Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

# Giving ChatGPT A Virtual Body

Semester Thesis


Han Xi

hxi@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich


**Supervisors:**
Ard Kastrati, Dushan Vasilevski
Prof. Dr. Roger Wattenhofer

February 17, 2025

# Acknowledgements

I would like to express my sincere gratitude to my supervisors Ard Kastrati and Dushan Vasilevski, for their continuous support and valuable discussions throughout the entire duration this project. Additionally, I extend my special thanks to the Magic Leap 2 team for providing access to the device, which made the development and evaluation of this project possible. Their support has been crucial in enabling hands-on experimentation and real-world testing of the system.

# Abstract

This project presents a novel pipeline for guided object detection and retrieval on head-mounted devices, with a particular focus on real-time object finding tasks. Our approach leverages state-of-the-art Vision-Language Models, Large Language Models, and open-vocabulary object detectors to interpret user queries, identify target objects in mixed reality environments, and provide intuitive guidance to the wearer. By integrating these components into the Magic Leap 2 headset, our system is able to understand the spatial environment and visualize the location of objects directly in the user's field of view. In doing so, we demonstrate how combining real-time object detection with VLM-based guidance can expedite the process of locating, tracking, and retrieving items, addressing a variety of real-world scenarios such as finding misplaced tools or navigating dynamic workspaces. Moreover, we showcase the system's extensibility, which opens up new avenues for future mixed reality applications, including collaborative assistance, inventory management, and enhanced accessibility support.

# Contents

Chapter 1

# Introduction

## 1.1 Background & Aims

Recent breakthroughs in deep learning have propelled Large Language Models (LLMs) and Vision-Language Models (VLMs) to the forefront of cutting-edge research. These models, capable of interpreting text, images, or a combination thereof, have demonstrated remarkable performance across various tasks, including natural language understanding, image captioning, and object detection. Their ability to perceive, reason, and generate contextually relevant outputs has enabled exciting new applications in fields such as robotics, healthcare, and interactive systems.

Meanwhile, advancements in Virtual Reality (VR) and Augmented Reality (AR) have revolutionized how users interact with digital content, merging virtual elements with physical environments to create immersive, user-centric experiences. Building upon these concepts, Mixed Reality (MR) extends the synergy between the real and virtual worlds by integrating virtual overlays that are spatially aware of the user's surroundings. The combination of deep learning models with MR platforms presents a unique opportunity to harness intelligent perception and human-computer interaction, paving the way for robust, adaptive systems that cater to real-world tasks.

Recently, a growing number of applications have demonstrated the power of combining VLMs/LLMs with MR headsets to deliver context-aware experiences [1] [2] [3] [4] [5]. These systems leverage advanced spatial mapping, sensor inputs, and real-time rendering capabilities to empower AI-driven applications that enhance user perception and decision-making. By situating deep learning models within the user's immediate environment, MR devices can provide interactive guidance and immersive feedback that surpass traditional screen-based interfaces.

Against this backdrop, our goal is to harness these breakthroughs to create a novel mixed reality application that performs agent-based guided object detection. Specifically, we seek to address the real-time challenge of locating, identifying, and guiding users toward target objects in complex environments. By combining open-vocabulary object detectors with VLM-based reasoning, our pipeline can interpret user requests, analyze the visual scene, and offer on-the-fly assistance in finding and retrieving objects. We deploy this solution on the Magic Leap 2 (ML2) headset, leveraging its multimodal sensor inputs and robust spatial capabilities to deliver context-aware and user-friendly guidance.

This project outlines the development and evaluation of our system, highlighting how integrating LLMs/VLMs with MR can enhance daily tasks such as object finding. We further show that the proposed framework opens up many possibilities for future applications in areas like collaborative problem-solving, inventory management, and accessible mixed reality interfaces, demonstrating the promise of AI-infused MR for a wide range of real-world scenarios.

## 1.2 Related Works

There have been applications that combine VLMs with object detection to deliver more holistic and context-aware experiences [6] [7] [8]. They rely on robust and efficient object detection to maintain real-time responsiveness in tasks such as object placement, scene understanding, and user interaction.

On the one hand, these systems use state-of-the-art ready-to-use object detectors - often from the YOLO family [9] [10] [11] - to locate and classify targets within the user environment. The YOLO family of single-stage object detectors has become widely adopted in real-time applications for its favorable balance between speed and accuracy compared to two-stage object detectors such as the R-CNN family [12][13] or the Transformer-based [14] DETR models [15] [16] [17]. To better align object detection with dynamic user needs, some recent works have adopted open-vocabulary detectors. Open-vocabulary detection enables detection beyond pre-defined categories, making it highly suitable for applications requiring adaptability and generalization. Most remarkably, YOLOWorld [18] archieves real-time open-vocabulary object detection. By integrating such models, MR systems can dynamically adjust to user-defined object classes, improving real-time object retrieval and reducing reliance on pre-trained category sets.

In parallel with advances in object detection, VLMs have gained significant traction for tasks that require reasoning over both textual and visual inputs. Models like Gemini [19], Qwen [20], Seed-VLM [21] and GPT-4V [22] have been developed to handle tasks such as image captioning, visual question answering, and scene understanding.

Chapter 2

# Design and Implementation

This chapter details the design and implementation of the system, outlining its key components and explaining them in depth in the subsequent sections. Before delving into implementation specifics, we first provide a high-level perspective on the system's overall architecture. Our primary objective is to design an interactive agent capable of real-time user interaction. The agent's core function is to interpret the user's task, guide them toward finding the target object by analyzing and understanding its surrounding environment, and precisely indicate the object's position by visualizing a pin at its location. The functionality of the agent is driven by two fundamental components: input processing and memory management. Input processing is enabled by embedding specialized models into the agent, allowing it to interpret both visual and textual information. Memory is simulated through (i) a *buffer mechanism* that stores both past and present environmental data, enabling the agent to leverage historical context when generating guidance, (ii) a *persistent database* that stores information of the detected objects, allowing for retrieval in the future. This memory structure allows the agent to maintain continuity in its interactions and offer more context-aware assistance.

A crucial aspect of the system's implementation is determining the most effective deployment strategy for the agent, as it directly impacts performance, responsiveness, and ease of development. Our system adopts a client-server design, and we consider three primary types of deployment: server-side deployment, client-side deployment, and a hybrid model that distributes computations between the client and the server. Each approach has distinct advantages and trade-offs, influencing factors such as latency, computational efficiency, and system complexity. In our system, the client-side application runs on the ML2 [23] device, where the mixed reality interface is developed using the Unity [24] framework. Unity provides a powerful environment for real-time rendering and interaction, making it particularly well-suited for mixed reality applications. Meanwhile, the server-side infrastructure is implemented in Python, allowing access to mature deep learning frameworks such as PyTorch [25], which facilitate rapid integration of models and efficient execution. The choice between these deployment strategies requires careful consideration of computational load distribution, real-time constraints, and system development complexity. The following paragraphs explore these approaches in detail, evaluating their advantages and limitations in the context of our application.

A *hybrid deployment* can optimize throughput by distributing computational workloads across both ends. However, this comes at the cost of a distributed execution model, making it difficult to maintain a consistent and uniform interaction with the agent. Additionally, system development becomes more complex due to the challenges in synchronizing operations across multiple devices. Since our objective is to abstract the agent as a unified entity, this approach is not ideal.

An alternative approach is deploying the agent entirely on the *client side*. This strategy offers several advantages, particularly in terms of processing latency and direct access to unprocessed data. By executing the agent on the client device, frames from the ML2 camera can

be processed immediately, eliminating the need for additional communication layers through APIs. Furthermore, the model can be deployed directly on the device using ONNX-exported [26] models in combination with Unity's Barracuda library [1]. This approach minimizes network dependency and enhances system responsiveness, making it advantageous for latency-sensitive tasks like real-time object detection and user guidance. Additionally, object detection models such as YOLOWorld, which support open-vocabulary detection, can be executed locally to improve responsiveness and reduce dependency on network connections. Despite these benefits, client-side deployment also introduces significant challenges. One of the major drawbacks is the longer development time due to the limitations of the C# ecosystem. Compared to Python, C# offers less flexibility and a more restricted selection of deep learning libraries, making model integration and debugging more cumbersome. Another critical limitation arises in object detection. While state-of-the-art VLMs such as Gemini [19] and GPT-4V [22] can be accessed via API calls, the object detection model presents a key restriction. In this setup, the class names for detection must be updated frequently to accommodate user requests. However, once an ONNX model is exported, it does not allow for dynamic modification of variables like class names. This limitation means that each exported model is restricted to a fixed set of object classes, making it unsuitable for applications requiring flexible and adaptive object detection.

