both Italian and English, allowing the system to provide spoken feedback in multiple languages depending on the user interaction context. During development, an alternative lightweight TTS engine, **eSpeak**, was also evaluated. However, pico2wave was selected for the final system because it produces a more natural and human-like voice compared to eSpeak, resulting in a better overall user experience.

3 Algorithm Description

3.1 SSD-MobileNet-V3-L Object Detection Workflow

The object detection pipeline is implemented using the SSD MobileNet V3 model integrated with OpenCV DNN and Picamera2 for real-time inference.

Camera Initialization and Configuration The camera is initialized and configured to capture video frames at a fixed resolution. The Picamera2 interface is used to acquire RGB frames, which are then converted to BGR format to ensure compatibility with OpenCV.

Model Loading and DNN Setup The SSD MobileNet V3 model is loaded using OpenCV's DNN module by reading the TensorFlow model weights and configuration files. The OpenCV backend and CPU target are selected to perform inference on embedded or low-power systems.

Blob Creation for Neural Network Input For each captured frame, the image is preprocessed into a blob using cv2.dnn.blobFromImage. This step includes resizing to the network input size, pixel value scaling, mean subtraction, and channel reordering to match the input requirements of SSD MobileNet V3.

Forward Pass (Inference) The preprocessed blob is passed to the network, and a forward pass is executed to obtain detection results. The network outputs a list of detections containing class IDs, confidence scores, and normalized bounding box coordinates.

Detection Filtering and Post-Processing Post-processing is applied to filter detections based on a confidence threshold and to map class IDs to human-readable labels. Invalid or low-confidence detections are discarded. Finally, normalized bounding boxes are scaled to pixel coordinates and drawn on the original frame along with class labels and confidence scores, enabling real-time visualization of detected objects.

3.2 Lightweight Motion Detection Algorithm

Frame Downscaling Each input frame is first downscaled using nearest-neighbor interpolation to reduce spatial resolution and improve processing speed. This significantly lowers the number of pixels to be analyzed while preserving coarse motion information. The downscaled frame is converted to grayscale to reduce data dimensionality from three color channels to a single intensity channel, simplifying subsequent processing. A light Gaussian blur is applied to suppress high-frequency noise and small sensor fluctuations that could otherwise generate false motion detections.

Previous Frame Initialization If no previous frame is available, the current grayscale frame is stored and motion detection is skipped for the first iteration to establish a reference frame.

Frame Differencing Motion is computed by calculating the absolute difference between the current and previous grayscale frames. This operation highlights pixels that have changed between consecutive frames.

Binary Thresholding A binary threshold is applied to the difference image to filter out small intensity variations and retain only significant pixel changes, producing a binary motion mask.

Motion Decision The algorithm counts the number of changed pixels and computes the percentage of motion relative to the total frame area. This normalized metric allows resolution-independent motion evaluation. Motion is considered detected if the computed motion percentage exceeds a predefined minimum area threshold. The current frame is stored as the previous frame for the next iteration.

```python
def detect_objects(self, frame: np.ndarray) -> List[DetectedObject]:
        # - Mean (127.5, ...): Subtracts this mean from each channel for normalization.
        # - Scale factor (1.0/127.5): Scales pixel values from [0, 255] to [-1, 1].
        blob = cv2.dnn.blobFromImage(frame, 1.0 / 127.5, BLOB_SIZE, (127.5, 127.5, 127.5), True, False)

        # Set the blob as input to the neural network
        self.net.setInput(blob)

        # Run the forward pass to get the output.
        detections = self.net.forward()

        detected_objects: List[DetectedObject] = []

        # The output shape is [1, 1, N, 7], where N is the number of detections.
        for i in range(detections.shape[2]):

            # Extract confidence (probability) at index 2
            confidence = detections[0, 0, i, 2]

            # Filter out weak detections based on the minimum threshold
            if confidence > CONFIDENCE_THRESHOLD:

                # Extract the class ID (index 1)
                class_id = int(detections[0, 0, i, 1])

                # Ensure the class ID is valid and present in our label map
                if class_id < len(self.classes) and self.classes[class_id] != "N/A":

                    # Extract bounding box coordinates (indices 3 to 7).
                    x_min, y_min, x_max, y_max = detections[0, 0, i, 3:7]

                    class_name_str = self.classes[class_id]

                    # Store the result in a structured object
                    detected_objects.append(DetectedObject(
                        class_name=class_name_str,
                        confidence=confidence * 100,      # Convert to percentage
                        bounding_box=(x_min, y_min, x_max, y_max)
                    ))

        # Cache the latest detections (useful for asynchronous drawing or logic)
        self._last_detections = detected_objects
        return detected_objects
```

