POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Semantic Object Navigation in Edge Robotics: Fusing Open-Vocabulary Perception and LLM Orchestration

Supervisors

prof. Marcello Chiaberge
Ph.D. Mauro Martini
Ph.D. Francesco La Carpia
Dott. Eng. Ilario Gerlero

Candidate

Francesco Masin

Academic Year 2025/2026

Abstract

Achieving human-like, effortless navigation in mobile robots requires a conceptual understanding that extends beyond mere obstacle avoidance. This capability, particularly in *Object Navigation* (ObjectNav) tasks, demands both a comprehensive spatial awareness of the environment and a deep semantic comprehension of user instructions. Because natural language commands are often unstructured, Large Language Models (LLMs) and Vision Language Models (VLMs) are uniquely suited to interpret these inputs and extract actionable goals for the robot. However, deploying such high-fidelity perception and spatial reasoning systems on resource-constrained hardware presents significant computational challenges.

This thesis presents an end-to-end object navigation pipeline for indoor environment that is optimized for edge deployment, utilizing an LLM framework for high-level behavioral orchestration.

The perception stack is driven by an open-vocabulary architecture, which enables zero-shot, prompt-driven object detection. These detections are subsequently processed by SigLIP-2 to extract embeddings, in order to create an efficient, lightweight visual memory. This persistent semantic storage allows the system to retroactively query for specific objects using also visual features, enabling the retrieval and localization of instances even when the target objects are outside the active field of view. To ground these features physically, 2D detections are projected into a 3D space, resolving multi-view ambiguities and preventing instance duplication to maintain a stable, dynamically updated 3D map. The consolidated spatial data and persistent visual embeddings are then serialized into a lightweight structured format state representation.

This textual map serves as the physical grounding mechanism for the cognitive layer. The integrated LLM uses the serialized map to perform spatial reasoning, acting as the central orchestrator that dictates the robot's high-level task planning. Guiding the execution of these plans there is a behavior tree designed to manage all aspects of semantic navigation based on the LLM's instructions. The behavior tree is structured to mimic human search strategies. In order to achieve this the agent must leverage *prior semantic knowledge* to explore the environment intelligently. By clustering mapped objects and evaluating the semantic relevance of those clusters, the system infers the most probable locations of the hidden target.

Experimental evaluations are conducted in both simulated and real-world indoor environments using a Clearpath Jackal robot equipped with an NVIDIA Jetson AGX Thor platform. To rigorously validate the perception and mapping pipeline in simulation the photorealistic Matterport3D dataset have been used. This framework

effectively narrows the sim-to-real gap, demonstrating a robust architecture that has a success rate in the ObjectNav task of 76.67% across multiple testing environments.

Table of Contents

List of Tables VI

List of Figures VII

1 Introduction 1
- 1.1 Thesis Objective . . . 2
- 1.2 Thesis Structure . . . 2

2 Background Concepts and Related Works 5
- 2.1 Semantic Navigation . . . 5
  - 2.1.1 Robot Perception and Scene Understanding . . . 5
  - 2.1.2 Processing Demands and Object Recognition Models . . . 6
  - 2.1.3 Evolution of Navigation Paradigms . . . 6
  - 2.1.4 Limitations of Trained Policies . . . 7
- 2.2 Foundation Models . . . 8
  - 2.2.1 The Emergence of Foundation Models . . . 8
  - 2.2.2 Foundation Models in Robotics . . . 8
  - 2.2.3 Current Limitations and Deployment Challenges . . . 9
- 2.3 LLMs and VLMs for Robotic Planning and Navigation . . . 10
  - 2.3.1 Human-Like Semantic Reasoning . . . 10
  - 2.3.2 Applications in Vision and Language Navigation . . . 11
  - 2.3.3 Current Challenges in LLM-Guided Navigation . . . 12
  - 2.3.4 Review of Relevant Architectures . . . 13
  - 2.3.5 Summary and Identified Limitations . . . 17
- 2.4 Open-Vocabulary Semantic Understanding . . . 18
  - 2.4.1 Contrastive Language-Image Pre-training (CLIP) . . . 19
  - 2.4.2 Review of Relevant Architectures . . . 21
  - 2.4.3 Summary and Identified Limitations . . . 27
  - 2.4.4 Architectural Differences Between CLIP and SigLIP . . . 28

3 Simulation Environment Setup 31
3.1 Sim-To-Real Gap in Semantic Navigation . . . 31
3.2 Preliminary Baseline: Gazebo Classic . . . 32
3.2.1 Technical Advantages of Game-Based Scenes . . . 32
3.2.2 Technical Limitations and the Sim-to-Real Bottleneck . . . 33
3.3 Photorealistic Simulation: Habitat-Matterport 3D (HM3D) . . . 34
3.3.1 Technical Advantages of World-Based Scenes . . . 34
3.3.2 Technical Limitations and Segmentation Bottlenecks . . . 35
3.4 Simulation Framework and Custom Middleware Architecture . . . 36
3.4.1 Habitat-Sim . . . 36
3.4.2 Habitat-ROS 2 Interoperability Bridge . . . 36
3.4.3 Coordinate System Transformations . . . 36
3.4.4 Sensor Data Serialization and Semantic Processing . . . 37
3.4.5 Ground-Truth Emulation for Algorithm Evaluation . . . 38

4 System Architecture 39
4.1 Open-Vocabulary perception . . . 39
4.1.1 Problem Setting and Requirements . . . 39
4.1.2 Detection and Segmentation . . . 40
4.1.3 Semantic Encoding . . . 42
4.1.4 Node Architecture and Implementation . . . 43
4.2 Persistent Semantic Mapping . . . 47
4.2.1 Depth Denoising and Geometric Back-Projection . . . 48
4.2.2 Geometric Representation with Oriented Bounding Boxes . . . 52
4.2.3 The Life Cycle of a Mapped Object . . . 53
4.2.4 Map Serialization . . . 58
4.3 Natural-Language Command Interpretation . . . 59
4.3.1 Background: Large Language Models . . . 60
4.3.2 From Instruction to Structured Command . . . 61
4.4 Autonomous Navigation Orchestration . . . 65
4.4.1 Behavior Tree [35] . . . 65
4.4.2 Implementation . . . 66

5 Experimental Results 73
5.0.1 Hardware Experimental Setup . . . 73
5.0.2 Results and Discussion . . . 76

6 Conclusions and Future Works 81
6.1 Future Works . . . 81

Bibliography 83

List of Tables

5.1 Timing statistics for the LLM script 77
5.2 Timing statistics for the perception module 78
5.3 Timing statistics for the mapper module 78

List of Figures

2.1 Three approaches to semantic navigation 6
2.2 Framework of ChatNav 14
2.3 Annotated Semantic Map in MapNav 15
2.4 OsmAG-LLM mapping method 16
2.5 Summary of CLIP Architecture 20
2.6 ConceptGraphs pipeline 21
2.7 Map structure of HOV-SG 23
2.8 OVO pipeline 25
2.9 Segmentation Example in FindAnything 26

3.1 AWS small house simulation layout 32
3.2 Habitat Matterport3D dataset 34
3.3 Coordinate transformations 37

4.1 SAM Oversegmentation Example 41
4.2 YOLOE pipeline 42
4.3 Example of the vision node processing using a single detection . . . 43
4.4 Example of the generated semantic map 48
4.5 Pinhole projection 50
4.6 Block diagram for filtering process in mapper node 52
4.7 Comparison between Axis Aligned Bounding Box (on the left) and Oiented Bounding Box (on the right) 54
4.8 object lifecycle diagram 54
4.9 Full pipeline that goes from LLM processing to persistant map . . . 60
4.10 behavior tree 67
4.11 Watershed example 70

5.1 Jackal Dimensions 74
5.2 Intel Realsense D435i camera 74
5.3 Jetson Thor 75
5.4 Laboratory setup 76

5.5 Top down views of the used environment 79

Chapter 1

Introduction

Autonomous navigation has been a central challenge in robotics since the inception of the field. The ability to localize and move within an environment has been one of the primary research objectives for decades, yielding excellent results that now underpin virtually every robotic system.

However, true autonomy requires more than geometric information and obstacle avoidance. The rapid development of Artificial Intelligence and the increasing suitability of AI models for online deployment has shifted interest toward solutions that incorporate semantic information directly into the navigation pipeline. From this emerged the concept of *semantic navigation*: a mandatory step toward genuine robot autonomy, going beyond classical navigation solutions [1].

Achieving semantic navigation requires an agent to address a rich set of interconnected questions. Some of these, like localization, motion control, and path planning, are inherited from classical navigation and have been studied for years. Introducing semantics, however, requires rethinking and reformulating most of them in a more complex manner. Agents must integrate object recognition and scene understanding seamlessly, and exploit this information to accomplish a wide variety of tasks.

An additional core aspect of semantic navigation is task reception. Semantic navigation is inherently tied to human-robot communication: the goal of this work is to bridge the gap between natural language instructions issued by a non-expert user and the structured, formal task representations required by robotic planning systems. This enables an end-to-end solution, from the reception and interpretation of a freely phrased instruction to its physical execution.

Early proposed solutions to the semantic navigation problem, however, remained significantly limited and still far from the efficient and flexible way in which humans navigate and explore unknown environments. The primary limitation lies in the perception models used to infer semantics: these were constrained to a closed set of object categories defined at training time. This limited the applicability

and made moving to a new deployment scenario tedious and time-consuming [2]. Furthermore, most systems treated the environment as unknown until directly observed, discarding the rich prior knowledge that humans naturally exploit when navigating, such as the expectation that a kettle is likely to be found in a kitchen, or that a bedroom is unlikely to contain a printer [3].

This thesis addresses both limitations. By adopting an *open-vocabulary* perception framework, the system is not restricted to predefined object categories, and by leveraging a Large Language Model as a source of world knowledge, the agent is equipped with semantic priors that guide exploration before direct observation [2].

1.1 Thesis Objective

The goal of this work is to develop an end-to-end semantic navigation system capable of receiving and executing tasks expressed in natural language, with no assumptions on a predefined object vocabulary.

To this end, the thesis makes the following contributions:

- An **open-vocabulary perception pipeline** that enables the agent to detect and identify objects beyond any fixed set of categories, allowing deployment in arbitrary environments without retraining.
- A **lightweight visual memory** that allows finding specific features in mapped objects without direct observation.
- A **semantic prior framework** based on a Large Language Model, which provide the agent with the world knowledge used to guide exploration and environment understanding before direct observation.
- An **end-to-end architecture** integrating natural language task reception, semantic environment understanding, and goal-directed navigation into a unified system.

1.2 Thesis Structure

The thesis is organized into 6 chapters, summarized as follows:

- **Chapter 2: Background Concepts and Related Works**: This chapter provides a more technical introduction to the various aspects of semantic navigation, supplying the theoretical background necessary for understanding the problem and reviewing currently available solutions.

- **Chapter 3: Simulation Environment Setup**: This chapter describes the simulation environment used during development. It is a key component for understanding the types of environments the system was designed for.
- **Chapter 4: System Architecture**: The central chapter of the thesis, presenting the methodologies adopted to achieve the desired result.
- **Chapter 5: Experimental Results**: Presents the experimental results.
- **Chapter 6: Conclusions and Future Works**: Presents the conclusions drawn from the work carried out and outlines possible future developments.

Chapter 2

Background Concepts and Related Works

2.1 Semantic Navigation

As stated in Chapter 1, in robotics, true autonomous navigation requires more than mere obstacle avoidance; it necessitates a comprehensive understanding of the surrounding area. Therefore, Semantic Navigation represents the natural and necessary step towards robot autonomy, moving from pure geometric path planning and obstacle avoidance to conceptually aware movement. This is done by incorporating high-level understanding of surrounding objects and places into a robot's navigation framework. This conceptual understanding aims to replicate the effortless navigation capabilities of humans. For instance, when a human attempts to find a glass of water in a previously unseen domestic environment, they naturally leverage semantic priors to locate the kitchen without necessarily searching other areas. By leveraging this form of prior knowledge, mobile robots can infer contextual information about their environment, enabling better planning, localization, and decision making in uncontrolled spaces like homes, hospitals, and schools [3].

2.1.1 Robot Perception and Scene Understanding

To achieve these efficient navigation capabilities, a robotic system requires accurate, reliable, and robust perception skills. The primary objective of robot perception is to process and analyze semantic information within a scene. Crucially, true scene understanding extends beyond basic object detection and recognition; it involves a deeper analysis and interpretation of sensor data to grasp the context of the environment [1].

2.1.2 Processing Demands and Object Recognition Models

Robot semantic navigation systems necessitate the use of object recognition models to classify various types of objects within the area of interest. Researchers have developed and deployed various models for this purpose, ranging from customized Convolutional Neural Networks (CNNs) to established architectures like YOLO, MobileNet, ResNet, and Inception [4, 5, 6, 7]. However, the extraction of high-level features from RGB and RGB-D images introduces intensive processing requirements. Handling tasks such as object detection, recognition, and localization demands significant computational resources and, consequently, it is crucial to employ highly efficient processor units to manage the data coming from onboard sensors. While incorporating personal computers (such as laptops) directly onto the robot platform can fulfill these processing needs, it introduces notable drawbacks by adding substantial size, power consumption, cost, and mechanical complexity [1].

2.1.3 Evolution of Navigation Paradigms

Since the beginning of robotics, methodologies for autonomous navigation have been researched and ,with recent developments in Artificial Intelligence, many researchers have tried to integrate the semantic information coming from AI models previously mentioned with standard Mapping and Navigation methods. Researchers in this field have pursued three different approaches [8].

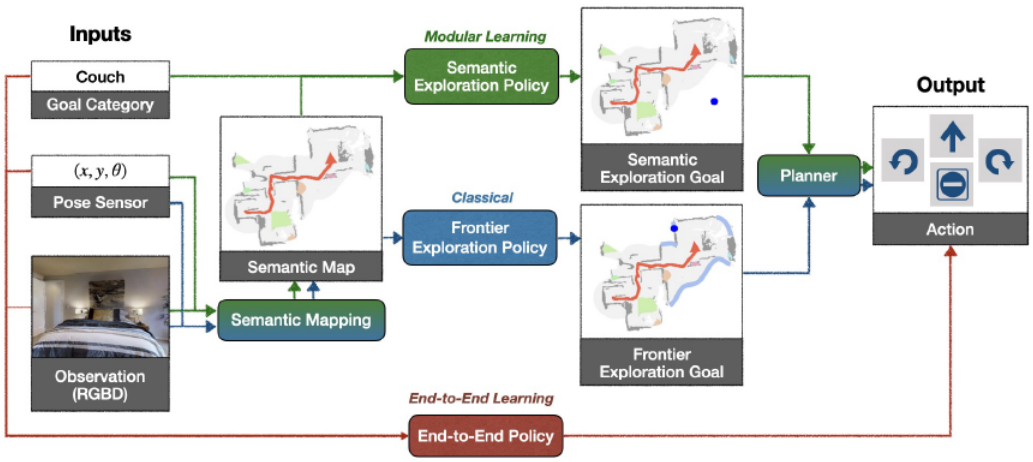


Figure 2.1: Three most common approaches to incorporate semantics into autonomous navigation [8]

- **Classical Approaches:** Many classical approaches to navigation require

access to pre-computed maps. For operations in unseen environments, classical methods typically construct a geometric map using depth sensors while simultaneously localizing the robot relative to this growing map. Building on this Simultaneous Localization and Mapping (SLAM) module, classical methods explore using heuristics, such as frontier-based exploration, and leverage analytical planners for low-level control. While Semantic SLAM methods naturally extend this to detect and keep objects in memory, they offer no inherent solution for efficient semantic exploration [8]. These approaches allow for semantic mapping that can later be leveraged for object navigation. However, these solutions do not integrate semantics into the exploration of the environment and do not allow for intelligent path planning if the object is not already present in the map.

- **End-to-End Learning Approaches:** Following recent advances in machine learning and computer vision, learning-based policies have emerged to learn these semantic priors. The most common methods employ deep neural networks (typically comprising a visual encoder followed by a recurrent layer for memory) to predict navigational actions directly from raw sensor observations. These end-to-end approaches are trained using backpropagation via Imitation Learning (IL) or Reinforcement Learning (RL) losses. However, results demonstrate the limitations of these approaches, which tend to perform well in simulation but drop significantly in the real world [8]. Recently, multiple researchers have tried to move from custom trained models to foundation models.

- **Modular Learning Approaches:** Following successful applications across robotics in autonomous driving, flight, and grasping, much of the current work focuses on designing modular learning approaches that aim to combine the benefits of both paradigms: leveraging the goal-directed exploration of End-to-End policies and the modularity of the classical pipeline that offers easier engineering and interpretability [8].

2.1.4 Limitations of Trained Policies

Despite the success of these learning-based and modular approaches, traditional trained semantic policies exhibit notable limitations. Because these end-to-end and RL/IL models are typically trained on closed, task-specific datasets, they often struggle with zero-shot generalization when deployed in truly novel, out-of-distribution environments. They are generally confined to a limited vocabulary of predefined object categories and lack the broad, generalized commonsense reasoning required to dynamically navigate complex, open-world scenarios [9]. Overcoming these limitations requires a much broader, generalized understanding of the world, one that cannot be easily achieved through standard reinforcement or imitation

training alone. To bridge this gap modern semantic navigation pipelines increasingly look toward the vast, open-vocabulary knowledge and reasoning capabilities provided by Foundation Models, which are detailed in Section 2.2.

2.2 Foundation Models

2.2.1 The Emergence of Foundation Models

The recent pursuit of *Artificial General Intelligence* (AGI) has driven the development of versatile, large-scale artificial intelligence architectures capable of executing a broad spectrum of downstream tasks without requiring task-specific labeled data. These architectures, denoted as "foundation models," represent a paradigm shift from specialized, narrowly tuned algorithms to highly generalizable systems [10].

Unlike traditional deep learning frameworks, which heavily relied on strictly constrained and manually annotated datasets, foundation models (such as BERT, GPT-3, and DALL-E [11, 12, 13]) are trained on vast corpora of raw, unlabeled data utilizing self-supervised learning techniques. This unlabeled training approach significantly alleviates the historical bottleneck of data acquisition and annotation. Consequently, while earlier neural networks achieved optimal results within their specific domains, they suffered from severe generalization limitations. Foundation models, by contrast, leverage their massive pre-training to adapt to novel scenarios with minimal fine-tuning [14].

2.2.2 Foundation Models in Robotics

Prior to the advent of foundation models, traditional robotics deep learning pipelines required the painstaking collection of distinct datasets for every specific task. Integrating foundation models removes these domain limitations, enabling the development of context-aware robotic agents with enhanced environmental perception and cognitive interaction capabilities, which is crucial to achieve high-level semantic navigation. This transformation is primarily evident across two distinct operational domains: *visual perception* and *high-level decision-making* [15].

Robotic Perception

In the perception domain, foundation models, particularly *Vision-Language Models*, have fundamentally changed how robots interpret their surroundings. Rather than relying on rigid, task-specific object detectors, these models provide comprehensive cross-modal understanding by learning deep associations between raw visual telemetry and textual semantics [16]. This enables robotic systems to move beyond

simple geometric mapping to achieve true semantic scene understanding, facilitating tasks such as zero-shot image classification and dynamic object detection. A notable result of this generalized training is the emergence of "open-vocabulary" capabilities; the system can comprehend and identify novel visual features based on natural language cues without being restricted to a predefined, hard-coded dataset. This allows the robot to contextually analyze its environment with unprecedented flexibility [15].

Decision-Making and High-Level Planning

Unlike visual processing, the decision-making and planning domains leverage *Large Language Models* (LLMs) as their cognitive engines for high-level reasoning. Operating in complex, unstructured environments requires robots to execute long-horizon tasks based on abstract human instructions. LLMs address this by acting as an interpretive bridge: parsing ambiguous or complex user prompts, extracting valuable semantic information, and breaking the overarching goal down into sequential, executable steps. By evaluating contextual data, LLMs construct structured task specifications. A prominent example of this integration is the SayCan architecture [17], which utilizes an LLM to generate high-level plans based on natural language commands, while simultaneously employing a learned value function to physically ground these abstract instructions into the kinematic and environmental constraints of the robot.

2.2.3 Current Limitations and Deployment Challenges

Despite their capabilities in natural language processing and computer vision, deploying large foundation models in real-world robotics introduces multiple critical challenges. Unlike pure software environments, robotics requires physical embodiment, making the generalization and fine-tuning of these models particularly complex. A primary limitation is *uncertainty quantification*; models often suffer from "hallucinations" and struggle with out-of-distribution scenarios or dynamic environmental shifts [15]. Additionally, their massive computational size imposes severe real-time performance constraints. The multi-second inference latency of typical VLMs and LLMs is currently incompatible with the high-frequency control loops required by mobile robots. Finally, these models lack physical grounding [15]. Because translating semantic reasoning into safe kinematic trajectories remains challenging, the proposed solution continues to rely on traditional robotics architectures for direct control and movement.

Efficient Alternatives for Edge Computation

Addressing these challenges requires a rigorous evaluation of model capabilities prior to integration. For this research, the primary objective is to achieve deep semantic understanding of dynamic scenes and objects within the robot's field of view. Consequently, it is necessary to implement a multimodal architecture. The selected model must not only process raw visual telemetry but also possess the cognitive capacity to comprehend complex user prompts, extract contextually valuable information, and map those semantic concepts directly to the physical environment, all while running efficiently on edge hardware.

To determine the optimal architecture for this pipeline, several prominent works in the field were evaluated. Multiple researchers have tried to implement LLMs and VLMs into robotics pipelines, the most relevant solutions will be studied in 2.3. However, large generative LLMs and VLMs were discarded due to their prohibitive inference times and massive computational footprints, the selection focused also on lightweight, discriminative models capable of real-time, open-vocabulary perception. To achieve deep semantic understanding while respecting the strict hardware constraints of robotic edge computers, models based on contrastive vision-language pre-training emerged as the ideal solution. Specifically, the *Contrastive Language-Image Pre-training* (CLIP) [16] architecture and its highly optimized successor, SigLIP (*Sigmoid Loss for Language Image Pre-Training*) [18], provide the necessary zero-shot classification capabilities without the latency of text generation of VLMs. The architectural details, specific advantages, and integration of these models into the navigation pipeline will be comprehensively explored in the following sections.

2.3 LLMs and VLMs for Robotic Planning and Navigation

In object goal navigation tasks, the robot's ability to understand and leverage semantic relationships within its environment is a key factor to successfully localizing target objects. Conventional navigation systems typically depend on preprogrammed rules and raw sensor data, an approach that often overlooks the subtleties and complexities of real-world environments [19]. To bridge the gap between rigid geometric mapping and adaptive autonomy, modern research has increasingly turned toward integrating high-level semantic reasoning into robotic control loops.

