Forget-Me-Not

Giancarlo Zaniolo, Ethan Muchnik and Swati Anshu

Department of Electrical and Computer Engineering, Carnegie Mellon University

Forget-Me-Not seeks to help users track and find commonly misplaced items indoors. It accomplishes this using a Raspberry Pi with an attached camera to periodically record pictures of a room, run an ML object detection model on the image, and store the results such that they can be searched using a Web App, or a microphone voice assistant. We sought to achieve 80% object detection accuracy within a 10x10 ft room under well-lit conditions. Our testing showed that we achieved 75% accuracy onboard the Raspberry Pi (YOLO) and an 83% accuracy when running a larger model on the cloud (Grounding DINO) for predefined objects. We were successfully able to add new objects to the model as well.

***Index Terms*—Camera-based tracking, computer vision, object detection, object tracking, Raspberry Pi, real-time location, YOLO, machine learning, indoor environments**

I. Introduction

The "Forget-Me-Not" system addresses the common problem of misplacing everyday household items such as keys, wallets, and remote controls in indoor environments. The primary use case is for families, students, and forgetful individuals who often lose track of these items due to busy schedules or shared living spaces. Searching for lost objects can be frustrating and time-consuming, particularly when people are in a rush or trying to manage multiple tasks.

Existing solutions have several limitations. For example, Bluetooth trackers are often inconvenient because they require physical attachment to each item, and have a difficult time providing exact location data, all while being too expensive to use for all but a few items. Similar complaints can be said about GPS trackers or AirTags, which cost a starting $30 per tag. What sets Forget-Me-Not apart from thse other technologies is that it does not require attaching any physical tags to your items. Instead, it relies on advanced machine learning models, like YOLOv11 and Grounding DINO, to visually detect objects in real-time using a Raspberry Pi camera. Our approach is non-intrusive, works in the background, and leverages a secure web application or voice assistant for user interaction, making it both convenient and scalable.

By focusing on user-centric design and leveraging state-of-the-art AI models, our project aspires to provide a practical and reliable solution to a common daily problem.

II. Use-Case Requirements

We have determined that Forget-Me-Not must meet several of the following critical requirements to effectively meet the needs of our envisioned customers.

Our first requirement concerns the capabilities that our system should have. Firstly, we would like our system to be able to take photographs of the user's room at a rate of 1 image every 5 seconds. This would be necessary as we would believe giving the user the opportunity to see a photo of their room in a state where their object was last seen would be conducive to helping find an object.

Secondly, our system should provide the user the capability to query the system, both through a "voice assistant" frontend and a visual frontend. We believe that both are necessary, as the voice assistant would provide the user a means to query the system if they have lost the device they are looking for, and the visual frontend would assist in the case that the model is not completely accurate, or the object to be found was the second-to-last seen version of said object. Each of the frontends have unique constraints that help make the user interaction with our system more efficient.

The audio frontend, we think it would be reasonable for it to take 30 seconds between speaking a query and receiving a result. This time was chosen because in most cases, finding an object on your own should take more than 30 seconds. This statement is corroborated with calculations based on data found on the "Lostings Lost and Found Statistics" [1] webpage. This page claims that the average person spends 2.5 days per year searching for lost objects, and that the average person can lose up to 9 items per day. With some calculations, as shown below, we arrive at the very conservative estimate that

$$2.5\,\frac{days\ searching}{year} * \frac{1}{365\ days} * \frac{1}{24\ hours} * \frac{1}{60 min} * \frac{1}{9}\,objects\,\frac{lost}{day}$$

$$= 1.096\ min/object$$

1.096 min/object spent searching on a given day. If we could cut this number in half, it could save the user lots of time in the long run.

For the web frontend, we would like to ensure that the user receives the answer for a query in 10 seconds. This is because according to Uptrends, user attention suffers if a webpage is stuck loading for more than 10 seconds [2].

For both frontends, we would like to ensure that our system returns queries based on data that is at most 30 seconds old. According to the National Library of Medicine, short-term memory lasts for 30 seconds [3].

Regarding our model, we have determined that the system needs to achieve at least 80% accuracy in detecting and identifying predefined objects within a 10x10 ft room under

well-lit conditions (≥3000 lumens per square foot). We have chosen these numbers because we believe it is reasonable for our tool to miss 1/5 voice queries and still leave users satisfied, and because we believe the area and lighting constraints provide a reasonable environment for which it would be nontrivial for users to remember which objects were always present. Additionally, if the object is not initially found, we believe it is reasonable for the users to find the unsuccessfully queried objects by themselves. The intention of our system is to provide support in day-to-day life, and not necessarily be a tool to help forgetful dementia patients in life-or-death situations. As with any system, we cannot promise a 100% accuracy due to limitations of existing technologies.

For our objects, we are choosing to initially support finding phones, wallets, keys, as they are the most commonly lost objects according to the "Lostings Lost and Found Statistics" webpage [1]. We also intend to add pencils, pens and markers to the list of objects identify as a proof of concept of the fact that our search domain can be expanded based on user preference. Even more objects may be added in the future.

Additionally, we have chosen cost constraints which dictate that the hardware setup should not exceed $300, while cloud services for continuous usage should be kept under $40 per user per month. These numbers account for 3 hardware setups across a 2-bedroom house. We based these numbers off security.org's SimpliSafe, an existing product which charges between $250 and $730 for the hardware, and $32/month as a monthly subscription. As this is a successful product, we have reason to believe people will buy our product for similar prices [4].

Lastly, privacy is a priority, meaning that access to the system must be restricted to authorized users on authenticated local devices. These requirements are designed to create a system that is both accurate and cost-effective, while maintaining a focus on user privacy and efficiency.

III. Architecture and/or Principle of Operation

From a software standpoint, our high-level system architecture involves 4 components. An "image capture" python script, a "sound capture" python script, our webserver, and our "external compute" server. The "image capture" python script's job is to periodically capture pictures of the room, and send them to our webserver for processing. The "sound capture" python script's job is to continuously send microphone input to our webserver for processing. The webserver's job is to accept requests from our "sensor" scripts, do the necessary processing on the desired inputs, and to make the processed data available to the user through other HTTP endpoints. One way that this is done is by serving the that users use to interact with our data. Our "external compute" server exists because the Raspberry Pi's computing power is too limited for some of the tasks we expect of it (especially from a latency standpoint). As such, the "external compute" server provides http endpoints which perform the necessary high-compute tasks, and return the value back to the user.

Some of the logic used within the web server is itself interesting, so we will go into greater detail regarding them. In particular, we would like to give paranoid users the choice to run a working system without ever having to invoke the "external compute server". Our system can be configured to use "Local mode" by changing a simple setting. The exact semantics of what is done in "local mode" will be covered in the "System Implementation" section.

The hardware for our project primarily exists in the form of providing all of the necessary hardware functionality in a convenient package. As can be seen in Figure 4, we have a Raspberry pi case to hold the Raspberry Pi, a "camera direction" subsystem, which holds the camera in place, can be mounted to a wall or lamp post, and can be angled to point in whatever direction best shows the room the system will take images of.

When interacting with the system the users have two interfaces, a web interface and a voice interface. The website provides a user-friendly interface designed to help users efficiently interact with the Forget-Me-Not object-tracking system. Each page has been carefully structured to deliver a seamless and intuitive experience, ensuring users can quickly navigate through the platform's features. The first interaction begins with the Login Page, where users provide their credentials to access the system. For new users, the Create an Account Page enables them to register by entering their desired username and password. Validation ensures secure and accurate submissions.