Figure 3.1: Object Detection Workflow

3.3 Object Notification Algorithm

User Request The process starts when the user issues a natural language voice command requesting to be notified upon detection of a specific object (e.g., “Notify me when you see a bottle”).The user’s spoken request is converted into text using a speech-to-text (STT) system. This step transforms the raw audio input into a textual representation suitable for natural language processing.

LLM-Based Intent and Entity Extraction The request is sent to a Large Language Model (LLM) for semantic interpretation. The LLM analyzes the user request to determine: Whether the user is requesting a notification (notification intent) and target objects the user wants to be notified about. If the LLM detects a notification intent, it returns a structured JSON response containing: a boolean flag indicating that a notification request is presentand a list of object class names requested by the user. The format of the prompts and responses are described in 2.3.1

Notification Rule Creation The list of target objects returned by the LLM is stored internally as active notification rules. These rules define which object classes should trigger a user notification.

Object List Matching For each new video frame, the object detection pipeline (SSD MobileNet V3) is executed to identify all visible objects in the scene. The system compares:the list of objects detected in the current frame and the list of target objects requested by the user for notification. This matching is performed at the class-name level.

Notification Trigger If at least one detected object matches one of the user-requested target objects, the system triggers a notification to the user, informing them that the requested object has been detected in the camera view.

```python
def detect_motion(self, frame: np.ndarray) -> bool:
    # 1. Downscale for faster processing
    # Uses Nearest Neighbor interpolation for maximum speed
    small = cv2.resize(frame, None, fx=self.scale_factor, fy=self.scale_factor,
                       interpolation=cv2.INTER_NEAREST)

    # 2. Convert to grayscale
    # Reduces data dimensionality (3 channels → 1 channel)
    gray = cv2.cvtColor(small, cv2.COLOR_BGR2GRAY)

    # 3. Light blur to reduce noise
    # Prevents false positives from sensor noise or small vibrations
    gray = cv2.GaussianBlur(gray, (5, 5), 0)

    # Initialize previous frame if this is the first iteration
    if self.prev_gray is None:
        self.prev_gray = gray
        return False

    # 4. Compute absolute difference
    # Calculates |current_frame - previous_frame|
    frame_delta = cv2.absdiff(self.prev_gray, gray)

    # 5. Binary threshold
    # Filters out small pixel changes below a certain intensity
    _, thresh = cv2.threshold(frame_delta, self.threshold, 255,
cv2.THRESH_BINARY)
    # 6. Count changed pixels and calculate percentage
    motion_pixels = cv2.countNonZero(thresh)
    total_pixels = gray.shape[0] * gray.shape[1]
    motion_percent = (motion_pixels / total_pixels) * 100

    # Update previous frame for the next loop
    self.prev_gray = gray

    return motion_percent > self.min_area_percent
```

Figure 3.2: Motion Detection Algorithm

3.4 Inter-Process Communication

Detection Process (Producer) The object detection process continuously acquires frames from the camera, performs object detection, and acts as a producer of detection results and annotated frames. After each detection cycle, the detection process pushes the structured detection metadata into a shared `detection_queue`. This queue contains the list of detected objects. In parallel, the detection process pushes the annotated video frame (with bounding boxes already drawn) into a separate `frame_queue`.
Conversation Process (Consumer) Using the detection metadata from the `detection_queue`, the conversation process: Evaluate user-defined notification rules and determine whether a detected object should trigger a user alert. Answer user queries about which objects are currently visible in the camera view. The `frame_queue` is consumed by the UI, It Retrieves the latest annotated frames and displays the live camera stream with bounding boxes and labels.

4 Prototype and Setup description

The system is composed of both embedded and desktop-class hardware components, designed to support image acquisition, processing and large language model (LLM execution).