2.3.1 Human-Like Semantic Reasoning

As previously mentioned, traditional learning-based methods are inherently limited by their training data. They can only recognize the specific semantic relationships

explicitly provided during the training phase. Consequently, when deployed in unfamiliar settings characterized by novel semantic contexts or out-of-distribution spatial layouts, these models perform poorly [2].

On the other hand, humans possess a rich, generalized understanding of diverse semantic relationships that enables advanced, spontaneous reasoning. For instance, if a human recognizes a space as a living room, they can immediately image that the room might contain a television or a sofa. If a robot could perceive and reason about semantic information from the objective world In this human-like manner, it would significantly enhance its navigation performance [2].

Recent breakthroughs in artificial intelligence have created new opportunities to achieve this human-robot cognitive parity. The rapid advancement of Large Language Models (LLMs) and Vision-Language Models (VLMs) (such as BERT, LLaMA, and the GPT series) has garnered widespread attention across research domains. Particularly since the emergence of ChatGPT-3.5, the powerful knowledge summarization and contextual understanding capabilities of these models have demonstrated profound potential. By leveraging the vast corpora of text these models are trained on, robotic systems can perform zero-shot inference of explicit semantic relationships and spatial layouts without requiring additional domain-specific retraining [2].

2.3.2 Applications in Vision and Language Navigation

The integration of these foundation models into robotics has opened new frontiers in task planning, multimodal reasoning, human-robot interaction, and autonomous driving [19]. A prominent area of interest within this intersection is *Vision and Language Navigation* (VLN). VLN integrates visual perception with natural language understanding to enable embodied agents to navigate an environment based on written (or spoken) instructions.

Sice LLMs already possess knowledge about the semantic priors of everyday environments, they exhibit exceptional high-level planning capabilities in VLN contexts. They can analyze contextual information, anticipate potential hazards [19], and offer highly adaptive search strategies. For example, while an LLM might generally suggest searching for scissors in a kitchen based on common household priors, providing the robot with a broader contextual understanding of the building allows the LLM to infer that the scissors might be found much quicker in an adjacent workshop [20]. This ability to dynamically adjust search heuristics based on semantic context is a massive leap forward for autonomous exploration.

2.3.3 Current Challenges in LLM-Guided Navigation

Despite this significant potential, using text-based foundation models to guide robot navigation proves to be a novel and unconventional task, introducing several critical deployment challenges. Main limitations related to the use of foundation models in robotics have been already stated in Section 2.2.3, now those limitations will be related to the scope of LLM-Guided Navigation:

- **The Semantic-to-Kinematic Gap:** A primary obstacle in VLN arises from the fact that LLMs frequently lack *physical grounding*, since Models are trained purely on internet text or static images, thus they lack an inherent understanding of physical laws (e.g., gravity, friction, mass). For this reason LLMs cannot handle the low-level movement guidance required for actual robotic kinematics and obstacle avoidance. Translating semantic understanding into safe, executable kinematic trajectories remains an ongoing area of research [15].
- **Prompt Engineering and Hallucinations:** Because navigating a physical space is an inherent domain shift for an LLM trained primarily on internet text, it can sometimes struggle with reliability. This manifests as (i) instance-level uncertainty, including language ambiguity or "hallucinations" where the model confidently predicts incorrect actions; (ii) distribution-level uncertainty, where the model encounters out-of-training scenarios; and (iii) distribution-shift, where the real-world physical environment dynamically changes in ways the static training data cannot account for [15]. Consequently, designing prompts capable of consistently steering an LLM toward rational, safe, and physically executable inferences poses a significant engineering challenge [2].
- **Over-Reliance on Map Knowledge:** Previous works on LLM-guided object navigation often rely on highly descriptive, explicit language instructions (e.g., “find a bean bag in the office on the first floor”). These requests inherently embed significant map knowledge into the task input. However, in true autonomous exploration, this knowledge may not be available to the user, or the environment might be too large and complex to provide such explicit guidance [20].
- **Latency and Power Constraints:** Large generative models are computationally massive, and running them on-board a mobile robot incurs inference times often on the order of seconds per query. Such latency is incompatible with the high-frequency control loops required for smooth, reactive navigation, forcing the agent into a disjointed "stop-and-think" behavior. Moreover, the sustained computational load imposes a significant power cost, which is a critical concern for battery-powered platforms operating under a fixed energy

budget. These constraints make the direct deployment of heavy LLMs and VLMs in the online navigation loop impractical on resource-constrained edge hardware [15].

Bridging this gap between abstract, high-level semantic reasoning and autonomous, low-level environmental exploration remains a central focus of current research. To address these challenges, numerous recent architectures have proposed novel methods for integrating LLMs and VLMs into the navigation pipeline, which are examined in Section 2.3.4.

2.3.4 Review of Relevant Architectures

To address the challenges of prompt engineering, map reliance, and the semantic-to-kinematic gap, numerous recent architectures have proposed novel integration strategies.

ChatNav [2]

One notable approach is ChatNav, an object goal navigation pipeline that leverages the zero-shot reasoning capabilities of LLMs to infer explicit semantic relationships in unknown environments. The ChatNav architecture operates in three sequential stages: semantic cue collection, LLM reasoning, and goal point selection. To overcome the lack of scene priors, the system dynamically builds a semantic map, a MxM 2D matrix that contains accumulated observations in which each numerical entry in the map corresponds to different meanings: 1 and 2 are explored areas and obstacles, and other numbers correspond to different categories of objects. Objects detected are limited to 15 common objects and the detection is performed by a fine tuned MASK-RCNN [2].

All detected objects are grouped using a spatial *clustering algorithm*. These object clusters then serve as the contextual semantic cues fed into the LLM. The clustering is decided on 3 conditions: 1) Objects are spatially close to each other, 2) No obstacle separates the objects, 3) there is a known connecting passage between the objects. To ensure that clustering satisfies all three conditions, a strategy based on object dilation is designed. Each region is then translated in a formatted text that includes the number of the region and the objects contained [2].

The obtained cues from the candidate regions interact with the ChatGPT model by calling the API online. Additionally, ChatNav proposes a fallback solution for offline reasoning that uses a fine-tuned local model based on the pre-trained RoBERTa-Large model [2].

A key innovation of ChatNav lies in its *prompt engineering strategy*. Recognizing that generative models often produce irrational outcomes when tasked with spatial

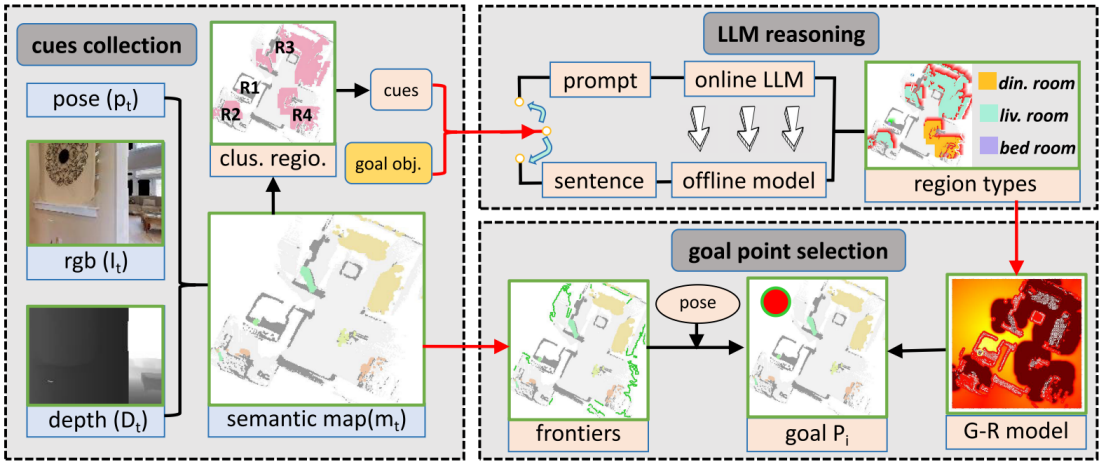


Figure 2.2: Framework of ChatNav [2]

robot navigation, the system translates the robotic task into a familiar, common-sense dialogue. Instead of prompting the LLM as a robot navigating a geometric map, the model is instructed to act as a "mover" placing "packages" (representing the clustered candidate regions and their detected items) into specific room types [2].

Finally, to bridge the semantic-to-kinematic gap, the LLM's high-level spatial inferences are not used as direct movement waypoints. Instead, they are transformed into a gravity-repulsion field. Target-related regions exert an attractive force, while unrelated regions exert a repulsive force. This field seamlessly overlays onto the robot's geometric map, guiding a traditional frontier-based exploration algorithm toward the target object without requiring complex, LLM-generated kinematic commands [2].

Regarding the implementation, ChatNav is evaluated entirely in simulation, within the Habitat simulator on the HM3D dataset and the Gibson dataset [2].

MapNav [21]

While architectures like ChatNav translate map data into text-based common-sense prompts, other recent methodologies attempt to feed structural map data directly into Vision-Language Models (VLMs). A prominent example is MapNav, an end-to-end architecture that addresses the severe computational and storage overhead associated with processing long sequences of historical RGB frames. To achieve this, MapNav replaces traditional historical visual buffers with an *Annotated Semantic Map* (ASM), which serves as a highly structured, lightweight memory representation.

The MapNav pipeline operates by continuously updating this ASM during

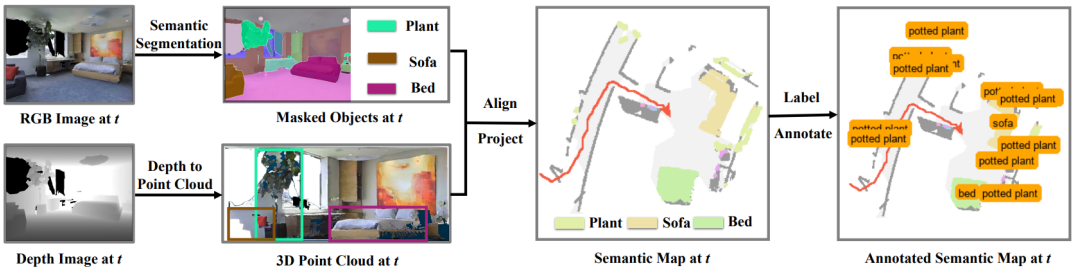


Figure 2.3: ASM Generation Process [21]

exploration. At each timestep, the system converts raw RGB-D observations, coming from a close-set segmentation model (Mask2Former), and pose data into a 3D point cloud, projecting it into a top-down semantic map. To make this abstract spatial data explicitly comprehensible to a VLM, the architecture performs region clustering and overlays physical textual labels (e.g., "chair," "plant," "bed") directly onto the corresponding geometric areas on the map.

For its VLM integration, MapNav utilizes a multimodal architecture based on LLaVA-Onevision that processes the current egocentric RGB frame and the annotated top-down ASM concurrently. These visual inputs are passed through a shared vision encoder, projected into a shared embedding space, and aligned with the user's natural language instructions. Rather than outputting exploration frontiers or target waypoints to a classical path planner, the VLM in MapNav acts as an end-to-end continuous controller. It directly leverages its language capabilities to predict the next executable navigation action (such as "move forward" or "turn left") in natural language, which is subsequently parsed into discrete low-level robotic commands via pattern matching capable of recognizing various linguistic expressions of the same navigational intent [21].

The pipeline has been tested in simulation using the R2R and RxR datasets and in a real setup composed by five environments: office, conference room, lecture hall, living room and tea room. For the real world testing the computation was performed on a server powered by NVIDIA A100 GPUs [21].

osmAG-LLM [20]

In contrast to MapNav's dense representation, osmAG-LLM explores purely textual maps to exploit the language-processing strengths of LLMs. It extends *osmAG*, a hierarchical, topometric map in the *OpenStreetMap* (OSM) textual format, to the object level. The guiding idea is that a high-detail object map quickly becomes outdated as objects are moved, so the map is treated as a lightweight source of context, combined with the semantic priors of an LLM and refined by active online

perception [20].

The map is built offline by augmenting a base osmAG (drawn manually in the JOSM editor) with two complementary node types extracted from RGB-D trajectories. *Object-nodes* are produced by the *LabelMaker* pipeline (with *Mask3D* for instance masks), which is geometrically precise but limited to a closed category set. To add open-vocabulary coverage, *viewpoint-nodes* and room descriptions are generated by prompting a Vision-Language Model (ChatGPT-4V) to describe the objects and scene at each location. The result is a fully textual, hierarchical map: buildings contain rooms, and rooms contain object nodes.

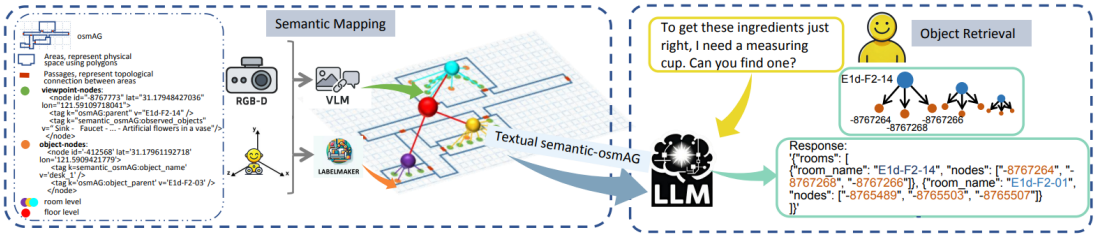


Figure 2.4: OsmAG-LLM mapping and LLM-based object-retrieval solution [20]

Because the map is textual, it can be parsed directly by an LLM with no multimodal projection. Given a query, the LLM does not output motion commands; it returns a structured JSON list of candidate *response nodes*, grouped by room and ranked by likelihood, deliberately mimicking how a human exhausts one room before moving to the next. The robot then visits these nodes in order (planning with A* over an occupancy grid rendered from the osmAG). Since the map may be outdated, the target is confirmed at each node by a two-stage check: an open-vocabulary detector (YOLO-World) proposes candidate boxes, and a VLM (StepFun) verifies each with the prompt "Is there a [object] here?". This detect-then-verify scheme is what lets the system find dynamic, relocated, or entirely unmapped objects [20].

The approach is evaluated in Habitat on eight multi-room, multi-floor HM3D-SEM scenes and deployed on a real robot in a campus building, using the HOV-SG protocol with queries at three granularity levels (object; object and room; object, room, and floor). It reports substantially higher retrieval success than HOV-SG on the hardest queries and in dynamic or unmapped-object cases, while producing a far more compact map [20].

Guide-LLM [19]

Extending text-based map representations into assistive human-robot interaction, Guide-LLM is an embodied LLM agent designed to guide people with visual impairments (PVI) through large indoor environments. Like osmAG-LLM, it avoids

the computational burden of dense 3D maps by using a purely *text-based topological map*: the environment is abstracted into nodes (placed at points where the robot must turn or act) and edges, described in natural language (e.g. "RoomA is connected to HallwayA, 0.5 m north"). The map deliberately favours straight paths and right-angle turns, which are easier for PVI to follow [19].

The central agent is *GPT-4o*, driven by a Chain-of-Thought system prompt that has it interpret the user's conversational query, select a destination, and reason step by step about the route. A path-planning module consults the text map and enumerates candidate routes with a depth-first search, allowing the agent to choose among them according to hazards or user preferences. Localization is handled visually rather than metrically: images along the route are embedded with *CLIP* and stored in a vector database, and the robot confirms it has reached a node by matching its current view against the stored image with cosine similarity (recovering via a broader database search if the score falls too low). A low-level planner converts the agent's high-level decisions into robot motion. Beyond routing, the LLM's commonsense reasoning is used to detect hazards (e.g. a wet-floor sign) and warn the user, proposing alternative paths [19].

The system is evaluated entirely in simulation, in the *iGibson* simulator with a *TurtleBot* equipped with an RGB camera, across two indoor environments of different scale (a large house and an office). Experiments cover navigation success, localization-error recovery, hazard detection, and personalization; an ablation confirms that both the system prompt and the path-planning module are essential (removing the system prompt drops the success rate to zero). All experiments were run offline on a single laptop (Intel Core i9, 32 GB RAM, RTX 4070). The authors note that hazard detection remains imperfect and that real-world testing with PVI is left to future work [19].

2.3.5 Summary and Identified Limitations

The integration of LLMs and VLMs into robotic control loops has demonstrated the profound potential of using foundation models for high-level semantic reasoning. As seen in the reviewed architectures, abstracting complex 3D environments into lightweight, textual representations effectively circumvents the computational overhead of dense geometric processing. This abstraction successfully allows robots to leverage the vast commonsense priors of LLMs to infer target locations and explore unknown environments with human-like intuition.

However, a critical limitation persists when deploying these generative models on physical robotic systems. Solutions that rely on an active, step-by-step interaction with an LLM or VLM to drive local navigation fundamentally bottleneck the robot's control loop. For autonomous mobile robots operating on edge hardware, the inference time required for a single LLM or VLM query typically spans several

seconds. Because physical robotics requires high-frequency control loops to ensure safe and continuous movement, imposing a multi-second latency at every decision junction entirely disrupts the navigation process. This forces the robot into a disjointed "stop-and-think" paradigm, resulting in stuttering trajectories that are highly inefficient in dynamic real-world environments.

Therefore, while utilizing textual maps for high-level, global reasoning is highly effective, to achieve smooth navigation, real-time local semantic perception must be decoupled from heavy generative inference. For this reason the approach adopted in this work follows the state-of-the-art strategy of using an LLM to produce an high level plan starting from a clustered semantic map but departs from the related works in a key respect: the LLM is employed purely as an up-front orchestrator. It is queried once to interpret the command and select a target region, after which it is removed from the control loop and the local decision making is handled to lightweight discriminative models. This allows to achieve truly smooth, continuous semantic navigation on edge hardware.

Since the large generative model is now employed as an orchestrator it is necessary to find an alternative solution that guarantees the real-time semantic perception. As explored in the following section, contrastive architectures provide a solution to this latency bottleneck, offering robust open-vocabulary semantic understanding at speeds fully compatible with real-time robotic control.

2.4 Open-Vocabulary Semantic Understanding

Enriching a geometric map with semantic information is a foundational stepping stone toward a flexible and versatile navigation system. In recent years, researchers have augmented traditional 3D reconstructions of the environment using deep learning-based object detection and segmentation models, achieving impressive results in dense semantic mapping. However, these traditional methodologies share a fundamental constraint: they are *closed-vocabulary* [22]. Their applicability is strictly limited to the predefined object categories annotated within their training datasets, rendering them inflexible in dynamic, real-world environments where novel or unmapped objects frequently appear.

To overcome the limitations of closed-vocabulary systems, there has been a significant surge of interest in leveraging foundation models to build robust 3D representations. Large vision-language (VL) models, such as Contrastive Language-Image Pre-training (CLIP), have excelled at tackling open-vocabulary challenges in 2D vision. Rather than restricting object classification to a rigid set of integer labels, models like CLIP store semantic information as high-dimensional feature embeddings. These embeddings can be matched against human language descriptions in real time, granting the robotic system incredible flexibility and enabling

true open-vocabulary indexing of observed environments [23, 24].

However, translating these powerful embeddings into a 3D robotic framework introduces two critical limitations. First, assigning a high-dimensional semantic feature vector (consisting of hundreds of floating-point values) to every single point in a map is highly redundant. This approach consumes substantially more memory than traditional integer-based class labels, greatly limiting the system's scalability to larger environments on resource-constrained hardware. Second, these dense, per-point representations do not admit an easy structural decomposition. Consequently, achieving real-time, open-vocabulary semantic understanding in large-scale unknown environments remains an open challenge, necessitating architectures that can harness the power of models like CLIP without succumbing to their massive memory footprints [24].

2.4.1 Contrastive Language-Image Pre-training (CLIP)

Contrastive Language-Image Pre-training (CLIP) represents a fundamental paradigm shift in computer vision designed to move beyond rigid, predetermined object categories. Unlike traditional state-of-the-art vision systems that are strictly supervised to predict a fixed set of predefined class labels, CLIP learns transferable visual representations directly from raw natural language supervision. By pre-training on a massive, diverse dataset of 400 million image-text pairs collected from the internet, the model learns a broad spectrum of visual concepts without the need for manual, task-specific annotations or constrained categorical lists [16].

At its core, CLIP operates by mapping two fundamentally different data types (images and text) into a single, shared mathematical environment known as a multimodal latent space. Rather than attempting the highly complex task of generating the exact wording of a caption for an image, CLIP employs a simpler, highly scalable contrastive learning objective: predicting which text snippet correctly pairs with which image [16].

Architecture and Contrastive Learning Objective

To achieve this, the architecture jointly trains two separate neural networks: an image encoder (typically a Vision Transformer) and a text encoder (a standard Transformer). During the training phase, the system processes a batch containing N (image, text) pairs. The encoders process their respective inputs and project them into the shared latent embedding space [16].

The network then computes the pairwise cosine similarities across all possible combinations within the batch. The model is optimized using a symmetric cross-entropy loss function. This loss function explicitly trains the network to maximize

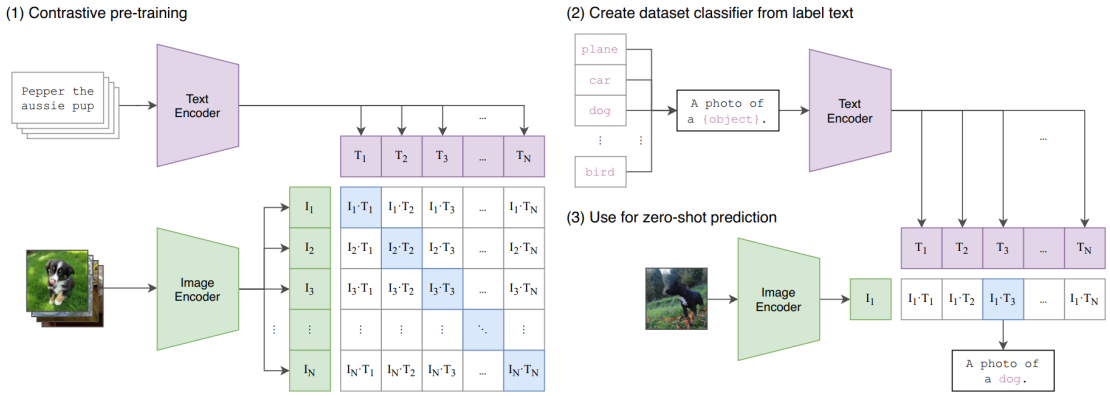


Figure 2.5: Summary of CLIP Architecture [16]

the cosine similarity for the N correct (matching) image-text pairs, while simultaneously minimizing the cosine similarity for the $N^2 - N$ incorrect (mismatched) pairings. Over time, this forces the model to deeply align visual features with their corresponding semantic language descriptions [16].