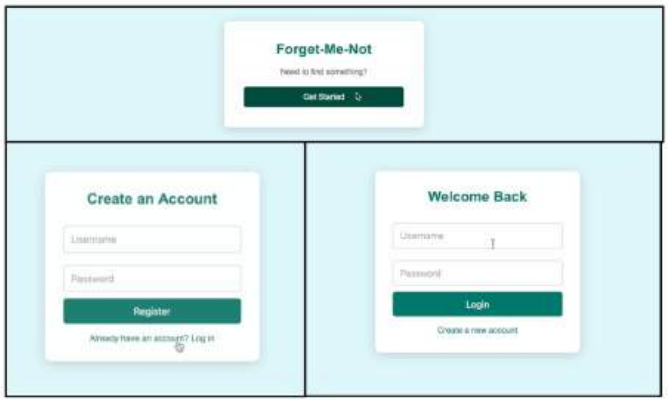


Fig 1: Home Page, Create an Account Page, Login Page

Once logged in, users land on the Homepage, which acts as the main control center. The homepage features a prominent search bar at the top, allowing users to type in the name of an object they wish to locate. Along the left panel, a categorized list of all tracked objects in the database is displayed, enabling users to select an item directly without needing to search. This organization ensures that frequently searched objects are always within easy reach. When a user initiates a search, the main screen dynamically updates to display the most recent image of the object's location, along with the time and date it was last detected. On the right panel, a scrollable history of previous sightings is displayed, showing thumbnail images and

timestamps of earlier detections. This allows users to trace the object's movements and explore its location history manually.

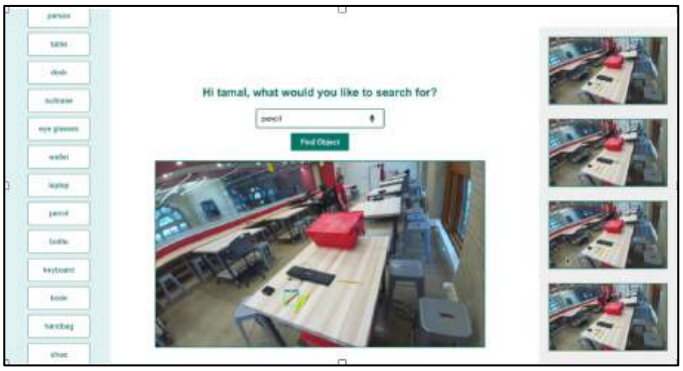


Fig 2: Main Page

For hands-free operation, users can switch to the Audio Interface by clicking the microphone button located prominently on the page. This opens a dedicated audio-query page featuring a sleek wave animation to visually represent the live audio input. Below the wave animation, the system displays a transcription of the user's query and the system's response, ensuring clarity in communication. The interface is designed to provide a modern, interactive feel while maintaining accessibility.

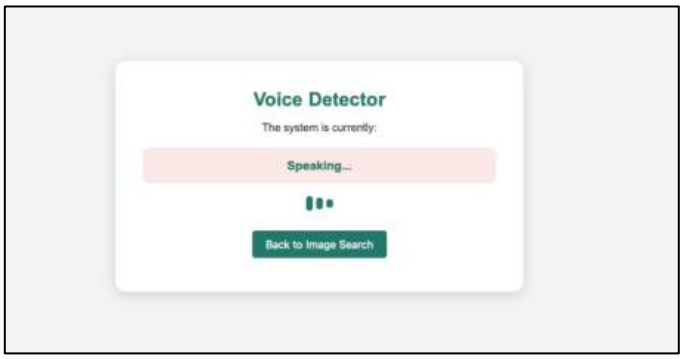


Fig 3: Audio Query Page

A consistent navigation bar at the top allows users to switch between pages easily, such as logging out or accessing account settings. The system's layout is logical and user-focused, with attention to detail that ensures all features are easily accessible.

The voice interface operates much like a home assistant. Our "sound capture" script uses the microphone to listen to its surroundings and send the data to our webserver. When the webserver converts the query to text, and hears the words, "Hey John, <User's question>", the webserver will run the user's question through our query processing pipeline. Once the query is finished processing, the user will hear a response, explaining the found object's location through our system's speakers.

There are a few significant differences between our current system architecture, and what we planned out for our design report. Primarily, we have abandoned the idea of hosting our service purely in the cloud. One of our main advantages is that we store all user information locally, and we have determined that having a cloud backend will require substantially higher development costs, to the point where we are no longer pursuing it.

Additionally, we have added an "external compute" server. As some of our planned functionality would have prohibitively high latency when executed on a raspberry pi, we decided to create a server which can be run on a more powerful computer, which can do more of the heavy lifting for us.

Our design process used key engineering principles that combines modular design and abstraction. The system was broken into smaller components, including image capture, processing, querying, and storage, ensuring that each could be developed, tested, and refined independently. We used the principle of efficiency optimization and balanced it with the need for privacy. These guided our design and implementation decisions, such as incorporating YOLO for lightweight object detection on edge devices and offloading computationally intensive Grounding DINO to a high-performance server. The integration of Docker containers allowed for portable and consistent deployment across devices. Furthermore, systems engineering practices, such as feedback loops (e.g., MSE threshold filtering to avoid redundant computations), were applied to optimize resource utilization. Finally, user-centric design principles shaped the web and voice interfaces to ensure intuitive interactions.

Scientific principles that were employed included concepts from computer vision, machine learning, and natural language processing. This includes those related to feature extraction and bounding box detection, to recognize objects in images, and probabilistic frameworks to process the images. Our project was grounded in mathematical principles, particularly linear algebra and probability. Linear algebra was critical for object detection models, where operations like matrix multiplications underpinned convolutional layers in YOLO and Grounding DINO. Probability and statistics were integral for training the ML models, calculating object detection confidence scores, and evaluating performance metrics like mAP (mean Average Precision). Additionally, cosine similarity, a core concept in vector mathematics, was employed for matching user queries with stored object names in the database. Thresholding based on Mean Squared Error (MSE) involved basic statistical measures to identify significant changes between consecutive frames.

Another addition to our system is a CAD model designed to be put up on a wall that has two pivot points that allow for greater freedom in the degree of motion. It also has the option of customizing it using different arm lengths to account for any further customizations needed.