Considering these limitations, the final implementation adopts a *fully server-based* deployment strategy. Deploying the object detection model on the server allows for the dynamic updating of detection class names through client-server communication, making the system more adaptable to different user requirements. This approach also benefits from Python's extensive ecosystem, which provides a rich set of libraries for deep learning, making development more efficient and flexible. Additionally, a server-based deployment ensures that the agent remains a unified entity, avoiding the complexities associated with distributed execution. Centralizing all processing on the server makes the system scalable and maintainable, that supports efficient object detection and guidance generation. This design choice allows the agent to provide seamless interaction with users while maintaining adaptability and robustness in real-time object-finding tasks.

[1] https://github.com/Unity-Technologies/barracuda-release

## 2.1 Overview

Our application consists of three main components: the *client*, the *server*, and the *agent*, which is deployed on the server. These components work together to facilitate real-time object detection, scene understanding, and user guidance in a MR environment. The following sections detail the implementation of each component, describing their roles in enabling real-time MR-based object detection and retrieval.

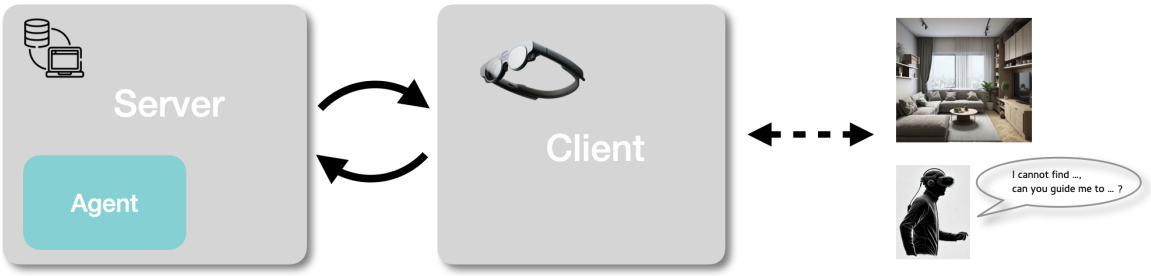


**Figure 2.1:** *A high-level overview of our application*

During an application session, the user, wearing the MHD, prompts the agent with a specific object-finding task while actively moving through the environment. The captured visual information and processed results are constantly exchanged between the client and the server. The agent processes this data, performs object detection and reasoning, and periodically provides guidance to assist the user in locating the target object. This results in two asynchronous data communication streams between the client and the server:

1. **Frame-Based Data Stream** – The first stream consists of visual and spatial data captured per frame by the MHD (see top figure of 2.2). This information is continuously sent to the server, where the agent processes it, applies object detection, and stores relevant results. This stream ensures that the agent maintains an up-to-date representation of the user's surroundings.

2. **Guidance Stream** – The second stream involves periodic guidance messages generated by the server-side agent (see bottom figure of 2.2 ). The agent analyzes the current scene, detection results, and user input before sending structured feedback to the client. These guidance messages help the user navigate toward the target object efficiently by providing context-aware instructions.

## 2.2 Agent

The agent is designed with (i) VLM for chat-based interaction and reasoning, (ii) an object detection model for visual localization, and (iii) an LLM to process (minor) textual information. This architecture is chosen based on the functional requirements outlined below.

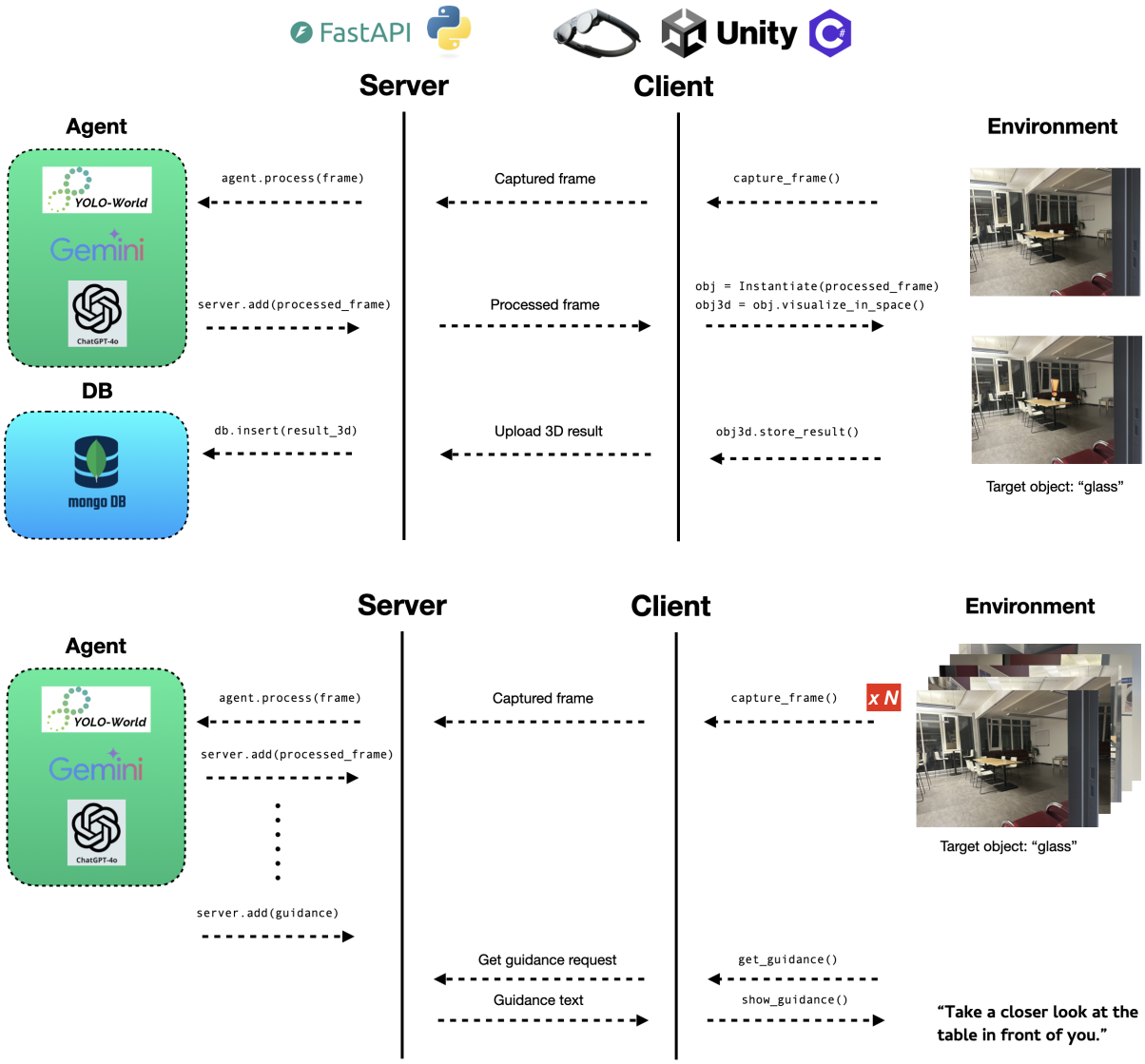


**Figure 2.2:** *Data flow per frame (top) and data flow per guidance (bottom)*

The agent is responsible for interacting with and guiding the user in locating objects. To achieve this, it must:

1. **Identify the user's task** – Understanding the objective based on the provided prompt.
2. **Detect objects within the user's environment** – Processing visual information to recognize and locate objects of interest, where the visual information is given by real-time camera frames.
3. **Provide guidance for locating the target object** – Leveraging user input, environmental context, and historical data to offer effective navigation assistance.