Zero-Shot Inference for Robotic Perception

This massive contrastive pre-training results in highly robust zero-shot classification capabilities, which are incredibly valuable for robotic navigation. At inference time, the system does not need to be retrained on a new dataset to recognize a novel object. Instead, the learned text encoder synthesizes a zero-shot classifier by embedding arbitrary natural language descriptions or target object names into the latent space [16].

Therefore, when a robot observes its environment, the image encoder extracts the embedding of the current visual frame. The system then calculates the cosine similarity between this real-time visual embedding and the synthesized text embeddings of the target objects. The text query that yields the highest similarity score corresponds to the recognized object. Because this classification relies entirely on flexible natural language rather than a fixed, hard-coded vocabulary, CLIP enables true open-vocabulary scene understanding, allowing robots to seamlessly identify and navigate toward novel objects they were never explicitly trained to detect [16].

2.4.2 Review of Relevant Architectures

ConceptGraphs [23]

One prominent architecture addressing the need for structured, open-vocabulary representations is ConceptGraphs, which builds an open-vocabulary 3D *scene graph* from a sequence of posed RGB-D images. The pipeline begins by applying a class-agnostic 2D instance segmentation model (Segment-Anything) to each incoming RGB frame to extract candidate object masks. To infuse these geometric regions with semantic meaning, the image is cropped to each detected region and the resulting crop is passed through the *CLIP image encoder*, yielding a high-dimensional visual descriptor for the object. The masked region is then projected into a 3D point cloud, and the corresponding CLIP embedding is attached to it, forming a semantically rich 3D object node. This object-level abstraction significantly reduces memory overhead compared to dense per-point feature maps and makes the representation amenable to downstream robotic planning [23].

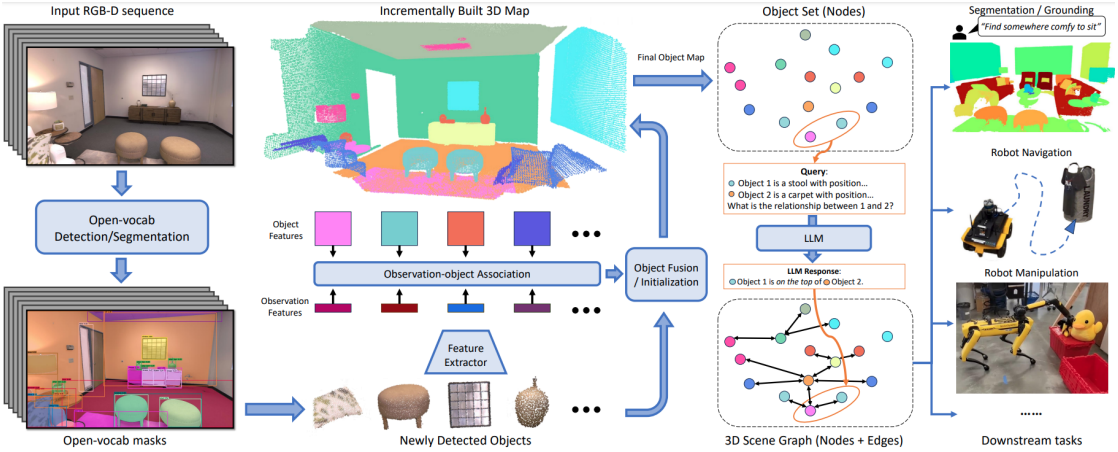


Figure 2.6: ConceptGraphs pipeline [23]

During the incremental mapping process, ConceptGraphs relies on these CLIP embeddings for multi-view object association. When a new detection is projected into 3D, the system must decide whether it is a newly discovered object or another view of an existing one. It computes a semantic similarity, defined as the cosine similarity between the new CLIP descriptor and those of existing map objects, and combines it with a geometric overlap term. If the detection is associated with an existing node, the two CLIP features are fused by averaging over time, producing a robust, multi-view semantic embedding for that object [23].

The distinguishing feature of ConceptGraphs, and what makes it a scene *graph* rather than a flat object map, is a subsequent Large Vision-Language Model and

LLM stage that populates the graph with captions and relations. Each mapped 3D object is rendered from its best viewpoints, and an LVLM (LLaVA) is prompted to produce a natural-language caption of the object; these per-object captions constitute the semantic content of the graph nodes. To build the *edges*, the captions and spatial relationships of nearby object pairs are passed to an LLM (GPT-4), which infers the inter-object relations (e.g. “on top of", “next to") that connect the nodes. The result is a structured, textual 3D scene graph that can be serialized directly into a prompt for downstream LLM-based task planning.

This design yields two complementary querying modes. For simple open-vocabulary retrieval, a target described in natural language is embedded with the CLIP *text* encoder and matched against the stored object embeddings by cosine similarity, an operation lightweight enough to run on-board on a CPU. For complex queries that involve negation, composition, or multi-object reasoning, where raw CLIP similarity is known to fail, the textual scene graph is instead handed to an LLM (GPT-4), which reasons over the captions and relations to identify the target object. This LLM-based mode is more capable but, unlike the CLIP mode, requires access to a large model at inference time, an important consideration for edge deployment [23].

HOV-SG [22]

Expanding upon the object-centric paradigm, the HOV-SG (Hierarchical Open-Vocabulary 3D Scene Graphs) architecture introduces a multi-level structural abstraction to enhance language-grounded robot navigation in large, multi-story environments. Rather than a flat object map, HOV-SG constructs a hierarchical graph spanning *floor*, *room*, and *object* concepts, each enriched with open-vocabulary features [22].

The pipeline first builds an open-vocabulary segment-level 3D map. Each RGB-D frame is segmented with a class-agnostic model (SAM), and every 2D mask is back-projected into the global 3D frame using the known poses. The distinctive step is how each segment is given a semantic feature: rather than encoding a single crop, HOV-SG computes a per-segment CLIP embedding as a *weighted sum of the masked sub-image and its context-free (background-removed) counterpart*, which mitigates the influence of salient background objects while retaining some context. As the same physical surface is observed across many frames, the corresponding 3D segments are merged, and instead of averaging their CLIP features (as done in ConceptGraphs), HOV-SG applies *DBSCAN* clustering over the accumulated point-wise features and selects the dominant cluster as the segment’s representative embedding. This yields a more robust, less noise-sensitive open-vocabulary descriptor and is the main source of HOV-SG’s improved semantic-segmentation accuracy over prior methods.

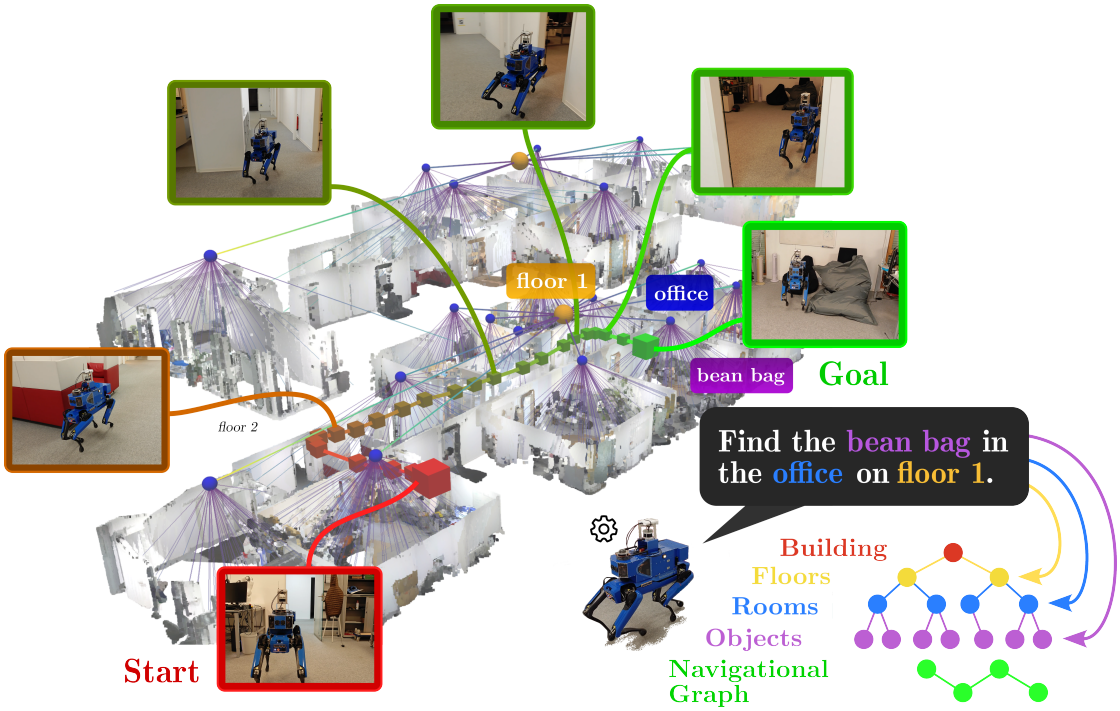


Figure 2.7: Map structure of HOV-SG [22]

From this segment-level map, the hierarchy is built bottom-up. Object nodes are formed by grouping segments; *floors* are identified by clustering the point cloud along the height (gravity) axis; and *rooms* are extracted by segmenting each floor's free space. Every node at every level, object, room, and floor, carries its own aggregated CLIP feature, so semantics are attached not only to individual items but to the spatial regions that contain them. For actionable navigation, the free space is finally encoded as a *cross-floor Voronoi graph*, in which per-floor navigation graphs are linked through stair-wise subgraphs, enabling traversal of multi-story buildings without a separate geometric map [22].

Because every concept in the hierarchy is represented by an open-vocabulary embedding, the graph is directly promptable. HOV-SG handles compound instructions by *decomposing* an abstract query such as "towel in the bathroom on the upper floor" into its constituent tokens and scoring each against the appropriate level of the hierarchy, floor, then room, then object, rather than matching a single embedding against a flat object list. This hierarchical grounding, combined with the compact graph abstraction (which the authors report reduces representation size by roughly 75% relative to dense open-vocabulary maps), is what allows HOV-SG to scale open-vocabulary navigation to real, multi-floor buildings [22].

OVO [25]

While architectures like HOV-SG and ConceptGraphs rely on offline processing, the OVO (Open-Vocabulary Online) pipeline integrates open-vocabulary semantic descriptors directly into an online 3D semantic mapping process that runs alongside SLAM. Rather than assuming pre-computed ground-truth poses, OVO is demonstrated end-to-end on top of two full SLAM backbones (Gaussian-SLAM and ORB-SLAM2), making it, to the authors' knowledge, the first open-vocabulary online 3D mapping system with loop-closure support [25].

Given a stream of posed RGB-D keyframes, OVO segments each frame with a class-agnostic model (SAM 2.1) and back-projects the resulting 2D masks into 3D. These are tracked and fused across frames as persistent 3D segments: incoming detections are matched to existing segments using a combination of geometric overlap and descriptor similarity, so that the same physical object observed from many viewpoints is consolidated into a single node. Because the system is coupled to SLAM, a loop closure that corrects the trajectory also triggers a re-fusion of segments that had been erroneously split by tracking drift, merging them back into one instance.

The core contribution of OVO is its learned *CLIP-merging* strategy for computing each segment's semantic descriptor, designed to overcome the computational and memory bottlenecks of online mapping. For a given viewpoint, three vision-language embeddings are extracted with a SigLIP encoder (ViT-SO400): one from the full image, one from the isolated masked segment without background, and one from the segment's bounding-box crop including its immediate surroundings. Rather than combining these with fixed hand-crafted weights (as HOV-SG does) or by simple averaging (as ConceptGraphs does), OVO feeds the three embeddings into a small trained network, a five-layer transformer encoder followed by a multi-layer perceptron, that predicts per-dimension fusion weights and outputs a single robust descriptor. This merging network is trained once on the ScanNet++ dataset and then generalizes to unseen classes and environments. To bound cost, the final per-segment descriptor is aggregated only over the highest-quality viewpoints (ranked by the pixel area of the segment's 2D mask), rather than over every frame in which it appears [25].

The result is an online pipeline that reportedly achieves lower computational and memory footprint than offline baselines such as ConceptGraphs and HOV-SG, while attaining better segmentation accuracy, and that recovers both semantic and instance labels at any point during mapping [25].

FindAnything [24]

The architectures discussed so far are not designed for the strict memory and computational limits of edge robotic platforms. FindAnything directly targets this

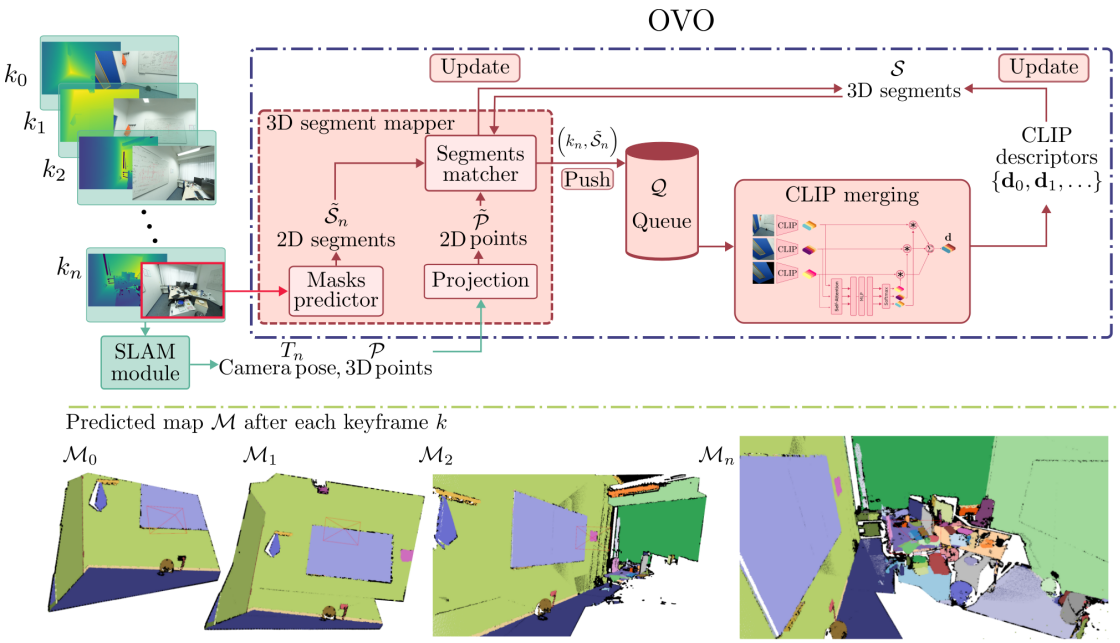


Figure 2.8: OVO pipeline [25]

gap, proposing an object-centric open-vocabulary mapping framework built for real-time, onboard operation on resource-constrained devices. Its inputs are IMU measurements and stereo images, processed by a Visual-Inertial SLAM (VI-SLAM) backend based on OKVIS2; the estimated poses are used to integrate depth into a set of volumetric occupancy *submaps* rather than one monolithic map. This submap structure enables drift correction: when the SLAM system closes a loop and updates its trajectory, the submaps are rigidly transformed accordingly, keeping the global reconstruction consistent without any external ground-truth pose [24].

To combine dense geometry with high-dimensional semantics without exhausting memory, FindAnything decouples voxel resolution from language representation. Rather than storing a language vector in every voxel, it aggregates semantics at the object level. For each RGB frame, a CLIP vision encoder (ViT-L/14 at 336 px) produces a *per-pixel* 768-dimensional feature map, and *eSAM* (EfficientSAM), a lightweight SAM variant, generates class-agnostic segment masks. The pixel-wise CLIP features falling inside each mask are then pooled into a single averaged descriptor that is attached to the corresponding 3D segment, so only one embedding is stored per object rather than one per voxel. To keep eSAM tractable on embedded hardware, it is prompted with only a sparse grid of points (64 equally-spaced prompts per frame) and its outputs are filtered by mask-quality and stability thresholds [24].

Segments are maintained across frames by an “as-fine-as-possible" tracking

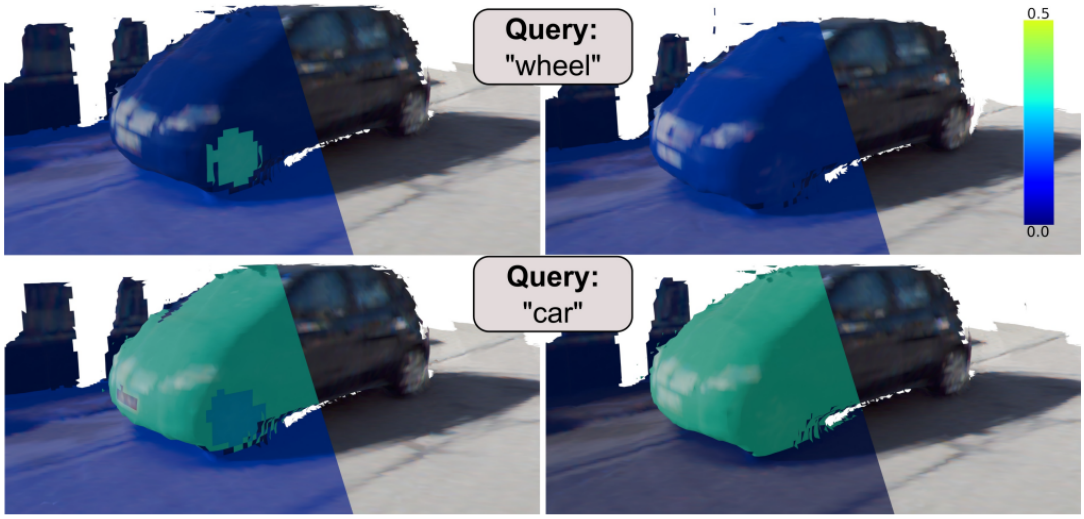


Figure 2.9: Segmentation example in FindAnything [24]

scheme: existing 3D map segments are rendered back into the current view and matched to the new eSAM masks by mask overlap (IoU). Notably, segments are only ever kept or further split, never merged, since eSAM tends to over-segment objects at close range; this keeps the representation reusable for arbitrary downstream tasks rather than tied to a fixed object definition. As a segment accumulates observations, its CLIP descriptor is refined by a running weighted mean over the contributing pixels. This tracking runs entirely on the CPU, leaving the GPU free for the foundation models, so that eSAM and CLIP inference remain the only bottleneck [24].

At query time, a natural-language prompt (e.g. “fire extinguisher" or “exit") is encoded into a CLIP text embedding and matched against the stored segment descriptors by cosine similarity, letting a frontier-based exploration planner steer the robot toward semantically relevant regions in real time, with an MPC executing the planned paths. Most relevantly for this work, FindAnything is deployed *online and fully onboard* a resource-constrained micro aerial vehicle (MAV) using an *NVIDIA Jetson Orin NX*, and is presented as the first open-vocabulary object-centric mapping system to achieve this. Running the foundation models on such hardware forces explicit compromises, the sparse eSAM prompting noted above, half-resolution input, and a software-imposed cap that still integrates depth at 3 Hz when segmentation or CLIP inference cannot keep pace, constraints that closely mirror the challenges addressed by the present thesis, making FindAnything the closest point of comparison to the edge-deployed pipeline proposed in the following chapters [24].

2.4.3 Summary and Identified Limitations

The primary bottleneck across the majority of the reviewed works (such as ConceptGraphs, HOV-SG, and OVO) comes from their reliance on class-agnostic 2D segmentation model. While models like the Segment Anything Model (SAM) provide excellent geometric boundaries, they are inherently prone to severe over-segmentation: they frequently partition continuous surfaces into redundant fragments and aggressively segment task-irrelevant background such as blank walls, ceilings, or patches of floor. Because these pipelines aim to build comprehensive, dense scene graphs, they must run a heavy vision-language encoder (CLIP, or SigLIP in the case of OVO) on nearly every proposed mask, executing tens of high-dimensional encodings per camera frame. For an autonomous mobile robot operating on edge hardware, this volume of transformer inference introduces a prohibitive latency that disrupts the high-frequency control loop required for smooth, real-time navigation. Crucially, the cost is spent indiscriminately: the same expensive encoding is applied to task-irrelevant background as to the objects that actually matter for the query.

Architectures that instead prioritize efficiency, such as FindAnything, resolve the latency problem but introduce a different trade-off in semantic fidelity. To remain real-time on constrained platforms such as MAVs, FindAnything computes a single dense per-pixel CLIP feature map per frame and pools these features within each segment mask, storing one averaged descriptor per object. Because the per-segment descriptor is obtained by averaging pixel-wise features over a coarsely-prompted mask, rather than by encoding a clean, background-isolated object crop, the resulting representation is less discriminative for fine-grained distinctions between individual entities. The efficiency is therefore bought at the cost of the precise, object-level semantic fidelity that crop-based encoders provide, which is precisely the fidelity required to distinguish, for instance, a specific object by its visual attributes rather than merely its category.

Therefore, a critical gap remains in the current literature: an open-vocabulary semantic navigation pipeline that achieves the precise, object-level semantic fidelity of crop-based approaches while respecting the strict real-time inference constraints of edge computing. The pipeline proposed in the following chapters addresses this gap by inverting the prevailing segment-then-encode paradigm. Rather than exhaustively segmenting and encoding every region, it detects only relevant objects with an open-vocabulary detector and encodes only those with a discriminative vision-language model, confining the expensive inference to task-relevant entities. These object-level embeddings are retained in a persistent semantic map, so that non-target objects observed during exploration remain available for later, potentially multi-step, queries. A separate large language model is used purely as an up-front orchestrator to interpret the command and guide the search, and is removed from

the real-time control loop entirely.

2.4.4 Architectural Differences Between CLIP and SigLIP

Having established how CLIP aligns visual and textual data within a shared multi-modal latent space (Section 2.4.1), it is now possible to examine how SigLIP departs from it. Although both models pursue the same goal of cross-modal alignment, they differ fundamentally in the mathematical objective function used to relate image and text pairs, and this difference has direct consequences for their behaviour under the resource constraints of edge deployment.

The Softmax Bottleneck in Standard CLIP

As described in Section 2.4.1, CLIP optimizes its encoders with a symmetric cross-entropy loss applied over softmax-normalized similarities. The limitation of this formulation lies in the normalization step: the softmax requires a global view of all pairwise similarities across the entire batch, since the score of each correct pair is defined relative to every other pair competing within the same batch. This coupling has two practical consequences. First, the memory and processing cost scales quadratically with the batch size, as the full similarity matrix must be materialized to compute the loss. Second, because performance depends on the diversity of in-batch negatives, the approach effectively requires very large batches to perform well, placing a hard demand on available memory and compute.