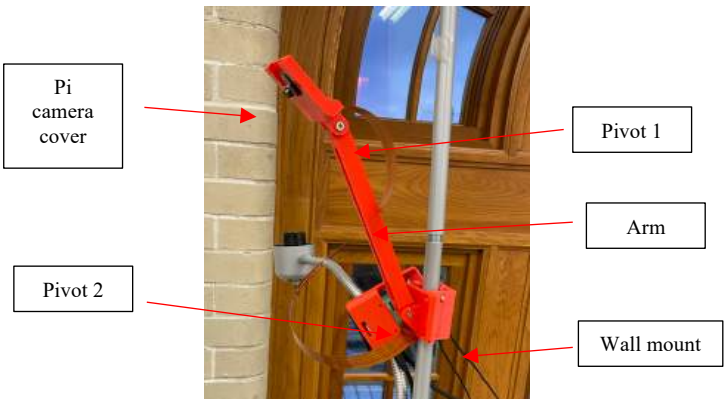


Fig 4. CAD print camera holder

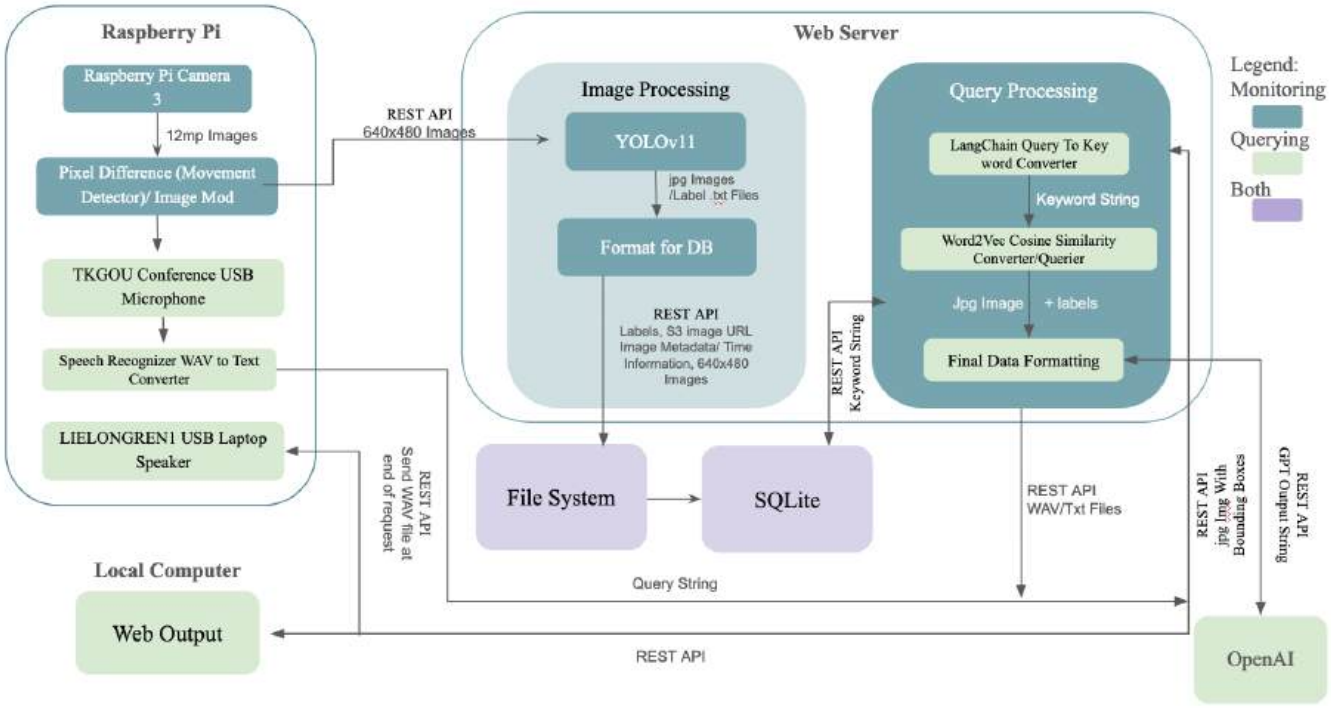


Fig 5: Original block diagram for implementation

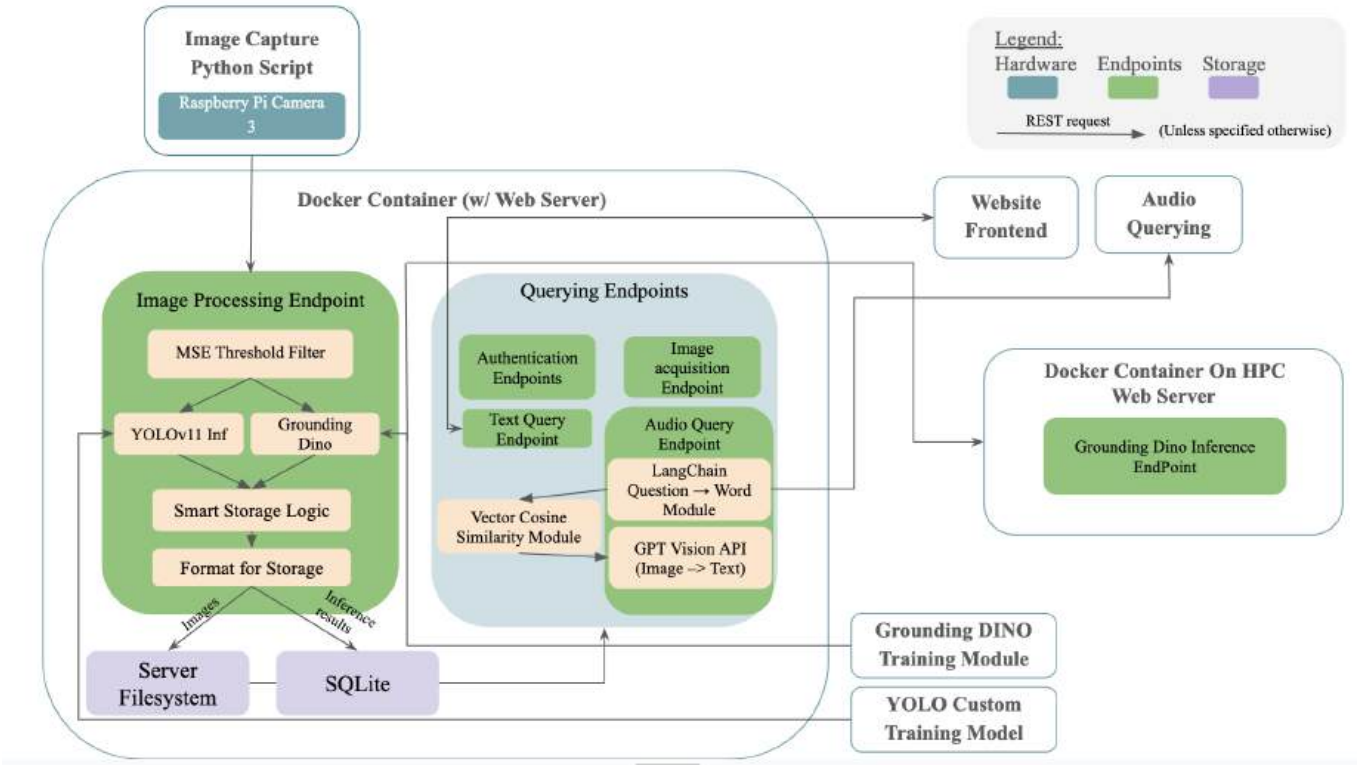


Fig 6: New block diagram for implementation

I. Design Requirements

The system's design requirements are primarily driven by the use case of tracking and identifying misplaced objects in indoor environments, and they can be categorized into Hardware latency, price, and privacy constraints.

A. Hardware Constraints

The system requires cameras with a resolution of 1080p to ensure that the objects are captured in sufficient detail for accurate detection. This resolution strikes a balance between providing clear images and managing data size for processing. The cameras need to capture images every 5 seconds to maintain up-to-date information on object locations without overburdening the system's processing capabilities. Notably, it took one of our group members 5 seconds to slowly walk across their room, meaning our system should reasonably capture any instance where an object is briefly placed, before being picked up and transported again. Additionally, each camera must have a field of view (FOV) of 40 degrees to ensure adequate coverage of the room. While this FOV might seem narrow, it helps the system focus on specific areas and reduces image distortion from the fisheye effect, optimizing object detection.

B. Latency Constraints

To ensure real-time operation, the system must meet specific latency targets. For the "Monitor" workflow, which involves capturing and processing images, preprocessing the image data must occur within 1 second, while object detection through machine learning models should take 5 seconds. The webserver "ping" latency should be within 1 second, and the entire process, including database writing and cross-component latencies, should be complete within 13 seconds, ensuring that the system remains responsive.

We chose these values because they seemed like reasonable targets for each of our subsystems. Ultimately, what matters is the performance of our whole pipeline, since that is what is ultilately observable by the end user. These requirements serve more as general guidelines than necessarily hard and fast rules. Even then, it is still useful to compartmentalize latency goals, to give individuals working on the project a more concrete target to aim for. Our object detection latency will realistically take the most time out of our full pipeline. Given that the inference frameworks we have easy access to are largely single-core, and we want to be able to take a picture every 5 seconds, it seems reasonable to require that object detection takes 5 seconds. This generates an additional implicit hardware requirement where given that a 1080p image has 2 million pixels, and an estimate that we should be using ~500 instructions per pixel of processing power on the original image, our processor should be able to support 2MFLOPS.

Regarding our database latency, our end querying from the user will require at least a 2 calls from the database, one to get the URLs associated with an image, and a second to retrieve the image (regardlesss of architecture). Given our desired end-to-end time of 10 seconds 5 seconds seemed like a good choice. Additionally, by pinging google.com, we can assume that a relatively "normal" ping is somewhere around 20-100ms, which should fit within our 1 second margin unless there are issues with the system.

For the "Query" workflow, which is responsible for providing users with information about the location of objects, speech-to-text processing should take no more than 10 seconds (it will also occupy a large portion of our runtime), followed by the webserver ping latency, which should take less than 1 second. Next, we list that the database read latency should require less than 5 seconds, leaving 10 seconds for an additional processing we would like to do using LLMs. The total latency for a query response should not exceed 30 seconds, allowing users to quickly find misplaced items. The rationale for this was discussed above.

C. Privacy

One of our system's main selling points is that users can run a self-contained system without needing to share private information with the outside world. As such, we require that at the very least, our users are able to run some version of the system (minimum object detection, one querying pipeline) completely locally. If anything is not run locally, we should guarantee that no private image is ever persistently stored off of the Raspberry Pi.