To effectively identify the user's task, the agent relies on direct user prompting, structured according to well-defined system principles. Given a suitable VLM, it is straightforward to prompt it with task-specific instructions. Moreover, we also want to localize the object so that the user is provided with a visual cue. In this context, localizing objects based on the user's prompt corresponds to the problem of Visual Grounding (VG). Several existing models, such as Owl-ViT [27] and GroundingDINO [28], have been developed specifically for VG. These models

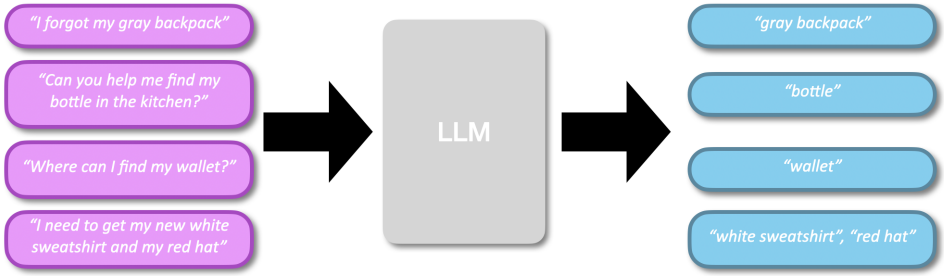


**Figure 2.3:** *User task prompts (left) are converted to a list of class names (right) using an LLM. The object detector will set its target classes to this list.*

demonstrate strong capabilities in localizing objects based on textual descriptions, but present two key challenges: (1) their inference times are relatively long, making them unsuitable for real-time interactions, and (2) they are not optimized for providing step-by-step guidance to users. These limitations make them less practical for our application. Additionally, industry-scale VLMs like GPT-4V [22], while powerful for joint text-visual reasoning, are not trained or fine-tuned to produce accurate bounding boxes or other localization cues per se[2] [3]. Consequently, we will have to rely on an additional model for localization. For this purpose, object detection has been extensively optimized for real-time performance while achieving relatively high detection accuracy. Recent advancements, such as YOLOWorld [18] and OmDet-Turbo [29], have further extended the capabilities of object detection by enabling open-vocabulary detection, allowing the model to recognize objects beyond a fixed set of categories. This makes real-time open-vocabulary object detection models an ideal choice for the agent's localization module.

Now the challenge lies in defining the appropriate input for the detection model. While open-vocabulary object detection models can handle a broad range of object categories, they struggle with long and complex textual descriptions, especially when multiple or potentially ambiguous objects are mentioned, or a long sentence is used to depict one object. To overcome this limitation, we integrate an LLM to process user input. The LLM extracts relevant object classes from the user's prompt, simplifying the textual input into distinct object categories. These extracted object names are then used as input for the object detection model (see Appendix A for details on the prompt formulation).

To facilitate real-time tracking and historical context, the system dynamically stores camera frames along with object detection results on the server during the application's runtime (see section Server Implementation). This history buffer enables the agent to maintain an evolving understanding of the environment over the course of the interaction.

To close the loop, the agent harnesses information from user input, real-time camera frames, and object detection results to generate meaningful guidance. This is achieved by, as mentioned in the beginning, leveraging the VLM, which interprets the contextual information and produces step-by-step instructions to help the user locate the target object efficiently. In order to maintain a cohesive API design and simulate the presence of a unified agent, we abstract all functionalities into a single Agent class. Tasks are executed through a standardized method call, following the format: `agent.process_TASK()`, with `TASK` being the task name. This approach ensures a modular and intuitive interface, allowing seamless execution of various agent functionalities while preserving the perception of a single, autonomous entity. An illustration of the agent is given by the Figure 2.4.

[2] https://blog.roboflow.com/gpt-4v-object-detection/
[3] https://community.openai.com/t/gpt4-v-object-detection-bounding-box-value-incorrect/846566

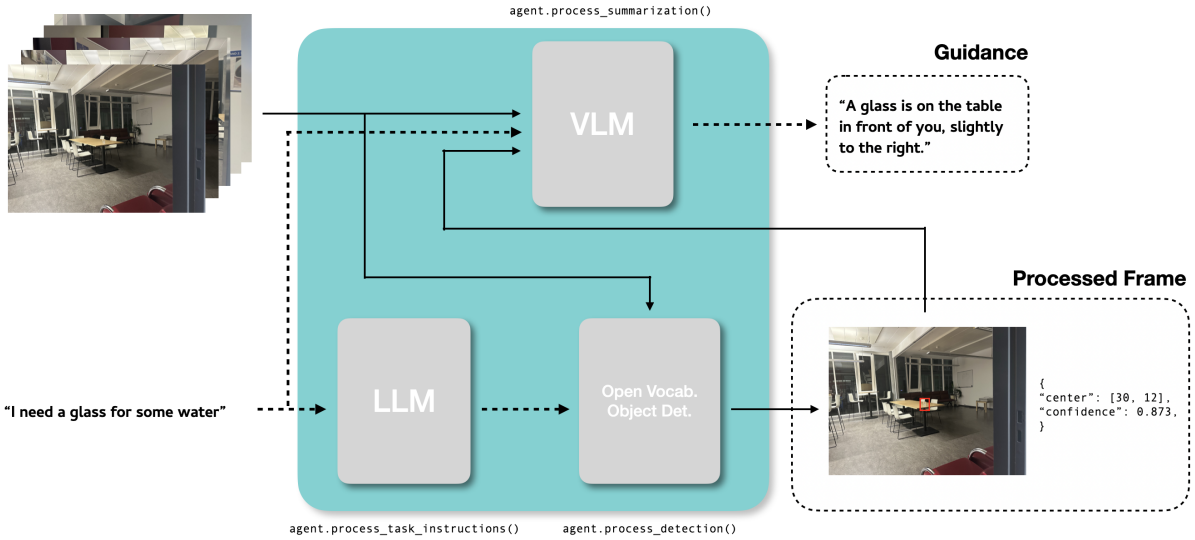


**Figure 2.4:** *Overall architecture of the agent. Dashed lines represent text data flow and solid lines represent visual data flow. The input to the VLM consists of the user task, camera frames and the JSONified detection results (center and confidence) in the history buffer. The agent produces an output stream of processed frames and a guidance stream (Figure 2.2)*

## 2.3 Server Implementation

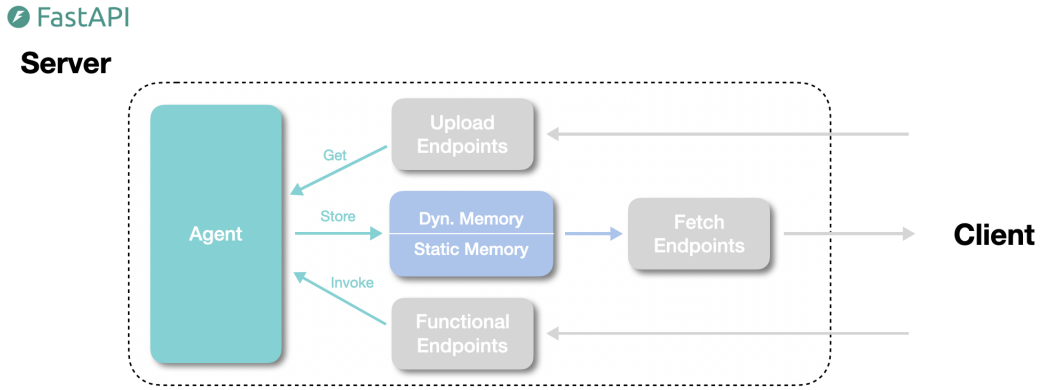


**Figure 2.5:** *Server Implementation*

### 2.3.1 Server

The agent is deployed on a server backend that manages real-time processing, object detection, and interaction with the user. For this purpose, we use FastAPI [30], a modern web framework that offers high performance and asynchronous capabilities, making it well-suited for real-time applications. Compared to traditional frameworks like Flask [31] and Django [32], FastAPI provides native support for asynchronous execution, significantly improving response times when handling multiple concurrent requests—an essential feature for our application, where real-time perception and user interaction are critical. Additionally, FastAPI's built-in data validation and automatic OpenAPI documentation generation streamline development and debugging. The FastAPI server is designed with three primary types of endpoints, each serving a distinct function in managing data flow and interactions between the client and the agent. By structuring the

API with these categories, the server ensures a modular and efficient approach to managing real-time data flow, supporting both data-driven interactions and dynamic agent control:

**Uploads Endpoints** These endpoints allow the client to send data to the server for storage and processing. In our implementation, we include:

- `upload_frame` - Used to upload frames captured by the ML2 main camera, enabling real-time visual processing.
- `upload_detection` – Allows the client to upload detected objects along with their associated metadata, including their position in 3D space, ensuring the agent maintains an accurate representation of the environment.

