Semantic Object-Goal Navigation on a Quadruped Robot in Known Environments

Emad Razavi

Supervisors:
Dr. Angelo Bratta (IIT)
Dr. João Carlos Virgolino Soares (IIT)
Prof. Carmine Recchiuto (UniGe)
Dr. Claudio Semini(IIT)

In partial fulfillment of the requirements for the degree of

Master's degree in Robotics Engineering

December, 2025

Declaration of Originality

I, Emad Razavi, hereby declare that this thesis is my own work and all sources of information and ideas have been acknowledged appropriately. This work has not been submitted for any other degree or academic qualification. I understand that any act of plagiarism, reproduction, or use of the whole or any part of this thesis without proper acknowledgment may result in severe academic penalties.

Acknowledgements

I want to express my deepest gratitude to my supervisors, Dr. Angelo Bratta and Dr. João Carlos Virgolino Soares, who taught me how to do research. When I started this project, I was like a child who did not yet know how to walk; they patiently guided my first steps, stayed close when I was unsure, and slowly helped me find my own pace. Whatever I have managed to achieve in this work is built on their patience, clarity, and trust. Words cannot fully express how much I appreciate their guidance and support.

I am also sincerely grateful to Prof. Carmine Recchiuto for his continuous guidance, careful feedback, and for always finding the time to point me in the right direction, even when his schedule was already full. I would like to thank Dr. Claudio Semini for the opportunity to join the Dynamic Legged Systems lab and gain this valuable research experience. His decision opened a door that changed the course of my studies and my life.

I want to thank all my colleagues at DLS for making the lab a supportive and inspiring place to work. In particular, I am thankful to Gabriel Fisher Abati and Dr. Miguel Ivo Fernandes for their help in solving technical problems, for the time they spent debugging with me, and for the many practical suggestions that kept this project moving forward. I am also deeply grateful to Amir Rad, Giovanni Minelli, and Alberto Sanchez Delgado for their emotional support during difficult moments and for the conversations that made long days lighter.

I would like to thank my close friends, whose presence and support reminded me that I was not alone in this journey, even when I felt far from everything familiar.

Finally, I want to thank my family. To my mother and my father, who have carried the worry, silence, and uncertainty that comes with distance so that I could carry this work instead, and my brother, who has always stood behind me like a mountain; their love, prayers, and quiet strength have been with me in every late night and every moment of doubt.

To all the people I love, who have stayed close to me and endured the distance with me: this thesis belongs to you as much as it does to me.

Thanks, all.

To all Iranian migrants who left loved ones and home behind,
carrying the weight of distance and loss in the hope that their
Dreams and Freedom might one day come true.

Abstract

This thesis presents a practical pipeline for object-goal navigation on a quadruped robot in real indoor spaces. The workflow has two stages. In the first stage, an operator teleoperates the robot along a short trajectory while SLAM Toolbox builds a two-dimensional occupancy grid, and the semantic stack runs in parallel. An object detector processes RGB-D images, confirmed detections are fused with depth and projected into the map frame, and they are written onto the grid as a semantic layer of labelled object instances. At the end of this run, both the 2D map and the semantic database are saved. In the second stage, during operation, a user selects a target from the list of recorded objects (e.g., a chair or a person). A small command-line client looks up the chosen instance in the semantic database and sends this goal to the navigator.

Navigation uses a layered costmap: the static map from the mapping stage, an obstacle layer from a two-dimensional laser scanner, and inflation that keeps a safe margin around people and obstacles. A classical global planner sets the path, and a local controller tracks it within speed and acceleration limits. We focus on integration, not on new methods; standard tools provide mapping and localization. During navigation, we do not run SLAM; the robot only localizes on this precomputed grid map.

All components run on a compact onboard computer mounted on the quadruped robot, with no discrete graphics card. A small supervision layer handles lost targets, short-term occlusions, and simple recovery actions so behavior remains steady and explainable. Despite its limitations, the work represents the first attempt in our lab to enable autonomous navigation for the Boston Dynamics Spot robot using only onboard sensing and computation. By adding an object-based semantic layer on top of the occupancy grid, the robot can also navigate goal-orientedly towards mapped object classes, rather than only following manually specified pose commands.

We demonstrate the system in three real indoor environments: a church, the Dynamic Legged Systems laboratory, and a large test room at the Istituto Italiano di Tecnologia. In both the church and the lab, we show that the mapping pipeline can produce usable occupancy grids in wide, cluttered spaces. In the IIT test room, we build semantic maps with chairs and a person, and we run object-goal navigation experiments in which the robot is commanded to visit specific objects using only the saved map and object poses. The evaluation is mainly qualitative, supported by trajectory plots and analysis of failure cases such as depth misalignment and localization errors. Overall, the results indicate that a lightweight, modular stack can provide semantic object-goal navigation for a quadruped robot under tight compute and cost constraints, while also highlighting the main limitations and directions for future improvement.

This work is the outcome of an internship at the Dynamic Legged Systems laboratory of the Istituto Italiano di Tecnologia. It reflects the design, integration, and experimental evaluation carried out there. An extended abstract focusing on the mapping and semantic-layer phase has been published in the proceedings of the Italian Conference on Robotics and Intelligent Machines (I-RIM 3D 2025).

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem statement

The development of embodied agents that can follow simple human requests such as “go to that chair” or “find the printer” is a long-standing theme in robotics and Artificial Intelligence (Chaplot *et al.*, 2020). This capability is often formulated as Object–Goal Navigation (ObjectNav), where a robot must navigate an environment to reach an instance of a target object category (Batra *et al.*, 2020). ObjectNav and the related area of semantic navigation are now established research topics with public benchmarks, challenges, and realistic simulation environments. Within this setting, a central question is how rich the representation of the environment needs to be to support reliable navigation (Chaplot *et al.*, 2020).

Many high-performing semantic navigation methods rely on rich, computationally expensive world models. A typical representation is a 3D metric–semantic map, which is a volumetric or mesh-based reconstruction that stores both geometrical data and semantic labels (e.g., object classes) (Agha *et al.*, 2021; Rosinol *et al.*, 2020). Other systems organize the same information as a *scene graph*, where nodes represent objects, rooms, or agents, and edges encode relations such as “on top of”, “inside”, or “in front of”. Frameworks such as Kimera are representative: they run real-time metric–semantic SLAM and output dense, labelled 3D reconstructions of the environment (Rosinol *et al.*, 2020). Maintaining these structures online requires frequent fusion of new depth and image data, significant memory, and often GPU support. More recent zero-shot navigation approaches add another layer by using large pre-trained vision–language models to interpret scenes and score candidate goals (Majumdar *et al.*, 2022). Systems like VLFM (Yokoyama *et al.*, 2023) keep dense maps and query large models for semantic reasoning, which typically demands powerful mobile GPUs to run in real time. This dependence on heavyweight perception and online dense mapping is a limitation for platforms with tight power, cost, or size constraints.

At the same time, there is growing interest in deploying these capabilities on quadruped robots in realistic indoor settings. Legged platforms are attractive for inspection and search-and-rescue tasks because they can traverse stairs, clutter, and uneven terrain where wheeled robots struggle. Recent work has demonstrated autonomous navigation on quadrupeds such as Mini Cheetah and Spot in real-world scenarios, using combinations of robust perceptive locomotion policies and map-based planning (Dudzik *et al.*, 2020; Koval *et al.*, 2022; Miki *et al.*, 2022; Raheema *et al.*, 2024). Open-vocabulary navigation on quadrupeds has also been explored, typically using embedded GPUs to host the semantic models (Busch *et al.*, 2024). These examples highlight a tension between rich semantic reasoning and the reality of resource-constrained, battery-powered legged robots.

This thesis focuses on that gap by studying ObjectNav for a quadruped robot in a known indoor environment under tight computational constraints. Instead of performing a dense 3D map online or running large vision–language models, the system assumes a pre-built 2D occupancy map and builds a lightweight semantic layer in real time from local RGB–D detections. The core idea is a modular pipeline: a static 2D map for localization and planning, plus a lightweight semantic layer that maintains persistent object hypotheses over time and exposes object–goal targets to the navigation stack. The goal is to retain the key behaviours of semantic navigation while running entirely on the robot's onboard computer, enabling practical object–goal navigation on a quadruped in real indoor environments.

1.2 Motivation

In recent years, legged robots have been deployed in real buildings for sustained, routine missions rather than remaining confined to lab demonstrations: teams have demonstrated long-range autonomous exploration in GPS-denied settings, robust perceptive locomotion in cluttered indoor spaces, and map-based navigation with Spot in real facilities. Figure 1.1 shows a Spot robot operating as part of one of these deployments, equipped with a field sensor payload in a real inspection scenario (Bouman *et al.*, 2020; Koval *et al.*, 2022; Miki *et al.*, 2022). These deployments share practical constraints such as limited energy budgets, tight timing and staffing windows, and strict requirements for predictable behavior around people and infrastructure, which make reliability, explainability, and low integration overhead as important as raw performance.

In parallel, many of the semantic navigation systems are built on assumptions that are hard to meet on a battery-powered quadruped: online dense 3D metric–semantic reconstruction or scene graphs that must be maintained in real time (Armeni *et al.*, 2019; Rosinol *et al.*, 2020), and, increasingly, zero-shot or open-vocabulary pipelines that lean on large vision–language models and modern detectors (Busch *et al.*, 2024; Majumdar *et al.*, 2022; Unlu *et al.*, 2024; Yokoyama *et al.*, 2023). While these choices deliver richer semantics and broader generalization, they typically drive up compute and power draw, enlarge the software stack, and complicate validation and maintenance, which are trade-offs that run counter to long-duration missions and conservative safety cases on mobile platforms.

A different operating point is both common and valid in labs and industrial sites: the environment is *known*. In these settings, a 2D occupancy map already exists (or can be created offline) and is sufficient for global localization and path planning; empirical studies with Spot in confined indoor environments confirm that a 2D map can support reliable autonomous operation without online dense 3D reconstruction (Koval *et al.*, 2022). What is missing, practically, is a lightweight semantic layer that turns per-frame RGB–D detections into persistent object instances that a standard navigator can target, so that object-centric tasks can be executed without inflating the perception stack.

This motivates a modular design that keeps the deterministic planners and localization on a known 2D map, adds a lightweight semantic layer only where it creates task value (object goals), and deliberately avoids heavy online 3-D processing or large VLM inference. The result aims to deliver a deployable stack that is easier to tune, and maintain over time, while still enabling the semantic behavior that makes indoor legged robots practically useful (Bouman *et al.*, 2020; Busch *et al.*, 2024; Koval *et al.*, 2022; Majumdar *et al.*, 2022; Miki *et al.*, 2022; Rosinol *et al.*, 2020; Unlu *et al.*, 2024; Yokoyama *et al.*, 2023).

1.3 Assumptions and Constraints

We work under a small set of practical assumptions. They shape the architecture, limit what we ask from the hardware, and define how we read the results of our experiments. We assume:

- Known, single-floor indoor environments (labs, offices, corridors);
- A 2D occupancy map for global localization and path planning;
- Static objects as targets (e.g, chairs or desks);
- Preference for lightweight and robust methods that run on limited hardware over more complex high-compute approaches.

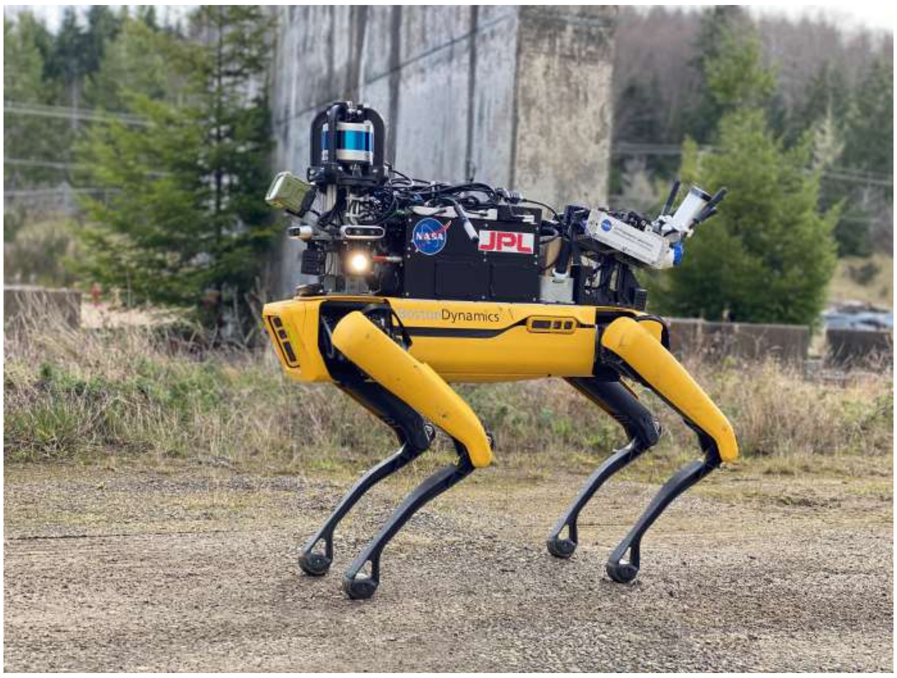

Figure 1.1: Boston Dynamics Spot integrated with the NeBula autonomy stack ("Au-Spot"): Spot base with the NeBula Sensor Package (NSP) and the NeBula Power and Computing Core (NPCC). The platform was deployed in the DARPA Subterranean Challenge and demonstrates building-scale autonomy on a legged robot (Agha *et al.*, 2021; Bouman *et al.*, 2020).