II. Design Trade Studies

A. SQLite3 database selection

Multiple database types were tested to determine the type of database we would use to maximize performance. Even though this wasn't the bottleneck of our system, it helps us understand and justify the use of the SQLite database for high frequency writes and updates.

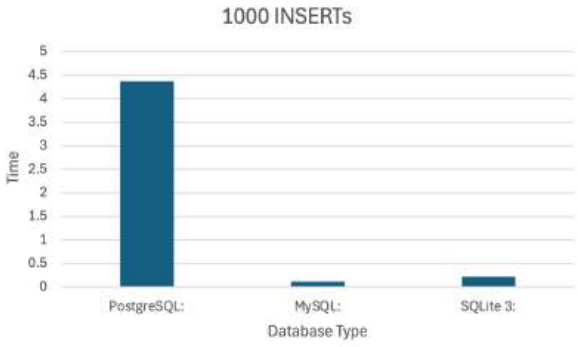


Fig. 7 1000 INSERT operations across the different database types

For 1000 inserts, SQLite 3 performs better than PostgreSQL (0.223s vs. 4.373s) but is slightly slower than MySQL (0.114s). SQLite still provides consistent and reliable performance.

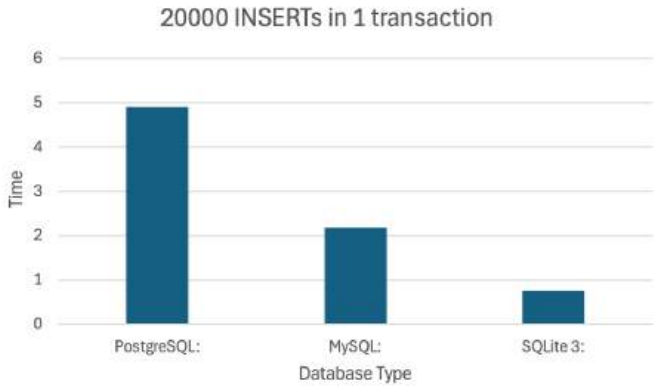


Fig. 8 2000 INSERT operations done in 1 transaction across the different database types

For 20,000 inserts in one transaction, SQLite 3 significantly outperforms PostgreSQL (0.757s vs. 4.900s) and is only marginally slower than MySQL (2.184s). This demonstrates SQLite's efficiency in handling batch operations when transactions are optimized.

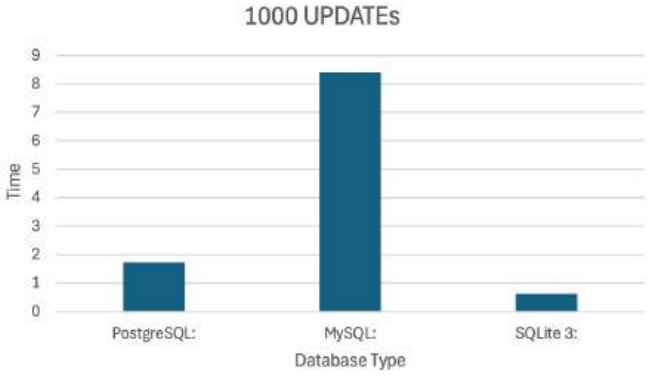


Fig. 9 1000 UPDATE operations across the different database types

SQLite 3 outperforms the other database types 1000 updates, completing the operation in just 0.638 seconds compared to PostgreSQL and MySQL, which take 1.739s and 8.410s, respectively. This makes SQLite a strong candidate for applications requiring frequent modifications to the data.

Based on this preliminary testing, along with the fact that SQLite operates without the need for a dedicated server process. This is ideal for resource-constrained environments like the Raspberry Pi, which has limited computational power and memory if we were to store it on the Pi. Since it is serverless, it requires no additional setup or maintenance unlike the other databases. This aligns perfectly with the project's design goals of simplicity and minimal overhead. For a system storing images' metadata, the data volume is likely not enormous. SQLite is optimized for smaller-scale databases and offers quick access without the complexity of managing a full-fledged database system. SQLite 3 was chosen for its lightweight, serverless architecture and robust performance in operations critical to the system, such as updates, deletes, and efficient batch transactions.

B. Model Selection

Apart from having to select the model's name YOLO/Grounding DINO etc, there also had to be testing with various different parameters. These parameters included but were not limited to amount of augmentations for each training image, epochs, and the size of training dataset. Thus both Grounding DINO and YOLO were trained/evaluated in a couple dozen different ways with custom built Training/Validation scripts. In order to validate the pros and cons of every model there were 2 main metrics used: MAP-50 and time to run. The former essentially takes care of assessing how good accuracy is while time just measures how long it takes to run each model. For the data below, we will show the direct comparisons for the models running on the GPU and Raspberry Pi.

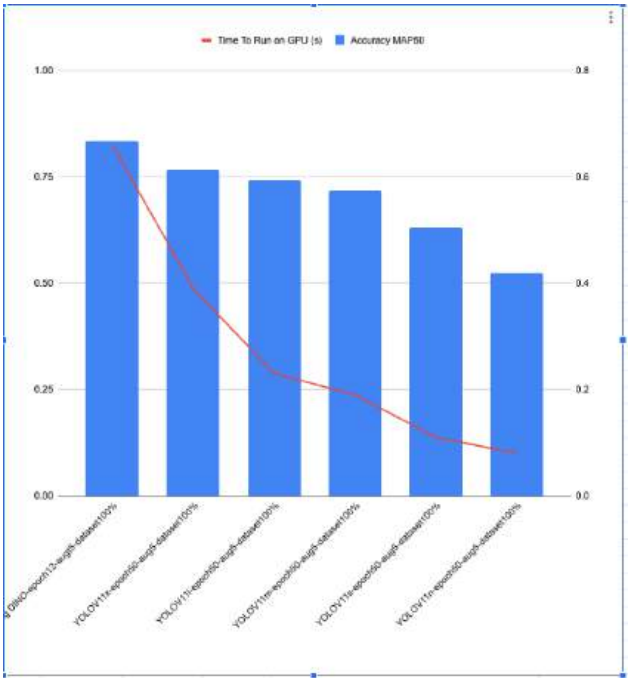


Fig. 10 GPU Metrics For MAP-50 vs Time for inference

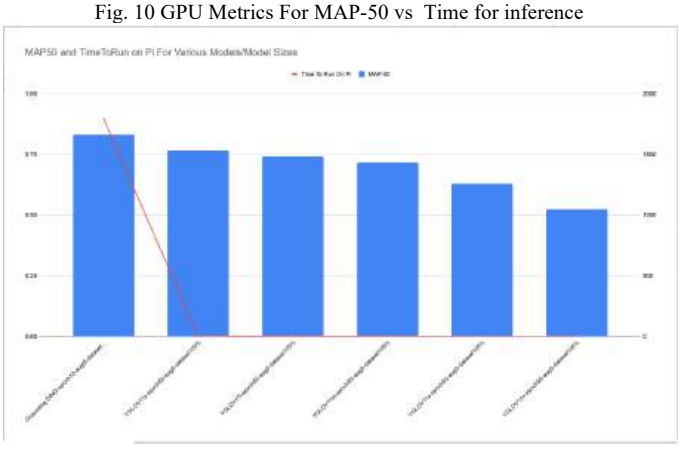
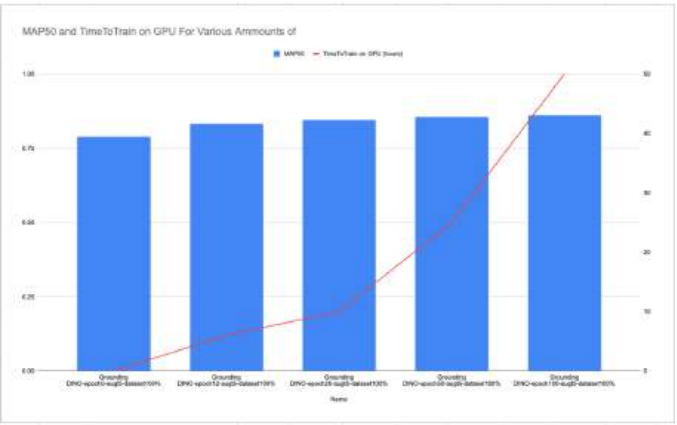


Fig. 11 Rasp Pi Metrics For MAP-50 vs Time for inference

So as one can see the larger and more accurate the model, the slower it is to run. Through the above charts we were able to see just how much slower. While larger YOLO models did indeed slow down significantly there were always at least

running at a speed within the same order of magnitude both on the GPU and Rasp Pi. GIven that there was not much to be gained from yolov11x vs l, it was decided to use yolov11l. Meanwhile the Grounding DINO model couldn't run on the Pi taking over 30 minutes which would make it impossible to fulfill the design requirement of image processing in 25 seconds. However, it could run the GPU and while it was a lot slower than all the YOLO models it was still fast enough that using it could be justified with it seeing almost 0.1 improvement over the next closest model in MAP50.