**Fetch Endpoints** These endpoints facilitate data retrieval, enabling the client to access processed information. The key endpoints include:

- `get_frame` - Retrieves frames that have been processed by the agent, providing access to the latest visual data.
- `guidance` - Gets guidance information generated by the VLM based on the history buffer.
- `find_object_by_reference` and `find_object` – Enable the retrieval of static objects stored in the database, ensuring efficient lookups for previously detected objects.

**Functional Endpoints** These endpoints provide control over the agent's internal state, allowing for adjustments and resets. For instance:

- `reset_classnames` – Resets specific fields within the agent, ensuring adaptability to different user inputs or session resets.

The primary data exchanged via the client-server API consists of camera frames, 2D (`ObjectDetectionResult`) and 3D (`ObjectDetectionResultInSpace`) object detection results. Detection results contain spatial and temporal metadata, which provide context for when and where objects were detected within the scene.

```python
@dataclass
class ObjectDetectionResult():
    frame_id: str
    center: Vector2d
    top_right: Vector2d
    bottom_left: Vector2d
    confidence: float
    class_name: str
    last_seen: datetime
    image: bytes
    image_size: Vector2d
    embedding_img: Optional[list[float]] = None
    embedding_cls: Optional[list[float]] = None
    embedding_desc: Optional[list[float]] = None
    description: Optional[str] = None
```

```python
@dataclass
class ObjectDetectionResultInSpace():
    frame_id: str
    center: Vector2d
    top_right: Vector2d
    bottom_left: Vector2d
    confidence: float
    class_name: str
    last_seen: datetime
    rel_location: Vector3d
    image: bytes
    image_size: Vector2d
    embedding_img: Optional[list[float]] = None
    embedding_cls: Optional[list[float]] = None
    embedding_desc: Optional[list[float]] = None
    description: Optional[str] = None
```

### 2.3.2 Storage

To emulate the agent's memory, we introduce a dual-memory system consisting of *static* and *dynamic* memory. This design enables a clear separation between information retained across multiple application runs and data relevant only to the current session. Such a distinction is particularly beneficial for object-finding tasks, as retaining past information allows users to locate objects more efficiently.

*Static memory* stores information from previous application runs, allowing the agent to recall and retrieve objects that may have been misplaced or forgotten over time. For example, the system can periodically scan and store metadata for objects prone to being lost, such as keys, wallets, or frequently used tools. By maintaining a persistent database, users can later query the agent to locate these items, retrieving their stored information and visualizing their estimated position in the environment. However, accurate retrieval depends on the availability of absolute spatial positioning. Unlike photo geolocation methods (e.g., GeoSpy AI [4]), which rely on GPS-based localization, object localization is in general geolocation-independent. To address this, we employ relative spatial positioning using marker tracking. By associating objects with known reference points in the environment, we can accurately estimate their relative locations.

In contrast, *dynamic memory* serves as a temporary buffer, storing real-time environmental data collected during the current application session. This allows the agent to track and analyze ongoing changes, which is crucial for generating context-aware guidance. Additionally, dynamic memory enhances object retrieval by enabling the system to recall recently seen objects, allowing users to retrieve information about items they have interacted with during the same session. By combining static and dynamic memory, the system achieves both long-term recall and short-term adaptability, ensuring that the agent can assist users effectively in both persistent object tracking and real-time navigation.

**Static Memory** We implement static memory using a MongoDB database [5] to store relevant information from object detection results, ensuring long-term accessibility to previously detected objects. To enhance retrieval capabilities, we incorporate text-based search mechanisms by leveraging text embeddings. Before inserting a new detection result into the database, we compute its text embedding using the SentenceTransformer model [33] from HuggingFace's Transformers library [34]. This embedding represents the semantic meaning of the object's class name and description in a high-dimensional space. By employing text similarity search, we enable users to retrieve objects based on natural language descriptions. Specifically, when a user queries the system, their input is converted into an embedding, and we compute the similarity between this embedding and the stored object embeddings. This allows the system to return the most relevant objects, even if the user's description does not match the stored class name exactly.

**Dynamic Memory** To maintain an accurate and context-aware understanding of the environment, we implement a history buffer on the server in form of a dictionary, which continuously stores processed frames along with their corresponding detection information throughout the current application session. This ensures that the agent has access to a temporal sequence of observations, rather than relying solely on the latest frame. Unlike using only the most recent frame, leveraging the entire history buffer as input to the VLM allows the agent to generate more coherent and contextually rich guidance. This is particularly important given the latency in processing visual information—updating the latest frame in real time requires careful synchronization to ensure that the most up-to-date detections and contextual data are incorporated.

[4] https://geospy.ai/blog/find-a-photo
[5] https://www.mongodb.com

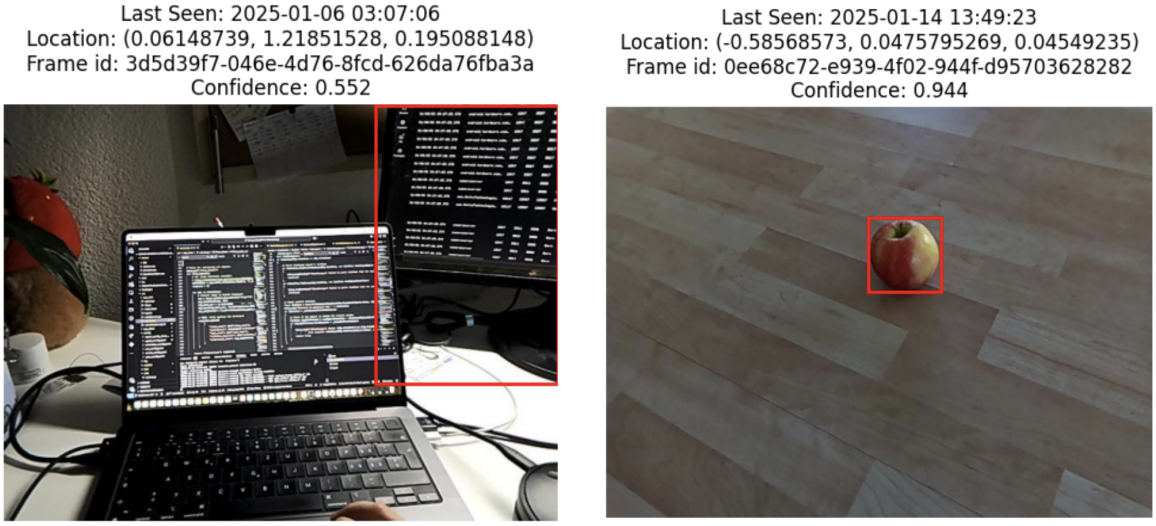


**Figure 2.6:** *Examples of retrieved object detection results from the MongoDB database (Left: "monitor", Right: "apple"). Note that the retrieved documents carry a 3D position.*

Improper synchronization could lead to an information fallback, where the user receives outdated guidance based on stale visual data rather than the current state of the environment.

## 2.4 Client Implementation

### 2.4.1 MagicLeap2 Subsystems

The client application runs directly on the ML2 device, granting it access to various subsystems that capture and process essential environmental information. We leverage ML2's OpenXR features [6] for Unity, allowing the application to efficiently process sensor inputs, track spatial information, and enhance user interactions without relying entirely on external computation.

**MLCamera** The most important feature for our application is the camera capture. The MagicLeap2 MLCamera allows us to capture real and virtual content inside our application. ML2 has two streams from the same physical camera, one being the `Main Camera` stream, and the other being stream coming from the `CV Camera`. In our case, we use the `CV Camera` to obtain uncompressed, raw frames.

**Marker Understanding** Since we aim to achieve accurate spatial positioning of objects detected in past application sessions, we implement a relative localization approach using markers. By associating detected objects with a known reference point, the system can consistently determine their position within the environment, across different sessions. ML2 OpenXR features include a `MarkerUnderstanding` subsystem that enables the detection of barcodes and markers. Markers such as Aruco, April Tags and QR Codes can also be tracked in 3D space.

