## Smart Retrieval Autonomous Car

**Members**:

Netanel Kuchma

Tal Harpaz

###### Or Amos



**About our project**

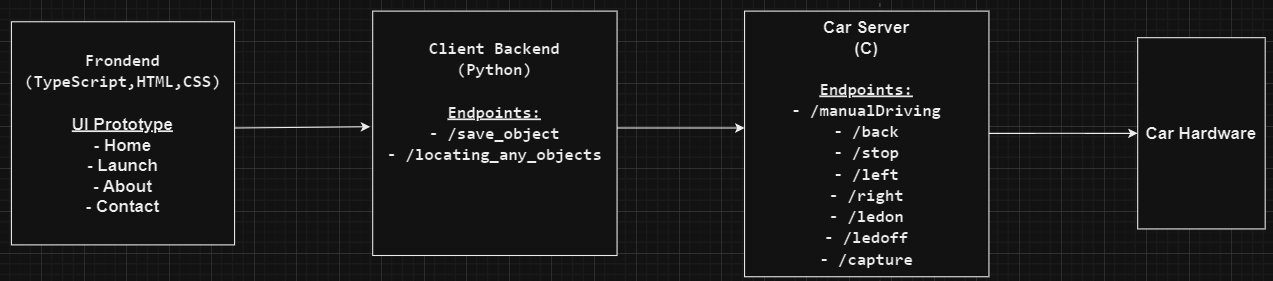
The project deals with the development of a Smart Retrieval Autonomous Car, designed for the efficient and swift retrieval of items within designated areas. This advanced vehicle integrates wireless WiFi remote control and a mobile camera for visual identification. The primary objective is to create an adaptable autonomous car that can autonomously navigate towards items, pick them up, and return them to their designated locations, guided by remote visual commands.

The Smart Retrieval Autonomous Car efficiently locates, retrieves, and delivers items within designated areas using advanced visual recognition and navigation capabilities. The process begins with the user accessing the interface, where they can choose an item from a predefined list, providing flexibility for various retrieval needs.

Once the user launches the operation, the car begins its mission, navigating the specified zone to identify the target item using its visual recognition capabilities. The car’s mobile camera analyzes the environment based on pre-programmed algorithms to locate the target item. Upon finding the item, the car securely attaches it using a magnet and autonomously navigates back to its starting point.

When the car returns, the user confirms the successful retrieval of the item through the interface, which provides real-time feedback on the mission’s accomplishment. After completing the mission, the Smart Retrieval Car is ready for another task, allowing the user to initiate a new retrieval mission and repeat the process for efficient and versatile item retrieval. This system offers a reliable and effective solution for tasks requiring accurate item identification, retrieval, and delivery within specified areas.

## **System Architecture Overview**



The system comprises a remote-controlled car equipped with an ESP32 camera module, a Flask web server for processing image data and controlling the car, and a YOLO-based object detection model for identifying and localizing objects within the captured images.

### System Overview and Implementation Strategy

Our project integrates several advanced technologies to create a robust, efficient, and scalable Smart Retrieval Autonomous Car. The system comprises multiple interconnected components, each playing a vital role in the overall functionality.

#### Data Model and Storage

The core data includes grayscale images stored locally on the server, which are used to match against query images. Annotated images with bounding boxes and centroids are also saved for visualization purposes, ensuring clear and precise feedback on item identification and retrieval.

#### User Interface Design

A user-friendly web interface is provided through HTML pages served by both Flask and ESP32 servers. This interface allows users to interact seamlessly with the car, initiate retrieval tasks, and view captured images in real-time.

#### Integration Strategy

The communication between Flask and ESP32 is handled via WiFi using HTTP requests. The YOLOv8 model, loaded on the Flask server, performs object detection, while OpenCV is utilized for image processing, manipulation, and annotation. This integration ensures that the system can effectively identify and locate items within the designated area.

#### Deployment Strategy

The deployment strategy involves setting up the Flask server, while the ESP32 car operates independently. Key steps include:

1. Deploying the Flask server with all dependencies.
2. Configuring the ESP32 car and connecting it to WiFi.
3. Starting both the Flask server and the ESP32 HTTP server.

#### Testing and Quality Assurance

Our testing methodology includes integration testing for endpoints. We use POSTMAN for endpoint testing to ensure correct request handling and response. The local development environment for Flask and a controlled test area for ESP32 facilitate thorough testing.