Environment and mapping. We assume known indoor spaces in which a 2D occupancy grid either already exists or can be built offline and is sufficient for global localization and planning. Field reports show reliable quadruped navigation in confined indoor environments using 2D maps with Cartographer and AMCL, without maintaining an online dense 3D reconstruction (Koval *et al.*, 2022). Recent work on low-cost legged platforms treats the 2D grid as the default navigation representation and sometimes augments AMCL with visual odometry to reduce drift in repetitive corridors (Aditya *et al.*, 2025). The navigation stack itself operates on a locally planar workspace. Elevation changes such as stairs are handled by the platform's locomotion controller and are out of scope here.

Representation and computation. Online 3D metric–semantic SLAM and scene graphs offer rich context but add substantial computational, memory, and software complexity compared to 2D navigation maps (Armeni *et al.*, 2019; Rosinol *et al.*, 2020). In practice, systems intended for field use often down-project to lower-dimensional maps (e.g., 2.5D height maps) because planning in full 3D is rarely practical for real-time operation on small platforms (Wellhausen & Hutter, 2023). Dense volumetric semantic pipelines can run at around 1 Hz on mobile hardware, which is already tight for closed-loop navigation (Grinvald *et al.*, 2019). For this work, we do not use any 3D representation at runtime and rely on the 2D map for deterministic planning and localization.

Semantic capability vs. acceleration. Zero-shot and open-vocabulary navigation and mapping provide more flexible semantic understanding, but typically rely on a GPU to run in real time. Recent systems report the use of RTX-class GPUs or embedded modules, such as the Jetson Orin AGX, for online open-vocabulary mapping and frontier selection (Busch *et al.*, 2024; Yokoyama *et al.*, 2023). Multi-floor zero-shot policies are likewise evaluated in multi-GPU execution environments (Zhang *et al.*, 2024). In this thesis, we avoid large vision–language inference at runtime; the stack is intended to run on the robot's onboard computer.

Quadruped deployment constraints. Legged deployments emphasise reliability, power awareness, and predictable behavior around people. Field reports document long-range autonomous exploration in GPS-denied sites, where sensing fusion compensates for noisy legged odometry (Bouman *et al.*, 2020), and broader autonomy stacks for subterranean missions stress offline-capable operation and robustness to communication dropouts (Agha *et al.*, 2021). Real inspection scenarios highlight hardware-independent operation without internet connectivity (Betta *et al.*, 2024) and raise human-factors concerns (social navigation and incidental encounters) that argue for navigation behavior that is easy to understand and to audit (Hauser *et al.*, 2023). These constraints favour modular stacks with limited complexity and stable failure modes.

Sensors and perception. The semantic layer relies on short-range RGB–D sensing within line of sight. Occlusions, motion blur, and low illumination degrade detections (Unlu *et al.*, 2022). Typical indoor camera setups operate within a few metres (with reliable depth usually below about 5 m), so standoff distances are kept conservative and transient observations are filtered across multiple views (Chen *et al.*, 2023). Dynamic agents, such as people, are not kept as persistent objects; the layer targets static objects and applies de-duplication and confirmation before exposing targets to the navigator.

1.4 Contributions

This work builds a navigation pipeline that takes an object–level command in a known indoor environment and drives a quadruped to an useful pose near the referenced object. In a first stage, an operator teleoperates the robot. At the same time, the SLAM Toolbox builds a two-dimensional occupancy grid, and the semantic nodes run in parallel, turning RGB–D detections into a small set of static object instances anchored in the map frame. At the end of this task, both the occupancy grid and the semantic database are saved. In a second stage, the robot is returned to the same environment, the saved map and semantic layer are loaded, and the operator selects a target object either from a command–line menu or via a short voice command. An object–goal client looks up the chosen instance in the semantic database, converts it into a navigation goal in the map frame, and sends this goal to Nav2 for planning and control.

The contribution of the thesis is a complete, deployable implementation of this pipeline on a real quadruped, with three tightly coupled pieces:

- A semantic layer aligned to a known 2D occupancy map, built online from RGB–D detections and designed to stay compact and stable over time.
- An object–goal generation method that turns a selected object instance into a `PoseStamped` goal in the map frame, positioned at a fixed offset near the object and oriented in a well-defined way relative to it.
- A ROS 2/Nav2 stack deployed on Boston Dynamics Spot, including a safety supervisor and practical launch/configuration, logging, and visualization tools tuned for repeated operation rather than one–off demos.

At runtime, the semantic mapper maintains a database of confirmed object instances in the map frame, each with a running confidence score. Per–frame RGB–D detections are projected into the map frame through TF and fused in a two–stage memory: new observations enter a proposal buffer, and only detections that are repeatedly supported within distance and angle gates are promoted to the static table. Near–duplicate instances are merged by spatial gating. A recorder node exports this static table to a YAML file, and a command–line client presents it as a menu of object instances. For each selected object, the client turns the stored instance pose into a navigation goal in the map frame and hands it to Nav2.

Planning and execution then rely on standard Nav2 layered–costmap navigation. The global planner computes a path on the static map, accounting for inflation, and the local controller tracks that path under speed and acceleration limits. Internal states (map, semantic markers, TF tree, global path, local trajectory, and goal pose) are visualised in RViz to support debugging and auditing.

Sensors and data flow are kept minimal on purpose. The robot is equipped with a tracking camera for odometry, a 2D LiDAR for range scans, and an RGB–D camera for object detections; the base accepts `/cmd_vel` commands. During the

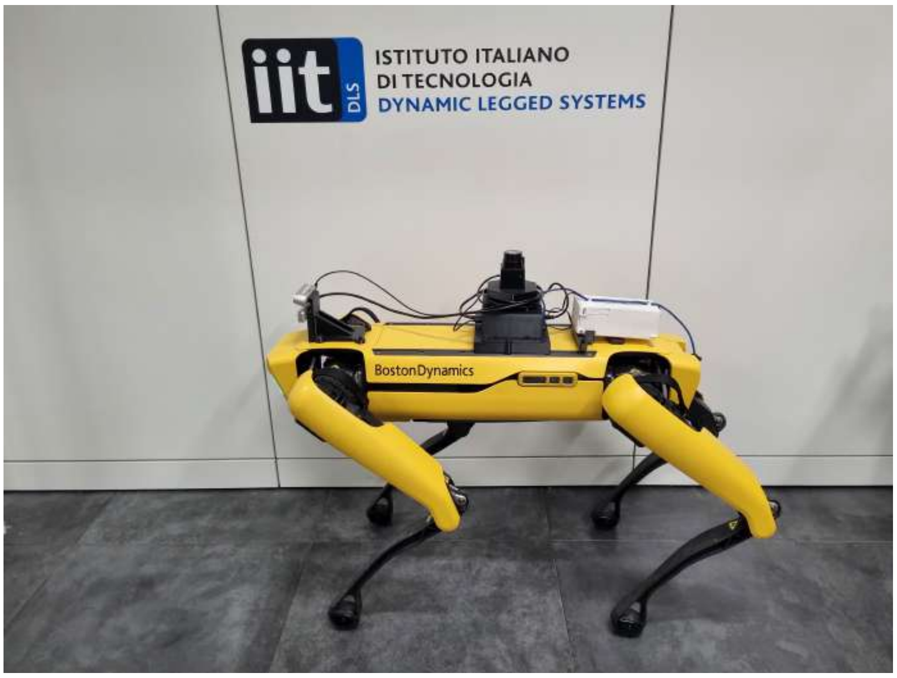

Figure 1.2: Our quadruped platform: Boston Dynamics Spot with the onboard computer and sensor payload used in this thesis.

teleoperated pass, SLAM builds the map while the semantic layer is populated. For navigation, the mapping nodes are stopped, and the system reloads the saved occupancy grid and semantic database. Nav2 uses this map, together with layered costmaps derived from the 2D LiDAR, to plan and execute paths to the goals produced by the object–goal client. A driver bridge forwards the velocity commands to the robot, and a small supervision node monitors progress and clearance, triggering stop, replan, or retreat actions when the robot gets stuck or new obstacles appear.

The next chapter surveys related work in quadruped object–goal navigation, semantic mapping, and ROS 2/Nav2–based navigation under similar constraints.

Chapter 2

State of the Art

2.1 Scope and Search Method

This review looks at recent work on legged robots that use semantic context to navigate to objects in indoor spaces. The emphasis is on real-robot studies, object-level goals, and methods that report enough detail to compare sensors, compute, and planning stacks. This section summarizes representative quadruped systems and highlights how they differ from our constraints (pre-run semantics, on-board compute, object-goal execution, conservative safety).

2.2 Quadruped Object–Goal Navigation

Object–Goal Navigation (ObjectNav) asks a robot to reach an instance of a requested object class. It is usually evaluated in Habitat on photorealistic indoor scans, with Success and SPL / Soft-SPL as the main metrics (Chen *et al.*, 2023; Yokoyama *et al.*, 2023). Two strands dominate current work: (i) modular, map-based approaches that build explicit representations and plan over them (for example, SemExp) (Chaplot *et al.*, 2020); and (ii) zero- or open-vocabulary variants that steer classical exploration with vision–language scoring or open-vocabulary features (Busch *et al.*, 2024; Qu *et al.*, 2024; Unlu *et al.*, 2024; Yokoyama *et al.*, 2023). A recurring issue is semantic reliability: false positives and unstable detections waste exploration budget, so filtering and confirmation layers have a clear impact on performance (Busch *et al.*, 2024). Many high-capacity systems also rely on accelerator-class hardware for real-time semantic inference. By contrast, known-map deployments on Spot show that a 2D occupancy grid is sufficient for robust global planning and localization when the environment is fixed, and that the main challenges then shift to execution reliability rather than online dense 3D mapping (Bouman *et al.*, 2020; Koval *et al.*, 2022).

For *quadruped* robots, the same ObjectNav problem comes with additional constraints that are less pronounced on wheeled bases. The stepping motion and body oscillations of legged platforms induce high-frequency perturbations on cameras and LiDAR, which degrade feature tracking and can corrupt SLAM estimates unless IMU, leg odometry, or stereo cues are tightly fused (Aditya *et al.*, 2025; Da Silva *et al.*, 2023; Jin *et al.*, 2024; Zhang *et al.*, 2024). On uneven or inclined terrain, quadrupeds may pitch enough that a 2D laser repeatedly detects the floor as an obstacle; recent systems therefore stabilize scans using inertial measurements or gate them based on pitch and roll (Da Silva *et al.*, 2023; Wellhausen & Hutter, 2023). Motion planning must respect stability and terrain constraints while remaining efficient, combining grid-based global paths with kinodynamic or reachability-based local controllers, and, in some cases, dynamic manoeuvres such as constrained jumping (Gilroy *et al.*, 2021; Liu & Yuan, 2024; Wellhausen & Hutter, 2023). At the same time, quadrupeds carry a limited payload and computing power while already running intensive state estimation and locomotion control. Inspection and ObjectNav systems, therefore, often keep safety-critical planning and control onboard and offload heavyweight open-vocabulary semantics or commonsense reasoning to Jetson-class modules or remote hosts (Bouman *et al.*, 2020; Ginting *et al.*, 2024; Jiang *et al.*, 2025; Xu *et al.*, 2024; Zhou *et al.*, 2025). These quadruped-specific constraints make lightweight, map-based ObjectNav with reliable but compact semantics an attractive operating point for field deployments, especially in the indoor environments considered in this thesis.

On-robot ObjectNav and adjacent systems. VLFM demonstrates zero-shot semantic navigation on Spot in office-like spaces: depth builds an occupancy grid while a pre-trained vision–language model scores frontiers; semantics are computed online and require a laptop-class GPU (Yokoyama *et al.*, 2023). OneMap constructs an open-vocabulary feature map from RGB-D to support single- and multi-object search on a quadruped, running in real time on a Jetson Orin; the map fuses CLIP-aligned features with uncertainty and guides target sequencing (Busch *et al.*, 2024). SEEK targets inspection on Spot, combining prior structure (e.g., floor plans) with online reasoning in a dynamic scene graph; sensing combines cameras and 3D LiDAR with GPU inference (Ginting *et al.*, 2024). Doubly Right demonstrates zero-shot object-goal navigation on a Unitree B1 in an apartment, building an online 2D semantic navigation map and verifying detections with a vision–language model; core modules run on the robot CPU, with an auxiliary host for commonsense reasoning (Unlu *et al.*, 2024). IPPON extends informative path planning with open-vocabulary cues, filling a 3D object-probability map and using a language model for commonsense guidance; validations include real indoor scenes (Qu *et al.*, 2024). Au-Spot (NeBula on Spot) reports long-range explo-

Table 2.1: Representative ObjectNav systems and adjacent deployments. Assumptions relevant to a known-map, lightweight-semantics, limited-compute setting.