The Pairwise Sigmoid Approach in SigLIP

SigLIP (Sigmoid Loss for Language Image Pre-Training) resolves this bottleneck by replacing the global softmax normalization with a simple, pairwise Sigmoid loss [18]. Rather than scoring each pair relative to all others in the batch, SigLIP operates solely on individual image-text pairs, treating alignment as a series of independent binary classification tasks: for a single image and a single text description, the model determines only whether they match or not, with no global view of the rest of the batch. This independence disentangles the loss calculation from the batch size, removing the quadratic memory coupling and the reliance on extremely large batches that characterize the softmax formulation.

Advantages for Zero-Shot Object Identification

For the specific task of object identification within a semantic navigation pipeline, the choice of SigLIP over standard CLIP is motivated by the operating regime of edge-deployed perception, where batch sizes and compute are constrained. The

transition from a competitive softmax loss to an independent sigmoid loss yields concrete benefits in this setting:

- **Stronger Performance at Constrained Batch Sizes:** Because the sigmoid loss is computed independently for each image-text pair, it does not depend on a large pool of in-batch negatives to remain effective. SigLIP has been shown to outperform softmax-based CLIP by a clear margin at small batch sizes, with the gap between the two losses narrowing as the batch size grows and the methods becoming comparable at very large batches [18]. For a pipeline that cannot rely on the massive batches used to train large-scale contrastive models, this property makes SigLIP the more reliable choice for accurate zero-shot recognition under fixed-resource constraints.
- **Improved Robustness to Noisy Associations:** Open-world perception requires the model to ground free-form language prompts (e.g., “find the fire extinguisher") to raw, imperfectly labelled visual data. Sigmoid-based pre-training has been reported to increase robustness to data noise relative to the softmax baseline, with models trained under the sigmoid objective retaining their advantage under increasing input corruption [18]. This robustness is desirable for a robot operating in uncontrolled environments, where visual observations are far noisier than curated training imagery.

Ultimately, by leveraging SigLIP, the proposed navigation pipeline aims to obtain a more precise and robust open-vocabulary understanding of target objects within the strict computational budget of on-board hardware, without incurring the large-batch requirements associated with traditional global-contrastive models.

Chapter 3

Simulation Environment Setup

3.1 Sim-To-Real Gap in Semantic Navigation

The problem of autonomous visual navigation is a foundational challenge in modern robotics, carrying significant implications for real-world applications ranging from household service robots to large-scale industrial automation. Exploring, perceiving, and interacting with the physical real world is crucial in these applications. However, conducting extensive real-world testing in domestic environments presents considerable challenges. Real-world evaluation is prohibitively slow, heavily constrained by battery life and human oversight, and carries safety risks to both the hardware and the delicate household surroundings. Consequently, high-fidelity simulated worlds have become indispensable virtual testbeds. Simulators allow for massive parallelization, highly reproducible scenarios, and the safe, accelerated testing of navigation algorithms across thousands of varied floor plans before physical deployment is ever attempted [26].

Despite the necessity of simulation, transferring policies learned in simulated environments to the real world remains challenging owing to the significant *sim-to-real reality gap*. Simulators typically fail to replicate visual realism and complex real-world geometry. This lack of realistic visual rendering limits the ability of these policies to support high-level tasks requiring RGB-based perception, such as ego-centric navigation and visual exploration [26]. When policies are deployed, they struggle because the pristine synthetic data they have been tested on does not match the noisy, complex visual data of the physical world.

3.2 Preliminary Baseline: Gazebo Classic

In the broader scope of simulation design, there are two primary methodologies for constructing virtual testbeds: game-based scene construction and world-based scene construction. The preliminary testing baseline utilizes the former through the use of Gazebo Classic, that is a widely used, open-source 3D robotics simulator that allows developers to test robotic systems in virtual environments before physical deployment.

Game-based scenes are constructed entirely by assembling individual 3D object assets into a cohesive environment. These are fundamentally asset-driven spaces, where the individual 3D objects, ranging from furniture to structural walls, are typically sourced from online asset stores or CAD libraries.

To evaluate the domestic navigation pipeline, the baseline utilized the AWS Small House environment within Gazebo Classic. This is a ready-made virtual home designed specifically for robotics simulation. Technically, it provides a standard, asset-driven layout containing common household objects, rooms, and boundaries. It offers a highly controlled, standardized setting intended to benchmark a robot's basic navigation and perception capabilities within a residential context.

Figure 3.1: AWS small house simulation layout

3.2.1 Technical Advantages of Game-Based Scenes

Using an asset-driven, game-based simulator provides several structural conveniences for early-stage robotics testing:

- **Inherent Physical Properties:** Because the environment is built from individual 3D models, the simulator's physics engine can easily assign discrete properties, such as mass, friction coefficients, and collision boundaries, to each distinct object in the scene.
- **Native Object Segmentation:** Object classes within these environments are perfectly segmented by default. Since a "chair" is a fundamentally separate 3D model from the "floor", the simulator inherently knows the exact boundaries of every item, providing flawless ground-truth data for evaluating perception modules.
- **Articulated Modeling:** This clear, asset-level segmentation makes it highly straightforward to model complex interactions. Assets can easily be programmed as articulated objects with movable joints (e.g., doors that open, cabinets that swing), which is advantageous for testing advanced navigation and manipulation behaviors.

3.2.2 Technical Limitations and the Sim-to-Real Bottleneck

Despite their structural advantages, game-based scenes present severe limitations when transitioning tested pipelines to physical robots. It is notoriously difficult to ensure the quality and consistency of 3D object models in asset-driven environments compared to scan-driven, real-world datasets. This discrepancy creates a significant simulation-to-real bottleneck, driven by three primary factors:

- **The Visual and Sensor Gap:** Game-based scene simulators like Gazebo Classic generally lack the high visual fidelity required for realistic robotics testing. They typically utilize low-polygon models, simplistic artificial lighting, and basic textures. Without the complex shadows, reflections, and camera sensor noise inherent to physical homes, semantic perception modules tend to overfit to the pristine synthetic data, causing them to fail when processing actual visual streams.
- **The Semantic Diversity Gap:** Asset-driven simulators are fundamentally limited by the specific assets placed within them. A pipeline tested in the AWS Small House might flawlessly identify the exact 3D model of a "sofa" used in that specific scene. However, this perfectly segmented, highly restricted dataset fails to prepare the semantic navigation system for the massive, long-tail variations of object shapes, colors, lighting conditions, and cluttered arrangements it will inevitably encounter in reality.

Consequently, the low-polygon models, artificial lighting, and lack of visual clutter in Gazebo Classic ultimately failed to provide the necessary realism for

this project. The resulting sim-to-real bottleneck proved too severe for the semantic segmentation pipeline to function accurately upon real-world deployment, necessitating a transition to more realistic testing environments.

3.3 Photorealistic Simulation: Habitat-Matterport 3D (HM3D)

To address the severe visual limitations of the preliminary Gazebo Classic baseline, the testing pipeline transitioned to a world-based scene construction utilizing the Habitat-Matterport 3D (HM3D) dataset [27]. Unlike game-based scenes constructed from disparate synthetic assets, world-based scenes are dataset-driven environments. They are constructed directly from real-world scans of physical spaces, capturing the holistic layout, objects, and architectural elements exactly as they exist in reality. HM3D provides a large-scale collection of 1,000 building-scale 3D reconstructions of real-world interiors—spanning multi-floor residences, offices, shops, and other indoor spaces—captured as textured 3D meshes. Compared to earlier scan-driven datasets such as Matterport3D and Gibson, images rendered from HM3D exhibit substantially higher visual fidelity and its reconstructions contain markedly fewer surface artifacts, making it particularly well-suited for evaluating RGB-based perception pipelines [27].

Figure 3.2: Habitat Matterport3D dataset [27]

3.3.1 Technical Advantages of World-Based Scenes

Using photorealistic, scan-driven datasets like HM3D provides significant advantages for evaluating domestic navigation pipelines:

- **High Visual Fidelity:** Real-world scans capture natural lighting, complex textures, and authentic sensor noise, providing a much more accurate visual representation of the real world compared to low-polygon game assets.

- **Bridging the Sim-to-Real Gap:** This high level of realism directly facilitates a better transfer of performance from simulation to the physical world, allowing perception modules to be tested against data that closely mirrors what physical cameras will actually see.

- **Authentic Semantic Complexity:** Testing with photorealistic RGB and depth data allows the navigation pipeline to incorporate a more robust semantic understanding of realistic, cluttered domestic environments.

3.3.2 Technical Limitations and Segmentation Bottlenecks

While HM3D significantly minimizes the visual appearance gap, it introduces new challenges for testing semantic pipelines due to how its data is annotated:

- **Geometry-Bound Annotations:** In HM3D, semantic annotations are defined per-vertex and are directly embedded into the 3D mesh geometry itself.

- **Inaccurate Object Boundaries:** Relying on mesh geometry discretization often fails to sufficiently capture the precise boundaries between objects, particularly on flat surfaces like walls, floors, or crowded tabletops. This can result in jagged, inaccurate semantic boundaries or missing annotations during evaluation.

- **Object Clustering:** During testing, pipelines may encounter ground-truth errors because the dataset tends to cluster smaller items (like decorative pieces or tabletop clutter) into larger furniture pieces rather than segmenting them individually.

Despite these segmentation artifacts, utilizing real-world scans ensures the semantic algorithms process data that closely mirrors physical reality. This naturally minimizes the visual sim-to-real gap and provides a much more robust foundation for evaluating the navigation pipeline than asset-driven simulators.

3.4 Simulation Framework and Custom Middleware Architecture

3.4.1 Habitat-Sim

The evaluation of a semantic mapping algorithm requires a simulation environment capable of rendering high-fidelity, photorealistic 3D scans of real-world environments while providing pixel-perfect ground-truth semantic annotations. As previously said traditional robotics simulators, such as Gazebo, are optimized for rigid-body physics in CAD-based environments and struggle with the computational overhead of rendering massive, unstructured scanned meshes like those found in the previously introduced Habitat Matterport 3D (HM3D) dataset [27].

Habitat-Sim was selected as the simulation platform because it is explicitly engineered for high-performance 3D navigation and dataset ingestion. It natively parses the HM3D dataset architecture, seamlessly linking visual geometry (`.glb`) with corresponding semantic metadata without manual scene reconstruction. By prioritizing rendering throughput over complex physics calculations, Habitat-Sim is capable of rendering multi-sensor arrays at high frame rates, which is crucial for feeding dense RGB-D and semantic data into a real-time mapping pipeline [26].

3.4.2 Habitat-ROS 2 Interoperability Bridge

Because Habitat-Sim is designed as a standalone Python library for embodied AI research rather than a native ROS 2 robotics simulator, it lacks built-in mechanisms to communicate with standard ROS 2 navigation and mapping stacks. To bridge this architectural gap, a custom middleware node, the `HM3DBridgeNode`, was developed. This script acts as a bidirectional translator between the simulator's internal memory and the distributed ROS 2 network, managing coordinate transformations, sensor data serialization, and ground-truth emulation.

3.4.3 Coordinate System Transformations

A critical function of the interoperability bridge is reconciling the differing coordinate systems used by computer graphics engines and standard robotics frameworks. Habitat-Sim utilizes a graphics-standard coordinate system where the Y-axis is up and the camera faces the negative Z-axis. Conversely, ROS 2 relies on a standard right-handed robotics coordinate frame where the Z-axis is up and the X-axis is forward.

To ensure spatial consistency within the mapping pipeline, the bridge actively intercepts the simulator's `AgentState` and performs matrix and quaternion transformations. The script converts Habitat positions (x_h, y_h, z_h) to ROS positions

(x_r, y_r, z_r) by mapping $x_r = -z_h$, $y_r = -x_h$, and $z_r = y_h$. Rotational quaternions are similarly transformed through a predefined basis matrix (`basis_h_to_r`), allowing the simulator's internal pose to be accurately broadcast to the ROS `tf2` transform tree.

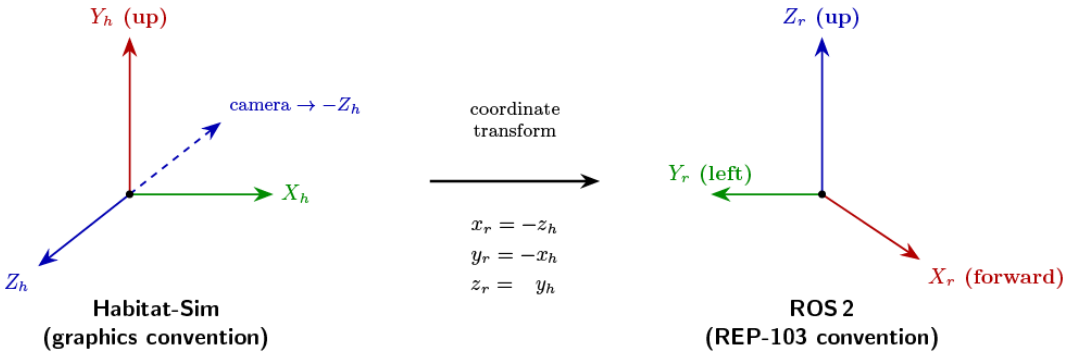


Figure 3.3: Coordinate transformations

3.4.4 Sensor Data Serialization and Semantic Processing

The bridge handles the heavy computational lifting of extracting raw observation matrices from the simulator and serializing them into standard ROS 2 `sensor_msgs/Image` formats at high frequencies.

- **RGB-D and Intrinsics:** The script extracts contiguous `numpy` arrays for both RGB and Depth observations. To enable accurate 3D point projection downstream, the node dynamically calculates and publishes the pinhole camera intrinsics (`CameraInfo`)—specifically the focal lengths (f_x, f_y) and principal points (c_x, c_y)—based on the configured Horizontal Field of View (HFoV) and image resolution.

- **Semantic Segmentation:** Habitat outputs raw semantic data as a matrix of 32-bit integer object IDs, which lack visual contrast. To facilitate human visualization in tools like RViz2, the bridge implements a custom color-mapping algorithm. By multiplying the raw integer IDs by distinct prime numbers (35, 90, 150) and applying a modulo operation, the script effectively hashes every unique semantic instance into a highly contrasting, distinct RGB color format before publishing the data stream.

3.4.5 Ground-Truth Emulation for Algorithm Evaluation

Because the primary objective of this research is to evaluate semantic mapping rather than Simultaneous Localization and Mapping (SLAM), the evaluation must be isolated from trajectory drift. The `HM3DBridgeNode` achieves this by simulating perfect localization and global mapping natively derived from the simulation mesh.

- **Perfect Odometry:** The script continuously queries the simulator for the agent's exact global coordinates and publishes this data as a dynamic transform from the `odom` frame to the `base_link` frame.
- **Global Occupancy Grid:** The bridge utilizes Habitat-Sim's built-in `Pathfinder` module to query the environment's 3D navigation mesh (`.navmesh`). By sampling the traversable floor space at a defined resolution and snapping coordinates to the nearest navigable area, the script generates and publishes a static 2D `nav_msgs/OccupancyGrid`. This provides a fixed, perfect `/map` reference frame, ensuring that any spatial errors detected during the evaluation are strictly the result of the semantic projection logic, rather than localization failures.

Chapter 4

System Architecture

4.1 Open-Vocabulary perception

The vision module is the perception front-end of the navigation system. It transforms a raw RGB stream into a structured set of object hypotheses, each carrying a geometric footprint (a per-instance segmentation mask and bounding box), a semantic label, a temporally consistent instance identity, and a semantic embedding suitable for open-vocabulary comparison against the navigation goal.

4.1.1 Problem Setting and Requirements

Semantic navigation requires the agent to ground a set of goal concepts in its visual field. This has become the first and most fundamental requirement for cutting edge semantic navigation solutions.

The second requirement follows from the navigation objective. Because the agent must travel to an object, each detection must yield spatial information precise enough to be back-projected through the depth image into a metric goal coordinate. A bounding box alone over-includes background pixels and biases the depth estimate, thus a per-instance segmentation mask is required, not merely a detection.

The third requirement is dictated by the deployment target. The pipeline runs onboard an edge compute platform under a real-time latency budget shared with localization, planning, and control. Any architecture must therefore be evaluated not only on accuracy but also on its computational footprint and, critically, on how that footprint scales with scene complexity.

4.1.2 Detection and Segmentation

The Class-Agnostic Segmentation Paradigm

As seen in Section 2.4.2, the prevalent design in the ObjectNav and open-vocabulary perception literature decouples the segmentation step from the semantic association step [23, 22, 25, 24]. Therefore, in many cases, it is first determined where objects are and later what they are.

For the segmentation step, the most common approach involves the use of a class-agnostic segmentation model (most prominently the Segment Anything Model (SAM) and its derivatives (e. g. , MobileSAM, NanoSAM, etc.)) [28] that partitions the image into all plausible object regions without assigning semantics.

Each region is then independently embedded by a vision–language model and compared against the embedding of the goal concept. The semantic decision is deferred entirely to the vision–language model, and open-vocabulary behavior emerges from the embedding space rather than from the segmenter [23, 22].

This paradigm cleanly separates geometric from semantic reasoning. However, assessed against the requirements above three structural problems emerge. The first two are intrinsic to the class-agnostic formulation; the third concerns the viability of the backbone on edge hardware.

Problems with Class-agnostic Segmentation

A class-agnostic segmenter has no notion of what constitutes a single nameable object, and consequently fragments compositional objects into their visually salient parts: a chair becomes seat, back, and legs; a desk dissolves into surfaces and devices. Let the semantically correct partition of an image be $S = \{s_1, \ldots, s_n\}$ and the agnostic model's output be $A = \{a_1, \ldots, a_m\}$ with $m \gg n$. The correspondence between the two is many-to-one, so recovering semantic objects requires an additional grouping stage that is itself error-prone. Under-segmentation, which merges distinct objects, is equally damaging to downstream goal localization.

This over-segmentation also carries a direct computational cost. Because the segmenter is agnostic, it cannot prune irrelevant regions: walls, floor patches, ceiling, and clutter are all segmented and all must be embedded, since relevance can only be assessed after the multimodal model scores each region. Writing C_e for the cost of a single embedding, the per-frame cost under the agnostic paradigm is

$$C_{\text{agnostic}} = |A| \cdot C_e = \mathcal{O}(m) \cdot C_e, \tag{4.1}$$

whereas a semantically-aware pipeline only ever embeds the n relevant objects, in practice a small subset matching the active goal class. Since $m \gg n$, the agnostic approach expends the dominant share of its embedding budget on regions that are

Figure 4.1: SAM Oversegmentation Example [29]

discarded immediately after scoring, and the cost grows unboundedly with scene clutter.

Distilled variants such as MobileSAM and NanoSAM reduce the backbone's parameter count and latency, but do not address either of the two structural problems above. A lighter encoder still performs a dense, class-agnostic pass over the entire frame and still emits a semantically ungrouped set of regions; quantization addresses the constant factor in the backbone, not the $\mathcal{O}(m)$ scaling of the embedding stage that follows it, nor the absent grouping semantics.

YOLOE

For these reasons the adopted architecture inverts the order of geometric and semantic reasoning. In this solution it is implemented a prompt-conditioned model that is told in advance which categories are of interest and returns only the corresponding instances, each associated with a mask and a label. YOLOE is a single-stage detector that accepts an open, text-specified vocabulary and produces, in one forward pass, bounding boxes, per-instance segmentation masks, and class assignments for the prompted categories [30].

Another advantage of using YOLOE is that it handles internally the tracking of the detections. Even though this introduces some overhead, it significantly simplifies the mapping step. In this case the tracker used is BoT-SORT, since it represents the best tradeoff between IDs stability and computational effort.

Open-vocabulary behavior is obtained by supplying the category names as a prompt rather than by retraining the detector. In the implementation, the active vocabulary is set at runtime and augmented on demand with the current goal class, so that the goal concept is always within the detectable set without any

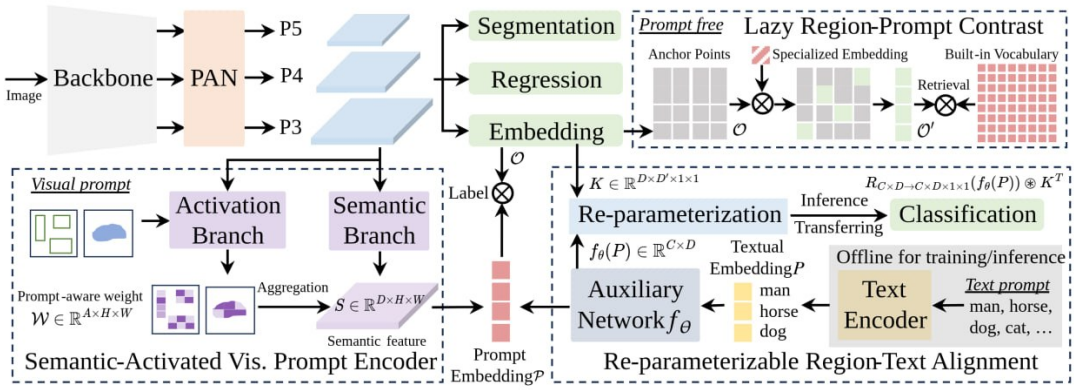


Figure 4.2: Overview of YOLOE pipeline [30]

optimization step.

Conditioning on the vocabulary also prunes the scene at the detection stage rather than after embedding. Regions that do not belong to any prompted category are never returned and therefore never embedded.

The foundation model is consequently relieved of the role of discovering objects and is reserved for the task at which it is strongest and which the detector cannot perform: fine-grained, open-vocabulary semantic scoring of an already-isolated object against the goal concept.

This allows the agent to access information about objects that are not normally accessible via YOLOE. The embeddings computed allow to understand specific features of each object and even their state, for example determine if the fridge is open or not, if the television is on or if a person is performing a specific action. Moreover, embeddings are the core element of the visual memory of the robot.

4.1.3 Semantic Encoding

For semantic validation, the node integrates a pre-trained Vision Transformer (ViT-B/16) optimized via SigLIP [18]. Unlike standard Contrastive Language-Image Pre-training (CLIP), which applies a softmax normalization across an entire batch of image-text pairs [16], SigLIP leverages pairwise sigmoid cross-entropy. This architectural shift is critical for robotics, as it evaluates each region-prompt pair independently, allowing the model to naturally output low probabilities for all classes if the target object is absent, rather than forcing a relative match.

4.1.4 Node Architecture and Implementation

The two chosen models are assembled into a single ROS 2 node. This section documents how, with emphasis on the decisions that make the pipeline meet its real-time budget.

The node subscribes to the RGB message coming from a stereo-camera, and publishes a custom detection-array message together with an optional annotated-image stream (used for debugging) and the goal text embedding.

Each frame passes through four stages: (i) detection, segmentation, and tracking with YOLOE; (ii) per-instance mask processing and crop preparation; (iii) batched semantic encoding with SigLIP; and (iv) assembly and publication of the custom message. Stages (i)–(ii) run synchronously on the ROS executor thread; stages (iii)–(iv) run on a separate worker thread. The two halves communicate through a bounded queue, giving the producer–consumer structure shown in Algorithm 1.