In addition to inference speed being important, training time was too. With a larger number of epochs, the training time scaled linearly with epoch growth while the performance of the model started levelling off. Thus, it was chosen that the breadth of exploration of various training parameters were more important than the incremental gains from each additional epoch. Thus, training for grounding DINO was stopped at 12 epochs. Similar tradeoffs were done with each parameter for training and from that a rough approximation of the optimal model for both YOLO and Grounding DINO was obtained.

III. System Implementation

A. Webserver

As every software component interacts with the webserver in some way, it will be useful to first cover how it was implemented.

The webserver was written in Python using Flask as its framework. Flask's primary capability is that it allows us to specify python functions to behave as "HTTP endpoints". In other words, when Flask receives a HTTP request, it sets up a global context with the contents of the request, and invokes the Python function which corresponds to the URL for that request.

Using Flask's built-in runtime by itself is not enough to guarantee properties that our system desires. As discussed previously, our webserver needs to have the ability to run both low-latency (website login, querying) tasks, and much longer-running tasks (ML Obj. Det. Inference). Given that Flask's default runtime is single threaded, short-running requests would get blocked behind long-running ones, unnecessarily increasing their latency. As such, we chose to run our Flask code with gunicorn. Gunicorn is a high-performance webserver that shares the Flask runtime's ability to interact with Flask python code, but has the added ability to start up multiple worker processes. As such, so long as our webserver has at least 1 more worker process than there are long-running tasks, short-length tasks should be able to run without needless latency spikes.

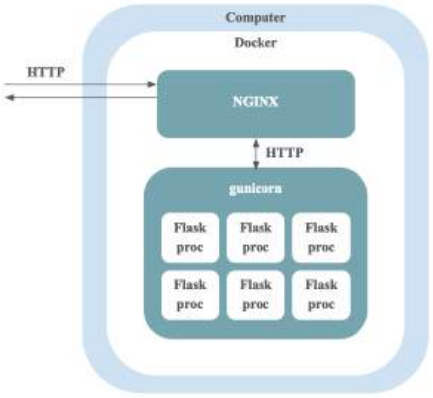


Fig X: Here is a diagram of how our webserver is laid out. Currently, the weberver is set up to run on the Raspberry Pi.

Lastly, NGINX is in charge of directing HTTP requests to gunicorn. NGINX is a second webserver capable of load-balancing requests between other servers. Our original idea was that using NGINX would allow us to balance requests between multiple gunicorn instances, allowing our previously proposed cloud implementation to be scalable. We no longer use this functionality, but as NGINX allows users to set up firewalls, we kept it in case we had time to dedicate towards additional security.

Throughout the development of the webserver, most things went relatively straightforwardly (Though it certainly was time-consuming). However, there were a few pivotal moments where we decided "change directions".

The first time was when we decided to pivot away from supporting a cloud backend and focus on our Raspberry Pi implementation. This decision arose from the fact that there seemed to be many concerns regarding the privacy of user data on the cloud. When considering the additional development cost involved with having privacy, scalability, and a cloud backend, all while still needing a hardware camera to record all of the necessary data, we decided to abandon the idea.

The second time came when after our interim demo, when the reviewers were thoroughly unimpressed with the performance of our local object detection model in a few specific scenarios. Because of this, we realized the importance of having as good of an object detection model as possible. At this point a group member had already been investigating the feasibility of a model much more advanced than YOLOv11, and had found that it was too large to be easily run on the Raspberry pi. As such, we changed our system architecture such that we could call a separate webserver running on a much more powerful machine to do our high-compute tasks.

B. Image Processing Pipeline

The image processing pipeline's primary task is to capture, process, and store information about an individual's room in a way that can be easily queried. The simplest way to describe how this subsystem works is to trace through what happens at each step.

The first part of this pipeline involves the "image capture" Python script. This script periodically captures an image using a camera, and sends it to the webserver through an authenticated HTTP message to an endpoint. In our presentation, this script was running on the Pi alongside the webserver, but as the two communicate over HTTP, this script could be running on any device with a camera. If the used libraries are not supported on the alternate device, it should be very easy to write a new camera module which can be imported by the existing script.

Once the image is sent, to the webserver, it performs a simple Mean-Squared-Error comparison against the previously sent image. If the error is below a certain threshold, the image is deleted, and a response is sent back to the script. This saves computation power, by ensuring that we only run ML object detection on relevant images. Presumably, we can say that if there is no difference between two images, not many objects have moved.

If it is sufficiently different, ML Object Detection is run on the image. If the system is in "local mode", YOLOv11 ran locally on the Pi. If the system is not in "local mode", the image is sent to the "extra compute" server over a HTTP request, where GroundingDino is run on the image, and its bounding boxes are returned. Both YOLOv11 and GroundingDino are only capable of detecting a closed set of objects.

After this, a heuristic, which quantifies how many new objects were seen in an image (compared to the previous image), is calculated. (In this heuristic, an object is considered the same as another object if their label is the same, and their bounding boxes are sufficiently close.) If a user has filled up their allocated storage, this heuristic is used to delete the image with the least new objects seen, with age serving as a tiebreaker. The image adjacent to the deleted image then recalculates its heuristic value.

Finally, the captured image, along with all of its bounding boxes, and heuristic value, are stored in an SQLite database.

Getting all of the pieces for this subsystem assembled was somewhat straightforward; It did not take much effort to just get the system working. However, there was additional testing done to optimize this subsystem, as the ML object detection is one of, if not the most expensive part of our system. We ran tests to find the largest model we could run on the Pi given our time constraints, tried different inference backends to see if any were faster, and explored various other options. What we ultimately learned was just how expensive ML really is, and that one of the best ways to save on CPU cycles is to reduce the amount of ML you really need to do.

C. Audio Processing Pipeline

The main purpose of the audio processing pipeline is to constantly listen for questions from the user, and to answer with the predicted location of the object when asked. It can also be effectively described by going through the steps it takes.

This pipeline begins with the "Audio processing" script. This script sets up the microphone to always be recording data, and performs a simple loop, where it sends captured microphone bytes to a webserver HTTP endpoint, and sends more once the HTTP request returns.

The webserver then sends the bytes to the "extra compute" server so that a OpenAI Whisper, a speech-to-text model can be run on it.

Once the webserver receives the translation of the speech bytes, it checks to see if the keywords, "Hey John" are present in the string. If they are, the webserver then uses LangChain, an LLM prompting framework, with a series of prompts with the goal of isolating the singular word referring to the object you are looking to find from the full query. As an examply, if you asked, "Where are my sunglasses?", this step would return the string, "sunglasses" This word may not necessarily be a word that is present in our database, so our webserver then converts this word into a Word2Vec embedding using the HuggingFace transformers library. Next, this embedding is compared to the embeddings of the closed set of objects previously detected, and in our database. The existing object with the closest embedding (calculated using cosine vector similarity) is chosen. As another example, "sunglasses" may not be a part of our closed set of detected objects, whereas "eyeglasses" are. Even though they may not be the same word, our ML object detection model would still detect sunglasses and categorize them as "eyeglasses", and this step would allow us to find the desired object in our database.