4.1 Embedded Components

Embedded Platform
The embedded platform used in this project is a **Raspberry Pi 3 Model B+**. This single-board computer is equipped with a Broadcom BCM2837B0 quad-core ARM Cortex-A53 processor running at 1.4 GHz, along with 1 GB of LPDDR2 SDRAM. The Raspberry Pi 3 Model B+ provides integrated Wi-Fi (802.11ac), Bluetooth 4.2, Ethernet connectivity, USB ports and GPIO interfaces, making it suitable for interfacing with peripheral devices and sensors.

Camera Module
For image acquisition, the system uses the **Raspberry Pi Camera Module V2 (v2.1)**. This camera module is based on the 8-megapixel Sony IMX219 sensor. It supports fixed-focus optics and is capable of capturing high-resolution still images and HD video streams. The camera is directly connected to the Raspberry Pi via the CSI (Camera Serial Interface), ensuring low-latency and high-bandwidth data transfer for real-time image acquisition and computer vision tasks.

4.2 Cloud Components

Host Computer for LLM Execution
The Large Language Model (LLM) is executed on a separate host computer equipped with an AMD Ryzen 7 8845HS processor and 32 GB of RAM. This host computer is used exclusively for LLM inference and does not perform other system-level processing tasks. Importantly, the system does not include a dedicated GPU; therefore, LLM execution is performed on the CPU.

4.3 System Architecture Overview

The embedded system communicates with the host computer by sending requests to an **Ollama server endpoint**, which exposes the LLM through a RESTful API. In this architecture, the Raspberry Pi acts as a client, issuing inference requests to the Ollama endpoint, while the host computer handles the computational workload associated with LLM execution. This design enables the Raspberry Pi to remain lightweight and focused on hardware interfacing and data acquisition, while offloading computationally intensive natural language processing tasks to the host system.

4.4 Installation Guide

INFO: Operating System

Raspberry Pi OS (Legacy, 64-bit) Lite

- Based on Debian Bookworm
- Includes security updates
- Headless/Lite version (no desktop environment)
- Maximizes resources for AI models

1. System Preparation (Raspberry Pi)

Ensure your Raspberry Pi is running the specified OS. Update the system and install system-level dependencies for audio and image processing:

```
# Update system packages
sudo apt update && sudo apt upgrade -y

# Install required dependencies
sudo apt install -y python3-pip python3-venv portaudio19-dev \
  libatlas-base-dev libttspico-utils ffmpeg libsm6 libxext6
```

2. Clone the Repository

Clone the VisionChat repository to your Raspberry Pi:

```
# Clone the repository
git clone https://github.com/andreabochicchio02/VisionChat

# Navigate to project directory
cd VisionChat
```

3. Set up the Python Virtual Environment

It is **highly recommended** to use a virtual environment to manage dependencies:

```
# Create virtual environment
python3 -m venv venv

# Activate virtual environment
source venv/bin/activate
```

4. Install Python Dependencies

Install the required Python libraries using `pip`:

```
# Install Python dependencies
pip install -r requirements.txt
```

5. Setup the LLM (External PC)

1. **Install Ollama**

   Download and install Ollama from: `https://ollama.com/`

2. **Pull the Llama 3.2 Model**

```
# Download the Llama 3.2 model
ollama pull llama3.2:3b
```

3. **Configure Network Access**

   Ensure the Ollama server is running and accessible via the network. You may need to configure the `OLLAMA_HOST` environment variable on the PC to accept external connections.

WARNING: Network Configuration

Both the Raspberry Pi and Host PC must be on the **same Wi-Fi/LAN network** for proper communication.

4.5 Usage

Hardware Connection

Connect the peripherals to the Raspberry Pi in the following order:

1. **Camera Module**: ensure the ribbon cable is seated correctly
2. **USB Microphone**
3. **Speakers/Headphones**
4. **Power on the Raspberry Pi**

Configuration

INFO: Network Requirements

Verify that:

- The Host PC is on the **same Wi-Fi/LAN network** of the Raspberry Pi
- The Ollama service is running on the Host PC
- Firewall settings allow communication between devices

Run the Application

On the raspberry pi, navigate to the project directory and execute the startup script:

```
# Make the script executable (if needed)
chmod +x run.sh

# Run the application
./run.sh
```

On the cloud computer execute:

```
ollama serve
```

INFO: First Run

The first time you run the application, it may take a few moments to initialize all components and establish the connection with the Ollama server.