**Meshing** The ML2 meshing subsystem enables the application to access a spatial mesh representation of the environment, providing a crucial foundation for accurate spatial reasoning. This

[6] https://developer-docs.magicleap.cloud/docs/guides/unity-openxr

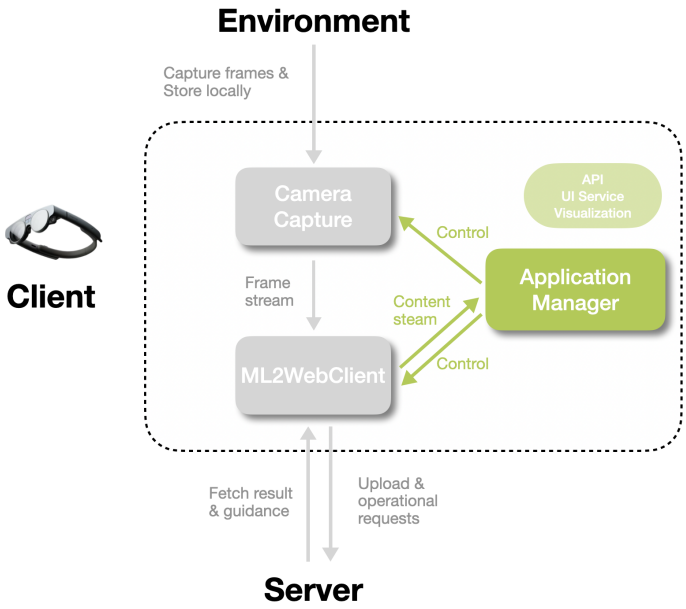


**Figure 2.7:** *Client Implementation*

feature is particularly important for visualizing object detection results on 2D image frames, as it allows the system to perform spherecasting to infer an object's position in 3D space based on its 2D image coordinates. Compared to the raw depth values obtained from the ML2 camera, the ML2 mesh offers a more stable and reliable world reference. ML2 depth measurements are more prone to noise and inconsistencies, which can introduce inaccuracies in object localization.

### 2.4.2 Client Application

The client application is structured around an application manager, which acts as the central control hub for various system components. This manager oversees the user interface, facilitates communication with the *web client*, and manages *camera capture* from the ML2 device. Additionally, it handles essential background operations, including resource management, permission handling, and pin visualization. Once the application starts, it initializes key components and then maintains the application's execution flow.

#### Camera Capture

The system periodically sends frames captured by `MLCamera` to the server after performing initial preprocessing on the client side. Specifically, instead of transmitting raw frames, the system first converts them to JPEG format. This compression allows for higher throughput and more frequent updates without overwhelming the network. Furthermore, to ensure that the application displays world information consistently, the camera capture must operate at an adequately high frame rate, such as 30 FPS. However, maintaining real-time performance requires a careful balance between frame capture rate and network throughput. Since transmitting every captured frame would overload the network and increase latency, we decouple frame capture from frame upload, and use an upload frequency of 1 FPS. We implement a local queue that temporarily stores the frames to be uploaded. The `ML2WebClient` is responsible for dequeuing frames from this queue and uploading them to the corresponding server endpoint.

### ML2 Web Client

The `ML2WebClient` serves as the communication interface between the client and the server, implementing client-side counterparts of the server's endpoints. This component is responsible for handling data exchange between the ML2 device and the agent running on the server. The `ML2WebClient` performs three primary functions:

1. **Uploading Frames for Processing** - It sends frames captured and enqueued by `CameraCapture` to the server, where they undergo processing.
2. **Fetching Processed Results** - It can retrieve processed frames, detected object information, and stored results from the server database. Once received, these results can be visualized within the mixed reality environment, allowing users to see currently or previously detected objects and interact with them. Additionally, it fetches guidance information generated by the VLM to assist users in locating their target objects.
3. **Sending Operational Requests to the Agent** - The client application interacts with the agent by issuing operational commands, such as setting target object classes for the object detector. This allows for dynamic adaptation of the detection pipeline based on user input, ensuring that the agent remains flexible and responsive to different tasks.

To ensure a consistent behavior and properly synchronize the latencies when *finding a new object*, we implement a structured dispatching schedule for different client-server interactions. The details of the different dispatch frequencies can be found in Chapter 3.

### Pin Visualization

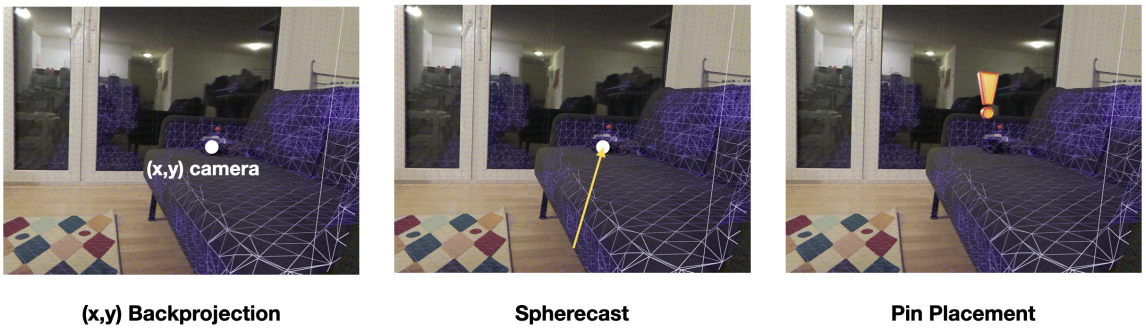


**Figure 2.8:** *For a detected object (here the camera) with center coordinates $(x, y)$, its position in space can be determined via spherecast. The pin will be placed in the proximity of the hitpoint with the spatial mesh.*

A detected object can be visualized in 3D space with the aid of the ML2 meshing subsystem. To determine the object's position, we perform raycasting from the camera origin through the object's 2D coordinates on a virtual projection plane that is located $l = 1$ meters away from the camera origin. This projection plane is scaled to match the resolution of the camera capture ($640 \times 480$), with a virtual plane width of $w = 1.3$ meters, ensuring that the ray accurately represents the object's position in the image. The point at which the ray hits the spatial mesh serves as the object's estimated position in space. Additionally, during the process of finding a new object, we perform a simple object tracking mechanism to prevent object that has been seen to show up more than once, by only visualizing objects that have not been seen in the current application session. An object $O_A$ is defined as *seen* if there is at least one object $O_B$

of the same class detected in the last $t$ seconds, within a radius of $r$ meters centered around the the center of $O_A$. For our application, we use $t = 10, r = 0.3$. Once the object pin is visualized in the environment, the user can interact with it by clicking the pin prefab. This action opens an information panel displaying the image frame at which the object was originally captured. Additionally, it presents the last seen timestamp along with the detection accuracy. Beyond viewing object details, the user can use the panel to upload the detection result to the server-side database for future retrieval and localization.

Chapter 3

# Experiments

## 3.1 Configurations

**System Configurations** The system configuration of our application is summarized in Table 3.1. In particular, for the object detector, we selected YOLOWorld-L with the YOLOv8-L backbone due to hardware constraints. Originally, YOLOWorld-L achieves an FPS of 52.0 on an NVIDIA V100 GPU and attains a zero-shot performance of 35.4 AP on the challenging LVIS [35] dataset. Although the hardware limitations of our MBP reduce the throughput to approximately 5 FPS, this remains sufficient for our application. For the LLM and VLM components, we implemented a flexible configuration that allows switching between GPT-4o, Gemini, and LLama. However, due to the long response times observed with LLama, we discarded it as a viable option. For our application tests, we maintained the final configuration as detailed in the table. More details about the model specifications are provided in Appendix B.

| | **Client** | **Server** |
|---|---|---|
| **Hardware** | MagicLeap2 PMF012 | MacBook Pro M3 (MBP) |
| **Software** | Unity 2022.3.47f11<br>C# 12.0 (.NET v8.0.403)<br>MagicLeap SDK v2.5.0<br>MagicLeap MRTK3 v1.2.0 | **Backend**<br>FastAPI v0.115.6<br>MongoDB v8.0.4<br>Python 3.10.4<br>**Agent**<br>VLM: GPT-4o mini<br>LLM: Gemini-1.5 Flash<br>Obj. Det.: YOLOWorld (yolov8l) |

Table 3.1: Hardware and software breakdown for client and server

**Runtime Configurations** Since the server's components are largely fixed, runtime configurations are primarily handled on the client side. After testing, we set an upload frequency of 1,500 ms and a fetch frequency of 800 ms, ensuring that network bandwidth usage remains manageable and that accumulating detection results do not overwhelm the system. Additionally, we request guidance every 5,000 ms. This is due to the VLM taking approximately 4 seconds to generate a response in average, hence we let the VLM automatically generate guidance either when the history buffer is full or at the five-second mark. As a result, the client's history buffer provides a sufficient batch of frames to the VLM, maintaining an effective balance between real-time responsiveness and computational overhead.