## Dependencies and External Services

* **YOLOv8 Model**: Loaded using an external library.
* **OpenCV**: For image processing.
* **Flask**: Web server framework for Python.
* **ESP32 Libraries**: Camera and WiFi libraries for ESP32.

## Component Breakdown

תמונה שמכילה טקסט, גלגל, צמיג, צילום מסך

התיאור נוצר באופן אוטומטי

### Front End Technology and Architecture

Chosen Front End Technology: React.js

React.js is chosen for the frontend development. React.js is a popular JavaScript library for building user interfaces, particularly single-page applications where we need a fast, interactive, and dynamic user experience.

Reasons for Choosing React.js:

1. **Component-Based Architecture:** React's component-based architecture makes it easier to build and maintain complex UIs by breaking them down into smaller, reusable components.
2. **Virtual DOM:** React's virtual DOM improves performance by minimizing direct manipulation of the actual DOM.
3. **Rich Ecosystem:** React has a large ecosystem of libraries and tools, including Redux for state management and React Router for routing.
4. **Community and Support:** Being one of the most popular frontend frameworks, React has extensive documentation, tutorials, and a large community for support.

Architecture: React Frontend with Flask Backend:

* **Frontend (React.js):**
  + **Components:** Modular UI components for different parts of the application (e.g., Navbar, Home, Test).
  + **Routing:** Managed by React Router to handle different routes in the application.
  + **State Management:** Using React's useState and useEffect hooks, and potentially Redux for more complex state management.
  + **HTTP Requests:** Axios or Fetch API to communicate with the Flask backend.
* **Backend (Flask):**
  + **API Endpoints:** RESTful API endpoints to handle requests from the frontend.
  + **Object Detection:** Using YOLO model to perform object detection on images captured by the ESP32 camera.
  + **Motion Control:** Endpoints to control the movement of the ESP32 camera car.

**Goal of the UI:** The primary goal of the user interface (UI) is to provide a seamless, intuitive, and engaging experience for users interacting with the Smart Retrieval Autonomous Car. The UI is designed to ensure that users can easily select items for retrieval, monitor the car's progress, and receive feedback on the completion of tasks.

**Considerations for the UI:**

1. **Comfortable Visualization:** The UI should present information in a clear and visually appealing manner, ensuring that users can easily understand and interact with the system.
2. **Clearness of the Interface:** The interface should be straightforward, avoiding clutter, and guiding users through the process of item selection and retrieval without confusion.
3. **Item Selection:** Users should be able to effortlessly choose the item they want the car to pick up from a predefined list. This selection process should be intuitive and efficient.
4. **User Engagement:** The interface should make users feel involved in the retrieval process. This can be achieved through real-time updates, visual feedback, and interactive elements that keep the user informed and engaged throughout the operation.

By considering these aspects, the UI will not only facilitate the efficient retrieval of items but also enhance the overall user experience, making the system user-friendly and effective for various retrieval needs.

**YOLO against SIFT**

**YOLO (You Only Look Once):** YOLO is a popular real-time object detection algorithm known for its speed and accuracy. Unlike traditional object detection methods that repurpose classifiers or localizers to perform detection, YOLO frames object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. It processes images in a single pass, making it highly efficient and capable of achieving high frame rates, which is essential for applications like real-time video processing.

**SIFT (Scale-Invariant Feature Transform):** SIFT is a feature detection algorithm used to identify and describe local features in images. It is particularly known for its robustness to changes in scale, rotation, and illumination. SIFT detects key points in an image and generates descriptors for these points, which can then be used to perform image matching and object recognition. This makes SIFT highly effective for tasks such as image stitching, 3D reconstruction, and object tracking where invariant feature detection is crucial.

At first we tried to use in our project the SIFT detector. Soon we realized that YOLO is designed better for real-time object detection, meaning it can process images very quickly and detect objects in near real-time. This is crucial for applications where timely detection of objects is required, such as in autonomous vehicles or surveillance systems.

In addition, YOLO is based on deep learning techniques, specifically convolutional neural networks (CNNs). Deep learning models like YOLO can learn rich feature representations directly from raw pixels, potentially capturing more complex patterns and variations in objects compared to handcrafted feature-based methods like SIFT.

Moreover, YOLO can learn rich feature representations directly from raw pixels, potentially capturing more complex patterns and variations in objects compared to handcrafted feature-based methods like SIFT.