5 Performance Evaluation

5.1 Video Process Metrics

Object Detection and Inference The average inference time for object detection is 384.09 ms, with a standard deviation of 81.60 ms indicates a moderate variability, which can be mainly attributed to changes in scene complexity and to the number of objects present in each frame. This behavior is consistent with the computational limitations of the Raspberry Pi 3 Model B+, which provides limited CPU and memory resources. These fairly short times confirm the good choice of the object detection model

Metric	Value
Average Inference Time (ms)	384.09
Minimum Inference Time (ms)	255.93
Maximum Inference Time (ms)	859.09
Standard Deviation of Inference Time (ms)	81.60
Inference Time per Object (ms/object)	321.77

Table 5.1: Object Detection and Inference Metrics

FPS Metrics

The frame rate results indicate an average FPS of 4.44. These low-frame-rate values are typical for embedded platforms performing on-device video processing and inference. In particular, the average classification FPS is 1.41 FPS. The classification stages represent one of the main performance bottlenecks in the processing pipeline.

Metric	Value
Average FPS	4.44
Min FPS	2.84
Max FPS	4.69
Average Classification FPS	1.41

Table 5.2: FPS Metrics

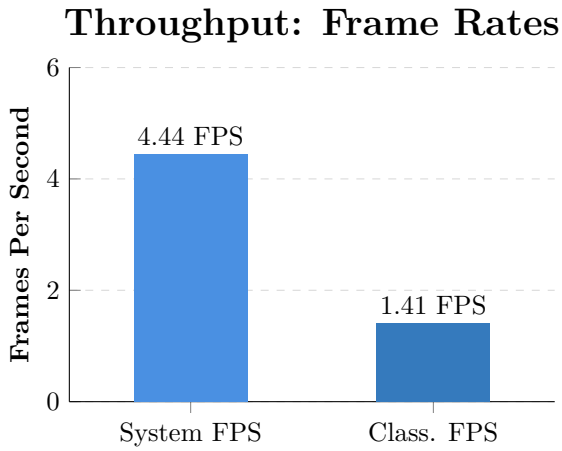

Figure 5.1: FPS Metrics: System vs. Classification

Frame Processing

These results indicate that frame acquisition and basic preprocessing are not the primary limiting factors in system performance. Instead, the dominant computational cost is associated with inference and classification. This suggests that future optimizations should primarily focus on reducing the complexity of the vision model

Metric	Value
Avg Frame Capture Time (ms)	54.96
Avg Frame Encoding Time (ms)	26.15
Avg Motion Detection Time (ms)	1.43