## 3.2 Test Cases & Discussions

We demonstrate the performance of our application by highlighting both successful use cases and failure scenarios, and we analyze the underlying factors that lead to these outcomes. In successful cases, the system accurately detects and localizes objects in real time, providing clear guidance that helps users retrieve their targets quickly. By contrast, failure cases—such as scenarios with highly occluded objects, noisy sensor data, or inaccurate raycasting—reveal practical limitations related to environmental conditions, model accuracy, and interaction design. As part of our initial development phase, we conduct tests in an indoor environment, specifically within an apartment setting. This allows us to assess the system's robustness under typical household conditions, before extending experiments to more complex or dynamic environments.

### 3.2.1 Normal Case

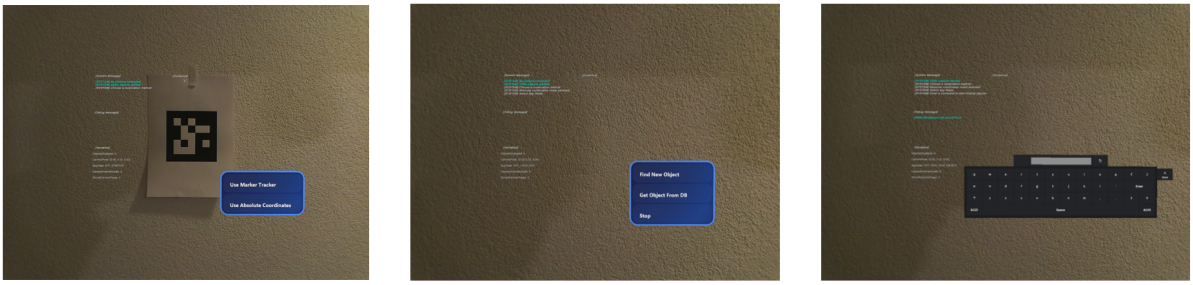


**Figure 3.1:** *The startup UI of the application, including localization (left), starting a finding process/get an object from the database (middle), and input handling when finding a new object (right).*

The application begins with a localization process, during which the user must choose between adopting absolute coordinates (relative to the session's XR Origin) or marker tracking. When selecting marker tracking, the system searches for a physical marker and records its position, ensuring that subsequent object detections can be instantiated relative to this marker's coordinates. After localization, the user can choose to place an object from the database in the environment, or to start finding a new object. Assuming that no objects are yet stored in the database, we proceed with the process of finding a new object. In this initial implementation, users input their search queries via a native keyboard interface, allowing them to specify the target object manually.

Now, let us attempt to find a forgotten camera using our system. For this demonstration, we simply type "camera" into the keyboard and submit the query (The system can also handle natural language instructions like "*I lost my camera somewhere, please help me find it*"). Upon receiving the input, the client sends a request via the `reset_classnames` endpoint, allowing the agent to process the task instruction and dynamically update the object detector's target class names (see Figure 2.3). Once the agent has completed this step , the object detection process begins, and the finding process begins. From this point onward, the system continuously detects objects matching the specified class, providing real-time feedback and guidance to assist the user in retrieving the misplaced camera. Figure 3.2 illustrates how the user is assisted by the VLM-generated guidace in navigating towards the camera. Although neither the VLM nor the object detector was able to precisely locate the camera during the process, the guidance system provided clear and actionable instructions, helping the user gradually reach the intended goal. One important remark is that the user needed to move slowly to allow the VLM-generated guidance to adapt effectively. Since the guidance was generated at fixed 5-second intervals, this update frequency appeared somewhat long for a typical indoor object-finding scenario. Despite

this limitation, the VLM demonstrated its ability to assist the user, by progressively refining the search direction.

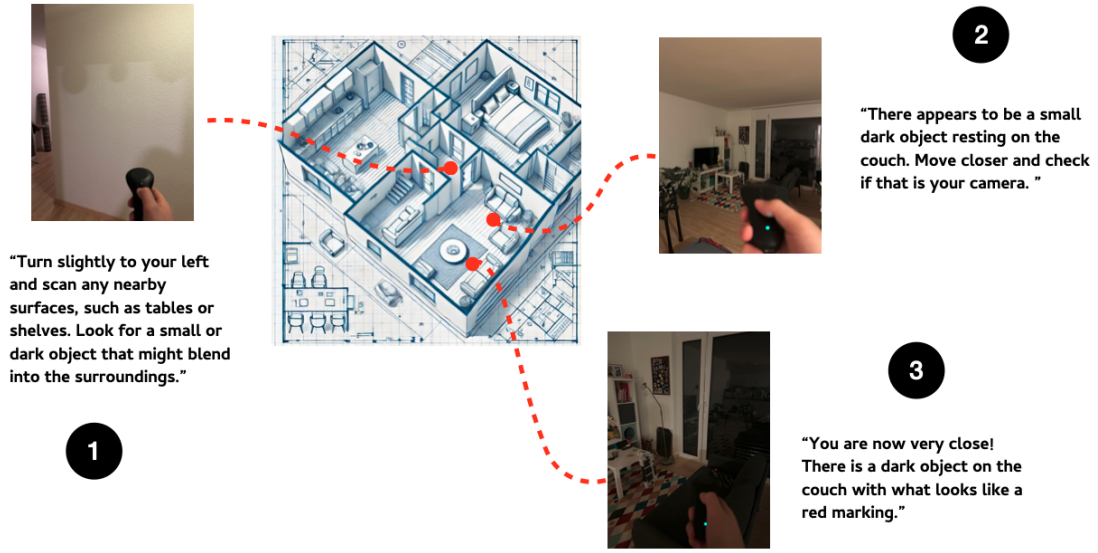


**Figure 3.2:** *The finding process of the user follows the step-by-step order illustrated in this image, showing how the system processes the request, detects objects, and refines VLM-generated guidance over time. Note that in total six guidance updates were generated in this trial, with three selected for illustration as they were the most informative.*

Once the user moves closer to the object, the object detector successfully identifies it and places a pin at its reprojected position in space (see Figure 3.3). By clicking on the pin, a panel appears displaying the detection information, providing additional context about the identified object. At this stage, the user can store the detected object in the database by selecting the upload button. Since the position is recorded relative to the marker, this ensures that in subsequent sessions, the system can accurately reinstantiate the object's location based on the same reference marker. This functionality enables persistent spatial memory, allowing the agent to assist users in retrieving previously detected objects across different application sessions.

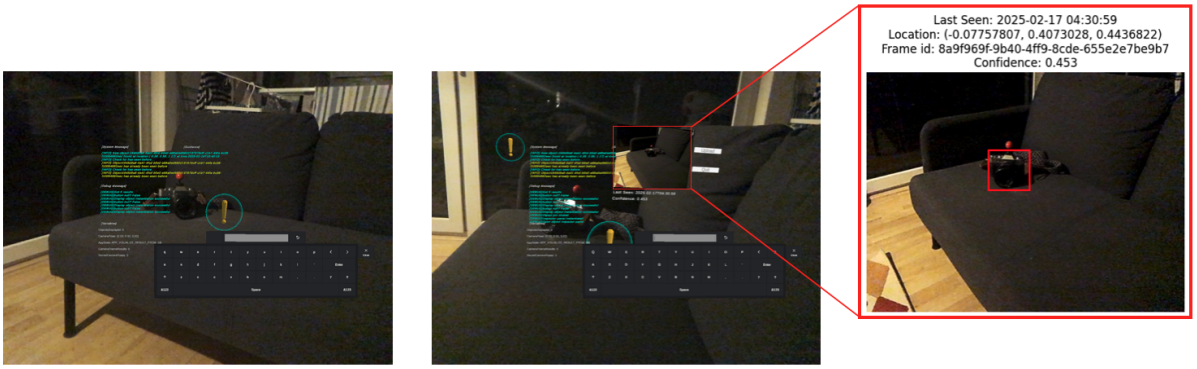


**Figure 3.3:** *Visualization of the placed object pin with its coordinates stored in the database, relative to the detected marker. Note that in the center image, another pin appears behind the one in focus, pointing to the same camera object. This duplication results from raycasting inaccuracies caused by variations in distance. The rightmost figure displays the object entry as stored in the server's database.*

### 3.2.2 Failure Cases

One direct consequence of the misalignment between the visual reasoning abilities of the object detector and the VLM is their differing strengths and limitations. The VLM excels at recognizing fine-grained details in an image but lacks the ability to generate bounding boxes, whereas the object detector can localize objects but sometimes fails to recognize even common items. This disparity can lead to inconsistent user experiences. For instance, in cases where the object detector fails to identify an object, but the VLM can still describe it, the system may generate guidance such as, "*There is nothing detected in the given frames; however, I can see [the target object] lying in front of you.*" Since no bounding box is produced, the object remains invisible in the MR environment, making it difficult for the user to act on the information. Conversely, in cases where the object detector misclassifies or fails to detect less common objects, the detection confidence threshold may need to be adapted dynamically. More complex object descriptions or longer queries tend to result in lower detection confidence, whereas single-word queries often yield higher confidence scores. Adjusting detection thresholds based on query specificity and incorporating VLM-assisted re-ranking of detections could help mitigate these discrepancies, improving alignment between reasoning and detection in our system.