Method	**Map used**	**Online 3D**	**GPU at run**	**Semantics**	**Planner / Policy**	**Platform**	**Reported metric**
SemExp (Chaplot *et al.*, 2020)	2D semantic grid (built online)	No	Yes	Closed-set detector (Mask R-CNN)	Map-based explorer	Habitat (sim)	65.0% / 33.0 (Gibson)
VLFM (Yokoyama *et al.*, 2023)	2D occupancy + VLM-scored frontiers	No	Yes	Open-vocabulary VLM (BLIP-2)	Frontier (value map)	Spot (real)	52.5% / 30.4 (HM3D)
OneMap (Busch *et al.*, 2024)	Open-vocabulary feature map (RGB-D)	No	Jetson Orin	CLIP-aligned features + uncertainty	Greedy / multi-object sequencing	Quadruped (real)	Qualitative + ablations
SEEK (Ginting *et al.*, 2024)	Prior + dynamic scene graph (online)	Often (LiDAR)	Yes	Lightweight RSN + relational reasoning	Probabilistic global + local control	Spot (real)	Inspection task success
Doubly Right (Unlu *et al.*, 2024)	2D semantic nav map (online)	No	Mixed (aux host)	VLM-verified detections	Frontier + Fast Marching	Unitree B1 (real)	Qualitative trials
IPPON (Qu *et al.*, 2024)	3D voxel probability map	Yes	Yes	Open-set cues + LLM commonsense	Informative path planning	Real + sim	Challenge / real demos
Au-Spot (NeBula) (Bouman *et al.*, 2020)	2D/3D geometric maps (online)	Yes	Onboard PC	Artifact-centric detection	Mission / global / local stack	Spot (real)	DARPA SubT field results
Koval et al. (Koval *et al.*, 2022)	2D occupancy (known)	No	No	Geometric stack	Classical (A* / Nav2)	Spot (real)	System eval. (confined env.)
This thesis	2D occupancy (known) + thin semantic layer	No	No	RGB-D detections + confirmation (closed set)	Map-then-act (Nav2; standoff + facing)	Spot (real)	Real-world eval. (Success, SPL*, errors)

SPL* only where benchmark shortest-path ground truth applies.

ration and artefact search in DARPA SubT with LiDAR-centric mapping and a mission / global/local planning stack, with a geometric focus rather than explicit "go-to-class" semantics (Bouman *et al.*, 2020).

Our operating point differs on four axes: we assume a known 2D map, we target low computational cost, we use a small, closed set of object classes, and we favour conservative, explainable behaviour over raw coverage. The table below contrasts representative systems with these choices and with the constraints of this thesis.

Summary. Map-based ObjectNav remains competitive when semantics are reliable and closely integrated with the planner (Chaplot *et al.*, 2020). Zero-shot and open-vocabulary approaches broaden the range of possible queries but typically assume GPU-class inference and have to deal with additional semantic noise (Busch *et al.*, 2024; Unlu *et al.*, 2024; Yokoyama *et al.*, 2023). On quadruped robots, where sensing is less stable and computing is often tighter, these costs are even more pronounced. For fixed indoor spaces, prior fieldwork on Spot and other quadrupeds supports a simpler stack: a 2D map for planning and localization, a lightweight semantic overlay to confirm static objects, and conservative standoff-and-facing goal generation (Aditya *et al.*, 2025; Bouman *et al.*, 2020; Koval *et al.*, 2022; Wellhausen & Hutter, 2023). This is the operating point adopted and evaluated in this thesis.

2.3 Metric and Semantic Mapping

State-of-the-art metric mapping for indoor navigation still relies on 2D occupancy grids. They are compact, planner–friendly, and integrate cleanly with mature localization and planning stacks (AMCL, layered costmaps) (Hess *et al.*, 2016). On quadruped platforms in confined or structured interiors, prior deployments show that a prebuilt 2D map is sufficient for global localization and path planning without maintaining an online dense reconstruction (Aditya *et al.*, 2025; Koval *et al.*, 2022). For legged robots operating in the field, layered costmaps are the common interface to planners and controllers, with inflation and obstacle layers shaping safe motion (Unlu *et al.*, 2024; Wellhausen & Hutter, 2023).

Beyond pure 2D, two families dominate. The first uses height / 2.5D or ESDF / TSDF layers to expose traversability and clearance for local planning; these maps trade memory for local geometric fidelity and are widely used when foot clearance or step height matters (Oleynikova *et al.*, 2017; Wellhausen & Hutter, 2023). The second builds dense 3D or scene-graph representations to couple geometry with semantics. While these structures enable richer reasoning (objects, rooms, relations), they carry higher memory and compute costs and typically assume accelerator hardware; several recent open-vocabulary pipelines report sub-Hz to ∼1 Hz updates or GPU-bound components that complicate tight real-time control on mobile bases (Armeni *et al.*, 2019; Jiang *et al.*, 2025; Longo *et al.*, 2025).

Semantic mapping spans a spectrum from lightweight overlays to dense metric–semantic maps. Open-vocabulary mapping approaches fuse CLIP-aligned features or VLM scores into maps to support text queries and object search; strong results are reported on quadrupeds, often with embedded or laptop-class GPUs. A recurring issue across ObjectNav systems is semantic reliability, because false positives and drifting labels reduce true success; as a result, confirmation filters and de-duplication are now standard practice in high-performing stacks (Busch *et al.*, 2024). Recent work on resource-limited inspection introduces distilled, compact models to reduce runtime costs while preserving the class signal for navigation (Ginting *et al.*, 2024). In parallel, Betta et al. build a multi-layer semantic grid on top of a 2D LiDAR map for search-and-rescue: each layer stores per-cell confidence values for doors, people, cracks, and stairs, sharing the same 2D substrate as the obstacle map rather than lifting semantics into full 3D (Betta *et al.*, 2024). Figure 2.1 shows their door and person layers overlaid on the obstacle map at two test sites.

In summary, today's mapping choices reflect a clear trade space. 2D occupancy grids plus layered costmaps remain the default substrate for indoor navigation because they are stable, efficient, and well supported by localization and planning toolchains (Hess *et al.*, 2016; Koval *et al.*, 2022). Height / ESDF layers add local clearance structure when needed (Oleynikova *et al.*, 2017; Wellhausen & Hutter,

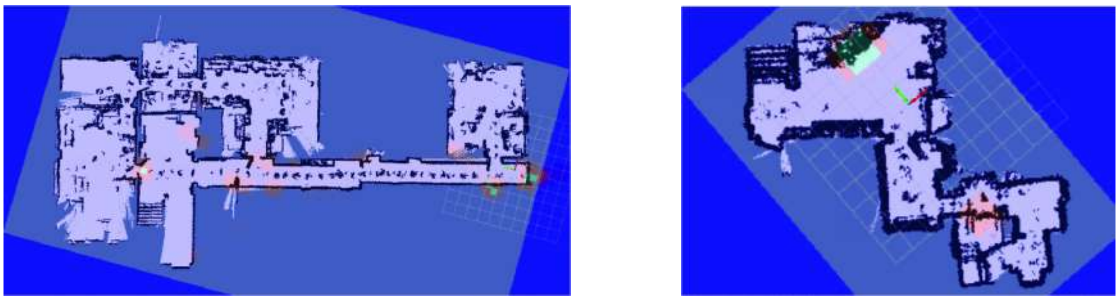

Figure 2.1: Example multi-layer semantic maps from Betta et al. (Betta *et al.*, 2024). Left: obstacle layer (black) with Door cells (yellow) at the first location. Right: obstacle layer (black) with Person cells (yellow) at the second location.

2023). Dense 3D and scene graphs offer richer semantics and global reasoning at the price of memory, latency, and accelerator assumptions (Armeni *et al.*, 2019; Jiang *et al.*, 2025; Longo *et al.*, 2025). Multi-layer semantic grids, such as those of Betta et al. keep semantics in raster form on top of 2D maps when full object reasoning is not required (Betta *et al.*, 2024).

2.4 Quadruped Navigation with ROS 2 / Nav2

For indoor legged platforms, the mainstream pattern is a ROS 2/Nav2 stack with a global planner on a 2D prior and a high-rate local controller over layered costmaps (Macenski *et al.*, 2020). In recent ObjectNav implementations, the planner is delivered as a Nav2 plugin, and the world model combines a *static* map with *obstacle* and *inflation* layers so trajectories respect geometry and clearances (Raheema *et al.*, 2024; Unlu *et al.*, 2024). The costmap serves a continuous occupancy/clearance field that global planners can optimize over (Raheema *et al.*, 2024). Spot deployments in confined interiors confirm that a prebuilt 2D map with AMCL suffices for global localization and planning, so no persistent online dense reconstruction is required. (Aditya *et al.*, 2025; Koval *et al.*, 2022).

At the global level, planning is usually straightforward graph search on the inflated grid, typically using A* or Dijkstra, trading a little path length for extra clearance (Raheema *et al.*, 2024). The local layer then turns that path into dynamically feasible commands within the robot's kinematic limits. In practice, legged stacks use simple waypoint followers (pure-pursuit/PID) in structured spaces and switch to kinodynamic MPC when fast disturbance rejection or risk-aware traversal is needed; in both cases, the local plan is refreshed at a high rate (tens of hertz) to cope with sudden perception changes and tight geometry (Agha *et al.*, 2021; Wellhausen & Hutter, 2023).

Robustness sits one level up, in a behavior tree or compact FSM that sequences actions and recoveries such as clearing costmaps, rotating in place to reacquire, backing up, and replanning. It rides through the communication hiccups typical of real facilities (Agha *et al.*, 2021). Safety for legged bases is encoded directly in parameters: conservative inflation to reflect footprint and human-space buffers, explicit limits on speed and acceleration, and asymmetric bounds for forward vs. reverse motion that controllers must obey (Aditya *et al.*, 2025; Koval *et al.*, 2022; Liu & Yuan, 2024).

Object–goal layers tend to plug into the same Nav2 backbone in two patterns (Macenski *et al.*, 2020). In simulator-centric work, exploration is steered by open-vocabulary scoring from vision–language models or CLIP-style feature maps; it works, but it assumes accelerator-class inference at run time (Busch *et al.*, 2024; Yokoyama *et al.*, 2023). On hardware, teams lean on the costmap and behavior-tree stack: some add a lightweight semantic bias on top of a known 2D map, others fold floor-plan priors and room cues into online reasoning to direct the search, all of it still funneled through the same planner, controller, and recoveries (Unlu *et al.*, 2024). Across papers, the constants are the same layered costmaps for planning and control, high-rate local replanning, BT/FSM supervision, and conservative speed and clearance limits encoded in the parameters.

2.5 Online 3D vs. 2D SLAM

Why is a pre-run 2D map enough here? Indoor navigation on legged robots is most stable when localization and planning run on a compact 2D substrate and the heavy lifting is done before deployment. Mature LiDAR-based systems like Cartographer deliver real-time mapping and loop closure at centimeter resolution (Hess *et al.*, 2016); the resulting occupancy grid works directly with AMCL and layered costmaps, which is exactly what prior Spot deployments in confined interiors rely on (Koval *et al.*, 2022). Recent quadruped studies echo the same pattern: 2D grids remain the de facto baseline for path planning and localization in buildings, with pose-graph tools (e.g., SLAM Toolbox) used for efficient back-end optimization and map management (Aditya *et al.*, 2025).

Full 3D pipelines add detail and semantics but at a clear computational cost. Metric-semantic stacks (e.g., Kimera) combine VIO, reconstruction, and factor graph inference to produce dense 3D maps or scene graphs (Rosinol *et al.*, 2020). These representations are powerful, yet they typically assume accelerator-class compute and accept lower update rates; recent open-vocabulary 3D mapping reports sub-Hz around the fusion loop, which is a poor fit for tightly coupled local control on mobile platforms (Jiang *et al.*, 2025). Field work on legged navigation highlights where richer geometry helps, for example, with uneven terrain, step heights, and clearance checks, which are often handled with 2.5D or ESDF layers

Table 2.2: Canonical SLAM toolchains for mobile robots.

Name	**Primary source**	**Typical sensors**	**Map type**	**Core method**	**Loop closure**	**Runtime profile**
Cartographer	(Hess *et al.*, 2016)	2D/3D LiDAR, IMU	2D occupancy	Branch&bound scan-to-submap; pose graph	Scan/submap constraints	CPU-only; high rate (≈ 5.3× RT reported)
RTAB-Map	(Labbé & Michaud, 2019)	RGB-D, 2D/3D LiDAR, IMU	2D grid; 3D cloud/mesh	Appearance-based LC + graph opt.	Visual/scan proximity	CPU (dense can use GPU); updates ≈ 1 Hz
SLAM Toolbox	(Macenski & Jambrecic, 2021)	2-D LiDAR, odom	2D occupancy	Pose-graph (g2o); lifelong mapping	Graph constraints	CPU-only; ROS 2 native; real-time
Kimera	(Rosinol *et al.*, 2020)	RGB-D/IMU (VIO)	3D mesh; DSG	Factor-graph VIO; meshing/semantics	Pose-graph	High; CPU/GPU; metric–semantic outputs
GMapping	(Grisetti *et al.*, 2005)	2D LiDAR, odom	2D occupancy	RBPF (adaptive proposals)	Implicit via particles	CPU-only; classic 2D baseline
ORB-SLAM2/3	(Campos *et al.*, 2021; Mur-Artal & Tardós, 2017)	Mono/stereo/RGB-D (+IMU)	Sparse 3D map	Feature-based BA, keyframes	DBoW2 place recog.	CPU (feature/BA heavy); high-rate VIO
LOAM	(Zhang & Singh, 2017)	3D LiDAR	3D cloud	Decoupled odom+mapping (edge/plane ICP)	Geometric LC	CPU-oriented; real-time
LIO-SAM	(Shan *et al.*, 2020)	3D LiDAR+IMU	3D cloud (factor graph)	Tightly-coupled smoothing & mapping	Scan-context	CPU for odom; back-end heavier; robust long halls
KinectFusion	(Newcombe *et al.*, 2011)	RGB-D	Dense 3D TSDF	Dense ICP + TSDF fusion	Local ICP	GPU; real-time dense indoor recon
LOCUS/LAMP (NeBula)	(Agha *et al.*, 2021)	3D LiDAR, IMU, VIO	3D geometric + 2D proj.	Factor-graph back-end (multi-modal)	LiDAR/visual LC	High-end CPU/GPU; centralized opt. when available

rather than maintaining a full global 3D reconstruction (Wellhausen & Hutter, 2023). For long corridors and other repetitive geometries, teams routinely fuse VO/VIO or wheel/IMU odometry to stabilize localization when scan matching drifts (Aditya *et al.*, 2025; Agha *et al.*, 2021).