After this step, the webserver would find the most recent appearance of this object in our database, and submit the image (containing the bounding box) along with a prompt requesting relational information to GPT vision. GPT vision would return a sentence about where the desired object is located in relation to the objects around it. Finally, this sentence will be read out loud through Forget-Me-Not's speakers using the macSpeaker text-to-speech python library.

D. Web Interface

The web interface allows users to interact with the object tracking system. It is designed to be intuitive and efficient, enabling both manual and voice-based interactions. The interface seamlessly connects to the backend infrastructure for querying, displaying object locations, and managing user accounts.

The pipeline begins with the user's interaction on the frontend. If the user submits a text-based query through the search bar, the input is sent via an API call to the querying endpoint. For voice-based queries, the pipeline involves capturing audio input, transcribing it into text, and extracting the object of interest. Once the query reaches backend, the webserver processes the query by parsing the text or the audio inputs. The object name is matched against existing database entries using vector cosine similarity to find the closest match. The most recent image containing the matched object is retrieved. The system also fetches a history of images to provide a visual timeline. For the audio query, it displays the transcribed text.

The backend supports multiple RESTful endpoints, each catering to specific functionality:
Authentication Endpoint: Handle user login, logout, and account creation. These endpoints validate input data, communicate with the database, and return success or failure responses.
Image Query Endpoints: Allow users to search for specific objects. When a query is made, the backend processes the request and retrieves relevant image data from the database.
Audio Query Endpoint: Captures and processes voice commands, leveraging machine learning models for speech-to-text conversion and object detection queries.
Image Acquisition Endpoint: Manages incoming images from the Raspberry Pi, validates them, and stores processed metadata in the database.

Authentication in the system is handled using JSON Web Tokens (JWT) for secure, stateless communication. During login, users submit their credentials, which are verified against hashed values in the SQLite database. Upon successful authentication, the server generates a signed JWT containing the user's identifier and session metadata. This token is sent to the client and included in subsequent requests via the Authorization header. The server validates the token's signature and expiry before processing requests, ensuring only authenticated users can access protected endpoints.

E. Hardware

The physical design of the Raspberry Pi and camera holder required a detailed and iterative approach to meet the functional requirements of the project. The holder needed to be wall-mounted with an adjustable angle to maximize the field of view for object detection while maintaining stability. The design also had to account for the compact size of the Raspberry Pi and camera module, ensuring it could house and protect the hardware components without obstructing functionality or airflow. To achieve adjustability, I initially designed a ball-and-socket joint mechanism. This mechanism allowed for a full range of motion and precise camera positioning, which seemed ideal for flexible installations. Using CAD software, I created multiple models, each with different socket tolerances and ball dimensions to ensure smooth movement while preventing slippage. However, after testing, this design failed due to strain-induced deformations in the joint material under the weight of the camera and Pi. Specifically, the plastic used in 3D printing deformed at the socket under prolonged stress, reducing adjustability over time. Simulations run on the CAD models highlighted the stress concentrations at the socket due to the uneven distribution of the camera's weight. Additionally, during real-world testing, the joint loosened over time, failing to hold the camera steady. This was particularly problematic because even slight vibrations or shifts in the camera angle could significantly affect the object detection system's performance. To address these issues, during the design phase, I used CAD simulations to analyze the performance of the hinge-based setup. Finite Element Analysis (FEA) was employed to simulate the stress distribution on the hinge and mounting bracket under the combined weight of the Raspberry Pi and camera. This revealed that the critical stress points occurred near the screw joints and along the hinge pivots. To address this, I increased the hinge thickness and selected screws with a larger thread diameter to distribute the load more evenly. I also simulated repeated angle adjustments to ensure the system could withstand regular use without material wear. These simulations demonstrated that the hinge design was far more robust than the ball-and-socket joint, with minimal deformation under load.

The modularity of the design also made it easy to mount and dismount, providing flexibility for customization with different arm lengths and camera placements. This hinge-based solution was a significant improvement over the initial ball-and-socket design, addressing stability issues while maintaining the adaptability needed for the project.

F. ML Training subsystem

While in previous sections it was taken for granted that the models work as expected and categorize relevant objects there was a lot of work that went into the training infrastructure. The reason we could not just use the default models is because said models were not optimized for the types of objects we wanted to detect. For example, most of the data the models were trained on was on outdoor pictures which our model would not have to really deal with as an at home solution. And with every extraneous class that the model had to detect the overall accuracy went down forcing us to investigate training.

Because there were two models – Grounding DINO and YOLO – there were two separate frameworks created for training said models and evaluating them.

For YOLO there was an extensive custom-built framework which allowed one to train any model with various amounts of epochs, batch size, parameters. In addition, functionality was added in one could add custom data that was augmented to the training/validation/test datasets. On top of the training script in the framework there was also a validation script which could use the exact same datasets specified for the training data with the typical train/validation/test breakdown to validate the model. That ensured training data was not being used to test the model. So, when both the training/validation scripts were run relevant metrics were outputted into a specific folder which contained each of the dozens of values for the parameters in its title so one could know which training runs were conducive to the best results.

For grounding DINO there was less work to do implementing a training framework as there was already an opensource training/testing framework called mmdetection. Through trial and error with various prompts for the object class sets and epochs we were able to satisfactorily train Grounding DINO to primarily focus on the objects were most interested in.

IV. Test, Verification and Validation

Note: ML testing results are probably more important and should go first

A. Object Detection Model Performance

Table I: Model Accuracy

	Req	MAP-50	MAP-50-95	Pass
YOLOv11m-e20	0.8	0.669	0.530	No
YOLOv11l-e50	0.8	0.745	0.598	No
Grounding DINO-e12	0.8	0.833	0.642	Yes

The following measurements were meant to assess just how accurate the underlying models are. The number next to the metric means the amount of overlap with the ground truth needed to classify the labelling as a success. Thus MAP-50 counts something as a success when the overlap is 50% of total are of both boxes while MAP50-95 takes metrics at intervals of 5 from 50-95 and averages the scores. We opted to go with MAP-50 because we don't actually care about the bounding box being perfect, just that it identifies the object correct and can roughly point our query answerer in the right direction of where that object may be.

Given the requirement was 0.8 Accuracy, only the Grounding DINO model truly passed. And from the tradeoff chart it was determined that the speed drawbacks of the Grounding DINO model were not too extreme if inference was run on the GPU.

B. System Latency Measurements

This section directly relates to our timing requirements in our use-case and design requirements sections. Getting results for these sections was relatively straightforward; All we had to do is run the necessary workflows and time how long they took.

Here are our results:

Table II: Image Processing Pipeline

Event	Requirement	Measured	Pass?
Camera Preprocessing	1s	0.0253s	Pass
Webserver Overhead	1s	8.00*10^-7s	Pass
Object Detection (YOLO – on RPI)	5s	3.35s	Pass
Object Detection (DINO – On Desktop)	5s	0.66	Pass
Database Write	5s	0.12s	Pass
Total	13s	3.54s	Pass

Table III: Website Query Pipeline

Event	Requirement	Measured	Pass?
Serve Webpage	1s	0.00297s	Pass
Text query roundtrip	1s	0.0007954s	Pass
Serve photo	1s	0.0079801s	Pass
Total	3s	0.0110s	Pass

Note: These times were recorded from the perspective of the server. It is difficult to time how long your browser takes to load images, etc.

Table IV: Voice Query Pipeline

Event	Requirement	Measured	Pass?
Speech-To-Text model	10s	1.53sec	Pass
Web Server Overhead	1s	8.00*10^-7s	Pass
Database Read	5s	0.00254s	Pass
Additional ML	10s	6.53s	Pass
Total	30s	8.06254	Pass

From what can be seen in this section, all our components pass the outlined latency requirements. If any values appear to be different from what was observed at the demo, it is likely because they were taken in more ideal conditions – the Raspberry Pi may have been less hot, the server may have been running for less time, and there may have been more background noise. In particular, we believe there may have been a bottleneck could be fixed with a little bit of extra debugging, but we couldn't get around to it by the time of the presentation.

C. Cost Calculation

This section relates to our cost requirements.