Chapter 4

# Conclusion

In this work, we presented a novel pipeline for agent-based object detection in mixed reality, integrating the strengths VLMs and open-vocabulary object detection. Our approach leverages the perceptual precision and contextual reasoning of VLMs alongside the spatial localization capabilities of object detection models. While VLMs excel in capturing fine-grained semantic details and generating text-based guidance, object detectors—such as those from the YOLO family—provide real-time bounding box outputs, albeit at slightly lower accuracy. Our system fuses these complementary modalities to offer a balanced solution, and we acknowledge ongoing work in related fields, including emerging research on Agent Object Detection [36] and enhancement to models like YOLO-UniOW [37].

Despite these promising contributions, our work faces several limitations. For instance, our current approach for converting 2D detection outputs to 3D visualizations is sensitive to mesh noise and ray direction inaccuracies. This issue is particularly pronounced in crowded scenes where occlusion further complicates the accurate placement of object pins. Moreover, the pipeline currently lacks a flexible interaction flow; users must manually reset tasks to initiate a new object-finding session, rather than having this process managed automatically by the agent. Also, the objects are not consistently tracked over extended periods, which leads to duplicate pin placements of the same (static) object. This can be improved by using a more advanced 2D and/or 3D object tracking algorithm, which can also help us track moving objects.

Looking ahead, our pipeline opens up multiple avenues for future research and application development. Potential extensions include integrating voice interaction for hands-free object retrieval, enhancing MR-based guidance for navigation in complex environments, and enabling direct manipulation of virtual objects within mixed reality spaces. Moreover, the framework could be adapted for use in, for example, autonomous driving systems or as a daily assistant to improve task efficiency in various settings. Refining the search process by enabling the agent to adapt dynamically to the user's task—by continuously listening to the user's inputs and incorporating additional object attributes into the search criteria—is another promising direction that could further enhance the system's precision and usability.

Overall, our contributions provide a step towards more intelligent and interactive mixed reality systems, while also highlighting key challenges and opportunities for future work in this rapidly evolving domain.

# Bibliography


[1] F. D. L. Torre, C. M. Fang, H. Huang, A. Banburski-Fahey, J. A. Fernandez, and J. Lanier, "Llmr: Real-time prompting of interactive worlds using large language models," 2024. [Online]. Available: https://arxiv.org/abs/2309.12276

[2] V. Parikh, S. Mahmud, D. Agarwal, K. Li, F. Guimbretière, and C. Zhang, "Echoguide: Active acoustic guidance for llm-based eating event analysis from egocentric videos," 2024. [Online]. Available: https://arxiv.org/abs/2406.10750

[3] M. Konenkov, A. Lykov, D. Trinitatova, and D. Tsetserukou, "Vr-gpt: Visual language model for intelligent virtual reality applications," 2024. [Online]. Available: https://arxiv.org/abs/2405.11537

[4] C. Xu, M. Chen, P. Deshpande, E. Azanli, R. Yang, and J. Ligman, "Enabling data-driven and empathetic interactions: A context-aware 3d virtual agent in mixed reality for enhanced financial customer experience," 2024. [Online]. Available: https://arxiv.org/abs/2410.12051

[5] Z. Li, C. Gebhardt, Y. Inglin, N. Steck, P. Streli, and C. Holz, "Situationadapt: Contextual ui optimization in mixed reality with situation awareness via llm reasoning," 2024. [Online]. Available: https://arxiv.org/abs/2409.12836

[6] A. Farasin, F. Peciarolo, M. Grangetto, E. Gianaria, and P. Garza, "Real-time object detection and tracking in mixed reality using microsoft hololens," in *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2020) - Volume 4: VISAPP*, INSTICC. SciTePress, 2020, pp. 165–172.

[7] D.-H. Kim, Y.-G. Go, and S.-M. Choi, "An aerial mixed-reality environment for first-person-view drone flying," *Applied Sciences*, vol. 10, no. 16, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/16/5436

[8] M. Łysakowski, K. Żywanowski, A. Banaszczyk, M. R. Nowicki, P. Skrzypczyński, and S. K. Tadeja, "Real-time onboard object detection for augmented reality: Enhancing head-mounted display with yolov8," 2023. [Online]. Available: https://arxiv.org/abs/2306.03537

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016. [Online]. Available: https://arxiv.org/abs/1506.02640

[10] R. Varghese and S. M., "Yolov8: A novel object detection algorithm with enhanced performance and robustness," in *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, 2024, pp. 1–6.

[11] G. Jocher and J. Qiu, "Ultralytics yolo11," 2024. [Online]. Available: https://github.com/ultralytics/ultralytics

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016. [Online]. Available: https://arxiv.org/abs/1506.01497

[13] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2014. [Online]. Available: https://arxiv.org/abs/1311.2524

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[15] Y. Zhao, W. Lv, S. Xu, J. Wei, G. Wang, Q. Dang, Y. Liu, and J. Chen, "Detrs beat yolos on real-time object detection," 2024. [Online]. Available: https://arxiv.org/abs/2304.08069

[16] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," 2020. [Online]. Available: https://arxiv.org/abs/2005.12872

[17] Z. Zong, G. Song, and Y. Liu, "Detrs with collaborative hybrid assignments training," 2023. [Online]. Available: https://arxiv.org/abs/2211.12860

[18] T. Cheng, L. Song, Y. Ge, W. Liu, X. Wang, and Y. Shan, "Yolo-world: Real-time open-vocabulary object detection," 2024. [Online]. Available: https://arxiv.org/abs/2401.17270

[19] Google, "Gemini: A family of highly capable multimodal models," 2024. [Online]. Available: https://arxiv.org/abs/2312.11805

[20] J. Bai, S. Bai, S. Yang, S. Wang, S. Tan, P. Wang, J. Lin, C. Zhou, and J. Zhou, "Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond," 2023. [Online]. Available: https://arxiv.org/abs/2308.12966

[21] doubao, 2024. [Online]. Available: https://team.doubao.com/en/direction/multimodal

[22] OpenAI(2023), "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[23] MagicLeap, 2023. [Online]. Available: https://www.magicleap.com/

[24] Unity Technologies, "Unity," 2023, game development platform. [Online]. Available: https://unity.com/

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019. [Online]. Available: https://arxiv.org/abs/1912.01703

[26] O. Community, "Open neural network exchange (onnx)." [Online]. Available: https://github.com/onnx/onnx

[27] M. Minderer, A. Gritsenko, A. Stone, M. Neumann, D. Weissenborn, A. Dosovitskiy, A. Mahendran, A. Arnab, M. Dehghani, Z. Shen, X. Wang, X. Zhai, T. Kipf, and N. Houlsby, "Simple open-vocabulary object detection with vision transformers," 2022. [Online]. Available: https://arxiv.org/abs/2205.06230

[28] S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, Q. Jiang, C. Li, J. Yang, H. Su, J. Zhu, and L. Zhang, "Grounding dino: Marrying dino with grounded pre-training for open-set object detection," 2024. [Online]. Available: https://arxiv.org/abs/2303.05499

[29] T. Zhao, P. Liu, X. He, L. Zhang, and K. Lee, "Real-time transformer-based open-vocabulary detection with efficient fusion head," 2024. [Online]. Available: https://arxiv.org/abs/2403.06892

[30] S. Ramírez, "FastAPI." [Online]. Available: https://github.com/fastapi/fastapi

[31] M. Grinberg, *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.", 2018.