Table 5.3: Frame Processing

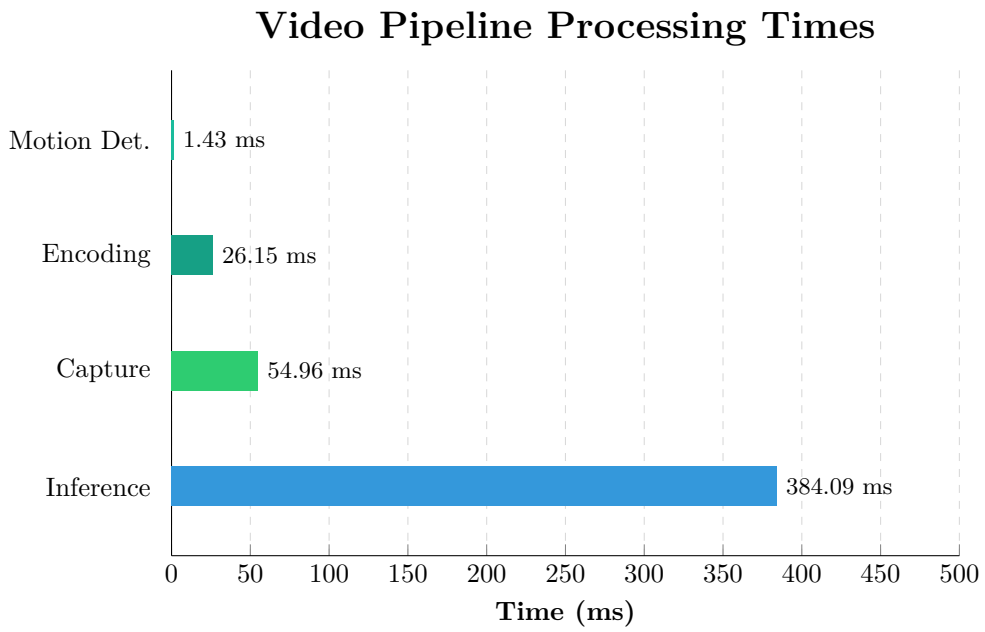

Figure 5.2: Video Pipeline Metrics

5.2 LLM Metrics

LLM Request Metrics

The LLM-related timings reported are significantly higher than the local processing times. The overall average LLM request time of 7301.37 ms clearly indicates that interaction with the language model represents the main performance bottleneck of the entire system. The average time to first token (3410.96 ms) highlights a substantial initial latency, which can be attributed to network communication overhead as well as to model initialization and generation latency on the remote host. The use of response streaming allows the system to partially mask this latency, enabling the user to perceive a progressive response rather than waiting for the full completion of text generation.

Metric	Value
First LLM Request Time (ms)	9492.38
Second LLM Request Time (ms)	3948.81
Average LLM Request Time (ms)	7301.37
Avg Time to First Token (ms)	3410.96
LLM Time per Word (ms/word)	982.88

Table 5.4: LLM Request Metrics

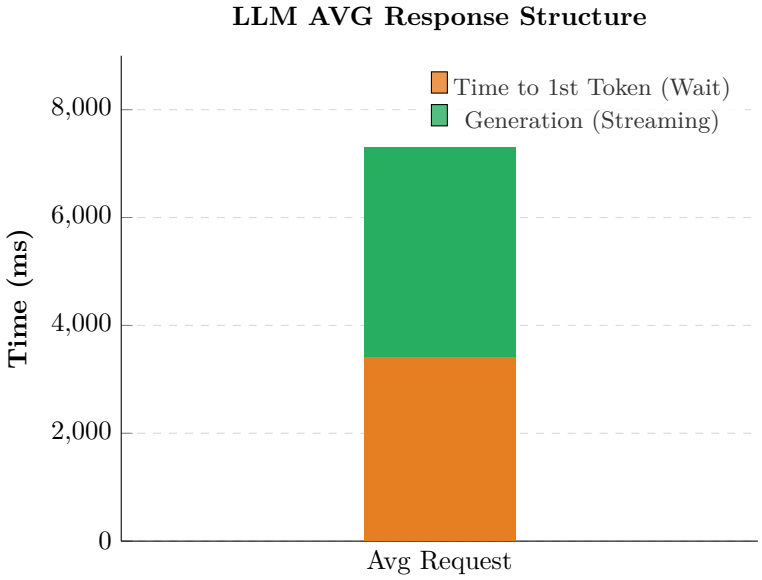

Figure 5.3: LLM Response Structure Metrics

6 Structure of the Demo

In the demo, several interactions with the system were recorded in order to showcase the main functionalities of the proposed architecture. The demonstrated features include:

- Real-time object recognition with spatial position estimation, executed on the Raspberry Pi 3.
- Speech-to-Text (STT) and Text-to-Speech (TTS) models running locally on the Raspberry Pi 3.
- Web-based user interface implemented using Flask and executed on the Raspberry Pi 3.
- Open-source Large Language Model (LLM) executed locally on a separate host PC.
- User requests for notifications when a specific object is detected in the scene.
- User requests for notifications when object motion is detected.
- Streaming visualization of the LLM response in the user interface, without waiting for the complete response to be generated.
- User-selectable dual-language interaction, supporting both Italian and English for STT and TTS.