Given those trade-offs, a pre-run LiDAR pass to build a static 2D map (Cartographer/SLAM Toolbox), followed by runtime AMCL on that prior, with optional VO/VIO fusion, is a pragmatic operating point for a quadruped operating in known offices and corridors. It keeps memory and latency predictable, leaves headroom for perception and safety checks, and aligns with the Nav2 planning stack used across recent systems (Aditya *et al.*, 2025; Koval *et al.*, 2022). Representative SLAM toolchains and their runtime profiles are summarized in Table 2.2.

Chapter 3

Methods

3.1 System Overview

This chapter describes how the system is put together on the real robot. We begin with the hardware, including the quadruped platform and its sensors. We then describe the ROS 2 architecture running on the onboard computer and finally explain how data and transforms flow through the system. The aim is to provide a clear picture of what runs where before delving into mapping, semantic mapping, and object–goal navigation.

3.1.1 Robot and Sensors

All experiments are carried out on a Boston Dynamics Spot quadruped equipped with a small onboard computer and a minimal sensor payload. The robot is controlled via velocity commands in the body frame (`/cmd_vel`); low-level locomotion, joint control, and state estimation within the robot are treated as black boxes. From the navigation stack point of view, Spot behaves as a mobile base with reliable odometry and a safety-rated stop.

An Intel NUC-class computer is mounted on the robot as the central compute unit. It runs Ubuntu and ROS 2 Humble and hosts all mapping, localization, semantic mapping, and navigation nodes. There is no discrete GPU and no external compute; all perception and planning run on the onboard computer. The NUC communicates with Spot over Ethernet, and all external sensors are wired directly to the NUC. This setup matches the constraints discussed in Chapter 1: everything runs locally on the robot, with no cloud or offboard compute, and bring-up is reproducible from a single machine.

Figure 3.1 shows the platform with the full payload used in this thesis (Razavi *et al.*, 2025).

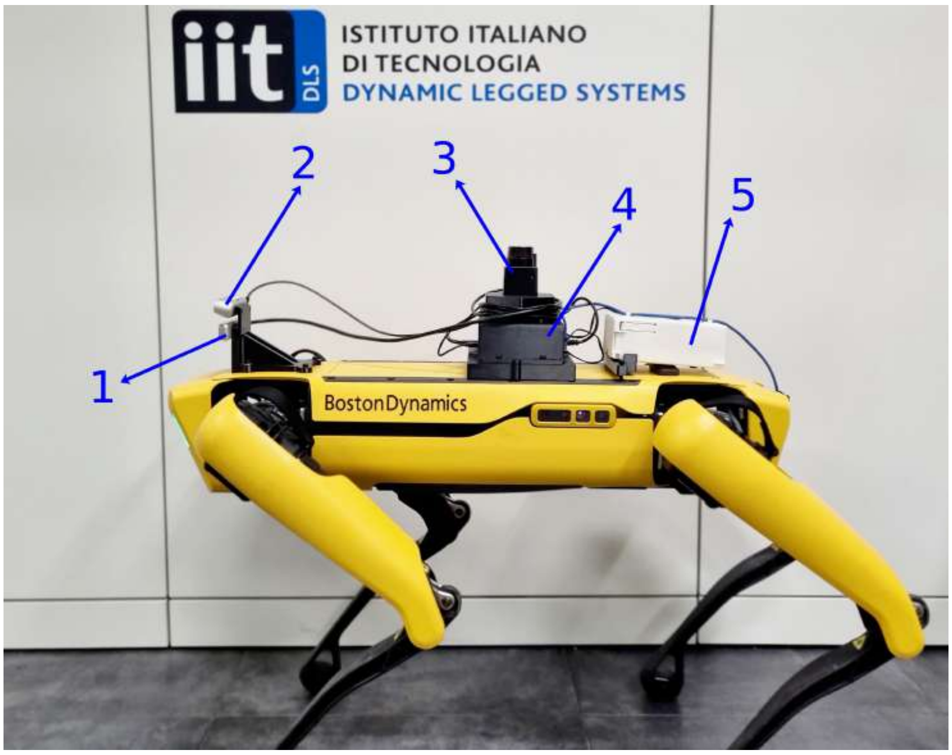

Figure 3.1: Spot by Boston Dynamics with the onboard payload used in this work: 1) Intel RealSense T265 tracking camera, 2) Intel RealSense D435 RGB–D camera, 3) 2D LiDAR, 4) Intel NUC 11 i7-1165G7 onboard computer, and 5) external lithium-ion battery powering the NUC.

The external sensor suite is intentionally small and focused on the needs of an object–goal navigation pipeline:

- **2D LiDAR.** A planar LiDAR is mounted on the robot with a clear view of the surroundings. Its scans are used together with odometry in a short pre-run to build a 2D occupancy grid with SLAM Toolbox. During navigation, the same sensor feeds the obstacle and inflation layers of the Nav2 costmaps and supports AMCL. The LiDAR therefore provides the system's metric backbone: it defines the static map and reflects newly appearing obstacles in the scene.

- **Camera pair (Intel RealSense D435 and T265).** The robot carries a pair of Intel RealSense cameras: a D435 RGB–D camera and a T265 tracking camera, mounted on a common bracket. This follows the tracking and depth configuration described in the Intel whitepaper on combined tracking and depth sensing (Schmidt *et al.*, 2019). The D435 provides aligned RGB and

Figure 3.2: Intel RealSense Depth Camera D435 and Tracking Camera T265 used together in this work (image adapted from (Schmidt *et al.*, 2019)). The RGB–D camera provides colour and depth for object detection, while the tracking camera provides visual–inertial pose estimates used to consistently place detections in the global map frame.

depth images; the RGB stream is passed to an object detector, and depth is used to back-project detections into 3D. The T265 provides visual–inertial odometry in a frame aligned with gravity. In our setup, the T265 pose and the calibrated extrinsics between the two cameras are integrated into the TF tree, so that each 3D detection from the D435 can be consistently expressed in the `map` frame without requiring dense 3D reconstruction or heavy registration.

Spot's internal sensors (IMU, joint encoders, body cameras) are not used directly in the semantic mapping pipeline, only for the manufacturer's state estimator and locomotion controller.

This combination of a 2D LiDAR and a compact camera pair (D435 + T265) is sufficient to support the proposed pipeline: the LiDAR and odometry provide the information needed for mapping, localization, and costmaps, while the RGB–D camera and detector supply the semantic observations used to build the semantic layer. The following subsections describe how these sensors are wired into the ROS 2 architecture and TF tree, and how their data flow into mapping, semantic mapping, and object–goal navigation.

3.1.2 ROS 2 Architecture

All software runs on the onboard Intel NUC under Ubuntu and ROS 2 Humble. Everything is on a single machine: the nodes that communicate to Spot, the mapping and localization stack, the semantic layer, and Nav2. This keeps the overall setup simple and aligns with the constraints from Chapter 1: all processing stays on the robot, with no cloud or offboard compute. Figure 3.3 gives an overview of the central nodes and how they are connected.

The interface to the robot is a ROS 2 driver that bridges the Spot API to standard topics and services. It publishes odometry and the base pose, exposes basic robot state (battery, faults, E-stop), and accepts velocity commands on `/cmd_vel`. From the rest of the system, and as already noted in Section 3.1.1, Spot is seen as a mobile base with a fixed `base_link` frame and a safety-rated stop; low-level locomotion and state estimation inside the robot stay opaque.

Mapping and navigation are split into two phases. In a short pre-run, a SLAM node based on SLAM Toolbox (Macenski & Jambrecic, 2021) subscribes to the 2D LiDAR and odometry topics from the T265 and builds a 2D occupancy grid of the environment. A map saver node writes this grid to disk. During navigation, SLAM is turned off. Instead, a `map_server` node loads the saved grid, and an AMCL node estimates the robot pose in the `map` frame using LiDAR scans and the same T265-based odometry. Together, `map_server` and AMCL provide the usual Nav2 interface: a static 2D map and a continuous transform between `map`, `odom`, and `base_link`.

Separate drivers handle the two RealSense cameras (D435 and T265). The D435 runs in its own container and publishes colour and aligned depth images, while the T265 driver runs together with the mapping stack and publishes a tracking pose (Schmidt *et al.*, 2019). An object detection node subscribes to the D435 RGB stream and outputs per-frame detections with class labels and scores. A small projection node then combines each detection with the aligned depth image to recover a 3D point (or small cluster) in the D435 frame. Using the TF tree and the calibrated extrinsics between the cameras, these points are transformed to `base_link` and then to `map`. This whole detection–projection pipeline runs in parallel with SLAM Toolbox during the teleoperated pass, so object hypotheses are placed directly in the same `map` frame that is being built. This is the only place where the D435 depth and T265 tracking information are fused; there is no dense 3D reconstruction in the loop.

On top of this, a custom `semantic_mapper` node subscribes to projected detections and TF. It maintains a compact, confirmed-only database of static object instances in the `map` frame. New observations first enter a proposal buffer, and only detections that are seen repeatedly within distance and angle gates are promoted to the static table; near-duplicate instances are merged. A separate `semantic_recorder` node writes the static objects to a YAML file, and a navigation client (`semantic_nav_cli`) reads this file and presents a list of object instances. The operator chooses which object to go to, and the client converts the stored instance pose into a goal pose in the `map` frame, which is sent to Nav2 as a `NavigateToPose` action.

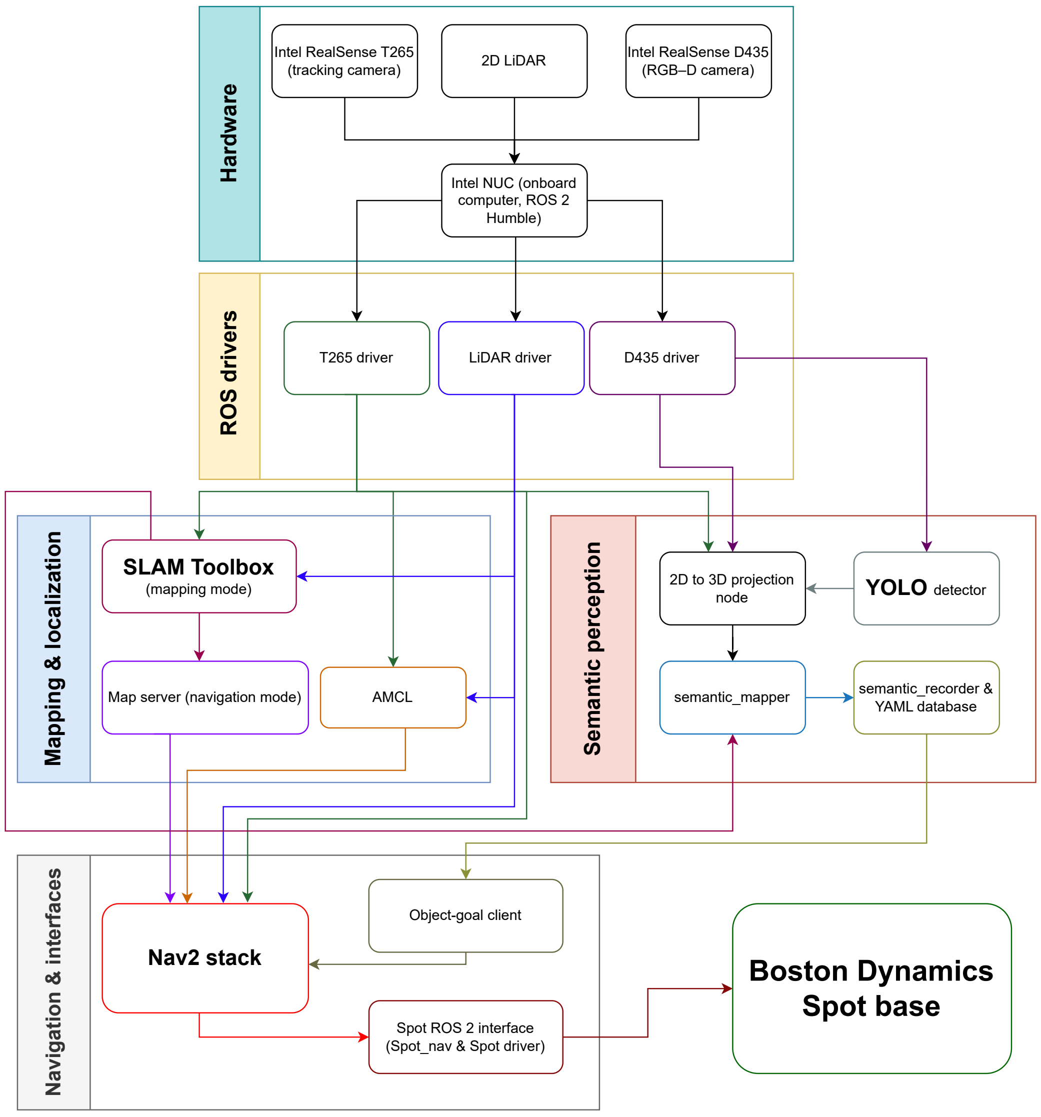
Hardware
Intel RealSense T265 (tracking camera)
2D LiDAR
Intel RealSense D435 (RGB–D camera)
Intel NUC (onboard computer, ROS 2 Humble)
ROS drivers
T265 driver
LiDAR driver
D435 driver
Mapping & localization
SLAM Toolbox (mapping mode)
Map server (navigation mode)
AMCL
Semantic perception
2D to 3D projection node
YOLO detector
semantic_mapper
semantic_recorder & YAML database
Navigation & interfaces
Nav2 stack
Object-goal client
Spot ROS 2 interface (Spot_nav & Spot driver)
Boston Dynamics Spot base

Figure 3.3: System overview of the onboard hardware and ROS 2 pipeline used in this work. The Intel RealSense T265, 2D LiDAR, and Intel RealSense D435 are connected to the Intel NUC, whose low-level drivers feed three main subsystems: (i) mapping and localization (SLAM Toolbox in mapping mode, map server and AMCL in navigation mode); (ii) semantic perception and mapping (YOLO detector, 2D–3D projection node, `semantic_mapper`, and `semantic_recorder` with its YAML database); and (iii) navigation and interfaces (Nav2 stack, object–goal client, and the Spot ROS 2 interface bridging to the Boston Dynamics Spot base).