Therefore at the end we chose the YOLO model.

### Flask Web Server

### ObjectLocalizer Class

This class is designed to load and manage a collection of reference images and detect objects within them using a YOLOv8 model.

* **Initialization (\_\_init\_\_ method)**:
  + Loads the YOLOv8 model.
  + Collects grayscale images from a specified directory (objects\_path).
  + Stores these images in a list (self.image\_collection) for later use in object detection.

### find\_and\_localize\_object Method

This method detects objects in a query image and attempts to match them with objects in the preloaded image collection.

* **Object Detection in Query Image**:
  + Uses the YOLO model to detect objects in the provided query image.
  + Initializes variables to track the best match.
* **Matching Process**:
  + Iterates through each image in the collection, detecting objects in each.
  + Calculates a match score based on the number of detected objects.
  + Updates the best match if the current image has a higher match score.
* **Localization of Objects**:
  + If a match is found, calculates the centroid of the detected objects in the query image.
  + Compares the centroid's position to the image's center to determine its location (left, right, middle).
  + Annotates and visualizes the detection by drawing bounding boxes, centroids, and lines on the query image.
  + Saves the annotated image to a specified path.

The method returns information about the best match, the distance of the centroid from the image center, and the location (left, right, middle). If no match is found, it indicates so in the response.

* **HTTP Endpoints**:
  + **/save\_object: Capture and save image using car's camera.**
  + **/locating\_any\_objects: Continuously capture images until an object is found.**

### 1. /save\_object

This endpoint captures an image from a car-mounted camera, processes it, and saves it as a grayscale image with a specified name.

* **Directory Creation**: Ensures the objects directory exists for saving images.
* **Parameter Validation**: Checks for the presence of car\_address and name parameters.
* **LED On**: Turns on the car's LED to improve image capture quality.
* **Image Capture**: Captures an image from the car camera and decodes it.
* **Image Processing**: Rotates, converts to grayscale, and crops the image.
* **Save Image**: Saves the processed image in the objects directory.
* **LED Off**: Turns off the car's LED after image capture.
* **Response**: Returns a success message with the saved image path.

### 2. /locating\_any\_objects

This endpoint continuously captures and processes images from a car-mounted camera to detect objects, returning the index of the best match found.

* **Directory Creation**: Ensures the images directory exists for saving captured images.
* **LED On**: Turns on the car's LED for better image quality.
* **Image Capture Loop**: Repeatedly captures and processes images until an object is detected.
* **Image Processing**: Rotates, converts to grayscale, and crops each captured image.
* **Object Detection**: Uses ObjectLocalizer to find objects in the processed images.
* **Camera Adjustment**: Moves the camera if no object is detected, to capture new angles.
* **LED Off**: Turns off the car's LED after an object is found.
* **Response**: Returns a success message with the index of the detected object.

### ESP32 Camera Car

* **Camera Module**: Captures images for processing.
* **Motor Controllers**: Controls car movement (forward, backward, left, right, stop).
* **HTTP Server**: Provides endpoints for manual control and camera actions.

### Car Server

### Endpoints Overview:

1. **/capture (capture\_handler):**
   * **Description:** This endpoint captures an image using the ESP32 camera.
   * **Response:** Returns the captured image in JPEG format.
2. **/manualDriving (go\_handler):**
   * **Description:** This endpoint controls the car's movements based on direction and delay parameters provided in the query string.
   * **Response:** Depending on the direction parameter, it moves the car forward, backward, left, or right for the specified duration, then stops. Returns "OK" if successful or an error message if parameters are missing or invalid.
3. **/back (back\_handler):**
   * **Description:** This endpoint moves the car backward.
   * **Response:** Returns "OK" after executing the backward movement.
4. **/stop (stop\_handler):**
   * **Description:** This endpoint stops the car.
   * **Response:** Returns "OK" after stopping the car.
5. **/left (left\_handler):**
   * **Description:** This endpoint captures three consecutive images using the ESP32 camera.
   * **Response:** Returns the last captured image in JPEG format.
6. **/right (right\_handler):**
   * **Description:** This endpoint moves the car to the right.
   * **Response:** Returns "OK" after executing the right movement.
7. **/ledon (ledon\_handler):**
   * **Description:** This endpoint turns on an LED.
   * **Response:** Returns "OK" after turning on the LED.
8. **/ledoff (ledoff\_handler):**
   * **Description:** This endpoint turns off an LED.
   * **Response:** Returns "OK" after turning off the LED.