Hardware Costs:

The hardware costs for our project can be estimated using our bill of materials.

For a system with 1 Raspberry Pi and its affiliated hardware, the costs are as follows:

Raspberry Pi 5: $80.00
Raspberry Pi Camera Module: $35.00
Camera Module Connector Wire: $8.00
3d Printer Filament: ~$5.00
USB Microphone: $20.00
USB Speaker: $16.00
Velcro Strips: $9.00
= $173.00

These costs fall below our initially required hardware cost quota of $300.

Additionally, if the user would like to add an extra camera to an already existing system, the cost will be substantially cheaper than our calculated $173. Given that our entire system is build using materials which facilitate development, we believe we can drastically cut the total cost of our product in the future.

Ongoing Costs (Cloud):

This section will show some rudimentary calculations which estimate the cost per-user of using the "extra compute" server.

In our current design, we have 2 workloads running on the "extra compute"; The ML object detection model from the image processing pipeline, and the speech recognition from the audio query pipeline.

We are choosing not to include the speech recognition workload in this pipeline. This is because once we optimize our design, we do not believe it will use much GPU time. While running our webserver, we have noticed that CPU utilization seems to be generally low. As a result, we believe that it should not be difficult to create a small, simple model that always runs on the Pi, whose only job is to recognize the model's keywords. With such a model, we would only have to run the large, expensive speech-to-text model when we run a query, and since users are probably not submitting queries all the time, the GPU cost of this workload will be negligible compared to the time spent doing ML object detection.

To calculate the cost of ML object detection per user with GroundingDino, we rented an 'NV6ads A10 v5' instance on Azure, and measured the runtime of GroundingDino while running on the VM.

Cost of VM/Month (pay as you go): $331.42

First, we calculate how many inferences we can do in a month.

GroundingDino Inference Latency: 0.656s

(1 / 0.656 s/inf) * 60s/m * 60m/hr * 24hr/day * 30day/mo
=3985800 inferences/month

Next, we calculate how many inferences a client is expected to make in a month. Based on our own tests, in an empty room, (where nothing can move, as there is no one in the room to move anything) MSE thresholding was able to prevent ML from running on 207/209 images, or 99.04% of images. Additionally, as an estimate, we imagine the average person might spend 8 hours moving around in their room, the rest being spent outdoors or sleeping. For this test, we will be making the conservative assumption that we must do ML object detection on every image where a person is in their room, even though MSE should be able to filter some out. Remember that the cameras take an image every 5 seconds by default.

0.2 imgs/sec * 1/3 of day spent in room + .0096 * (2/3 of day spent in room) = 0.0730 imgs/sec on average

0.0730 imgs/sec * 60s/m * 60m/hr * 24hr/day * 30day/mo =
= 189388 inferences/mo

189388 inferences/mo / 3985800 inferences/mo =
4.75% of GPU time taken by the average user per month

$331.42 * 0.0475 = $14.26 per user per month

This value falls far below our goal of $40 per user per month.

V. Project Management

A. Schedule

The schedule is split up according to individual responsibilities with slack time of an average of 2 days included in the timeline. The largest time allocation is for system integration, with slack time of a week built-in. As seen in Table X attached in the appendix. The green bars show the actual time allotted for the work described in the first column whereas the red bars display the buffer time of having tit completely done. There were no significant changes in the schedule from the one in the design report. The 3 of us were able to stick to our schedules and used the buffer time if it was necessary. We worked with each other and played off each other's strengths and knowledge to help speed up the work.

B. Team Member Responsibilities

Ethan's primary responsibility in the project is the machine learning component, where his focus was on optimizing the model for accuracy, performance, and generalization. He worked on improving the detection algorithms, and to train and integrate Grounding DINO with the system, to ensure that the system can identify objects in various indoor environments with high precision. Ethan also put in significant amount of work to integrate LangChain, vector embeddings and GPT Vision API calls to finalize and implement the audio pipeline. He also played a secondary role in optimizing performance across the overall system, ensuring that both the web server and database integrate efficiently with the machine learning model. Lastl

Swati worked on setting up the database and hardware configurations along with the CAD models, which are critical for the system's backend infrastructure. She ensured that data related to detected objects is stored correctly and accessed efficiently. In addition to database management, she also worked on the audio processing pipeline instead of the preprocessing optimization, by establishing a rudimentary pipeline to transcribe, call to GPT Vision and output the relational information. She also created the frontend UI and endpoints to integrate the backend and frontend. Her secondary responsibility involved collaborating with Ethan and Giancarlo to troubleshoot any performance issues that arise in the database or hardware setup, ensuring smooth communication across all system subsystems.

Giancarlo was primarily responsible for the web server structure, setup, and instrumentation. His role ensures that the server can handle incoming data and process requests efficiently, providing users with quick access to object detection results. He also worked closely with Ethan to integrate the machine learning model into the web server and with Swati to ensure smooth data flow from the database to the user interface. This coordination allowed the system to operate seamlessly, balancing the needs of each subsystem. He also worked on creating a second inference server for Grounding DINO integration and execution. His secondary responsibility involved performance testing and latency testing in order to determine the necessary optimizations.

C. Bill of Materials and Budget

Table X shows the detailed breakdown of parts that we require for our system and the estimated total cost for its development. We did not use the Jetson Nano due to resource constraints. Only addition is the Raspberry Pi 5 camera cable since the existing samples weren't compatible.

D. Risk Management

Critical risks for this project included the fact that none of us have worked with databases before so figuring out the correct configuration and the integration, post-setup will be a challenge. This just required time and communication to figure out. Swati and Giancarlo coordinated to ensure the database specifications matched the webserver and ML model interfaces. Secondly, the amount of data needed to train the ML model is substantially large and there are limited online datasets for the indoor images we need – hence we created our own bounding boxes for around 1000 images to train the model further. Ethan and Swati worked on this easy, but time-consuming task. There was no significant risk other than time commitment which was managed by spreading this task over 2 weeks. Lastly, another critical risk was the final integration of the entire system, as none of us have worked on creating interfaces for multiple different systems and different datatypes. This involved extra research and debugging. Giancarlo spent a lot of time ensuring that the interfaces match and also created dummy inputs and outputs to test his systems before the actual ones were created to ensure faster and smoother integration.

Hence, our primary risk was training the ML model for our system to meet the design specification. These risks were mitigated by allotting time for just for the creation of the training dataset manually and using annotation tools to speed up the process. Moreover, Ethan spent a lot of time researching on other alternatives and found Grounding DINO as a solution that would ensure that it meet our design specifications. Other than that, we approached the integration risks modularly and debugged each subsystem and interface which helped minimize complications. Lastly, we figured out how to use services we hadn't used before by learning on the spot, understanding the demands of our system and implementing the necessary steps efficiently. Furthermore, dividing the team responsibilities, with each member focusing on mastering a particular service or software ensured a smoother integration process.

VI. Ethical Issues

Our product raises several ethical concerns, primarily regarding privacy, security, and equitable access. Our system relies on capturing and analyzing images within personal spaces, which may lead to the surveillance of users and visitors. Users might be unaware of the system's operation, or stored data could be vulnerable to unauthorized access. These risks could adversely affect individuals' privacy, leading to potential misuse of sensitive information or even exploitation in malicious scenarios.

To mitigate these concerns, we have implemented multiple safeguards. All captured images are stored locally on the device, minimizing exposure to third-party access or centralized data breaches. Additionally, user authentication is managed through JWT (JSON Web Tokens) to ensure only authorized individuals can access the system. Consent is emphasized throughout setup, and users are offered tools to configure retention policies and control data collection. Users are given the option of using local compute or the remote compute and are made aware of all the risks associated with remote data transfer.

The users would be made aware of all the potential and foreseeable risks associated with using our system in order to warn them against potential risks. We have made an effort to keep it as secure as possible but will continue working on encryption algorithms and secure data transfers to make the system even more robust.

VII. Related Work

Several existing projects and products are similar to the system we are proposing. A notable example is Tile [5], a small Bluetooth-based tracking device that helps users find misplaced objects like keys, wallets, and phones. Tile's strength lies in its simplicity and user-friendly mobile app interface, which allows users to track items using a connected smartphone. However, Tile requires physical attachment of the tracking device to each item, which our project aims to avoid by using computer vision to detect items in an environment without the need for individual trackers.