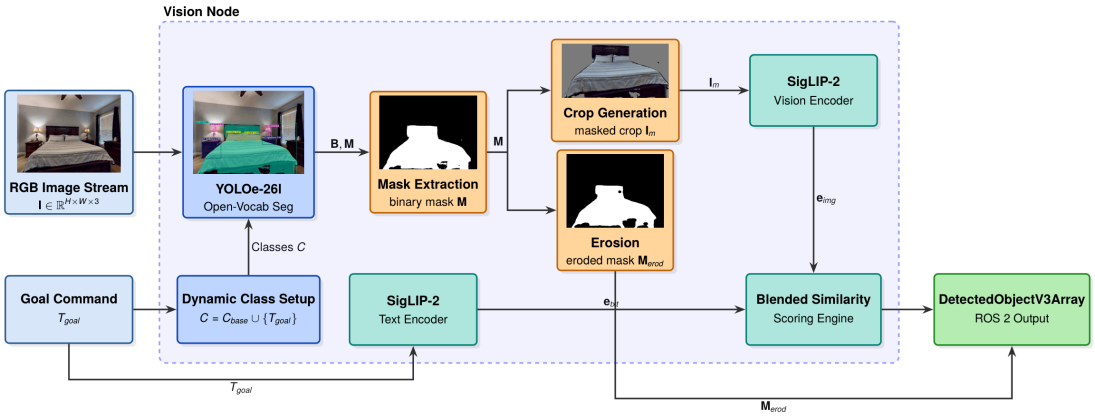


Figure 4.3: Example of the vision node processing using a single detection

Vocabulary Management

The detector is initialized with a base list of category names and, before the first inference, the node reads the active goal from a command file and appends it to the vocabulary if absent. This guarantees that the goal class is always detectable without retraining. The same command file supplies the SigLIP text prompt; both are re-read on a timer so that a change of goal at run time updates the vocabulary and the target embedding without restarting the node.

Algorithm 1: Two-stage perception pipeline (per frame)

Input: RGB frame I, goal text embedding t

Executor thread:
  $R \leftarrow$ YoloE.track(I) **if** R *has no detections* **then**
    return
  $(\mathbf{b}, \mathbf{c}, \mathbf{id}, \mathbf{M}) \leftarrow$ ToCpu(R)
  $F \leftarrow \emptyset$; $Q \leftarrow \emptyset$
  foreach *detection* i **do**
    clamp box; erode mask M_i; build entry e_i
    $F \leftarrow F \cup \{e_i\}$
    if *doSiglip* **then**
      prepare masked/unmasked crops; $Q \leftarrow Q \cup \{e_i\}$
  enqueue (header, F, Q) **non-blocking**; **drop frame if queue full**
Worker thread (loop):
  (header, F, Q) $\leftarrow$ **dequeue (blocking)**
  if $Q \neq \emptyset$ **then**
    $E \leftarrow$ SigLIP.encodeBatch(crops of Q)
    attach embeddings E to the corresponding entries of F
  foreach *entry* $e \in F$ **do**
    if e *has embedding* ***and*** $t \neq \varnothing$ **then**
      $e.sim \leftarrow$ BlendedScore(e, t)
  publish DetectedObjectV3Array(F, header)

Producer–Consumer Decoupling and Frame Dropping

The single most important real-time decision is the separation of detection from semantic encoding across two threads connected by a bounded queue (`maxsize` = 2). YOLOE inference and SigLIP inference have different and variable latencies; running them sequentially on the callback thread would couple the camera-callback rate to the slower of the two and induce unbounded queueing inside the ROS middleware. By offloading SigLIP to a worker thread, the executor thread returns quickly and remains responsive to incoming frames.

The bounded queue implements an explicit latency-over-completeness policy. When the worker cannot keep pace, the queue fills and the producer drops the current frame rather than blocking.

This bounds end-to-end latency at the cost of frame throughput — the correct trade-off for navigation, where a recent, timely detection is more valuable than an exhaustive but stale one.

Temporal Subsampling of the Semantic Stage

Detection and tracking run on every frame, but SigLIP is invoked only every N_{skip} frames. The justification is that object identity is maintained across frames by the tracker (BoT-SORT), so a semantic embedding computed for an instance remains associated with that instance on the intervening frames through its persistent identity; recomputing the embedding each frame would duplicate work without adding information. Detection must run at full rate to keep the geometry and tracking current; semantic encoding, being the more expensive stage, runs at a reduced cadence.

Mask Processing and Crop Preparation

For each instance the raw mask is thresholded to a binary map and eroded with a small elliptical kernel before being resized to full resolution. Erosion pulls the mask boundary inward, which prevents background pixels at the object edge from leaking into the foreground crop, since edge bleed would contaminate the embedding with non-object context. Two crops are then prepared from the eroded mask and the related bounding box: a masked crop, in which the background is replaced by a neutral grey so the embedding concentrates on the object, and an optional unmasked crop that retains the surrounding context. To satisfy the strict input resolution topologies of downstream Vision Transformers, the standard bounding box $\mathbf{B}_i = [x_{min}, y_{min}, x_{max}, y_{max}]$ undergoes a square-crop normalization. Let $w = x_{max} - x_{min}$ and $h = y_{max} - y_{min}$. The center of the box is denoted as $C = (x_c, y_c)$. A target bounding dimension is calculated using a configurable scale factor γ (typically $\gamma = 1.2$):

$$L_{target} = \gamma \cdot \max(w, h) \tag{4.2}$$

The normalized, squared bounding box $\mathbf{B}'_i$ is defined symmetrically as $[x_c - \frac{L_{target}}{2}, y_c - \frac{L_{target}}{2}, x_c + \frac{L_{target}}{2}, y_c + \frac{L_{target}}{2}]$, which is subsequently clamped to the absolute dimensions of the image frame.

Similarity Scoring

Let f_v and f_t denote the ViT vision encoder and the text encoder, respectively. The target text T is embedded to a latent vector $\mathbf{v}_{text} = f_t(T) \in \mathbb{R}^d$. Similarly, an image crop (masked or unmasked) is embedded to $\mathbf{v}_{img} = f_v(\mathbf{I}) \in \mathbb{R}^d$.

The unnormalized similarity logit l is calculated using the dot product of the flattened embeddings, dynamically scaled by a temperature parameter τ_{scale} and shifted by a bias β_{bias} learned during pre-training (e.g., on the WebLI dataset):

$$l = \left(\mathbf{v}_{img}^T \mathbf{v}_{text}\right) \cdot \tau_{scale} + \beta_{bias} \tag{4.3}$$

To map this unbounded logit into a stable probability density, a safely clamped sigmoid function is applied to prevent numerical overflow:

$$\sigma(l) = \frac{1}{1 + \exp(-\text{clip}(l, -60, 60))} \tag{4.4}$$

The base probability score $p \in [0, 100]$ for a given embedding pair is obtained by scaling $\sigma(l)$ by 100. In the first iteration, when both a masked and an unmasked embedding are available, the final score blends the two,

$$\text{score} \;=\; w_m\, \sigma\Big(\ell(\mathbf{v}_m, \mathbf{v}_{text})\Big) \;+\; w_u\, \sigma\Big(\ell(\mathbf{v}_u, \mathbf{v}_{text})\Big), \qquad w_m + w_u = 1, \tag{4.5}$$

with the masked term dominant ($w_m = 0.85$, $w_u = 0.15$ by default).

In theory the masked embedding provides object focus while the small unmasked contribution restores a measure of context, hedging against cases where the mask is imperfect or the object is only meaningful together with its surroundings. The blended masked–unmasked scoring strategy was originally proposed by OVO [25], which combines a masked crop for object focus with an unmasked crop to recover context, hedging against imperfect masks. This pipeline initially adopted the same strategy. However, OVO builds on CLIP, whereas the present pipeline uses SigLIP, whose pairwise sigmoid scoring already yields low scores for absent or poorly-matched objects without relying on relative, batch-level comparison. Under SigLIP, the unmasked context term was found to be redundant: using the masked crop alone proved both faster (one embedding per instance instead of two) and at least as accurate as the blend. For this reason, the unmasked crop was retained only as an optional component.

Heterogeneous Inference Backend

The semantic stage uses two backends deliberately. The image encoder, executed in batches every N_{skip} frames is compiled to a TensorRT engine in half precision (FP16) for throughput. The text encoder, by contrast, runs in PyTorch, because text embeddings are needed only when the goal prompt changes and therefore do not justify the engine-compilation effort. The asymmetry matches optimization effort to call frequency.

A consequence of mixing backends is the image-encoding data path. Preprocessing is performed on the GPU with PyTorch (resize on the CPU, then normalization, and channel reordering as vectorized GPU operations), producing a CUDA tensor. To hand this tensor to TensorRT without a round trip through host memory, the node performs a device-to-device copy directly from the PyTorch tensor's pointer into the TensorRT input buffer, after synchronizing the PyTorch stream so the preprocessing is guaranteed complete.

Avoiding the GPU to CPU to GPU detour removes a transfer that would otherwise sit on the critical path of every batch.

Prompt Ensembling for the Text Embedding

The goal text embedding is not computed from a single sentence. The label is expanded into a small ensemble of templated prompts (for example, “a photo of a {label}”, “a cropped photo of a {label}”, and similar variants); each is encoded, the resulting features are L2-normalized and averaged, and the mean is normalized to give a single robust embedding. Averaging over templates reduces the sensitivity of the match to any one phrasing and is a standard means of stabilizing zero-shot scoring.

Output Message

All per-frame results are packaged into a single custom message, `DetectedObjectV3Array`, whose elements carry the class name, instance identity, detection confidence, the segmentation mask, the masked and unmasked image embeddings, the goal text embedding, and the computed similarity. Packaging geometry and semantics together and publishing them atomically per frame ensures that downstream consumers receive a temporally consistent snapshot, avoiding the synchronization hazards that separate topics would introduce.

4.2 Persistent Semantic Mapping

The vision module described in the previous section produces, for each frame, a set of object detections: a segmentation mask, a class label, a confidence, a tracker identity, and a pair of semantic embeddings. These detections are inherently frame-local and transient, they describe what the camera sees at one instant, in the camera’s own reference frame. Navigation, however, requires a persistent object-level model of the environment in a fixed frame that accumulates evidence over time. Bridging this gap is the task of the mapper module.

The mapper consumes each detection together with the aligned depth image and maintains a global semantic map in which every entry corresponds to a single physical object, equipped with a three-dimensional pose, an oriented bounding box, a fused semantic embedding, a consensus class label, and a goal-similarity score. Building such a map from a moving, noisy sensor raises three coupled difficulties that are explained in this section.

The first is measurement noise. The depth camera produces corrupted and spurious returns, and the objects to be reconstructed have different size and point density, so no single filtering configuration can serve them all.

The second is geometric representation. Each object must be summarized by a compact, pose-invariant volume that downstream reasoning data association, duplicate detection, and navigation can use directly. An axis-aligned box, tied

Figure 4.4: Example of the generated semantic map

to the arbitrary orientation of the world frame, would loosely enclose a tilted or elongated object and overstate its footprint. The mapper instead fits an oriented bounding box (OBB) to each object's point cloud, aligned with the object's own principal axes, giving a tighter and more stable descriptor that does not change as the robot's viewpoint changes.

The third, and most consequential for the integrity of the map, is the unreliability of the detector itself. YOLOE does not emit clean, correct detections: it produces occasional false positives on background clutter, drops objects between frames, and might mislabel objects. A map that admitted every detection on first sight would faithfully reproduce all of these errors, populating the world model with phantom objects and wrong labels that the planner would then try to navigate to. A single noisy detection therefore must not be allowed to create a permanent map entry. This motivates the central safety mechanism of the mapper: a staged object life cycle, in which an object progresses from an unconfirmed tentative state to a permanent one only after it has been observed repeatedly, persistently over time, and with sufficient confidence. The life cycle is, in effect, a temporal filter on the detector's output, the counterpart at the object level to the geometric filtering applied at the point level.

4.2.1 Depth Denoising and Geometric Back-Projection

The depth sensor is the principal source of error in the pipeline. Its returns are corrupted by three characteristic phenomena: zero-mean measurement noise that grows with range; flying pixels at depth discontinuities, where a single pixel

integrates two surfaces at different depths and is projected to a spurious intermediate location; and sparse, isolated false returns. Left untreated, these errors inflate object extents, displace centroids, and seed phantom objects. Because the same sensor must reconstruct objects spanning orders of magnitude in size and point density, the filtering cannot be governed by a single fixed configuration. The node therefore applies a graded sequence of filters, two of which are made adaptive to the density of each individual cloud.

Image-Domain Denoising

Filtering begins before projection, on the depth image itself, with an edge-preserving bilateral filter. A linear smoother is inadmissible here: it would average depth across object silhouettes and manufacture exactly the flying pixels that must be removed. The bilateral filter instead weights each neighbour by both spatial and range proximity,

$$\hat{D}(\mathbf{p}) = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q}\in\mathcal{N}(\mathbf{p})} D(\mathbf{q}) \exp\left(-\frac{\|\mathbf{p}-\mathbf{q}\|^2}{2\sigma_s^2}\right) \exp\left(-\frac{\Big(D(\mathbf{p})-D(\mathbf{q})\Big)^2}{2\sigma_r^2}\right), \tag{4.6}$$

where:

- $\mathbf{p}$ and $\mathbf{q}$ are pixel coordinates in the depth image, with $\mathbf{p}$ the center pixel being filtered and $\mathbf{q}$ ranging over its neighborhood;
- $D(\mathbf{p})$ and $D(\mathbf{q})$ are the raw depth values (in meters) at those pixels;
- $\hat{D}(\mathbf{p})$ is the filtered depth value at $\mathbf{p}$;
- $\mathcal{N}(\mathbf{p})$ is the set of pixels within the filter window centered on $\mathbf{p}$;
- $W_{\mathbf{p}} = \sum_{\mathbf{q}\in\mathcal{N}(\mathbf{p})} \exp\Big(-\frac{\|\mathbf{p}-\mathbf{q}\|^2}{2\sigma_s^2}\Big) \exp\Big(-\frac{(D(\mathbf{p})-D(\mathbf{q}))^2}{2\sigma_r^2}\Big)$ is the normalization constant ensuring the weights sum to one;
- σ_s is the spatial bandwidth, controlling how steeply the influence of a neighbour falls off with pixel distance;
- σ_r is the range bandwidth, controlling how steeply the influence falls off with depth difference.

With σ_r set to a few centimeters, the range exponential vanishes whenever two pixels straddle a depth gap larger than that threshold, so the filter smooths within surfaces but refuses to smooth across boundaries.

Pinhole Back-Projection

For each detection, the binary mask is aligned to the depth resolution (nearest-neighbor resampling, to avoid introducing fractional mask values) and every in-mask pixel with a finite depth in the admissible range $[z_{\min}, z_{\max}]$ is back-projected through the pinhole model. Given intrinsics (f_x, f_y, c_x, c_y) and a pixel (u, v) with depth z, the camera-frame point is

$$X = \frac{(u - c_x)\, z}{f_x}, \qquad Y = \frac{(v - c_y)\, z}{f_y}, \qquad Z = z. \tag{4.7}$$

The range gate $z \in [z_{\min}, z_{\max}]$ (here $[0.3,3.0]$ m) discards returns that are too near to be valid or too far to be reliable, the latter being important because depth noise grows with range. The result is a per-detection cloud expressed in the camera optical frame.

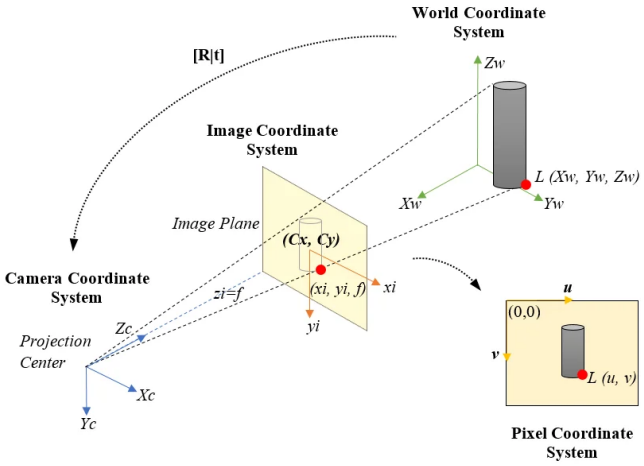


Figure 4.5: Pinhole projection [31]

Transformation to the Map Frame

Each cleaned cloud is transformed into the global map frame by the rigid transform looked up from the TF tree at the depth frame's timestamp,

$$\mathbf{x}^{\text{map}} \;=\; R\,\mathbf{x}^{\text{cam}} + \mathbf{t}, \qquad T_{\text{cam}}^{\text{map}} = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^{\top} & 1 \end{bmatrix} \in SE(3), \tag{4.8}$$

where the rotation R is obtained from the (normalised) quaternion reported by TF. Performing association in the map frame rather than the camera frame is what makes the map persistent: an object retains its coordinates as the robot moves, and detections from different viewpoints become directly comparable.

Density-Adaptive SOR and Voxel Filtering

The two principal cloud-domain filters are voxel-grid downsampling and statistical outlier removal.

- **Voxel-grid downsampling.** Space is partitioned into a regular grid of cubic cells of side ℓ (the leaf size); all points falling in a cell are replaced by their centroid. This bounds cloud density and fixes the computational cost of subsequent steps. The leaf size ℓ trades resolution against density: a small ℓ preserves fine structure (essential for small objects) but retains noise; a large ℓ aggressively simplifies (appropriate for large objects) but erases detail.
- **Statistical outlier removal (SOR).** For each point the mean distance $\bar{d}_i$ to its k nearest neighbours is computed; assuming these mean distances are approximately Gaussian with mean μ and standard deviation σ over the cloud, a point is rejected when
$$\bar{d}_i \;>\; \mu + \alpha\,\sigma, \tag{4.9}$$
where k (the neighbourhood size) and α (the standard-deviation multiplier) are the two parameters. SOR removes the isolated "floating" points characteristic of depth-sensor error. Its behaviour is again scale-dependent: a large k estimates local density robustly in a dense cloud but is meaningless in a sparse one where the object may have fewer than k points in total.

Linear Parameter Adaptation

Let n be the raw point count of a detection's cloud and let $[n_{\min}, n_{\max}]$ be a configured density band. Each filter parameter $\theta \in \{\ell,\ k,\ \alpha\}$ is scheduled between a small-object value $\theta_{\min}$ and a large-object value $\theta_{\max}$ by the clamped linear map

$$s(n) \;=\; \mathrm{clamp}\Big(\frac{n - n_{\min}}{n_{\max} - n_{\min}},\, 0,\, 1\Big), \qquad \theta(n) \;=\; \theta_{\min} + s(n)\,\Big(\theta_{\max} - \theta_{\min}\Big). \tag{4.10}$$

The interpolation factor $s(n)$ saturates outside the band, so that

$$\theta(n) = \begin{cases} \theta_{\min}, & n \le n_{\min} \quad \text{(small/sparse object)}, \\ \theta_{\min} + s(n)\,(\theta_{\max} - \theta_{\min}), & n_{\min} < n < n_{\max}, \\ \theta_{\max}, & n \ge n_{\max} \quad \text{(large/dense object)}. \end{cases} \tag{4.11}$$

The reasoning is that a sparse cloud, which almost certainly corresponds to a small object or a distant one, receives a lighter filtering using the smallest voxel (preserving every available point), the smallest neighbourhood k, and the mildest rejection threshold so that its limited geometry is not destroyed. A dense cloud, by contrast, can afford a coarse voxel and an aggressive outlier test, which suppress

noise and bound cost without losing the object. The mapping is continuous across the band, so objects of intermediate size are filtered proportionally rather than being forced into one of two regimes. This single mechanism, recomputing ℓ, k and α from n on every detection via Equation (4.10) is the central solution to the noise-versus-scale issue.

The same adaptation law is applied a second time, on the accumulated cloud, whenever new geometry is fused into an existing object; there the relevant count is the size of the merged cloud, so a growing object naturally migrates toward the coarser regime as evidence accumulates.

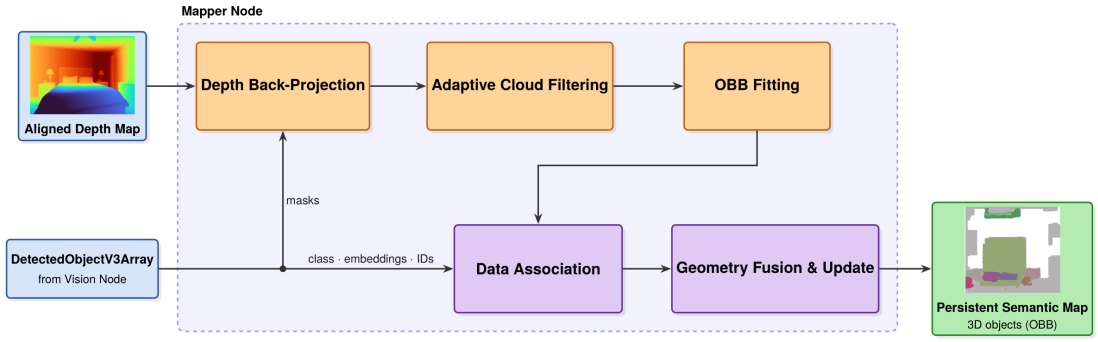


Figure 4.6: Block diagram for filtering process in mapper node

4.2.2 Geometric Representation with Oriented Bounding Boxes

Each object in the map is summarized by a compact volumetric descriptor computed from its accumulated point cloud. The natural first choice is axis-aligned bounding box (AABB), that is the smallest box with faces parallel to the world axes that contains the cloud. It is trivial to compute but is tied to the arbitrary orientation of the map frame: a long, thin object lying diagonally in the world frame is enclosed by a large AABB whose volume is dominated by empty space. For data association, duplicate resolution, and navigation, a descriptor that depends on the world frame in this way is undesirable.

The mapper therefore uses an oriented bounding box (OBB): a compact box that contains the cloud, but free to align with the object's own dominant directions rather than with the world axes. The orientation is recovered from the cloud's second-order statistics by principal component analysis (PCA), which makes the descriptor tight and approximately invariant to the object's pose in the map frame.