[32] Django Software Foundation, "Django." [Online]. Available: https://djangoproject.com

[33] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 11 2019. [Online]. Available: https://arxiv.org/abs/1908.10084

[34] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[35] A. Gupta, P. Dollár, and R. Girshick, "Lvis: A dataset for large vocabulary instance segmentation," 2019. [Online]. Available: https://arxiv.org/abs/1908.03195

[36] LandingAI, "Agent-object-detection," 2025. [Online]. Available: https://landing.ai/agentic-object-detection

[37] L. Liu, J. Feng, H. Chen, A. Wang, L. Song, J. Han, and G. Ding, "Yolo-uniow: Efficient universal open-world object detection," 2024. [Online]. Available: https://arxiv.org/abs/2412.20645

Appendix A

# Prompts

## A.1 Prompt to Class Name Conversion

```
"""
You're a sharp-eyed visual assisstant, and I want you to help me find objects.
↪  Specifically, I want you to help me extract the object(s) I'm looking for based on
↪  a text. If there are multiple objects, separate them with commas.
Example: I'm looking for a man in a red T-shirt
Output: man in red T-shirt
Example: Have you seen a bottle or a cup?
Output: bottle, cup
Example: I'd like to buy oranges
Output: oranges
Example: Can you look for sunflowers, chocolate, and boots?
Output: sunflower, chocolate, boots
Example: Have you seen my phone? It's black with a pink phone case
Output: black phone with pink phone case
Example: I lost my glasses somewhere near me.
Output: glasses
Example: I cannot find my backpack with a yellow, shiny ornament on it
Output: backpack, yellow and shiny ornament
Example: Did you see that dog wearing a sweater?
Output: Dog wearing a sweater
The output should be the answer only, truncate anything unrelated to it.
"""
```

## A.2 Prompt VLM Guidance

```
"""
You are an intelligent assistant helping a user locate specific objects in a 3D indoor
↪  space. The user is wearing a headset that captures frames of the environment.
↪  Based on the task description provided by the user, your role is to:
    1. Analyze the recent scene as summarized from the captured frame history.
    2. Identify and assess the objects detected in the frames, including how closely
    ↪  they match the user's objectives, using confidence scores.

Based on your analysis, you should generate concise, human-readable instructions
↪  that:
    1. Suggest where to look next by offering simple and straight forward exploration
    ↪  tips (e.g., "Look around", "Move forward," "Look to your left," "Search near
    ↪  the table").
    2. Avoid technical jargon and ensure the guidance feels intuitive and friendly.

If the text input's DETECTED OBJECT section contains objects found by the detection
↪  model, describe the position of them relative to the user's current pose assuming
↪  the user is facing the captured frame in the forward direction
Combine your knowledge with the detection model's result, and the user's past
↪  actions.

Example outputs:
''Look closer to ...''
''The [object you are looking for] on the table to your left.''
''Go straight ahead and look on the shelf.''
''The [object you are looking for] is behind you.''
"""
```

Appendix B

# Configurations

## B.1 Agent Configuration

```yaml
---
max_workers: 10
embedder: # only text embedder now
  model_class: 'sentence_transformer'
  model_name: 'sentence-transformers/all-MiniLM-L6-v2'
  api_url:
  ↪  'https://api-inference.huggingface.co/models/sentence-transformers/all-MiniLM-L6-v2'
  api_key: ...
  hub: True
  max_sentences: 100
language_model:
  model_class: 'gemini'
  model_name: 'gemini-1.5-flash'
  api_key: ...
  system: ['llm_prompt2cls_instruct']
  predict_kwargs:
    response: 'instruct'
    max_tokens: 600
vision_model:
  model_class: 'yolo'
  model_name: 'yolo-world'
  model_weight_path: 'src/models/yolo/weights/yolov8l-world.pt'
  mode: ['detection']
  initial_classes: ['']
  predict_kwargs:
    verbose: False
    stream: True
    conf: 0.3
vl_model:
    system: 'vlm_guidance_instruct'
    model_class: 'gpt' #'blip'
    model_name: 'gpt-4o-mini' # 'Salesforce/blip2-opt-2.7b'
    api_key: ...
    use_model_config: False
    max_new_tokens: 51 # default on HF
    add_image_token: False
    response_type: null
    predict_kwargs:
      response_type: 'captioning'
      response_format: null
      max_tokens: 600
```

## B.2 Server Configuration

```yaml
---
db:
  db_name: object_detection
  collection_name: detections
  systemLog:
    destination: file
    path: "/var/log/mongodb/mongod.log"
    logAppend: true
  processManagement:
    fork: true
  net:
    bindIp: 127.0.0.1
    port: 27017
  setParameter:
    enableLocalhostAuthBypass: false

logger:
  version: 1
  disable_existing_loggers: false
  formatters:
    default:
      format: "[%(asctime)s] [%(levelname)s] [%(name)s] %(message)s"
      datefmt: "%Y-%m-%d %H:%M:%S"
  handlers:
    console:
      class: logging.StreamHandler
      formatter: default
      level: INFO
  loggers:
    my_app_logger:
      handlers: ["console"]
      level: INFO
      propagate: false
    uvicorn:
      handlers: ["console"]
      level: INFO
      propagate: false

server:
  detection_buffer_size: 100
  host: 127.0.0.1
  port: 8000
  reload: true
```

Appendix C

# Agent Implementation

```python
class Agent():
    """
    The agent performes vision and language processing from observed env information

    The methods are named as 'process_[task_name]'
    """
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self._init_models()
        print("Agent initialized")

    @property
    def language_model(self) -> LanguageModel:
        return self._language_model

    @property
    def vision_model(self) -> VisionModel:
        return self._vision_model

    @property
    def vlm(self) -> VLM:
        return self._vlm

    @property
    def embedder(self) -> Embedder:
        # only text embedder now
        return self._embedder

    def _init_models(self):
        self._language_model: LanguageModel =
        ↪  resolve_language_model(self.cfg.language_model)
        self._vision_model: VisionModel = resolve_vision_model(self.cfg.vision_model)
        self._vlm: VLM = resolve_vlm(self.cfg.vl_model)
        self._embedder: Embedder = resolve_embedder(self.cfg.embedder)

    def process_summarization(self, instruct, scene) -> None:
        """scene: an image or a list of observed images"""
        return self._summmarize_scene(instruct, scene)

    def process_task_instructions(self, prompt: str, type: str = 'detection'):
        """
        post init process, called upon receiving user instructions
        """
```

```
    if type == 'detection':
        self._set_new_class_from_prompt(prompt)
    else:
        raise NotImplementedError(f"task instruction \"{type}\" not supported
        ↪ yet")

def process_chat(self, user_msg):
    return self._respond_to_msg(user_msg)

def process_detection(self, frames: list):
    """
    returns the results processed by the vision model
    """

    results = self.vision_model.predict(frames[0])

    processed_frames = []
    for result in results:
        boxes = result.boxes
        for box in boxes:
            processed_frames.append(
                {
                    "xyxy": map(int, box.xyxy[0]),
                    "conf": float(torch.round(box.conf[0], decimals=3)),
                    "cls": self._class_id_to_class_name(int(box.cls[0])),
                }
            )

    return processed_frames

def _respond_to_msg(self, user_msg):
    return self.language_model.predict(user_msg)

def _summmarize_scene(self, summary_instruct, scene):
    return self.vlm.predict(summary_instruct, scene)

def _prompt_to_class_names(self, prompt: str, **kwargs):
    """
    Extracts class names from the user prompt based on the prompt-to-class system
    ↪ principles
    Returns a string of llm-generated class names, where the names are
    ↪ comma-separated
    """
    return self.language_model.predict(prompt, **kwargs)

def _class_id_to_class_name(self, cls_id):
    if cls_id < len(self.vision_model.names):
        class_name = self.vision_model.names[cls_id]
    else:
        class_name = "Unknown"
    return class_name

def _set_new_class_from_prompt(self, prompt: str):
    """
    Converts the explicitly given string of class names into a list of class
    ↪ names
    """
```

```
assert hasattr(self.vision_model, 'set_classes'), 'cannot set classes for
↪  vision model'
new_classes = self._prompt_to_class_names(prompt).split(",")
self.vision_model.set_classes(new_classes + ['']) # adding a background class
↪  to improve performance
print(f"Updated classNames to {new_classes}")
```