Another similar product is the Apple AirTag [6], which uses Ultra-Wideband (UWB) technology along with the "Find My" app to locate lost items. Like Tile, it also requires physical

attachment to objects. Though it leverages precise location detection, it does not provide a solution for automatically tracking objects across a room in the way that our system does.

On the more technical side, systems such as Amazon Go "Just Walk Out" technology [7] stores utilize computer vision and machine learning for object detection and tracking. The Amazon Go system is much more advanced and tracks multiple users and objects in real time, relying heavily on cloud infrastructure. While impressive, the scale and complexity of Amazon Go surpasses what our project is targeting, which is smaller indoor environments.

Cortexica Vision Systems [8] also offers vision-based object recognition for retail and inventory management, similar in principle to our approach but with a focus on industrial use cases.

Our system differs from these existing solutions by focusing on a low-cost, camera-based, non-invasive solution for individual users, such as families or students, to locate commonly misplaced items without attaching tracking devices. Furthermore, the use of machine learning models for indoor object detection ensures that the solution remains scalable and adaptable to various environments.

VIII. Summary

Our system successfully met the majority of the design specifications, including real-time object detection, voice and text querying capabilities, and seamless integration between the Raspberry Pi, web server, and database. However, some performance limits remain, particularly in object detection accuracy during poor lighting conditions and when handling highly similar objects. The system's latency for certain operations, such as processing voice queries or running computationally intensive models like Grounding DINO, also revealed opportunities for optimization. Given more time, we could explore further fine-tuning of the object detection models, optimizing database queries, and leveraging hardware acceleration to improve overall efficiency.

A. Future work

While the project was developed within the scope of the semester, we are considering continuing this work to enhance its usability and performance. Future efforts could focus on building a more robust dataset for training models to recognize diverse objects and deploying a hybrid cloud-local processing architecture for faster response times. Additionally, we aim to improve the user interface to make it more intuitive and accessible, especially for users with disabilities, and explore the incorporation of advanced edge AI chips for better real-time processing on the Raspberry Pi.

B. Lessons Learned

For future student groups addressing similar applications, we would recommend prioritizing modularity in the system design to simplify debugging and scaling. Prototyping early and testing in real-world conditions helped us identify and address limitations quickly, which we found extremely helpful. Additionally, balancing computational demands between edge devices and external servers is necessary for creating a responsive yet cost-effective system. Lastly, keeping user needs and ethical considerations in mind while designing your system helps define the purpose of the project.

Glossary of Acronyms

RPi – Raspberry Pi
AWS – Amazon Web Services
FOV – Field of View
GPU – Graphics Processing Unit
ML – Machine Learning
RDS – Relational Database Service
YOLO – You Only Look Once
VM – Virtual Machine

References

[1] "Lost and Found Statistics, Trends & Facts 2023." Lostings, 2023, www.lostings.com/lost-and-found-statistics/. newauthors.ieeeauthorcenter.ieee.org/author-tools/

[2] "The Psychology of Web Performance | the Uptrends Blog." Blog.uptrends.com, 13 June 2018, blog.uptrends.com/web-performance/the-psychology-of-web-performance/.

[3] Cascella, Marco, and Yasir Al Khalili. "Short Term Memory Impairment." PubMed, StatPearls Publishing, 2020, www.ncbi.nlm.nih.gov/books/NBK545136/.

[4] Vigderman, Aliza, and Gabe Turner. "2024 SimpliSafe Home Security Package Costs & Monitoring Plans." Security.org, 16 Sept. 2024, www.security.org/home-security-systems/simplisafe/. Accessed 12 Oct. 2024.

[5] "Tile Tracker | Bluetooth Trackers for Keys, Wallets, Phones, and More." Tile ECommerce, 2024, www.tile.com/en-us?srsltid=AfmBOoqiPlO2RfW68YqSzWMciQHgULCpPVqbTAXsK1V-UDVidQUBQluf. Accessed 12 Oct. 2024.

[6] Basappa, Prashanth. "The Technology behind Apple's AirTag." Nerd for Tech, 20 July 2021, medium.com/nerd-for-tech/the-technology-behind-apples-airtag-c7983f9322b5.

[7] Gross, Ryan. "How the Amazon Go Store's AI Works." Towards Data Science, Towards Data Science, 7 June 2019, towardsdatascience.com/how-the-amazon-go-store-works-a-deep-dive-3fde9d9939e9.

[8] "Cortexica Vision Systems." Pitchbook.com, 2024, pitchbook.com/profiles/company/60147-64#overview. Accessed 12 Oct. 2024.

TASK TITLE	TASK OWNER	START DATE	DUE DATE	PCT OF TASK COMPLETE	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8	WEEK 9	WEEK 10	WEEK 11	WEEK 12	WEEK 13	WEEK 14	WEEK 15	WEEK 16
Project Conception and Initiation																				
Team Composition Form	Giancarlo	Week 1	8/29/24	100%																
Abstract	Swati	Week 2	9/5/24	100%																
Website Initialization	Giancarlo	Week 3	9/14/24	100%																
Proposal Presentation Slides	Swati	Week 3	9/15/24	100%																
CV Models Initial Research/Training	Ethan	Week 3	9/20/24	100%																
Text To Voice Proof Of Concept	Ethan	Week 4	9/18/24	100%																
Project Definition and Planning																				
Design Presentation Slides	Giancarlo	Week 3	9/19/24	100%																
Design Report	All	Week 3	10/12/24	100%																
Project Launch & Execution																				
Initial Equipment Procurement	All	Week 4	Week 16	100%																
ML model training (YOLO)	Ethan	Week 4	Week 13	100%																
Grounding DINO Research + Training	Ethan	Week 8	Week 16	100%																
RPi configuration with camera	Swati	Week 5	Week 5	100%																
SQL database prototype	Giancarlo	Week 5	Week 6	100%																
Camera Synchronization	Ethan	Week 8	Week 10	100%																
Pre-processing	Ethan	Week 10	Week 12	100%																
Webserver Prototype	Giancarlo	Week 5	Week 5	100%																
Define Interfaces	All	Week 6	Week 6	100%																
Assemble MVP	All	Week 8	Week 13	100%																
Improve Hardware - RasPi Case	Swati	Week 8	Week 8	100%																
Order Additional Hardware	Swati	Week 6	Week 7	100%																
Audio Interface/Word Cosine Analysis	Ethan	Week 10	Week 12	100%																
Project Performance/Monitoring																				
Construct Testing Dataset	All	Week 5	Week 15	100%																
Instrument Code for Testing	Giancarlo	Week 8	Week 10	100%																
Optimize Slow Code Based On Latency Tests	Ethan, Swati	Week 10	Week 13	100%																
Testing Integrated MVP on data/iterating on modules	All	Week 9	Week 15	100%																

Table V: Schedule and task breakdown

Description	Manufacturer	Quantity	Cost / item	Price Paid (in class)	Total	Total Paid (in class)
Raspberry Pi V4 8GB	Raspberry Pi	2	$75.00	$0.00	$150.00	$0.00
Raspberry Pi V5 8GB	Raspberry Pi	1	$80.00	$0.00	$80.00	$0.00
NVIDIA Jetson Nano 4GB Developer Kit	NVIDIA	1	$300.00	$0.00	$300.00	$0.00
Raspberry Pi Camera Module 3 Wide	Raspberry Pi	3	$35.00	$0.00	$105.00	$0.00
3D Printer Filament	PLA Printer Filament 1kg	1	$20.00	$0.00 (previously owned)	$20.00	$0.00
TKGOU Conference USB Microphone	TKGOU	1	$20.00	$20.00	$20.00	$20.00
USB Laptop Speaker	LIELONGREN1	1	$16.00	$16.00	$16.00	$16.00
Velcro strips	VELCRO	1	$9.00	$9.00	$9.00	$9.00
Rpi 5 Camera Ribbon	Wonrabai	1	$8.00	$8.00	$8.00	$8.00

Total: $95.00

Table VI: Cost Breakdown