Let the cloud be $\{\mathbf{x}_i\}_{i=1}^{n}$ with centroid $\bar{\mathbf{x}}$ and covariance

$$\bar{\mathbf{x}} = \frac{1}{n}\sum_i \mathbf{x}_i, \qquad \Sigma = \frac{1}{n}\sum_i (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^\top. \tag{4.12}$$

As Σ is real symmetric positive semi-definite, it admits an orthonormal eigendecomposition $\Sigma = R\,\Lambda\,R^\top$ whose eigenvectors (the columns of R) are the principal axes of the cloud and whose eigenvalues are the variances along them. The orthogonal matrix R is the rotation from the world frame to the object's principal frame. Since an eigenbasis is fixed only up to sign, the third axis is set to $\mathbf{r}_3 \leftarrow \mathbf{r}_1 \times \mathbf{r}_2$ to force a proper right-handed rotation ($\det R = +1$), as required for the quaternion export and point transforms.

The extents follow by expressing the cloud in the centred principal frame and taking its axis-aligned span there,

$$\mathbf{x}'_i = R^\top(\mathbf{x}_i - \bar{\mathbf{x}}), \qquad e_k = \max_i x'_{i,k} - \min_i x'_{i,k}, \quad k \in \{1,2,3\}. \tag{4.13}$$

The box is then fully specified by the triple $(\bar{\mathbf{x}}, R, \mathbf{e})$ of centre, orientation, and extents. Its eight corners are the half-extent sign combinations mapped back to the world frame, $\mathbf{c}_j = \bar{\mathbf{x}} + R\,(\pm e_1/2,\, \pm e_2/2,\, \pm e_3/2)^\top$.

The same transform yields the containment test used by the association and merge logic: a world point $\mathbf{p}$ lies inside the box iff its local coordinates $\mathbf{p}' = R^\top(\mathbf{p} - \bar{\mathbf{x}})$ satisfy $|p'_k| \le e_k/2 + \varepsilon$ on every axis. Because R is orthogonal, this reduces the rotated-box test to a trivial axis-aligned comparison with no matrix inversion, and is the primitive from which the point-sampling overlap ratio between two objects is built.

4.2.3 The Life Cycle of a Mapped Object

As previously stated, the physical object does not enter the permanent map on first sight. Requiring it to do so would import every transient false detection of the noisy detector directly into the world model. Instead, each object passes through a three-state life cycle: UNDETECTED → TENTATIVE → PERMANENT, in which evidence must accumulate and pass explicit quality gates before an observation is granted permanence. The cycle implements a track-before-confirm discipline: it trades a short confirmation latency for strong protection against spurious objects. Algorithm 2 enumerates every path a single detection can take through this cycle; the three transitions are then derived in turn.

UNDETECTED → TENTATIVE

Every tracker identity begins as UNDETECTED: no record of it exists in either the TENTATIVE or the PERMANENT store. A detection can create a TENTATIVE

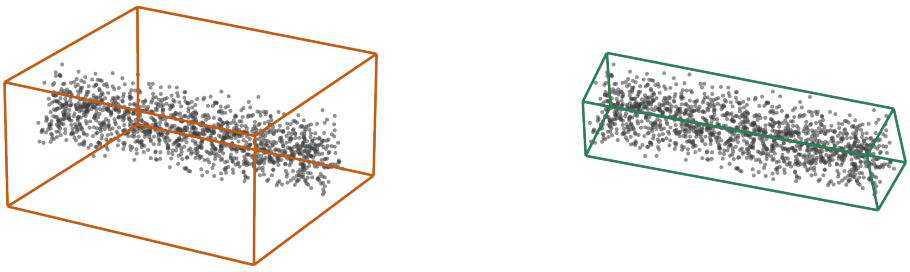


Figure 4.7: Comparison between Axis Aligned Bounding Box (on the left) and Oiented Bounding Box (on the right)

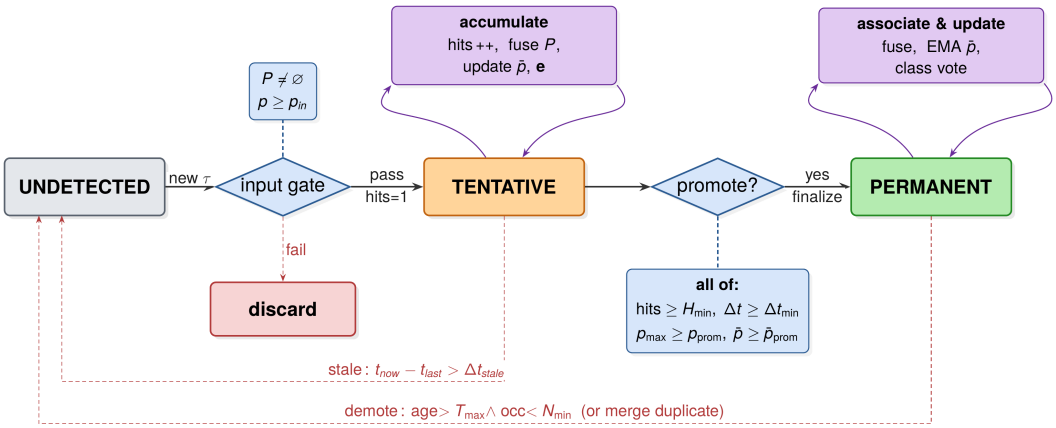


Figure 4.8: object lifecycle diagram

track only after clearing an input quality gate that rejects unusable evidence,

$$P \neq \varnothing \quad \text{and} \quad p \geq p_{\text{in}}, \tag{4.14}$$

where P is the detection's map-frame cloud and p_{in} is the minimum input confidence. A detection that fails Equation (4.14) is discarded and the identity remains UNDETECTED; this is the cheapest rejection in the system and keeps low-confidence noise out of the state entirely.

Algorithm 2: Routing of one detection through the object life cycle

Input: detection d with tracker id τ, class c, confidence p, map-frame cloud P, embeddings $\mathbf{e}$

Data: bindings $\mathcal{B} : \tau \to$ map id; permanent objects $\mathcal{O}$; tentative tracks $\mathcal{T}$

if $\tau \in \mathcal{B}$ ***and*** $\mathcal{B}[\tau] \in \mathcal{O}$ ***and*** $c = \mathcal{O}[\mathcal{B}[\tau]].class$ **then**
  UpdatePermanent($\mathcal{B}[\tau], d$)
  return

$j^\star \leftarrow$ Associate($d, \mathcal{O}$) **if** $j^\star \neq \varnothing$ **then**
  UpdatePermanent($j^\star, d$); $\mathcal{B}[\tau] \leftarrow j^\star$
  return

if $P = \varnothing$ ***or*** $p < p_{in}$ **then**
  discard return

if $\tau \notin \mathcal{T}$ **then**
  create tentative track t with hits=1, init stats from d
  $\mathcal{T}[\tau] \leftarrow t$
else
  $t \leftarrow \mathcal{T}[\tau]$; fuse P into t; hits + =1; update stats

if Promote(t) **then**
  $m \leftarrow$ new permanent object from t's accumulated evidence
  $\mathcal{O}[m] \leftarrow$ finalize(t); $\mathcal{B}[\tau] \leftarrow m$; remove t from $\mathcal{T}$

PruneStaleTentatives($\mathcal{T}$); RemoveWrongPermanents($\mathcal{O}$)

A detection that passes, and whose tracker id is not yet known, instantiates a fresh TENTATIVE track initialized with a single hit, the detection's cloud, its class, and its confidence (recorded both as a running sum and a maximum), together with the normalized embeddings if present. The identity has now moved to TENTATIVE. Crucially, a tentative track is held in a separate store and is never published as a map object, so nothing observed for the first time can yet influence navigation.

Accumulation Within the Tentative State

While TENTATIVE, repeated detections of the same tracker id fuse into the existing track rather than creating new ones. Each repeat increments the hit count

and updates the track's running statistics:

$$\text{hits} \leftarrow \text{hits} + 1, \tag{4.15}$$

$$\Sigma_p \leftarrow \Sigma_p + p, \qquad p_{\max} \leftarrow \max(p_{\max},\, p), \tag{4.16}$$

$$\mathbf{e} \leftarrow \frac{(\text{hits} - 1)\,\mathbf{e} + \mathbf{e}_{\text{new}}}{\|(\text{hits} - 1)\,\mathbf{e} + \mathbf{e}_{\text{new}}\|}, \tag{4.17}$$

where Σ_p is the confidence sum and Equation (4.17) is the count-weighted running mean of the embedding, re-normalized to the unit sphere. The accumulated cloud is fused by concatenation followed by the same density-adaptive voxel/SOR pass explained in previous section, so the track's geometry improves with each sighting instead of being overwritten. The average confidence used for promotion is the derived quantity

$$\bar{p} = \frac{\Sigma_p}{\max(1, \text{hits})}. \tag{4.18}$$

A tentative track that stops being seen is not kept indefinitely: a periodic reaper removes any track whose time since last update exceeds a staleness horizon,

$$t_{\text{now}} - t_{\text{last}}^{(\tau)} > \Delta t_{\text{stale}} \;\Rightarrow\; \text{delete tentative } \tau, \tag{4.19}$$

returning that identity to UNDETECTED. This is the path by which a one or two-frame false detection silently disappears without ever reaching the map.

TENTATIVE → PERMANENT

Promotion is the gate that admits an object to the permanent map and it requires all of four conditions to hold simultaneously,

$$\textsc{Promote}(t) \;=\; \Big(\text{hits} \geq H_{\min}\Big) \wedge \Big(\Delta t \geq \Delta t_{\min}\Big) \wedge \Big(p_{\max} \geq p_{\text{prom}}\Big) \wedge \Big(\bar{p} \geq \bar{p}_{\text{prom}}\Big), \tag{4.20}$$

with the track age $\Delta t = t_{\text{last}} - t_{\text{first}}$. The four terms guard against distinct failure modes and are not redundant:

- **Persistence in count**, hits $\geq H_{\min}$: the object must be seen at least $H_{\min}$ times, eliminating single-frame detections regardless of how confident they were.
- **Persistence in time**, $\Delta t \geq \Delta t_{\min}$: the sightings must span at least $\Delta t_{\min}$ seconds. This is independent of the hit count: a burst of detections within a single instant can satisfy $H_{\min}$ yet still fail here, which rejects a momentary flicker that the tracker happened to report several times.
- **Peak quality**, $p_{\max} \geq p_{\text{prom}}$: at least one sighting must have been strongly confident, ensuring the track is not built entirely from marginal evidence.

- **Sustained quality**, $\bar{p} \geq \bar{p}_{\text{prom}}$: the average confidence must clear a floor, so a single lucky high-confidence frame cannot by itself carry an otherwise weak track over the line.

Only when Equation (4.20) holds is a new permanent object instantiated. It inherits the tentative track's accumulated cloud, fused embeddings, and class, and is finalized through the standard permanent-update routine using the track's average confidence as its initial confidence. The tracker id is then bound to the new map object, $\mathcal{B}[\tau] \leftarrow m$, and the tentative record is deleted. The identity is now PERMANENT.

Associating Detections to PERMANENT Objects

Once an object is permanent, subsequent detections must be routed back to it so that its evidence keeps accumulating. This association runs in three layers of increasing cost, evaluated in order, so the expensive global optimization is reached only when the cheap paths fail.

identity binding. If the detection's tracker id τ is already bound to a permanent object and the class label agrees, the detection updates that object directly in constant time. This is the most common case, since the upstream tracker keeps ids stable across frames, and it requires no geometric computation at all.

gated screening. For an unbound detection, candidate permanent objects are first pruned by a hard spatial gate that scales with object size,

$$d(\text{det}, \text{obj}) > \max\Big(0.4,\ 1.2\,\text{maxSize}\Big) \ \Rightarrow\ \text{rejected}, \tag{4.21}$$

where d is the Euclidean distance between oriented-box centres and maxSize is the larger of the two longest extents; the radius grows with size because a large object's centroid legitimately shifts more between viewpoints. Survivors are ranked by a cheap distance-only cost and only the top-K are passed to the full evaluation, bounding the cost per detection regardless of map size.

combined cost and global assignment. Each shortlisted detection–object pair is scored by a combined cost,

$$C = w_d\, d + w_o\,(1 - \text{IoU}) + w_s\,\Big(1 - \cos(\mathbf{e}_{\text{det}}, \mathbf{e}_{\text{obj}})\Big) + P\,\mathbb{1}[\text{class mismatch}], \tag{4.22}$$

combining spatial distance, geometric non-overlap, semantic dissimilarity (the cosine of the L2-normalized embeddings), and a class-mismatch penalty. The distance and overlap weights w_d, w_o are themselves scheduled by object size through the

adaptation law of Equation (4.10): small objects weight distance heavily, large objects weight overlap. The resulting cost matrix over unmatched detections and candidate objects is solved for a globally optimal one-to-one assignment by the Hungarian algorithm,

$$\pi^\star = \arg\min_{\pi} \sum_{i} C\Big(i, \pi(i)\Big), \tag{4.23}$$

and an assignment is accepted only if its cost is below a rejection threshold. A detection whose best available match is still too costly is not forced onto a permanent object; it falls through to the tentative path of Section 17, where it may later form its own object, if it persists. This rejection branch is what lets a genuinely new object appear even when it happens to be near an existing one.

Updating the matched object. An accepted detection updates its permanent object by the recursive rules already introduced: geometry is fused and re-filtered, the embedding is updated by the count-weighted running mean, confidence by an exponential moving average

$$\bar{p}_{\text{obj}} \leftarrow (1-\beta)\,\bar{p}_{\text{obj}} + \beta\, p, \tag{4.24}$$

and the class label by weighted voting with hysteresis. The binding $\mathcal{B}[\tau] \leftarrow m$ is refreshed so that future detections of the same tracker take the Tier-1 fast path.

Demotion of False Permanents

Permanence is strong but not unconditional. A periodic cleanup removes any permanent object that has aged beyond a maximum horizon without having gathered enough corroborating observations,

$$\text{age} > T_{\max} \quad \wedge \quad \text{occurrences} < N_{\min} \;\Rightarrow\; \text{delete permanent object}, \tag{4.25}$$

This reaps objects that were promoted on a brief burst of evidence but were never seen again, the residual false positives that survive the promotion gate. Separately, two permanent objects of the same class whose oriented overlap exceeds a strict threshold are merged into one, the better-observed absorbing the other. Together these passes keep the invariant that the permanent map holds exactly one entry per real object and none for noise.

4.2.4 Map Serialization

The object map is persisted to disk as a JSON document on a periodic timer, so that the current world model is continuously available to the modules that consume it offline (like the language module of the previous chapter). Only permanent

objects are written; tentative tracks are excluded. The document is an associative store keyed by stable object identifier, each value recording the object's class, pose, oriented bounding box (extents, quaternion orientation, and corners), fused embeddings, and accumulated evidence. The schema of a single record is shown in Listing 4.1.

```
"map_obj_<id>": {
    "name": <string>,                 // consensus class label
    "frame": "map",
    "timestamp": { "sec": <int>, "nanosec": <int> },
    "pose_map": { "x": <float>, "y": <float>, "z": <float> },
    "bbox_type": "obb",
    "box_size": { "x": <float>, "y": <float>, "z": <float> },
    "bbox_orientation": { "x": <float>, "y": <float>,
                          "z": <float>, "w": <float> },
    "bbox_corners": [ { "x": <float>, "y": <float>, "z": <float>
    },
                      ... ],           // eight corners, map
   frame
    "occurrences": <int>,
    "similarity": <float>,
    "image_embedding_masked":   [ <float>, ... ],
    "image_embedding_unmasked": [ <float>, ... ],
    "embedding_confidence": <float>,
    "confidence": <float>
}
```

Listing 4.1: Structure of a serialised map object.

4.3 Natural-Language Command Interpretation

The components described so far give the robot a persistent, semantic model of its surroundings, but they do not address how a human expresses what the robot should do. The natural interface for a non-expert operator is ordinary language: an instruction such as *"go to the fork"* carries an intent that must be turned into a concrete, machine-readable plan. This chapter describes the module responsible for that translation. It listens to a free-form instruction in natural language and emits a single structured command that the downstream behavior tree can consume directly, organized so that the downstream behavior tree can consume it directly and decide what to do. The module is, in effect, the bridge between human intent and the symbolic plan that drives execution.

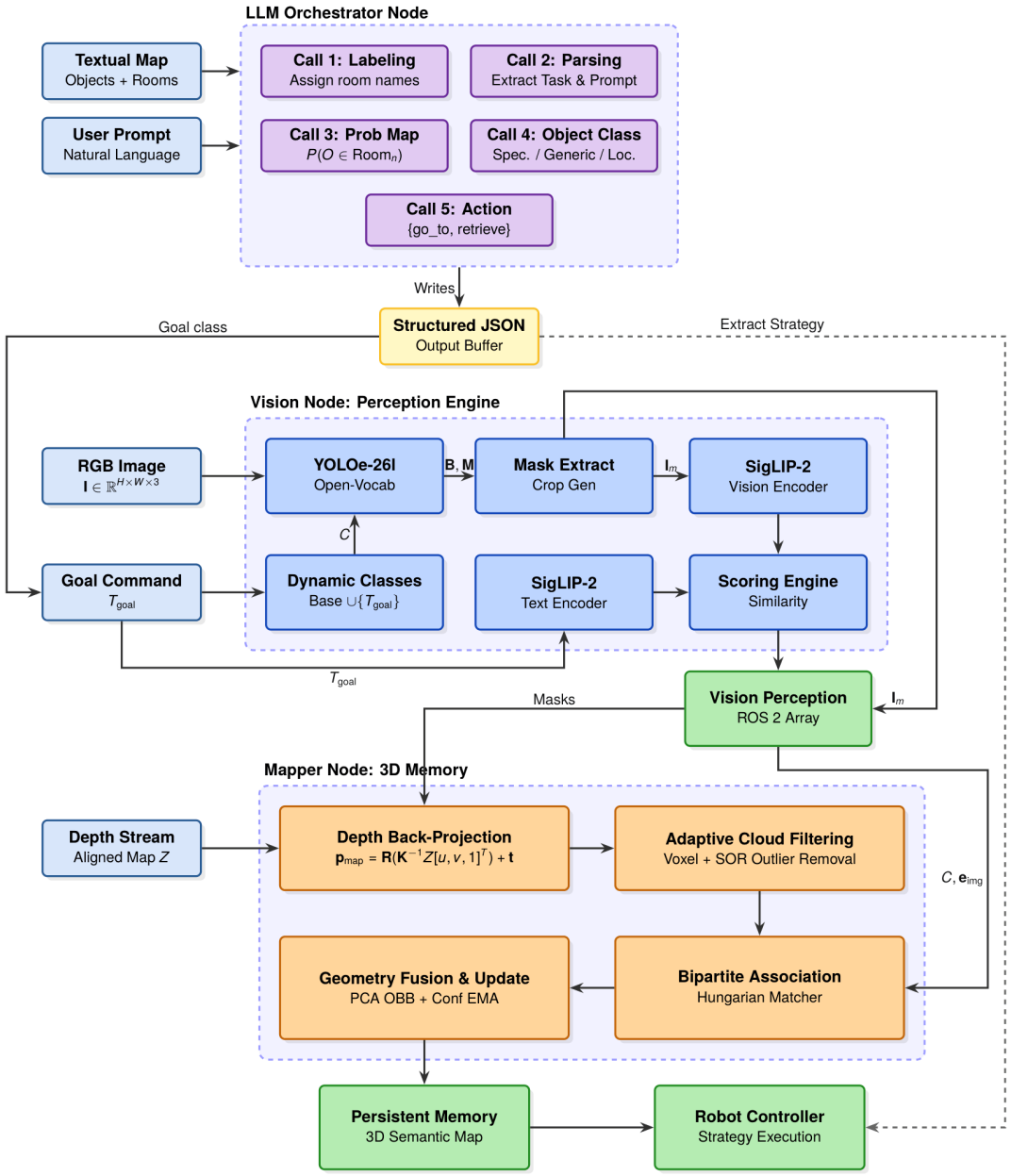


Figure 4.9: Full pipeline that goes from LLM processing to persistant map

4.3.1 Background: Large Language Models

A large language model (LLM) is a neural network, almost always based on the transformer architecture [32], trained on very large text corpora to model the

conditional distribution of the next token given those preceding it. A model trained at sufficient scale acquires a broad store of commonsense and world knowledge and the ability to follow instructions expressed in natural language.

Two properties of such models are what make them suitable here. First, their world knowledge lets them resolve implicit references that a rule-based parser could not: that a fork is typically found in a kitchen, or that "the thing I eat with" denotes cutlery, is knowledge the model already holds and that a fixed grammar would have to enumerate by hand. Second, instruction following allows the model to be steered, through a carefully written *system prompt*, into returning its answer as structured data rather than prose. This is decisive for an autonomy stack since a behavior tree cannot act on a paragraph of free text, but it can act on a JSON object with known fields.

The model is used strictly as a *zero-shot* reasoner: it is not fine-tuned for this application and receives no task-specific training. All task knowledge is supplied at inference time through the system prompt and the contextual information (the map and its clusters) injected into it. This keeps the module flexible since the behavior can be changed by editing a prompt rather than retraining, and avoids the cost and data requirements of fine-tuning.

Model Choice and Local Deployment

The module runs `Llama-3.1-8B-Instruct` [33], an instruction-tuned model of eight billion parameters, loaded locally through the `transformers` library [34] and served entirely offline. Local, offline operation is a deliberate requirement. A mobile robot cannot depend on a network connection to a remote inference service for its command interpretation, and on-board execution avoids the latency and privacy concerns of a cloud API. Accordingly, the weights are loaded in half precision (FP16) on the GPU to fit the on-board memory budget and to accelerate inference.

The eight-billion-parameter scale is chosen as a compromise: large enough to carry the commonsense reasoning the task needs, yet small enough to run on the robot's accelerator alongside the perception stack.

4.3.2 From Instruction to Structured Command

A single monolithic prompt that asked the model to emit the entire command at once would be fragile: the model would have to perform several unrelated reasoning steps simultaneously, and a mistake in any one would corrupt the whole output. The module instead *decomposes* the problem into a sequence of narrow sub-queries, each handled by its own LLM call with its own dedicated system prompt and its own small output schema. Each call asks one well-defined question, which both improves reliability and allows the sampling temperature to be tuned per task.

Algorithm 3: Instruction-to-command pipeline

Input: user instruction u (natural language)
Data: semantic map M; clustered map M_c; cluster labels L
(once, at start-up) $L \leftarrow \textsc{AssignClusterLabels}(M_c)$
$(\text{goal}, \text{clip}) \leftarrow \textsc{ExtractGoal}(u, M)$ clu $\leftarrow$ $\textsc{DetermineCluster}(u, \text{goal}, M_c, L)$ act $\leftarrow \textsc{ExtractAction}(u)$ log $\leftarrow$ $\textsc{DecideLogic}(u)$
validate & bind anchor to selected cluster
assemble NavResult; serialise to `robot_command.json`

The structured results are then assembled into the final command. The stages are summarised in Algorithm 3 and detailed below.