### Detailed Explanation of the Camera Endpoint (/capture)

**Endpoint:** /capture

**Handler Function:** capture\_handler

**Purpose:** The /capture endpoint is designed to capture an image using the ESP32-CAM module and send it to the client in JPEG format. This endpoint is crucial for obtaining real-time visual data from the camera, which can be used for various applications such as monitoring, visual recognition, or navigation.

**How It Works:**

1. **Request Handling:**
   * When a request is made to the /capture endpoint, the capture\_handler function is invoked.
   * The function prints a message "capture request" to the serial monitor for debugging purposes.
2. **Image Capture:**
   * The function attempts to capture an image by calling esp\_camera\_fb\_get().
   * If the camera fails to capture an image, an HTTP 500 response is sent, indicating an internal server error.
3. **Response Headers:**
   * If the image is successfully captured, the function sets the response type to "image/jpeg".
   * It also sets the "Content-Disposition" header to inline with the filename "capture.jpg".
   * The "Access-Control-Allow-Origin" header is added to allow cross-origin requests.
4. **Image Encoding and Sending:**
   * If the captured image is already in JPEG format, the image buffer is sent directly as the response.
   * If the image is not in JPEG format, it is encoded into JPEG using the frame2jpg\_cb function and sent in chunks.
   * After sending the image, the frame buffer is returned using esp\_camera\_fb\_return(fb).
5. **Timing and Debugging:**
   * The function measures the time taken to capture and send the image using esp\_timer\_get\_time() for debugging and performance analysis.

### Frame Buffer Problem and Solution

**Problem:** During the project, we observed that the ESP32-CAM did not always send the current image frame. This was due to the frame buffer holding onto an outdated image. The default behavior of the ESP32-CAM is to buffer frames, which can result in sending an old image instead of the most recent one.

**Solution:** To address this issue, we implemented the following steps:

1. **Set Frame Buffer to 1:**
   * In the setup function, we set the frame buffer size to 1, ensuring that only one frame is buffered at any given time. This is done using the camera configuration settings.
   * Note: The frame buffer size cannot be set to 0, so there is always at least one frame in the buffer.
2. **Clearing the Frame Buffer:**
   * When capturing an image, we read the frame buffer twice. The first read operation clears the buffer, and the second read operation ensures that the buffer is filled with the most recent image.
   * This method forces the ESP32-CAM to refresh its frame buffer and capture the latest image.

**By implementing these steps, we ensured that the ESP32-CAM consistently captured and sent the most recent image, improving the reliability and accuracy of the visual data obtained from the camera.**

### **Detailed** **Explanation of manualDriving Endpoint**:

The manualDriving endpoint is designed to allow users to control the car's movements manually without needing to re-upload code to the Arduino each time. This provides a way to validate the car's driving functionality, test delays, and control the car through tools like POSTMAN.

**How It Works:**

1. **Initialization:**
   * The car is stopped initially using the stopCar() function.
2. **Query Parameters:**
   * It reads query parameters from the URL to determine the direction (dir) and the delay (delay).
   * The direction can be forward, backward, left, or right.
   * The delay specifies how long the car should move in the given direction.
3. **Direction Handling:**
   * Depending on the value of the dir parameter, the corresponding wheel actions are executed to move the car:
     + forward: Moves the car forward.
     + backward: Moves the car backward.
     + left: Turns the car left.
     + right: Turns the car right.
   * If the direction is invalid, an error message is returned.
4. **Movement Duration:**
   * The car moves for the specified delay duration (in milliseconds) and then stops.
5. **Response:**
   * If successful, the endpoint responds with "OK".
   * If there are missing or invalid parameters, an appropriate error message is returned.

**Usage with POSTMAN:**

* Users can test the car's movements by sending HTTP GET requests to the /manualDriving endpoint with appropriate query parameters (dir and delay).
* Example: To move the car forward for 2 seconds, send a request to http://<ESP32\_IP>/manualDriving?dir=forward&delay=2000.
* This approach allows for real-time testing and validation of the car's movement without reprogramming the device.

#### Future Enhancements

Future plans for the Smart Retrieval Autonomous Car include expanding its capabilities to support a wider variety of items and improving item retrieval accuracy. Additional enhancements will focus on scalability, allowing the system to handle larger-scale operations efficiently.