Nav2 handles the navigation stack: the global planner, local controller, behavior-tree navigator, and costmap servers. These nodes use the occupancy grid from `map_server`, the AMCL pose estimate, and LiDAR scans to build layered costmaps and plan paths. The obstacle and inflation layers define a safety margin around mapped and observed obstacles, and Nav2's standard recoveries (spin, backup, wait) are used when the controller stalls or cannot progress.

In the last layer, additional safety is provided by the Spot driver itself. The driver exposes E-stop and fault handling, and the robot's internal controller enforces its own safety envelope (for example, stopping or modifying motion when it detects potential collisions). We do not add a separate safety node in ROS 2; instead, we rely on conservative costmaps and Nav2 recoveries at the planning level, and on Spot's built-in safety mechanisms at the locomotion level.

RViz is used for visualization and runs either on the NUC or on a tethered laptop. It shows the occupancy map, TF tree, costmaps, semantic markers, and the current Nav2 path and goal. During experiments, rosbag recording is used to log LiDAR scans, camera detections, semantic outputs, and Nav2 topics for offline analysis and debugging.

Overall, the architecture stays close to standard ROS 2 components (SLAM Toolbox, AMCL, Nav2, RealSense drivers, Spot driver) and wraps them with two custom pieces: the semantic mapping pipeline (including the semantic recorder) and the object-selection client used for object–goal navigation.

3.1.3 Dataflow and Modes of Operation

The system runs in two simple modes: a short *mapping* mode to build the 2D occupancy grid, and a *navigation* mode in which the robot uses that map, together with the semantic layer, to reach object goals. The hardware and TF tree are the same in both cases; what changes is which nodes publish commands and which mapping components are active.

The T265 driver (and a small helper node) provides the robot's pose, which we treat as the `odom` frame. A static transform relates the T265 tracking frame to `base_link`, so that all other sensors can be tied into the same tree. The D435 RGB–D camera is used only for detection and depth-based back-projection, not for visual odometry.

Mapping mode. In mapping mode, the operator drives the robot manually with a joystick. A standard teleop node publishes `/cmd_vel`, which the `spot_nav` package forwards to the Spot driver. The driver sends these velocity commands to the robot.

The T265 publishes its pose, which we expose as the transform `odom` $\rightarrow$ `base_link`. The 2D LiDAR publishes scans in a `lidar` frame that is rigidly attached to base_link. SLAM Toolbox subscribes to LiDAR scans and the T265-based odometry, estimates the robot pose in the map frame, and builds a 2D occupancy grid as the operator drives through the environment. When coverage is sufficient, a map saver node writes the grid to disk, and mapping mode ends.

The TF tree in this phase has a small backbone:

$$\texttt{map} \rightarrow \texttt{odom} \rightarrow \texttt{base\_link} \rightarrow \{\texttt{lidar}, \texttt{d435\_link}, \texttt{t265\_link}\},$$

with the last three transforms static and measured offline (or from the RealSense configuration).

Navigation mode. In navigation mode, SLAM is turned off, and the rest of the stack is brought up. A map_server node loads the saved occupancy grid. AMCL subscribes to LiDAR scans and uses the same T265-based odometry to re-estimate map → odom online. The Nav2 costmap servers use the occupancy grid and LiDAR scans to build global and local costmaps. The global planner and local controller run on top of these costmaps and output velocity commands on /cmd_vel.

The command path is:

$$\text{Nav2} \rightarrow \texttt{spot\_nav} \rightarrow \text{Spot driver} \rightarrow \text{robot}.$$

The spot_nav package acts as a thin bridge: it takes /cmd_vel from Nav2, adapts it to the Spot API, and forwards it to the driver. Spot's own odometry and higher-level behaviours remain unused; from the navigation stack's point of view, the robot is just a base that executes commanded velocities.

In parallel, the RealSense driver publishes RGB and depth images from the D435 and the tracking pose from the T265. As described in Section 3.1.2, an object detector subscribes to the D435 RGB stream, a projection node combines detections with depth to recover 3D points, and these points are expressed in map through the TF tree. The semantic_mapper subscribes to these projected detections, maintains its confirmed-only database of object instances, and publishes markers for visualisation.

When the operator wants to navigate to an object, they run the semantic_nav_cli client described in Section 3.4. The client loads the recorded semantic objects from the YAML file, presents a menu of instances (for example, the chairs in the environment), and, for the chosen entry, computes a goal pose in the map frame near the object. Nav2 then plans and executes the motion using the same T265-based odometry and LiDAR-based costmaps as in mapping mode.

In general, dataflow remains simple: the T265 provides odometry, the LiDAR supports mapping, localization, and costmaps, the D435 provides colour and depth for semantics, and everything is tied together through a small fixed TF tree centered on map, odom, and base_link. The next section focuses on the mapping mode in more detail and describes how the 2D occupancy grid used by the rest of the pipeline is built and managed.

3.2 Mapping and Localization

This section explains how the robot builds and then reuses a 2D map of the environment. The overall idea is simple: we first run a short mapping session to create a static occupancy grid of the lab and corridors, and later we use that same map for localization and planning during all navigation and semantic experiments. Mapping and localization are therefore separated in time. The mapping pass is done once per environment, under supervision, and the runtime system only has to keep track of the robot pose on a fixed map.

This design follows the assumptions from Chapter 1. The robot operates in known indoor spaces that do not change dramatically from one day to the next. In this setting, a 2D occupancy grid is usually enough for global navigation. It is compact, well supported by standard tools, and easy to inspect in RViz. At the same time, we still want the underlying SLAM to be robust: long straight corridors, glass panels, and occasional moving people can all confuse a naïve pipeline. For this reason, we use SLAM Toolbox (Macenski & Jambrecic, 2021) with the T265 as odometry and the 2D LiDAR as the main sensor, and we always check the resulting map visually before using it.

At runtime, localization is handled by AMCL and the Nav2 stack. The map created in the first step is loaded by the map server, and AMCL uses LiDAR scans and T265-based odometry to estimate the robot pose in the `map` frame. Nav2 then plans and controls motion entirely on the basis of this 2D map and the associated costmaps. In the rest of this section, we first describe how SLAM Toolbox is configured for mapping, then how we create and save maps, and finally how localization is done during navigation.

3.2.1 SLAM Toolbox Setup

The mapping pass is built around SLAM Toolbox (Macenski & Jambrecic, 2021). The goal is simple: during a short, manual run we want to build a clean 2D occupancy grid of the environment that we can later reuse for localization and planning. We keep the setup as standard as possible so that it is easy to rerun or adapt in other buildings.

All of our mapping runs use the same frames and topics. The T265 tracking camera provides odometry, which we expose as the transform `odom` → `base_link`. The 2D LiDAR publishes scans in the `laser` frame. In the SLAM Toolbox configuration, we set:

- map_frame = map,
- odom_frame = odom,
- base_frame = base_link,
- scan_topic = /scan.

This means SLAM Toolbox listens to the LiDAR scans on /scan and uses the T265-based odometry to predict motion between scans. It then corrects that motion with scan matching and closes loops when it revisits the same area.

We use the standard Ceres-based solver plugin provided by SLAM Toolbox (parameters solver_plugin, ceres_linear_solver, and so on). These control the optimization backend but are mostly left at the recommended values. More important for us are the parameters that control when new scans are accepted, how often the map is updated, and how aggressive loop closure is. For example:

- minimum_travel_distance and minimum_travel_heading set how much the robot needs to move before a new scan is used. We keep them at moderate values (0.5 m and 0.5 rad) to avoid overloading the system with almost identical scans.
- do_loop_closing is set to true, and the various loop_match_* parameters are tuned so that obvious loops are closed while avoiding spurious matches.
- resolution is set to 0.05 m, which gives a good balance between detail and file size for indoor corridors and labs.

We also enable use_odom_topic = true and provide_odom_frame = true. In our case, "odom topic" really means the T265 pose published through the RealSense driver (Schmidt *et al.*, 2019). SLAM Toolbox treats this as the motion prior and still corrects it with scan matching and loop closure. This setup is convenient: the T265 soaks up small motions and drift over short distances, and the LiDAR and SLAM Toolbox take care of long-term consistency.

The configuration enables use_map_saver = true, which allows us to save the current occupancy grid directly from SLAM Toolbox when the mapping pass is complete. The transform_publish_period and map_update_interval parameters control how often the map transform and the occupancy grid are updated. We keep them at a few Hertz so that RViz feels responsive but the CPU is not overloaded.

Finally, we use rolling_window = true during mapping. This keeps the active scan matching window around the robot and helps the solver stay focused on the area we are currently exploring, while still maintaining the global map internally. Combined with the loop closure settings, this is enough to produce a single, consistent 2D map of the lab and corridor areas used in the experiments.

Overall, the SLAM setup is intentionally conservative. We rely on standard components, avoid exotic settings, and use the T265 as a stable odometry source. This makes the mapping runs predictable and easier to repeat when we need to adjust the map or extend it to a new part of the building.

3.2.2 Map Creation and Saving

Map creation happens entirely in `mapping` mode. The workflow is straightforward and based on manual teleoperation.

We start by launching the mapping setup:

```
ros2 launch mapping_docker slam_nav_setup.launch.py mode:=mapping
```

At this point, the T265 node, LiDAR, static transforms, `pointcloud_to_laserscan`, SLAM Toolbox, and RViz are all running. In a separate terminal, we launch a joystick teleop node that publishes `/cmd_vel`. The `spot_nav` bridge forwards these velocity commands to the Spot driver, and the operator drives the robot through the environment.

While the robot moves, SLAM Toolbox builds the 2D occupancy grid in the `map` frame. In RViz we can see the live map, the robot pose, and the laser scans. This visual feedback is important: it helps the operator decide which areas still need to be covered, and it reveals problems like glass walls, reflective surfaces, or moving obstacles that might confuse the LiDAR.

We follow a simple strategy: first, trace the main corridor loop, then sweep into side rooms and around large objects, and finally close the loop again. Because loop closure is enabled, revisiting the same area allows SLAM Toolbox to correct drift and tighten the map. We try to avoid large dynamic obstacles during mapping (for example, groups of people moving around) and pick times when the environment is relatively quiet.

The SLAM configuration has `use_map_saver = true`, so once coverage looks good we save the map. This can be done either through the SLAM Toolbox service or through RViz's SLAM Toolbox panel. In both cases the result is a standard occupancy grid written to disk: a `.pgm` image file and a `.yaml` file describing the resolution, origin, and thresholds. In our setup, these maps are stored under the `mapping_docker/maps` directory. By convention, the final map used for experiments is called `final_map.yaml`.

In addition to SLAM Toolbox's own saver, we also integrate map saving with Nav2's map server format. The launch file passes the map YAML as `map_yaml`, and we use the same structure that Nav2 expects. This means the map produced in mapping mode can be loaded by the Nav2 `map_server` in navigation mode without conversion.

After saving, we usually restart RViz and load the saved map directly through the `map_server` node to check that the static map, the robot pose, and the laser scans all align well. If we see noticeable misalignment (for example, walls not matching scans, or drifting corridors), we rerun the mapping pass, sometimes with slightly adjusted SLAM parameters or a slower teleoperation pace.

In the end, map creation is not fully automatic, but the process is reproducible: one launch command, teleoperate the robot with joystick, watch RViz, save the map, and visually verify. Once a map for a given environment is stable, we reuse it for all later navigation and semantic mapping experiments in that environment.

3.2.3 Runtime Localization

Runtime localization happens in `navigation` mode and is handled by a standard combination of AMCL and Nav2, with the T265 as the odometry source.

We launch the system in navigation mode:

```
ros2 launch mapping_docker slam_nav_setup.launch.py mode:=navigation
```

The T265 node, LiDAR, static transforms, and `pointcloud_to_laserscan` still run as before. Instead of SLAM Toolbox, the launch file now starts:

- `nav2_map_server`, configured with `final_map.yaml`;
- `nav2_amcl`, using the same `map`, `odom`, and `base_link` frames as in mapping mode;
- the Nav2 planner, controller, behaviour tree navigator, and behaviour server;
- the Nav2 lifecycle manager, which brings all of them to the active state.

The map server loads the saved occupancy grid and publishes it as a static map. AMCL subscribes to the LiDAR scan topic and the T265-based odometry and estimates the robot pose in the `map` frame. Effectively, AMCL replaces SLAM Toolbox's localization component while keeping the same TF structure:

$$\texttt{map} \rightarrow \texttt{odom} \rightarrow \texttt{base\_link}.$$

Using AMCL on top of the T265 odometry is important for two reasons. First, it anchors the pose estimate to the static occupancy grid and compensates for the slow drift that accumulates in the visual–inertial odometry. Second, the particle filter in AMCL can recover from short tracking losses of the T265 by re-aligning the pose with the LiDAR scans of the environment. In practice, this combination is more stable over long traversals than relying on T265 odometry or scan matching alone (Aditya *et al.*, 2025).