Querying the Model and Recovering Structured Output

Every sub-query follows the same logic. A task-specific system prompt is loaded from a template file and combined with the user instruction into a chat message list, which is rendered into the model's expected format by its chat template and passed to the generation pipeline. The system prompt instructs the model to answer *only* with a JSON object of a prescribed shape.

Because a language model cannot be guaranteed to emit syntactically clean JSON (it may wrap the object in prose, in Markdown fences, or in explanatory text) the raw response is passed through a tolerant extraction routine that attempts, in order: parsing the whole response as JSON; locating a (possibly nested) JSON object by a brace-matching regular expression; and extracting an object from a fenced code block. The first candidate that parses and contains the expected keys is accepted. The extracted object is then validated against a `pydantic` schema — one per sub-query (`Goal`, `ClusterPrediction`, `Action`, `Logic_decision`) — which enforces the field names and types and supplies defaults for missing optional fields. Any failure in this chain triggers a bounded retry: the call is repeated up to three times before the error is propagated. This extract-validate-retry loop is the mechanism that turns a probabilistic text generator into a dependable producer of typed data.

Temperature as a per-Task Control

The sampling temperature is set independently for each sub-query. Decisions that are essentially classifications over a small fixed set, like the action and the logic branch, use a very low temperature (0.1–0.2) so the output is nearly deterministic.

Goal extraction, which must occasionally rephrase or normalize an object name, uses a low but non-zero temperature (0.2). The cluster and labeling steps, which are inferential and benefit from some diversity, use a moderate temperature (0.4). Temperature is thus used as a dial that trades determinism against flexibility, matched to the nature of each decision.

The Sub-Queries

Cluster labelling (start-up). Once at start-up, the model is shown the objects belonging to each spatial cluster of the map and asked to assign each a human-meaningful room label (for example, *kitchen* or *bedroom*). The resulting labels are cached and reused by every later call that needs spatial context. Where two clusters receive the same label, the module differentiates them with a numeric suffix, so the labels remain unique identifiers.

Goal extraction. The first per-instruction call identifies the target object. Its system prompt is parameterized with the list of object classes present in the map, so the model is steered toward naming a class the robot can actually perceive. It returns two fields: a `goal`, the object class to navigate to, and a `clip_prompt`, a short natural-language phrase describing the object that is later handed to the vision node's text encoder for open-vocabulary matching. If the model returns a phrase but no explicit goal, the phrase is used as the goal hint, so the pipeline degrades gracefully rather than failing.

Cluster determination and anchor selection. The second call performs the most substantial reasoning. Given the instruction, the extracted goal, and a textual description of every labeled cluster together with the candidate object identifiers it contains, the model predicts which cluster is most likely to hold the goal and, optionally, which specific object within that cluster should serve as a spatial *anchor* for the search. It also returns a free-text `reasoning` string and a `location_confidence` in [0,1]. This is where the model's commonsense is exploited directly: asked for a fork with no room specified, it infers the kitchen because that is where forks are typically found.

Action and logic. Two further low-temperature calls classify the instruction into a high-level `action` (the operation the robot must perform, e.g. navigating to an object) and a `logic` branch that selects which sub-tree the behavior tree should execute. Both are deliberately constrained, near-deterministic decisions.

Validation and Safeguarding of the Model Output

The model's output is never trusted blindly; it is treated as a proposal to be checked against the ground truth of the map. Several guards are applied deterministically after generation. A returned `cluster_id` is accepted only if it corresponds to a cluster that actually exists; otherwise the cluster selection is cleared. The `location_confidence` is coerced to a float and clamped to [0,1]. Most importantly, a proposed anchor object identifier is accepted only if it genuinely belongs to the selected cluster: if the model hallucinates an identifier, or names an object from a different cluster, the anchor is discarded rather than passed downstream. These checks ensure that even a malformed or over-confident generation cannot inject an invalid reference into the command that the behavior tree will act on.

The Output Command

The validated results of all sub-queries are assembled into a single message and serialized to `robot_command.json`, the file the behavior tree reads. It carries the originating prompt and timestamp, the goal class and its CLIP phrase, the selected anchor object and its class, the cluster context (including the model's reasoning and confidence), and the action and logic decisions. Listing 4.2 shows a representative command produced for the instruction *"go to the fork"*: the model has correctly extracted `fork` as the goal and, with no room named, reasoned that a fork belongs in the kitchen cluster, selecting a bottle in that cluster as the spatial anchor.

```
{
    "prompt": "Go to the fork",
    "goal": "fork",
    "clip_prompts": "fork",
    "anchor_object_id": "map_obj_000019",
    "anchor_object_class": "bottle",
    "cluster_info": {
        "cluster_id": 4,
        "reasoning": "No room mentioned and 'fork' is a common
    object in the kitchen.",
        "location_confidence": 0.6,
        "prioritize_location": false
    },
    "action": "go_to_object",
    "logic": "GENERIC_OBJECT"
}
```

Listing 4.2: A generated `robot_command.json` (abridged).

This command is the sole output of the module and the complete specification the behaviour tree needs: a typed, validated, self-contained description of what the robot has been asked to do, derived from a single sentence of natural language.

4.4 Autonomous Navigation Orchestration

4.4.1 Behavior Tree [35]

A Behavior Tree (BT) is a hierarchical, modular formalism for encoding reactive decision-making in autonomous agents. Originally developed in the video game industry for non-player character AI, BTs have become a standard architectural pattern in robotics due to their properties of reactivity, modularity, and readability, qualities that finite state machines (FSMs) and scripted sequences struggle to provide at scale [35].

A BT is a directed tree in which execution flows from a root node downward through composite nodes (such as Sequence and Fallback) that encode control logic, down to leaf nodes that represent atomic actions or conditions. At each tick of the tree, every active node returns one of three statuses: `SUCCESS`, `FAILURE`, or `RUNNING`, which propagate upward and drive the overall control flow. This tick-based execution model makes BTs inherently reactive: changes in the environment are reflected in the next tick without requiring explicit state transitions to be defined between every pair of states, a well-known scalability bottleneck of FSMs. To enable data sharing across nodes without introducing direct coupling between them, BTs rely on a blackboard: a shared key-value store accessible by all nodes in the tree. Leaf nodes read from and write to the blackboard through named ports, allowing actions and conditions to exchange state information — such as a navigation target, a detected object identifier, or a replanning flag — in a decoupled and thread-safe manner [35].

In the context of this work, the behavior tree is implemented using the `BehaviorTree.CPP` library within a ROS 2 node, and serves as the central orchestration layer of the semantic object navigation pipeline. At each tick, the tree reads two external data sources: a textual object map produced by the perception module and a command file generated by the Large Language Model, which encodes the high-level navigation intent, target object identity, and any auxiliary behavioral constraints. Both sources are parsed and written onto the blackboard at startup, making their contents uniformly accessible to all nodes throughout execution. Based on this structured input, the tree dispatches navigation goals to the robot's motion stack and continuously monitors execution, handling outcomes such as goal reached, target not found, and replanning triggers through dedicated subtrees. This design cleanly separates what to do from how and when to do it, keeping the architecture both interpretable and robust to runtime failures.

4.4.2 Implementation

What has been presented so far enables the extraction of semantic information about the navigation environment and the actions to be performed. However, the methodology that allows this information to be used in pursuit of a goal has not yet been introduced. The objective of this project is to develop a solution that allows an agent to navigate towards objects in a human-like manner, through the fusion of priors, spatial information, and semantic information about the environment.

As already discussed during the exploration, the robot is able to recognize objects and place them in space. All the information concerning the objects — including a vector representation through embeddings — is stored in a textual map. The situation in which the agent must navigate towards a generic, already-identified object is therefore trivial: in that case the agent selects the object of that class which is closest. This, however, represents only an extremely small portion of the object-navigation task that humans are normally able to perform.

Navigation Cases

The further situations that the pipeline must be able to handle are the following:

- **Specific object**. We are often interested in finding a specific object, rather than just any object belonging to a generic semantic class. The specificity of the object can be expressed by two factors: one related to its visual features and one related to its placement in the environment. One might wish to reach an object with a specific colour or a specific pattern (such as the phone with the light-blue case, or the water bottle with the PoliTo logo), or an object located in a specific position (for example, the book on the nightstand, or more generally the jacket left in the bedroom).
- **Undiscovered object**. A more difficult condition is the one in which the requested object has not yet been discovered. In this case, exploration of the environment is required in order to search for that object. In order to emulate human behavior in this condition as well, it is necessary to use the combination of priors and information extracted from the environment, in the same way that a human, when asked to reach a specific object (e.g. a glass), would use the knowledge they already have about the environment they are in together with the inferences they can draw about the request (for example, assuming that the glass is in the kitchen).

Logic Labels

As introduced in the section concerning the LLM, the language model extracts, from the natural-language prompt, a *logic label* that identifies the case in which

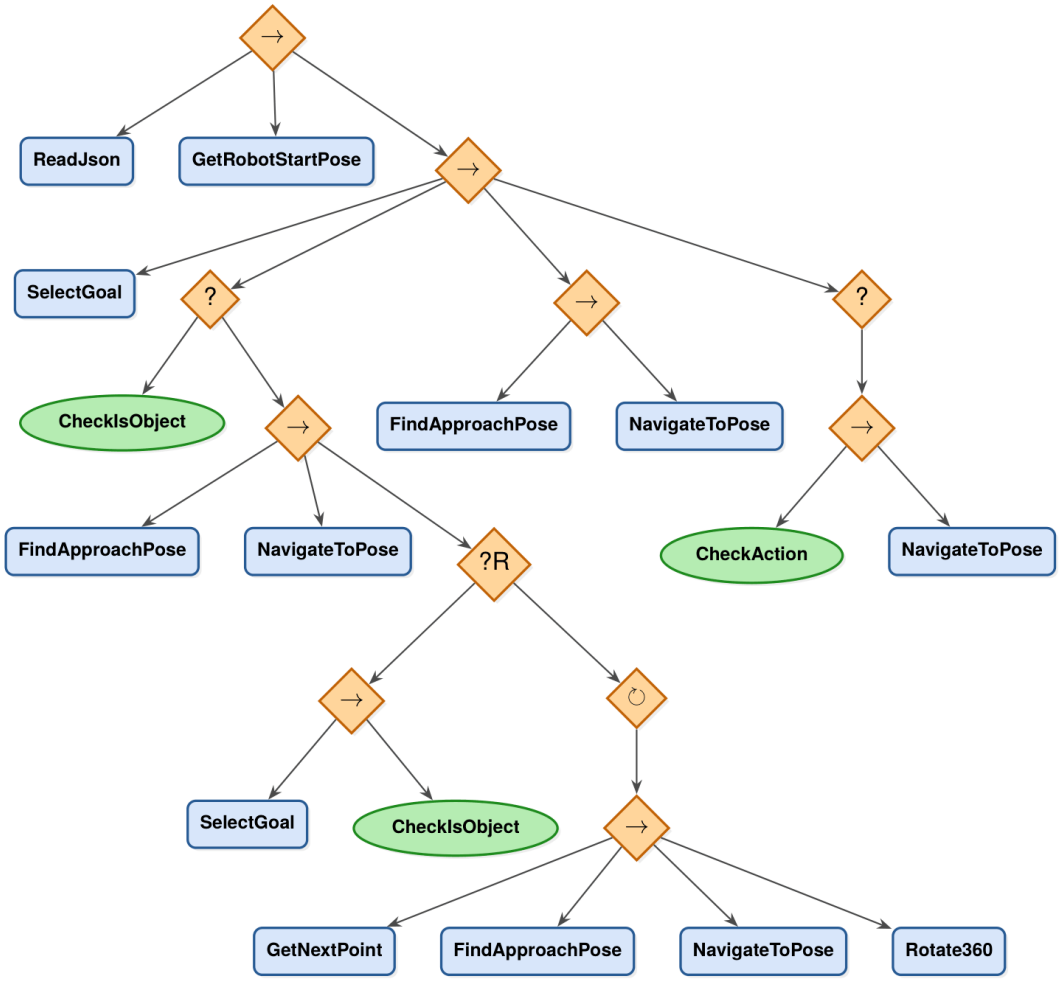


Figure 4.10: behavior tree

the agent must operate. Each case corresponds to a different type of behavior and a different decision process. The three possible labels are:

- `GENERIC_OBJECT`: a generic object is requested, without any specification of location or visual features.
- `GENERIC_OBJECT_SPECIFIC_LOCATION`: the specificity of the object is given by its geospatial relationship with the environment, that is, by the room it belongs to or by its proximity to another object.
- `SPECIFIC_OBJECT_WITH_FEATURES`: the desired object is identified by specific visual features.

Behavior Tree Structure

The entire mission logic is encoded in a single tree, whose root is a `Sequence` that chains together the three fundamental phases of the pipeline: command acquisition, decision, and navigation. The first phase is entrusted to two leaf nodes executed in succession. The `ReadJson` node reads the command file `robot_command.json` generated by the LLM and deserialises its contents onto the blackboard, exposing the goal class (`goal_class`), the target cluster (`cluster`), the action to perform (`robot_action`), the logic label (`logic`), and any information about the "anchor" object (`anchor_class`, `anchor_id`). The `GetRobotStartPose` node queries the `TF` transform and writes the robot's current pose (`start_pose`) onto the blackboard, which will serve as a reference for the subsequent proximity decisions.

The second phase, enclosed in the `DecideAndNavigate` sub-sequence, constitutes the reactive core of the tree. It begins with the decision node `SelectGoal` — described in detail in the following section — which, based on the logic label and the current state of the map, determines *what* the robot must reach and produces as output a boolean flag (`is_object`) indicating whether the identified target is an actual goal object or merely an exploration point. This flag drives a `Fallback` node (`ChooseNavigationStrategy`) that selects the navigation strategy: if `CheckIsObject` confirms that an object has been identified, the exploration branch is skipped and the tree proceeds directly towards the goal; otherwise, the tree activates the cluster-exploration subtree.

Once navigation is complete, a final phase (`PostArrivalLogic`) handles the actions following arrival: the `CheckAction` node verifies the nature of the requested action and, should it involve retrieving the object, the robot returns to its starting pose (`start_pose`).

The separation between what has been decided and the way in which it is executed — made possible by the blackboard, which decouples the nodes by letting them share state without direct dependencies — is what makes the tree at once readable and robust: each leaf node performs an atomic task and communicates exclusively through the input and output ports mapped onto the blackboard.

Generic Object

When a generic object is requested — that is, without any specification of location or visual features — the solution is trivial. The behavior tree selects the object of the desired class which is closest and navigates towards it, ignoring information about the position or appearance of the various objects (e.g. reach the nearest computer). This is the behavior when the selected label is `GENERIC_OBJECT`.

Object with Specific Features

The `SPECIFIC_OBJECT_WITH_FEATURES` case demonstrates the centrality of the visual memory developed through the use of SigLIP. In this situation the behavior tree compares the objects present in the map with the textual description of the visual features of the desired object. For each object a *similarity* value is computed which, if above a certain threshold specified as a parameter, identifies the object as a match. Should no object match the description — that is, the similarities remain below the threshold — the behavior tree assumes that the desired object has not yet been discovered, and adapts its behavior accordingly, as described in the section on exploration.

Object in a Specific Location

The `GENERIC_OBJECT_SPECIFIC_LOCATION` case is the one in which the specificity of the object is given by its geospatial relationship with the other objects. The indicated location has been simplified into two possible cases: proximity to an object, or membership of a room.

The first case is fairly trivial: the object is selected on the basis of its proximity to an "anchor" object. The second case, on the other hand, is not equally obvious, since it requires partitioning the map into rooms and assigning each object to the room it belongs to. The mechanism that realises this partition is described in the following section.

Room Segmentation and Object Clustering

Handling goals expressed in terms of a room requires partitioning the map into semantically coherent regions and assigning each detected object to the room it belongs to. This task is delegated to a dedicated ROS 2 node, which operates in parallel with the behavior tree and exposes the segmentation results to it through a pair of services.

Segmentation algorithm. The partitioning of the map is based on the *watershed* algorithm, applied to the occupancy grid generated during exploration. The choice of this algorithm is motivated by its ability to separate regions connected by narrow passages — such as the doorways between one room and another — without requiring any a priori geometric model of the environment nor a predefined number of clusters. The pipeline is structured into the following steps:

1. From the occupancy grid, a binary mask of the free space is extracted, considering as navigable only those cells with a value of zero.

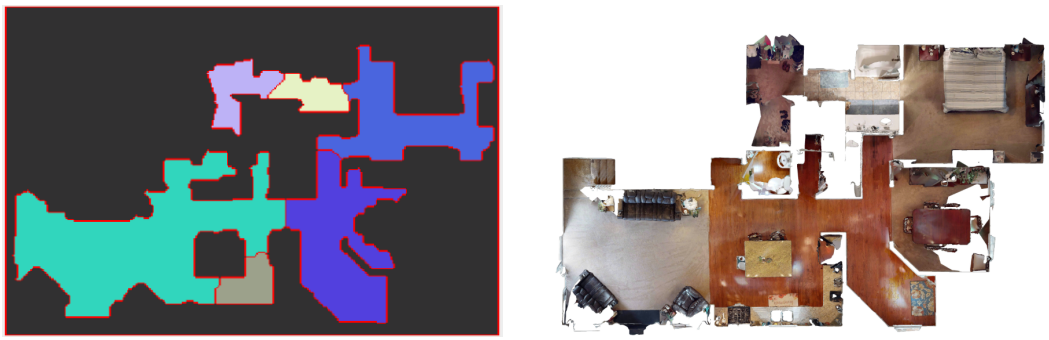
2. On this mask, the *distance transform* (the Euclidean distance of each free cell from the nearest obstacle) is computed. The local maxima of this transform correspond to the centers of the rooms, while the low values are concentrated in the vicinity of walls and doors.

3. By applying a threshold proportional to the maximum of the distance transform (regulated by the `dist_thresh_multiplier` parameter), the region *seeds* are obtained, that is, the central cores of each room, well separated from one another at the narrowings.

4. The seeds are labeled as distinct connected components, providing the initial markers. The region of uncertainty — the band between the seeds and the boundaries of the free space — is marked as unknown.

5. The watershed algorithm propagates the marker labels through the unknown region, progressively "flooding" the free space starting from the seeds until the basins collide at the narrow passages. The boundary lines thus generated (labeled with the value -1) materialize the separations between rooms.

The result is a segmented map in which each free cell is associated with the label of the room it belongs to. This segmentation is also republished as a colored `OccupancyGrid` to allow its visualization and debugging in RViz.



(a) Clustered map

(b) Firt floor of 00800-TEEsavR23oF environment [27]

Figure 4.11: Watershed example

Assigning objects to clusters. Once the map has been segmented, each detected object must be associated with the corresponding room. Rather than projecting the object's coordinates onto the segmented map a posteriori, the choice was made to assign each new object the room in which the robot is located at the instant of its discovery: at the moment of detection, the `TF` transform between the `map`

frame and the `base_link` frame is queried, and the watershed label at the robot's pose becomes the object's cluster. This assignment is computed only once and kept stable for the entire lifetime of the object, while the other attributes (refined pose, similarity) continue to be updated by the mapper. For each cluster, moreover, a centroid is computed, obtained as a collision-free point inside the room, which acts as the spatial reference of the region. The map thus enriched, in which each object is accompanied by its class, coordinates, similarity, cluster identifier, and room centroid, is serialized and published for the behavior tree.

Environment Exploration

When the object of interest is not in view — whether because it does not correspond to any object in the map, in the `SPECIFIC_OBJECT_WITH_FEATURES` case, or because it has not yet been detected — exploration of the environment is required.

As explained in the previous section, the LLM provides the room in which the object is likely to be found. Thanks to this information, the agent is able to restrict the exploration to a specific area that is semantically correlated with the goal to be found. To further direct the exploration towards semantically relevant areas, the search within the zone begins from an "anchor" object that is particularly associated with the goal.

If the object of interest is not discovered in the vicinity of the "anchor" object, the agent begins to navigate the room using a series of randomly generated waypoints. Periodically, during the exploration, the behavior tree checks whether the object of interest has been identified; should it be recognized, the exploration is interrupted in favour of navigation towards the object.

Chapter 5

Experimental Results

In order to test the pipeline's functioning as completely and exhaustively as possible, several tests were carried out both in a simulated environment and in a real-world environment. Chapter 3 described the method used to run the simulation. This section will now present the setup used for the real-world tests.

5.0.1 Hardware Experimental Setup

The real-world tests were carried out on the Jackal rover, equipped with a Jetson AGX Thor (128 GB), on which the majority of the computation is performed. A depth camera was used for image acquisition.

Jackal

The platform used for field testing is the Jackal, an unmanned ground vehicle (UGV) manufactured by Clearpath Robotics [36]. It is a four-wheeled, skid-steer robot with a high-torque 4x4 drivetrain, designed primarily as a research and development platform. Given its compact dimensions (508 x 430 x 250 mm), the Jackal represents an optimal solution for this experimentation, particularly considering the limited size of the testing area. Moreover it is built around ROS integration, making it straightforward to use.

In its standard configuration, the Jackal mounts a depth camera directly on its chassis; however, for the intended use case, this low mounting position significantly limits the camera's field of view, causing it to miss most objects relevant to semantic navigation. For this reason, an additional support structure has been mounted on top of the Jackal, on which the depth camera has been repositioned 500 mm higher. This elevated position provides the camera with a field of view that captures objects at heights typical of domestic environments, which is essential for semantic

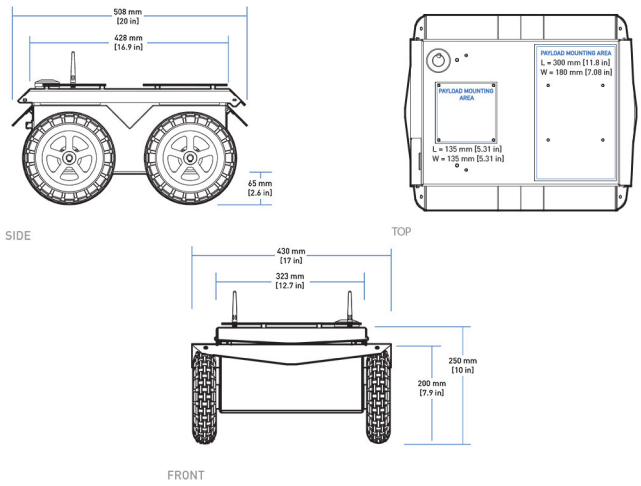


Figure 5.1: Jackal Dimensions [36]

navigation tasks. On the same structure, at a height of 340 mm, a lidar has been mounted, used to perform localization and obstacle avoidance.

Camera

The camera used is a stereo depth camera made by Intel. The camera is the RealSense D435i [37].