Two video demonstrations of the application are available and can be accessed at the following links:

- Video 1
- Video 2

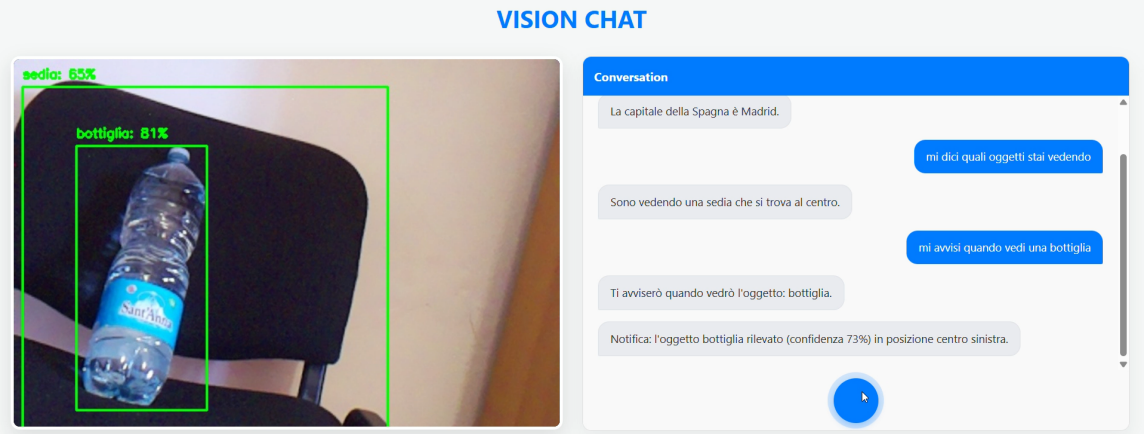

Figure 6.1: UI Demo

7 Conclusion

This project successfully presented the design of a prototype system based on a Raspberry Pi 3 B+ capable of real-time object detection and natural language interaction, demonstrating that intelligent vision-based systems can operate effectively on resource-constrained hardware.

7.1 System Validation

The solution validates a hybrid architecture where perception tasks run locally while reasoning is offloaded to a local cloud host.

- **Object Detection:** The **SSD MobileNet V3 Large** model proved a viable performance, offering a sustainable trade-off between accuracy and latency compared to heavier YOLO-based alternatives.
- **Interaction:** Integrating **Vosk** (STT) and **pico2wave** (TTS) enabled effective voice interaction without introducing excessive latency or external cloud dependencies.
- **Reasoning: Llama 3.2 3B** hosted via Ollama provided the necessary instruction-following capabilities, verifying that powerful LLMs can be leveraged via a client-server architecture in a GPU-free system.

7.2 Performance Summary

Experimental results confirm the system meets functional requirements, despite identified bottlenecks:

- The vision pipeline achieved an average throughput of **4.44 FPS** (inference avg. **384.09 ms**), sufficient for identifying objects in static or slowly changing environments.
- The primary bottleneck was LLM interaction (avg. request **7.3 s**). However, response streaming mitigated the impact, allowing users to read the response as it is generated.

7.3 Future Work

Future optimizations should target both the embedded hardware and the inference backend to overcome current performance bottlenecks:

- **Embedded Hardware Upgrade:** Transitioning to a next-generation Raspberry Pi or integrating dedicated accelerators (e.g., the Raspberry Pi AI Kit with Hailo-8L) would drastically improve the computer vision throughput. However, these on-device upgrades remain insufficient for running the full LLM locally.
- **LLM Acceleration:** To address the high latency observed with the current CPU-only host, the language model should be migrated to a machine equipped with a dedicated GPU or offloaded directly to a high-performance cloud-based API service.

In conclusion, this work demonstrates that by selecting lightweight models, advanced AI features can be democratized on low-cost hardware.

Bibliography

[1] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, Hartwig Adam, *Searching for MobileNetV3*, Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019.

[2] A. Zagitov, E. Chebotareva, A. Toschev, E. Magid, *Comparative analysis of neural network models performance on low-power devices for a real-time object detection task*, Computer Optics, 2024.

[3] Artificial Analysis, *Win Rate vs GPT-4o mini: Llama 3.2 vs Qwen 2.5*, Artificial Analysis, 2024.

[4] Aaron Grattafiori et al., *The Llama 3 Herd of Models*, arXiv:2407.21783, 2024.

[5] Andrew Howard et al., *Searching for MobileNetV3*, arXiv:1905.02244, 2019.