The Nav2 costmap servers subscribe to the map, LiDAR scans, and robot pose and build global and local costmaps. The global costmap is aligned with the static map and is used for long-range planning; the local costmap is centred on the robot and reacts to new obstacles.

During navigation experiments, goals are sent either manually from RViz or via the object-selection CLI described in Section 3.4. In both cases, Nav2 receives a PoseStamped goal in the map and handles planning and control.

Because odometry still comes from the T265, the behaviour of the system during navigation matches what SLAM saw during mapping. This helps keep the map and runtime localization aligned. If the drift of the T265 grows too large, AMCL can still pull the pose back towards the map using LiDAR scan matching. In practice, this combination was sufficient for the lab and corridor environments considered here.

RViz is used again as the main debugging and monitoring tool. It shows the static map, the AMCL pose estimate, the LiDAR scans, the global and local costmaps, and the current Nav2 path. This visualisation is also useful when running semantic mapping and object–goal experiments: it makes it easy to see whether failures are due to localization errors, poor costmap settings, or issues in the semantic layer.

3.3 Semantic Mapping Pipeline

The semantic mapping pipeline sits between perception and navigation. Its job is simple to state: turn a noisy stream of camera detections into a small, stable set of named objects placed on the existing 2D map. The rest of the stack does not care about single frames or image coordinates. It only needs to know that, for example, "there is a chair at (x, y) in `map`, with this confidence, and it has been seen many times". Once this layer is in place, object–goal navigation only needs to query this object list and turn it into goal poses.

The pipeline follows the same idea we used in our earlier work on online object-level mapping for quadrupeds (Razavi *et al.*, 2025). YOLO detector (Redmon *et al.*, 2016) runs on RGB images from the Intel RealSense D435 and produces 2D bounding boxes and scores for a small set of classes. Depth, aligned to the color stream, provides a distance for each box. Using the calibrated camera intrinsics and the transform between the camera and the robot body, we back-project each detection to a 3D point in the camera frame and then into the `map` frame. A light association and memory step merges near duplicates and keeps only confirmed, static objects over time. In this section, we focus on the first two stages: detector runtime and 2D–to–3D projection.

Figure 3.4 illustrates the standard YOLO detection pipeline that underlies our detector.

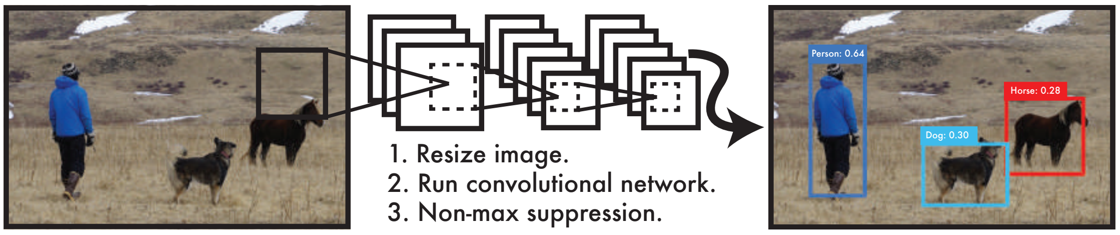

Figure 3.4: YOLO detection pipeline, adapted from Redmon *et al.* (2016). The input RGB image is first resized to a fixed resolution. A single convolutional network processes the entire image in a single forward pass and predicts, for each cell on a coarse grid, a small set of bounding boxes along with class scores and confidence values. A final non-maximum suppression step discards low-confidence and highly overlapping boxes, leaving a compact set of object detections for the semantic mapping layer.

3.3.1 Detector Runtime

The semantic pipeline starts from an RGB stream coming from the Intel RealSense D435. The camera runs at a moderate resolution and frame rate (for example, 640×480 at 15–30 Hz) so that the onboard NUC can keep up without a GPU. Depth is enabled and aligned to the color stream, but the detector itself only works on the color images. Depth is used later in the projection step.

For object detection, we use a YOLOv11 model, in a small configuration suitable for CPU inference (Redmon *et al.*, 2016). The goal is not to cover hundreds of classes, but to reliably detect a small, closed set of objects that are meaningful for navigation in indoor spaces: chairs, people, desks, doors, and similar man-made structures. In the experiments described in 4.2, we limited the semantic map to persons and chairs, which were sufficient to test the layer's stability and its usefulness for future navigation tasks.

Each RGB frame is pre-processed before it reaches the network. The image is resized to the network input size, normalized, and optionally cropped to remove parts of the field of view that are not useful (for example, the top of the image when there is only the ceiling). The detector node then runs a forward pass and produces raw bounding-box predictions: for each candidate box, it outputs a class label, a confidence score, and a 2D bounding box in pixel coordinates. Non-maximum suppression (NMS) operates on these raw outputs, keeping only the strongest box per object.

The detector uses a global confidence threshold, and we also allow per-class thresholds in the ROS 2 parameters. In practice, we noticed that some classes (such as "person") are detected very reliably, while others (for example, certain furniture classes) can be noisier. By exposing a parameter like `class_score_thresholds`, we can raise the minimum confidence only for those classes that tend to produce more false positives, while keeping a lower threshold for the classes that the model handles well. If a detection does not meet the threshold for its class, it is discarded immediately and never reaches the mapping step.

The detection node publishes its results as a ROS 2 message on a dedicated topic (for example `/yolo/detections`). Each message contains a header with the timestamp and frame ID (the camera frame), and an array of per-object entries. For each object, we store at least: the class name (as a lowercase string), the confidence score, and the 2D bounding box. In our implementation, this is a custom message type compatible with later stages and easy to visualize in RViz.

To debug and tune the system, the detector also publishes an overlay image that displays the bounding boxes and labels on top of the original RGB frame. This stream is not used anywhere in the pipeline; it is purely for the operator. When mapping or navigating, we often keep this view open in RViz to see when the detector is struggling, for example, under strong backlight, motion blur, or partial occlusions. This is important in practice because the semantic layer will only ever be as good as the input detections.

Runtime performance is a key constraint. The detector runs on the same onboard NUC as the rest of the pipeline, with no dedicated GPU. To keep latency under control, we keep the model size small, we avoid very high input resolutions, and we sometimes throttle the detector by skipping frames when the robot is moving slowly. In the experiments reported in section 4.1.3, the RGB camera stream runs faster than the detector, and the semantic layer runs slower again. The system is designed to tolerate this: semantic association works with whatever detection messages arrive, and the objects in the map remain visible even when no new frames are being processed.

A second important design choice is to limit the set of classes that the detector is allowed to pass to the mapper. The `show_object` parameter lets us specify a comma-separated list of class names (for example, "chair, person"). Only detections whose class is in this allow-list are forwarded. This avoids cluttering the semantic map with objects that are not meaningful for the navigation task at hand. For instance, in a lab we might want to see chairs and desks, but not small items like keyboards or mugs. Filtering at the detector layer also saves work in the later stages.

Finally, the detector does not try to do any temporal smoothing or tracking. Each message is a snapshot of what the camera saw at a given time. We deliberately keep the detector stateless and modular: it does not need to know about the occupancy map, the robot pose, or any past detections. All temporal logic (for example, deciding that a chair has been seen many times from different viewpoints and should be kept as a confirmed object) is handled by the semantic mapper. This separation keeps the detector simple, makes it easier to swap in a different model in the future.

In summary, the detector runtime has three responsibilities. It takes RGB frames from the D435, runs a lightweight YOLO model on CPU, and publishes a clean list of 2D detections (class, score, bounding box) in the camera frame. It filters out low-confidence and irrelevant classes, and it provides a debug overlay for the operator. It does not do mapping, memory, or navigation. The next subsection explains how we combine these 2D detections with depth to obtain 3D object positions that can be placed on the 2D map.

3.3.2 2D to 3D Projection with D435

The second step in the pipeline is to turn each 2D detection into a 3D point that we can place on the map. This happens in two stages: first we use the depth image and the D435 intrinsics to lift pixels from image space into the camera frame; then we use the calibrated transforms to move that point into the robot and `map` frames.

The Intel RealSense D435 provides synchronized color and depth streams with depth already aligned to the color image (Schmidt *et al.*, 2019). For every RGB frame used by the detector, there is a corresponding depth frame with the same resolution. The camera driver also exposes the intrinsic parameters of the color camera: focal lengths (f_x, f_y), the principal point (c_x, c_y), and the depth scale that converts raw depth units into metres. These values are published on the ROS 2 topic and are also saved in the calibration files.

Given a 2D bounding box from the detector and the aligned depth image, we first choose a representative point inside the box. In principle, we could use the exact center of the box, but we found that using a small window around the center and taking the median (or a trimmed mean) of the depth values is more robust to noise and outliers. For example, if the box covers a person, the depth at the very top of the box might be the wall behind them, while the center region is more likely to sit on the torso. In practice the window can be a fixed $k \times k$ patch (for example, 5×5 or 7×7 pixels) around the box center, ignoring invalid or zero depth values.

Once we have a valid depth value Z in metres and the pixel coordinates (u, v) of the chosen point, we compute the 3D point in the camera frame using the standard pinhole model,

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

This gives us a point (X, Y, Z) in the optical frame of the D435. Conceptually, this is a point in space that lies along the ray from the camera center through that pixel, at the distance given by the depth measurement. We also attach an orientation to this point, but since we are building a 2D semantic layer, we do not care about full 3D orientation. In the implementation, we later reset roll and pitch and keep only a yaw angle, so that objects "stand" on the floor plane.

Depth data is noisy and has its own failure modes. Close to the camera, very small depth errors can produce large lateral errors in (X, Y). On shiny or dark surfaces, the depth sensor may fail and return zeros. To reduce the effect of these issues, we apply a few simple filters at projection time. We discard detections whose projected range is outside a sensible interval (for example, less than 0.3 m or more than 5 m from the camera). We also discard boxes where the center patch has too many invalid depth pixels. These checks are cheap and remove many obviously bad cases before they reach the map.

The next step is to move the 3D point from the camera frame to the robot body frame and then into the `map` frame. We rely on the standard TF tree for this. During setup we calibrate and fix the static transform between the D435 and the robot body, so we have a transform from `base_link` to the camera frame. The Intel whitepaper on T265 and D435 describes the typical way to mount and align the two devices and highlights the importance of accurate extrinsics for tracking and depth fusion (Schmidt *et al.*, 2019). In our system we follow the same idea: the physical mount is rigid, the extrinsic transform is measured once and then kept constant in the TF tree.

At runtime, the SLAM Toolbox maintains a transform between `map` and `odom`, and the T265 provides odometry between `odom` and `base_link`. Combining these with the static transform from `base_link` to the camera, we can transform the 3D point from camera coordinates to the global `map` frame. In ROS 2 this is a standard TF lookup and transform operation, and it is exactly what the semantic mapper node does internally: given a detection message with a timestamp, it requests the transform chain up to `map` at that time and applies it to all points in that frame.

Because our semantic map lives on a 2D occupancy grid, we then "flatten" the 3D point onto the floor plane. In other words, we take the (x, y) coordinates in `map` and set z to a fixed value (usually zero or a small constant that matches the map height). For orientation, we keep only the yaw part of the quaternion and set roll and pitch to zero. This makes each object look like a small cube sitting on the map. This simplification is enough for the navigation tasks we target: we care about where the object is on the floor, not about its exact 3D shape.

In the thesis, this projection step is where the RGB–D camera, the T265 tracking camera, and the SLAM map all meet. The D435 and YOLO give us a class label and a direction in image space. Depth turns that direction into a physical distance. The T265 and SLAM Toolbox tell us where the robot was in the global map when the frame was taken. Putting these pieces together, we can say "there is a chair around (x, y) in `map` coordinates", with a confidence taken from the detector. This is the basic ingredient that the semantic layer needs to build its object list.

We also observed some limitations here. The depth plane of the D435 is not perfectly aligned with the LiDAR scan plane that was used to build the occupancy grid. As a result, there can be a small bias in range when placing objects on the map. In the lab this shows up as objects that are slightly closer to or further from walls than they should be. For this thesis we accept this limitation, since the main goal is to provide approximate object locations for navigation, not centimeter-accurate 3D reconstructions. A more detailed calibration or a joint optimization of LiDAR and depth data could reduce this error in future work.

To sum up, the 2D–to–3D projection step takes the output of the detector and turns it into 3D points in the `map` frame, using the D435 depth image, the camera intrinsics, and the TF tree that links the camera to the map. It filters out obviously bad depth readings, flattens the result onto the floor plane, and passes on a simple structure: class, confidence, position in `map`, and a yaw-only orientation. The next subsections (not shown here) use these projected detections to build and maintain a stable set of confirmed objects over time.

3.3.3 Transforms to the Map Frame

The semantic mapper relies heavily on TF. Every detection starts in a camera-related frame and must end up in the global `map` frame. If the transforms are wrong or missing, the whole semantic layer fails, so this part of the pipeline is quite careful.

The node exposes two main parameters for this: `global_frame` (by default `map`) and `fallback_frame` (by default `base_link`). The idea is simple: the semantic map lives in the global frame, so all object positions must be expressed there. At the same time, we want the node to tolerate small changes in how upstream nodes publish frames.

Each incoming detection message contains a header and an array of detections. In our setup, the detections come from the D435 RGB–D camera or from a small projection node that already did the image-to-3D step. The first thing the semantic mapper does is choose a "source frame" for the whole message. It checks each detection and tries to read a `frame_id` from the 3D bounding box. If it finds one, it uses that. If not, it falls back to the header frame. If that is still empty, it finally falls back to `fallback_frame`. This is a small detail, but it makes the node robust to small changes in the detector or projection code: we do not need to hard-code a specific camera frame name.