The stereo vision system works with two imagers paired with an infrared projector, which together compute depth by triangulating disparities between the two views (similar to human binocular vision). The camera also includes a standard RGB camera that outputs a color image at 1920x1080 and 30 fps. Alongside the RGB image the camera also provides an aligned and synchronized depth map that is used to compute the location of detected objects.

Figure 5.2: Intel Realsense D435i camera [37]

The camera has a fairly wide depth field of view (roughly 87° horizontal × 58° vertical), works across a range of roughly 0.3 m to about 3 m effectively (though

usable range varies with lighting and surface), and outputs depth at up to 1280×720 resolution. It connects via USB 3.1 and is natively supported in ROS, which fits well with the Jackal's ROS-based software stack.

Jetson AGX Thor

The Jackal rover is equipped with an internal computer, however its computational power is limited and not suited for heavy and complex AI models, like the ones implemented in the pipeline. To this end the Jackal has been integrated with an external module on which the majority of the computation is outsourced. This module is an NVIDIA Jetson AGX Thor that is the flagship edge AI and robotics computing platform [38].

Figure 5.3: Jetson Thor [38]

It is specifically designed to be the onboard "brain" of robots and other physical AI systems that need to run heavy AI workloads locally. The Jetson Thor is a System-on-Module (SoM) that combines Blackwell GPU architecture with 14 Arm Neoverse V3AE CPU cores and 128 GB of memory. In terms of raw throughput, it offers up to 2070 FP4 TFLOPS, with 40 to 130 W of configurable power draw, and up to 1035 FP8 TFLOPS. Compared to its predecessor, it delivers 7.5 times the AI performance and 3.5 times the energy efficiency of the AGX Orin.

These advanced computational abilities come at a significant power cost. The module manages power draw and performance through a system called `nvpmodel`, used across the Jetson family. Through `nvpmodel` it is possible to regulate parameters such as number of cores and clock frequency for the CPU, GPU, PVA (Programmable Vision Accelerator), and SoC engines such as VI (Video Input) and VIC (Video Image Compositor). By setting these parameters it is possible to regulate the power consumption of the device.

By default the Jetson AGX Thor supports two predefined modes: `MAXN` and 120W. `MAXN` is an unconstrained power mode that allows maximum number of cores and clock frequency, while 120W mode is the capped configuration that locks

frequencies, core counts and other parameters to a level that NVIDIA has validated as a thermally sustainable operating point.

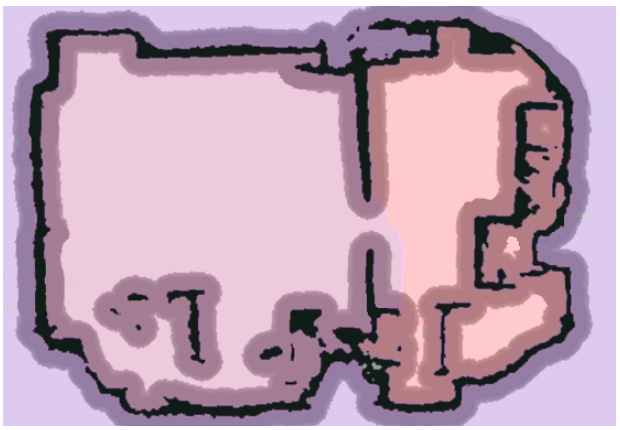
However, both Jackal and Jetson AGX Thor are powered by LiPo 6S batteries, thus, it is necessary to reduce power consumption in order to curb battery drain over long sessions. This has been done by setting a custom power profile that caps the power consumption to 90 Watts. This power setting has been selected by balancing power consumption and inference time of the pipeline.

5.0.2 Results and Discussion

In order to evaluate the pipeline several tests have been performed both in real world and in simulation. For several reason, however, the bulk of the experiments have been done in simulated environment.

Real-World Testing

The pipeline has been tested under real-world conditions using the hardware setup previously described. Due to the specific use case of the pipeline, the choice of test field constituted a critical aspect. The solution was tested in a laboratory room that also includes general household appliances (such as a fridge and a microwave) and has been divided in two by an artificial wall. This multi-room layout is necessary to evaluate room disambiguation and cross-room navigation. The room layout is reported below, with the room segmentation provided by the watershed algorithm.



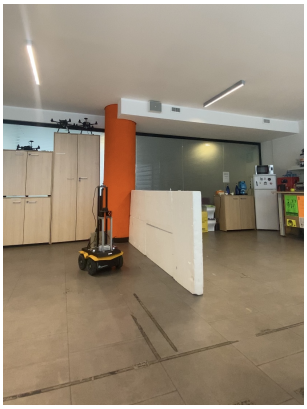


Figure 5.4: Laboratory setup

However, this setup does not constitute an optimal testing site, due to its limited complexity in terms of both layout and the variety of objects it contains; it is therefore not possible to accurately validate the full range of possible agent behaviors. Despite these limitations, the agent successfully reached the correct

target object in 5 out of 7 trials, confirming that the pipeline operates correctly end-to-end in a real-world setting. For this reason, field testing serves primarily as a proof of feasibility, demonstrating that the hardware is capable of handling computation at an online rate and that the models and algorithms are resilient to real-world noise, mostly arising from lighting conditions and camera depth artifacts.

The parameters evaluated in these tests are mostly related to timing statistics, whereas simulation testing focuses more on evaluating the agent's behavior. Timing must be computed separately for the LLM, the perception module, and the mapper module.

Since the LLM acts as an orchestrator and must run before navigation begins, its computational latency does not represent a critical parameter, but only an overhead that introduces a delay between the reception of the instruction and the start of navigation. Inference timing was computed across 30 consecutive calls, with timing statistics calculated for each LLM call and then averaged. Results are shown in the table below:

Step	*Avg (s)*
Assign Cluster Labels	2.633
Extract Goal	0.965
Cluster Prediction	**3.451**
Extract Action	0.764
Decide Logic	0.349
TOTAL callback	**8.162**

Table 5.1: Timing statistics for the LLM script

As the statistics show, the critical step is the cluster assignment node. This is due to the fact that the model receives a higher number of tokens as input and produces a higher number of tokens as output. However, the resulting time has been considered an acceptable overhead.

On the other hand, the timing statistics for the mapper and perception modules are more critical for the pipeline, as both must guarantee online performance. The main bottleneck is represented by the inference time of the AI models. Timing statistics for the perception module are shown in the table below.

During testing a notable variation in YOLO's inference time has been observed depending on the camera's publishing rate. This is due to the presence of the tracker, which requires more computation time at lower rates.

Another critical aspect concerns the mapper's timing statistics. Since heavy computation is also performed within the mapper node, its timing is likewise reported in the table below.

Step	*Minimum (ms)*	*Average (ms)*	*Maximum (ms)*
Yolo Inference	28.44	**33.57**	68.63
Detections Processing	1.92	2.17	2.86
SigLIP encoding	61.76	**65.12**	69.57
Publishing	2.23	3.14	4.89
Total Frame	34.81	**41.65**	78.22

Table 5.2: Timing statistics for the perception module

Step	*Avg/Frame (ms)*	*Total (%)*
Lock Wait (Mutex)	0.001	0.004
Depth Conversion	**5.521**	40.857
Point Extraction	1.900	14.063
Filtering (Voxel+Cluster)	1.314	9.723
Transform to Map	0.003	0.024
Batch Addition (Matching)	**3.001**	22.206
Refine Geometry	0.077	0.570
Resolve Duplicates	0.023	0.169
Cleanup Wrong Detections	0.004	0.032
Publish Semantic Map Msg	0.198	1.468
Publish Stable Pointcloud	0.156	1.152
Post-Batch Other	0.026	0.192
Unaccounted (other work)	1.289	9.538
TOTAL callback	**13.514**	100.000

Table 5.3: Timing statistics for the mapper module

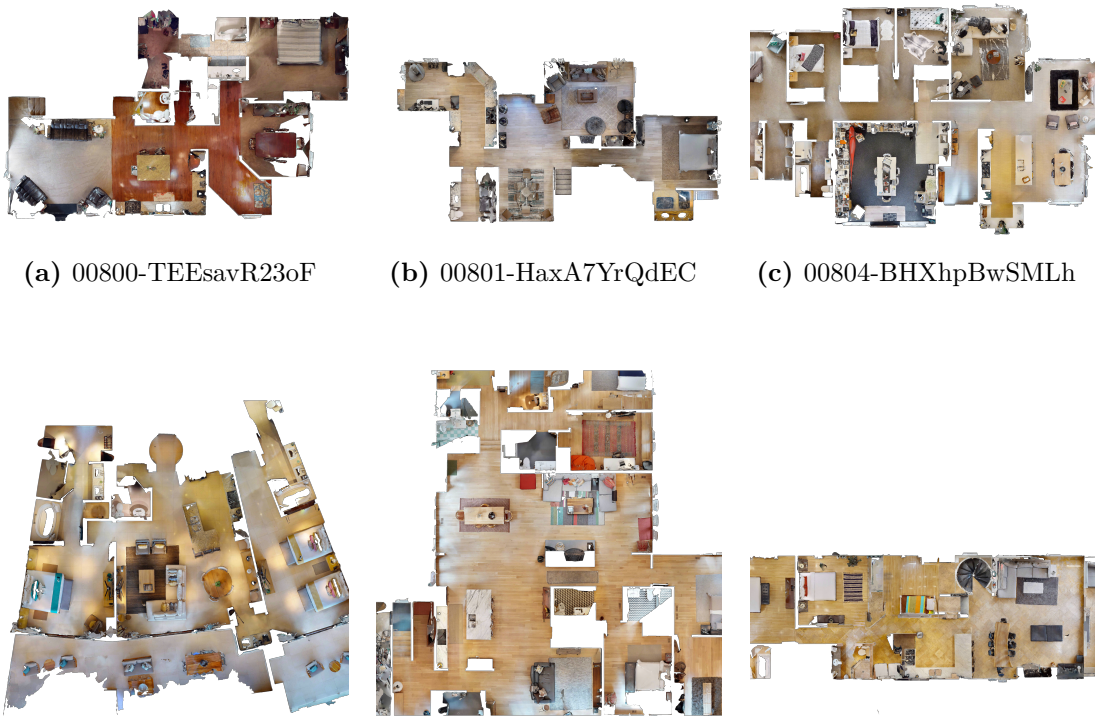
As mentioned previously, on-edge feasibility was tested in a limited environment, which nonetheless allowed the correct functioning of the system to be demonstrated across several tests. The Jetson Thor's computational capabilities prove more than sufficient both in terms of GPU computation (for all the AI models involved) and CPU computation (handled by the mapper).

Simulation Results

As indicated in Chapter 2, the simulation environment chosen is Matterport3D. Within the Matterport dataset, six different environments were selected:

- 00800-TEEsavR23oF

- 00801-HaxA7YrQdEC
- 00804-BHXhpBwSMLh
- 00805-SUHsP6z2gcJ
- 00807-rsggHU7g7dh
- 00808-y9hTuugGdiq



(a) 00800-TEEsavR23oF **(b)** 00801-HaxA7YrQdEC **(c)** 00804-BHXhpBwSMLh

(d) 00805-SUHsP6z2gcJ **(e)** 00807-rsggHU7g7dh **(f)** 00808-y9hTuugGdiq

Figure 5.5: Top down views of the used environment [27]

For each environment, 10 navigation tests were carried out, of which 4 required exploration. The limited number of tests per environment is also due to the limited size of the dataset and to image quality. Indeed, the number of objects that can be detected is limited, which prevents the formation of an object-rich map. The parameter measured is the navigation success rate, defined as the percentage of trials in which the robot reaches the object of interest, positioning itself in its vicinity with the object within frame. For each failure, the cause was assessed and categorized as one of the following:

- Object unreachable, n=1 (navigation failure; the simulated environment does not implement physics, and the object of interest was located in a narrow gap between a table and the wall that Nav2 determined to be unreachable)
- Detection failure, n=8 (the object is not recognized by the vision node; in these cases the object was difficult to recognize or partially occluded)
- Wrong object, n=3 (the object reached is not the object of interest, due to a missed detection)
- Object not matched, n=2 (the object is identified but not associated with the goal; SigLIP2 computed a similarity score below the threshold, an error strongly influenced by lighting conditions, which can obscure features)

Across the 60 tests carried out, there were 14 failures in total, distributed across these four categories as shown above. It is clear that the main source of error is the detection model, which proves too weak and misses several detections. The low number of unmatched objects demonstrates that SigLIP2 is sufficiently powerful to handle the full scope of the visual matching task. This limitation is closely related to the available hardware power, since, as indicated previously, class-agnostic segmentation models would represent an excessive bottleneck and would make edge deployment infeasible.

Chapter 6

Conclusions and Future Works

The presented semantic navigation pipeline offers a novel approach on the Object Navigation task performed on edge hardware. Through the Vision–Language model it is possible to build a lightweight persistent visual memory that exists in the embeddings space. Coupling the visual memory with a Large Language Model allows the introduction of semantic priors into the robot behavior without blocking the pipeline at every iteration, maintaining only lighter models in run time and using the heavy LLM only as an orchestrator.

With extensive experiments conducted across multiple different environments of the Matterport 3D dataset, it has been demonstrated that the presented pipeline has a significant success rate of 76.67%. Field testing, although limited to a smaller number of trials due to the constraints of the available testing environment, further supports this result, with the pipeline successfully reaching the correct target object in 5 out of 7 real-world trials, while maintaining a running rate between 15 and 20 Hz suitable for online edge computation. Nevertheless, the presented work is not without limitations. As emerged from the simulation experiments, the detection model suffers lack of generality and it has a poorer semantic understanding of the environment compared to the Vision-Language model. Moreover, the mapper has been developed considering a static environment, thus, it cannot handle changes in the setup.

6.1 Future Works

These limitations make future research indispensable. Future work should focus on the two major issues presented. Regarding the detection model's limited generality, as stated in the perception section of chapter 4, careful optimization of an agnostic

segmentation model is required. To overcome this problem the proposed pipeline uses an open vocabulary semantic segmentation model, which however has limited semantic capabilities, especially compared to the vision-language model. The introduction of an agnostic segmentation model would require careful filtering and fine-tuning, in order to avoid over-segmentation and background segmentation, as well as further optimization to keep its computational cost compatible with the edge hardware constraints discussed in Chapter 4.

Regarding the mapper's static-environment assumption, further work on the semantic mapper module is also necessary. The current version of the mapper has been developed with the limiting assumption of a static environment, which makes the pipeline sensitive to changes in the setup. It is therefore necessary to implement logic capable of removing stale entries for objects no longer present in the scene, as well as, as a further and more challenging extension, tracking and updating the position of objects that move within the environment.

Overall, despite these open limitations, the results obtained across both simulated and real-world experiments support the central claim of this work: LLM-orchestrated semantic navigation constitutes a viable and computationally efficient approach for object navigation on edge-deployed robotic platforms, and the limitations identified here represent natural directions for extending and strengthening this approach rather than obstacles to its feasibility.

Bibliography

[1] Raghad Alqobali, Maha Alshamrani, Reem Alnasser, Asrar Rashidi, Tareq Alhmiedat, and Osama Alia. «A Survey on Robot Semantic Navigation Systems for Indoor Environments». In: *Applied Sciences* 14 (Dec. 2023), p. 89. DOI: 10.3390/app14010089 (cit. on pp. 1, 5, 6).

[2] Yong Zhu, Zhenyu Wen, Xiong Li, Xiufang Shi, Xiang Wu, Hui Dong, and Jiming Chen. «ChatNav: Leveraging LLM to Zero-Shot Semantic Reasoning in Object Navigation». In: *IEEE Transactions on Circuits and Systems for Video Technology* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/TCSVT.2024.3485907 (cit. on pp. 2, 11–14).

[3] Santhosh Ramakrishnan, Devendra Chaplot, Ziad Al-Halah, Jitendra Malik, and Kristen Grauman. «PONI: Potential Functions for ObjectGoal Navigation with Interaction-free Learning». In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2022, pp. 18868–18878. DOI: 10.1109/CVPR52688.2022.01832 (cit. on pp. 2, 5).

[4] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You Only Look Once: Unified, Real-Time Object Detection». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91 (cit. on p. 6).

[5] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Apr. 2017. DOI: 10.48550/arXiv.1704.04861 (cit. on p. 6).

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 6).

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going Deeper with Convolutions». In: *2015 IEEE Conference on*

Computer Vision and Pattern Recognition (CVPR). June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594 (cit. on p. 6).

[8] Theophile Gervet, Soumith Chintala, Dhruv Batra, Jitendra Malik, and Devendra Singh Chaplot. «Navigating to objects in the real world». In: *Science Robotics* 8.79 (2023), eadf6991. DOI: 10.1126/scirobotics.adf6991 (cit. on pp. 6, 7).

[9] Naoki Yokoyama, Ram Ramrakhya, Abhishek Das, Dhruv Batra, and Sehoon Ha. *HM3D-OVON: A Dataset and Benchmark for Open-Vocabulary Object Goal Navigation*. Sept. 2024. DOI: 10.48550/arXiv.2409.14296 (cit. on p. 7).

[10] URL: https://blogs.nvidia.com/blog/what-are-foundation-models/ (cit. on p. 8).

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423 (cit. on p. 8).

[12] Tom B. Brown et al. «Language Models are Few-Shot Learners». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33. 2020, pp. 1877–1901 (cit. on p. 8).

[13] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. «Zero-Shot Text-to-Image Generation». In: *Proceedings of the 38th International Conference on Machine Learning (ICML)*. Vol. 139. Proceedings of Machine Learning Research. July 2021, pp. 8821–8831 (cit. on p. 8).

[14] Rishi Bommasani et al. *On the Opportunities and Risks of Foundation Models*. Aug. 2021. DOI: 10.48550/arXiv.2108.07258 (cit. on p. 8).

[15] Roya Firoozi et al. «Foundation models in robotics: Applications, challenges, and the future». In: *The International Journal of Robotics Research* 44.5 (2025), pp. 701–739. DOI: 10.1177/02783649241281508 (cit. on pp. 8, 9, 12, 13).

[16] Alec Radford et al. «Learning Transferable Visual Models From Natural Language Supervision». In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 8748–8763 (cit. on pp. 8, 10, 19, 20, 42).

[17] Michael Ahn et al. *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances*. 2022. DOI: 10.48550/arXiv.2204.01691 (cit. on p. 9).

[18] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. «Sigmoid Loss for Language Image Pre-Training». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2023, pp. 11975–11986 (cit. on pp. 10, 28, 29, 42).

[19] Sangmim Song, Sarath Kodagoda, Amal Gunatilake, Marc Carmichael, Karthick Thiyagarajan, and Jodi Martin. *Guide-LLM: An Embodied LLM Agent and Text-Based Topological Map for Robotic Guidance of People with Visual Impairments*. Oct. 2024. DOI: 10.48550/arXiv.2410.20666 (cit. on pp. 10, 11, 16, 17).

[20] Fujing Xie, Sören Schwertfeger, and Hermann Blum. «osmAG-LLM: Zero-Shot Open-Vocabulary Object Navigation via Semantic Maps and Large Language Models Reasoning». In: *IEEE Robotics and Automation Letters* 11 (2025), pp. 2426–2433 (cit. on pp. 11, 12, 15, 16).

[21] Lingfeng Zhang et al. *MapNav: A Novel Memory Representation via Annotated Semantic Maps for VLM-based Vision-and-Language Navigation*. June 2025. DOI: 10.22541/au.175131560.00300188/v1 (cit. on pp. 14, 15).

[22] Abdelrhman Werby, Chenguang Huang, Martin Büchner, Abhinav Valada, and Wolfram Burgard. «Hierarchical Open-Vocabulary 3D Scene Graphs for Language-Grounded Robot Navigation». In: *Proceedings of Robotics: Science and Systems*. Delft, Netherlands, July 2024. DOI: 10.15607/RSS.2024.XX.077 (cit. on pp. 18, 22, 23, 40).

[23] Qiao Gu et al. «ConceptGraphs: Open-vocabulary 3D Scene Graphs for Perception and Planning». In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 5021–5028 (cit. on pp. 19, 21, 22, 40).

[24] Sebastián Barbas Laina, Simon Boche, Sotiris Papatheodorou, Simon Schaefer, Jaehyung Jung, and Stefan Leutenegger. *FindAnything: Open-Vocabulary and Object-Centric Mapping for Robot Exploration in Any Environment*. 2025. DOI: 10.48550/arXiv.2504.08603 (cit. on pp. 19, 24–26, 40).

[25] Tomas Martins, Martin Oswald, and Javier Civera. «Open-Vocabulary Online Semantic Mapping for SLAM». In: *IEEE Robotics and Automation Letters* PP (Jan. 2025), pp. 1–8. DOI: 10.1109/LRA.2025.3617736 (cit. on pp. 24, 25, 40, 46).

[26] Manolis Savva et al. «Habitat: A Platform for Embodied AI Research». In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019, pp. 9338–9346. DOI: 10.1109/ICCV.2019.00943 (cit. on pp. 31, 36).

[27] Santhosh Ramakrishnan et al. *Habitat-Matterport 3D Dataset (HM3D): 1000 Large-scale 3D Environments for Embodied AI*. Sept. 2021. DOI: 10.48550/arXiv.2109.08238 (cit. on pp. 34, 36, 70, 79).

[28] Alexander Kirillov et al. «Segment Anything». In: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023, pp. 3992–4003. DOI: 10.1109/ICCV51070.2023.00371 (cit. on p. 40).

[29] URL: https://huggingface.co/docs/transformers/model_doc/sam (cit. on p. 41).

[30] Ao Wang, Lihao Liu, Hui Chen, Zijia Lin, Jungong Han, and Guiguang Ding. *YOLOE: Real-Time Seeing Anything*. 2025. DOI: 10.48550/arXiv.2503.07465 (cit. on pp. 41, 42).

[31] Luis Ortiz, Luiz Gonçalves, and Elizabeth Cabrera. *A Generic Approach for Error Estimation of Depth Data from (Stereo and RGB-D) 3D Sensors*. May 2017. DOI: 10.20944/preprints201705.0170.v1 (cit. on p. 50).

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is All you Need». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (cit. on p. 60).

[33] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. July 2024. DOI: 10.48550/arXiv.2407.21783 (cit. on p. 61).

[34] Thomas Wolf et al. «Transformers: State-of-the-Art Natural Language Processing». In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Ed. by Qun Liu and David Schlangen. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6. URL: https://aclanthology.org/2020.emnlp-demos.6/ (cit. on p. 61).

[35] Michele Colledanchise and Petter Ogren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, July 2018. ISBN: 9781138593732. DOI: 10.1201/9780429489105 (cit. on p. 65).

[36] URL: https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/ (cit. on pp. 73, 74).

[37] URL: https://www.realsenseai.com/products/depth-camera-d435i/ (cit. on p. 74).

[38] URL: https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-thor/ (cit. on p. 75).