Once the source frame is known, the node asks the TF buffer for a transform from this frame to the global frame (`map`). It tries to get the transform at the exact timestamp of the message, with a short timeout (for example 0.1 s). This is the ideal case: the 3D point is placed where the robot actually was when the camera captured that frame. If this lookup fails (for example, because TF is slightly delayed or the robot just started), the node does a second attempt with a "latest available" transform. Only if this second attempt also fails do we give up on transforming the detections from that message.

Even when we skip a message due to missing TF, we still keep the semantic layer visible. The node calls a small re-publish function that takes the last known header (or a fresh header with the current time) and publishes the current set of confirmed objects. This way, RViz still shows the semantic map, and downstream consumers still see the same objects, even if there is a short TF hiccup or a gap in detections.

After the transform from source frame to `map` has been found, each detection goes through two possible transforms. First, if its 3D pose is expressed in a sub-frame (for example, a local "depth frame" inside the D435), the node transforms it into the selected source frame. Then, it transforms it from the source frame into `map`. In code this is just two calls to the TF library, but conceptually we are walking along the chain:

$$\texttt{map} \rightarrow \texttt{odom} \rightarrow \texttt{base\_link} \rightarrow \texttt{camera\_frame} \rightarrow \texttt{depth\_frame},$$

or the reverse, depending on how the frames are defined in the actual tree.

Once a detection is in the `map` frame, the node optionally clamps it to a fixed plane and removes roll and pitch from the orientation. Clamping means we set the z coordinate to a constant value (for example, 0.0 m). Removing roll and pitch is done by extracting the yaw from the quaternion and creating a new quaternion with only that yaw. This matches the idea of a 2D semantic layer on top of a 2D occupancy grid: objects live on the floor plane and have a heading, but we do not care about their full 3D pose.

By the end of this step, all valid detections are represented as "points with yaw" in `map`. They share the same global frame as the SLAM map, the Nav2 costmaps, and the robot pose. This makes later operations much simpler. For example, downstream components can directly query costmap values around each object to reason about clearance.

A final point is that the semantic mapper never writes to the TF tree; it only reads it. All transforms are produced by other parts of the system: SLAM Toolbox or AMCL maintain `map` to `odom`, the T265 node publishes `odom` to `base_link`, and static transform publishers provide the camera and LiDAR frames. This separation keeps responsibilities clear: mapping and localization maintain the geometry; the semantic layer consumes it and stays read-only with respect to TF.

3.3.4 Static Memory and De-duplication

Once detections are in the `map` frame, the pipeline must decide which ones to keep and how to store them. Detections are noisy. The same chair might appear slightly shifted from frame to frame. False positives also appear from time to time. If we took every detection at face value and inserted it into the map, we would soon end up with a cloud of points instead of a clean list of objects.

To avoid this, the semantic layer implements two ideas: it only keeps static objects that have been seen many times, and it merges observations that are close to each other in the map. In other words, it does both confirmation and de-duplication.

The long-term memory of objects is a simple dictionary keyed by an integer ID. For each object we store:

- the class label (for example, *chair* or *person*),
- a fixed position (x, y, z) in `map`,
- a yaw-only orientation,
- a confidence score,
- a hit count (how many times we saw it),
- timestamps for when it was first created and when it was last seen.

These are the "static" objects that we publish to other nodes and show in RViz. We expect them to correspond to things like chairs, desks, or common spots where a person stands, not to transient noise.

To decide whether a new detection should contribute to an existing object or start a new one, we use a distance-based matching rule. For a given class, we look for the nearest confirmed object within a certain radius. If we find one, we treat the detection as another sighting of that object. We do not move the object's position; we only update its last-seen time, increase its hit count, and, optionally, raise its score if the new detection has a higher confidence. This choice is deliberate: we want the map to be stable. An object should not walk around the map just because the detector jittered a few centimetres.

If there is no confirmed object of the same class within this radius, we treat the detection as a potential new object. However, we do not immediately insert it into the static list. Instead, we use a short-term memory of “candidates” that aggregates repeated detections in roughly the same area.

Conceptually, each candidate keeps:

- class label,
- an averaged position and yaw in `map`,
- a list of confidence scores,
- a hit count,
- the time of the first and last sighting.

When a new detection comes in, we look for a nearby candidate of the same class. If we find one, we pull its stored position slightly towards the new detection (a simple exponential moving average), append the score to the list, and update the hit count and last-seen time. If no candidate is close enough, we start a new one at the detection position.

Candidates that are not seen again within a short time (for example 1.5 s) are dropped. This acts as a time-based filter that removes isolated detections: if we saw something once and then never again, it is more likely to have been noise than a real static object. The time window also encourages the robot to keep the object in view for a moment if it wants that object to be confirmed.

Promotion from candidate to static object follows three simple conditions:

- the candidate must have been seen at least a given number of times (for example 10 hits),
- all hits must have occurred within a given time window (for example 2 s),
- the average confidence score must be above a threshold (for example 0.5).

Only when all these conditions are met do we consider the object “confirmed”. At that point, we check once more whether there is already a confirmed object of the same class within a slightly larger radius. If not, we create a new static object with the averaged position, mean score, and hit count from the candidate. If there is an existing static object in that guard radius, we do not create a new one. Instead, we treat the candidate as additional evidence for that existing object.

This two-level radius scheme (one radius for merging into an existing static object, another for deciding whether to create a new one) helps to keep the semantic map compact. Objects that are very close together merge into a single entry. Objects that are further apart but still within the guard radius are treated carefully to avoid duplicates. The exact values of these radii are parameters and can be tuned depending on the size of the environment and the typical spacing of objects.

There is also de-duplication within a single frame, as described in the previous subsection. There, we avoid keeping multiple detections of the same object in the same image by only accepting the highest-score detection in a small area. Combined with the longer-term aggregation described here, this gives us three layers of filtering:

1. per-frame: only high-score detections and no multiple boxes for the same object;
2. short-term: only detections that repeat in the same place within a short time;
3. long-term: only objects that survive the promotion conditions and are not near an existing static object.

The end result is a small, stable set of objects in the `map` frame. Their positions do not flicker; their scores only ever go up when strong detections arrive; and their hit counts give a rough measure of how often the robot has seen them. This is exactly the behaviour we need for object–goal navigation: the robot can trust that "chair at (x, y)" is not going to vanish just because the detector missed it for a few seconds.

3.3.5 Data Structures and Interfaces

The semantic mapper node is meant to be a reusable component in a larger ROS 2 system, so its internal data structures and external interfaces are kept simple and explicit. This also makes it easier to debug and to connect to other parts of the stack, such as the semantic recorder and object-selection CLI.

Internally, the node keeps two main dictionaries in memory: one for confirmed objects and one for short-term candidates. Both are keyed by integer IDs. The choice of integer keys is practical: it makes it easy to assign a stable ID to each object for RViz markers and logs, without depending on contents that might change.

Each confirmed object entry stores a small set of fields: class, position, yaw, score, hit count, creation time, and last-seen time. These are plain Python types (floats, ints, strings). There is no nested structure, no dynamic fields, and no direct references to ROS messages. This makes it straightforward to copy these entries into outgoing messages or into markers. It also makes it easy to extend the structure in the future if needed, for example to store a small covariance or a per-object bounding radius.

Each candidate entry stores a similar set of fields, plus a list of past scores and two timestamps (first and last sighting). Again, the fields are simple scalars and lists. This design avoids heavy dependencies and keeps the memory footprint small, which is useful when the node runs for long periods.

On the output side, the main interface is a `Detection3DArray` message on the topic `/semantic/detections_map`. Each confirmed object becomes one `Detection-3D`. The center of the detection's bounding box is set to the stored pose in `map`, and the result list has one `ObjectHypothesisWithPose` entry. The class ID is stored as the class name (for example "chair"), and the score field carries the object's current confidence. This structure is intentionally simple: downstream nodes that already understand `Detection3DArray` can consume it without modification.

The node also publishes a `MarkerArray` on `/semantic/markers`. For each confirmed object, it creates two markers: a cube at the object position and a text label slightly above it. The cube uses a fixed scale (configurable via parameters) and a fixed colour. The text label shows the class name and hit count. Marker lifetimes can be set to infinite or to a finite duration. Because the publishers use transient-local durability, markers remain visible in RViz even if RViz is restarted or connects later.

For inputs, the node subscribes to the YOLO detection topic (`/yolo/detections_3d` in our setup). The QoS profile for this subscription is reliable, with a small history. This is enough because we do not need a large backlog: detections that arrive late are not very useful for mapping. The output QoS for the semantic detections and markers is also reliable and uses transient-local durability, so that new subscribers can immediately get the latest state without waiting for a new callback.

Configuration is handled through standard ROS 2 parameters. There are parameters for topic names (input, output, markers), frame names (`global_frame`, `fallback_frame`), TF timeouts, and all of the thresholds mentioned earlier: global minimum score, per-class thresholds, frame de-duplication radius, merge and guard radii, hit counts, time windows, and marker properties. In the thesis, we fix most of these values in a YAML file under the semantic mapping package, and we only adjust them when moving to a significantly different environment.

From a system perspective, the semantic mapper's interface to navigation is one-way. It does not talk directly to Nav2. Instead, the semantic recorder subscribes to `/semantic/detections_map` and periodically saves the confirmed objects to a YAML file, and the object-selection CLI loads this file, lets an operator choose a target, and converts that choice into a goal pose for Nav2. This separation keeps the roles clear: the semantic mapper is responsible for "what and where"; the navigator handles "how to get there".

Finally, the node is designed to behave well in long runs. A periodic timer re-publishes the current static objects and markers at a fixed rate, even when no new detections come in. This keeps the semantic layer visible in RViz and makes logging easier: rosbag recordings always contain a regular stream of semantic snapshots. Because the internal data structures are small and simple, this periodic republishing has a low cost and does not put pressure on the CPU.

Overall, the combination of simple internal structures and standard ROS messages makes the semantic mapper easy to integrate. It provides a clear, typed interface for "objects in the map" that can be consumed by navigation, monitoring, or logging components, while hiding the noisy details of raw detections and association behind a small, self-contained node.

3.4 Object-Goal Navigation

The previous chapters described how we build a 2D map, how we localize the robot on that map, and how we maintain a thin semantic layer of confirmed objects in the `map` frame. This section explains how that information is used to move the quadruped: given a target class and a specific object instance on the map, the system computes a goal pose with a safe offset and sends it to Nav2, which, in turn, drives Spot through the `spot_ros2` bridge.

A small recorder node converts the live semantic layer into a static YAML database of object poses, and a command-line client lets the operator choose either an object from that list or an initial "home" pose. For each choice, the client generates a standoff-and-facing goal in the `map` frame and submits it as a standard `NavigateToPose` action.

Once Nav2 accepts the goal, the rest of the pipeline is identical to classical map-based navigation: a global planner finds a path on the 2D occupancy grid, local controllers track it under speed limits, and the Spot driver consumes `/cmd_vel` commands (Macenski *et al.*, 2020). In this way, object–goal navigation is reduced to three pieces that are easy to inspect and debug: object selection, goal pose generation, and the existing Nav2 + `spot_ros2` stack.

3.4.1 Object Selection Logic

Once the semantic mapper has built a stable list of objects in the `map` frame, we still need a simple way to pick one of these objects as a navigation target. In this thesis, we keep this step very explicit and transparent. Instead of a black-box policy, we use a small "recorder + CLI" pipeline that makes the object list visible, stores it on disk, and allows an operator to choose which instance to visit.

From live semantic layer to static database. The first step is to convert the live semantic map into a persistent database. The node `semantic_recorder` subscribes to the confirmed object stream on `/semantic/detections_map` (the output of the semantic mapper described in Section 3.3). This topic carries a `Detection3DArray` message in the `map` frame. Each detection holds a pose and at least one hypothesis with a class label and a score.

The recorder maintains an in-memory list of `objects`. Each entry is a dictionary with:

- `class`: the object class as a lowercase string (e.g., *chair*, *person*),
- `x`, `y`, `z`: position in the `map` frame,
- `yaw`: heading in the plane, derived from the quaternion (q_z, q_w),
- `score`: the maximum confidence score seen so far for this object.

This list is periodically written to a YAML file (by default `semantic_objects.yaml`) under the mapping package. The file has a single top-level key, `objects`, and is human-readable, so it can be inspected or edited as needed.

On startup, the recorder can either clear the file or reload it, depending on the `reset_on_start` parameter. In the mapping runs used in this thesis, we set `reset_on_start = true` so that each recording session starts with an empty database and captures only the current environment. The node creates the directory if needed, removes the old file during a reset, and immediately writes an empty database so that the path always exists.

Merging nearby instances. When a new `Detection3DArray` message arrives, the recorder loops over its detections. For each detection, it selects the highest-score hypothesis (if there are several) and extracts the class label, pose, and score. The pose is already in `map`, so we only need (x, y, z) and the yaw angle. The yaw is recovered from the stored quaternion using a simple analytic formula.

Before inserting a new object into the list, the node calls a de-duplication function. The idea is that the semantic mapper may still publish several very close instances of the same class, either because two static objects are close together or because the confirmation logic allowed small shifts in the final position. In the object database, we want each physical object to appear at most once.

The de-duplication checks all existing entries with the same class and computes the squared distance in the map plane. If the closest one lies within a configurable radius (for example, 0.5 m), the recorder does not create a new entry. Instead, it updates the existing one with a smoothed position and a higher score:

- position is updated with an exponential moving average, so the object drifts slowly towards the new observation but does not jump,
- yaw is replaced by the latest yaw (we assume small errors here are acceptable),
- score is updated to the maximum of the previous and current scores.

If no existing entry of that class is close enough, the detection is treated as a new object and appended to the list.

This extra de-duplication step is intentionally simpler than the semantic mapper's internal memory. The mapper is optimized for online filtering of noisy detections; the recorder is only concerned with producing "one entry per object" and a reasonable pose for each entry. The smoothing and maximum score logic help remove residual noise while preserving the fact that there are only a handful of objects in the scene.

Periodic saving. To avoid heavy disk usage, the recorder does not write the YAML file on every message. Instead, it keeps track of the last save time and calls `_save()` only after a fixed interval (e.g., every 2 seconds). The save function dumps the current list to YAML:

```
objects:
  - class: chair
    x: ...
    y: ...
    z: ...
    yaw: ...
    score: ...
  - class: person
    ...
```

On shutdown, the node writes one final copy so that no recent changes are lost. This behaviour makes the database robust across runs: even if the node is interrupted, the file contains a recent snapshot of the semantic map.

Manual object selection via CLI. The second part of the object selection logic is implemented in the `semantic_nav_cli` node. This node does not process sensor data. Instead, it loads `semantic_objects.yaml`, prints a menu, and lets the operator pick which object to navigate to.

When `semantic_nav_cli` starts, it reads the YAML file into a list of dictionaries with the same structure as above. If the file is missing or invalid, it reports an error and shows an empty list. The main loop then prints a simple text menu:

- entry 0 is an "initial point", defined by parameters `initial_x`, `initial_y`, and `initial_yaw`. This is useful for sending the robot back to a known parking or start pose.
- entries 1, 2, ... list the recorded objects in order. For each object, the menu shows the class and the (x, y) position in `map`.

The user types a number to choose a target, or `r` to reload the database from disk (for example, after updating the semantic map), or `q` to quit. This simple interface keeps the control loop explicit: it is always clear which object the robot is going to, and the operator remains in the loop.

When the user selects entry 0, the node builds a goal at the initial point. When the user selects an object entry, the node calls a helper function to compute a standoff goal in front of the object (Section 3.4.2). In both cases, the result is a `NavigateToPose` goal that is sent to the Nav2 action server.

In summary, the object selection logic is built from two small components: a recorder that converts the live semantic layer into a clean YAML database of objects, and a CLI that loads that database and allows an operator to select a specific target or the initial pose. This keeps the behaviour transparent and debuggable: we can always inspect the YAML file, see the menu, and understand which object is being used for navigation.

3.4.2 Goal Pose Generation (Offset and Facing)

Once an object has been selected, the system must convert its stored pose into a navigation goal that is meaningful for Spot. The goal should not be precisely on the object centre; instead, the robot should stop at a safe distance and face the object. This subsection describes how the `semantic_nav_cli` node computes such goals in the `map` frame and hands them to Nav2.

Goal representation. Nav2 expects goals as `NavigateToPose` actions. Each goal wraps a `PoseStamped`:

- the header frame is the global frame (here `map`),
- the position is (x, y) in that frame (we keep $z = 0$),
- the orientation is a quaternion that encodes yaw only.

To keep things simple, the CLI uses a small helper that takes (x, y, yaw), sets the frame to `map`, stamps it with the current time, converts yaw to a quaternion, and builds the corresponding `NavigateToPose` goal.

Reading the robot pose. To compute a standoff point in front of an object, it is useful to know the robot's current location. The CLI uses the TF buffer to look up the transform from `map` to the base frame (by default `base_link`). From this transform it extracts:

- the robot position (x, y) in `map`,
- the robot yaw, again reconstructed from the quaternion's q_z and q_w.

If this TF lookup fails (for example, because localization has not started yet), the node falls back to a simpler rule described later. In normal operation, though, TF is available and the standoff point can be defined relative to the current robot pose.

Standoff along the line of sight. Assume we have an object at (o_x, o_y) and the robot at (r_x, r_y). The CLI computes the vector from the robot to the object,

$$d_x = o_x - r_x, \quad d_y = o_y - r_y,$$

and its length,

$$d = \sqrt{d_x^2 + d_y^2}.$$

We want the robot to stop at a fixed standoff distance d_{off} in front of the object (for example, 0.30 m), measured along this line of sight. If the robot is already closer than this distance, there is nothing to do; the goal is set to the current position. If it is further away, we move along the segment from robot to object, but we stop early:

$$d_{\text{stop}} = \max(d - d_{\text{off}}, 0), \quad \text{scale} = \begin{cases} 0 & \text{if } d \approx 0, \\ d_{\text{stop}}/d & \text{otherwise,} \end{cases}$$

$$g_x = r_x + d_x \cdot \text{scale}, \quad g_y = r_y + d_y \cdot \text{scale}.$$

The goal yaw is set so that the robot faces the object:

$$\text{yaw}_g = \text{atan2}(d_y, d_x).$$

The final goal pose is (g_x, g_y, yaw_g) in the `map` frame. Intuitively, this means: "walk straight towards the object, but stop d_{off} metres in front of it, looking at it".

This construction has a few nice properties. The line from the robot to the goal lies entirely on the line of sight to the object, so if the global map and costmaps are consistent, the robot should get a clear view of the object near the end of the trajectory. The standoff distance is configurable via the `standoff_m` parameter, allowing us to be more conservative around people and desks and a bit closer to walls or shelves if needed.

Fallback using object heading. If TF is unavailable or the robot pose cannot be computed for some reason, the CLI cannot define the standoff point relative to the robot's current position. In that case, it falls back to a rule based on the stored object heading. The object pose in the database includes a yaw angle yaw_o, which roughly indicates "which way the object is facing" in the map. For a person, this might be the direction they were facing in the frame when they were recorded; for a chair, it might be the direction the chair was pointing.

In the fallback mode, the goal position is computed as:

$$g_x = o_x - d_{\text{off}} \cos(\text{yaw}_o), \quad g_y = o_y - d_{\text{off}} \sin(\text{yaw}_o),$$

and the goal yaw is set to yaw_o. In words: we place the robot "in front" of the object along its heading, at the chosen standoff distance, and orient the robot in the same direction as the object. This works best when objects are near walls and their heading points into the room; the robot ends up between the object and the free space.

In practice, TF is usually available, so the line-of-sight rule is used. The fallback is primarily a safety net and a way to keep the CLI usable even when localization is not fully configured.

Sending goals and handling cancellation. Once the goal pose has been computed, the CLI sends it to Nav2 through an `ActionClient` on the configured action name (by default `navigate_to_pose`). The node waits for the action server to be available, then sends the goal and enters an "active navigation" loop. While the goal is active, the node:

- periodically spins ROS so that feedback and result messages are processed,
- watches the result future to see when Nav2 finishes,
- listens to the terminal input so the operator can type `s` or `stop` to cancel.

If the operator cancels, the CLI calls the action's cancel API and, as an extra precaution, publishes a zero `Twist` on `/cmd_vel`. This makes sure the robot stops even if the action cancellation takes a moment to propagate. When the navigation finishes normally, the node prints the result code and returns to the menu, ready for another selection.

The same goal-generation logic is used for all objects and for the initial point. For the initial point, the goal pose is taken directly from the configured (x, y, yaw) instead of being computed from an object. This is useful for quickly returning the robot to a known safe pose after visiting one or more objects.

Overall, the goal pose generation is intentionally geometric and straightforward. It uses the object pose in the map, the current robot pose (when available), and a single standoff parameter to produce a goal that is both practical (a clear view of the object, a conservative distance) and easy to reason about. This fits well with the rest of the system: the semantic mapping and object selection steps produce a small, interpretable list of targets, and the goal generation step turns those targets into standard Nav2 goals without hiding any of the geometry behind a black-box policy.

3.4.3 Integration with `spot_ros2`

The last step in the pipeline is turning Nav2 velocity commands into real motion on Spot. In this work we do not touch any low-level locomotion or internal state estimation on the robot. All of that stays inside the official `spot_ros2` driver and the Boston Dynamics SDK. Our code in the `spot_nav` package only does two things:

- send high-level trigger commands (*claim*, *power on*, *stand*, *sit*, *power off*, *release*, etc.) to manage the robot's lifecycle;
- bridge planar velocity commands from Nav2's `/cmd_vel` topic to the driver's namespaced `/cmd_vel`, with clamping and a small timeout.

This keeps the integration thin and easy to understand. From the mapping and navigation stack point of view, Spot behaves like a mobile base that accepts `/cmd_vel` in the `base_link` frame, publishes odometry and TF, and exposes a few services for startup and shutdown.

Trigger services via `SimpleSpotCommander`. The `spot_ros2` driver exposes a set of `Trigger` services under the robot namespace. In this thesis we wrap them in the `SimpleSpotCommander` class. The list of services we bind to is:

`claim`, `release`, `stop`, `self_right`, `sit`, `stand`, `power_on`, `power_off`, `estop/hard`, `estop/gentle`.

On construction, `SimpleSpotCommander` receives an optional `robot_name` (used as a ROS namespace) and a node handle. It then:

1. builds the fully qualified service name for each base command, optionally prefixing it with the robot namespace (for example `/my_spot/stand`);
2. calls `create_client(Trigger, service_name)` for each of them;
3. waits for each service to become available before returning.

All clients are stored in a small dictionary keyed by the base command name. The `command()` method then just looks up the right client and calls it with an empty `Trigger.Request`. If the command is unknown, it returns a failed response and logs the problem. This wrapper is used in two places:

- in a small standalone CLI that allows manual typing of commands like *stand*, *sit* or *self_right* (useful for debugging the driver on its own);
- inside the main `SpotControl` class, which uses a fixed sequence of commands for startup and shutdown.

In practice, each experiment follows the same script. Initially, we call `claim` to lock the robot, then `power_on`, and finally `stand`. At the end, we send a few zero velocity commands, then `sit`, `power_off`, and `release`. Emergency stop commands remain available via services, but are not part of the automatic logic in this thesis.

SpotControl: bridging `/cmd_vel`. The core of the integration is the `SpotControl` class in `spot_nav`. This class is started as a ROS 2 process using a small helper (`synchros2.process.main`), which sets up the node and then spins it. The constructor:

- creates or reuses a node;
- reads parameters that control how velocity is bridged;
- sets up publishers, subscribers, and a timer;
- constructs a `SimpleSpotCommander` instance.

The key parameters are:

- `in_cmd_vel_topic` (default `/cmd_vel`): the topic where Nav2 publishes velocity commands;
- `out_cmd_vel_topic`: the topic where the Spot driver expects commands; by default this is `/robot_name/cmd_vel` if a robot name is given, or `/cmd_vel` otherwise;
- `keepalive_rate`: how often the last command is re-published (for example, 20 Hz);
- `timeout`: how long we allow the last command to be reused before forcing a stop (for example 0.5 s);

- `max_linear`, `max_angular`: absolute limits on linear and angular speed.

A subscriber listens on `in_cmd_vel_topic`, and a publisher writes to `out_cmd_vel_topic`. The callback `_on_twist()` is triggered whenever a new `Twist` message arrives and:

1. creates a new `Twist` called `out`;
2. clamps each velocity component:
   - `linear.x` and `linear.y` are saturated in [− `max_linear`, `max_linear`];
   - `angular.z` is saturated in [− `max_angular`, `max_angular`];
3. publishes the clamped twist to the driver;
4. stores `out` as `_last_cmd` and records the time as `_last_time`.

All other fields of the original message are ignored. The clamp function is just a simple `max(lo, min(hi, v))`. These limits are an extra safety layer on top of Nav2's configuration. Even if a wrong parameter in Nav2 allowed higher speeds, `SpotControl` would keep them within the chosen bounds.

Keepalive and automatic stop. To avoid sending commands only when new messages arrive, `SpotControl` also runs a small timer at `keepalive_rate`. The timer callback `_tick()` checks how much time has passed since the last command:

- if the elapsed time Δt is less than or equal to `timeout`, it simply re-publishes `_last_cmd` on the Spot command topic;
- if Δt exceeds `timeout` and `_last_cmd` is non-zero, it publishes a zero `Twist` once and resets `_last_cmd` to zero.

This behaviour forms a basic "watchdog" for the driving commands. As long as Nav2 is alive and calling `/cmd_vel`, the bridge repeats the last command at a steady rate, ensuring the driver always receives fresh messages. If Nav2 stops for any reason (crash, stop, misconfiguration), the bridge notices that no new commands have arrived, and forces a stop after at most `timeout` seconds.

On shutdown, `SpotControl` cancels the timer and, before using the trigger services, sends a few more zero `Twist` messages with short delays. This is a small extra precaution to ensure the robot does not move when `sit` and `power_off` are sent.

Lifecycle: from ROS to the robot. Putting all the pieces together, a typical run in this thesis looks like this:

1. **Driver and sensors.** The `spot_ros2` driver is started, together with the T265, LiDAR, and external cameras. The driver publishes TF and odometry and exposes the trigger services.

2. **Bridge startup.** The `spot_nav` process starts. `SpotControl` creates the velocity bridge, connects `SimpleSpotCommander`, and calls `initialize_robot()` and `stand_up()`. If any of these steps fail (for example the driver is not ready, or the claim fails), the node logs the error and exits.

3. **Navigation.** Mapping or navigation nodes are launched. When Nav2 is running and receives a goal (either from RViz or from the object-goal selector), it publishes `/cmd_vel`. The bridge clamps and forwards those commands to the driver at the configured keepalive rate. The robot walks according to the Nav2 plan.

4. **Cancel and stop.** If the operator cancels a Nav2 goal from the CLI, the CLI cancels the action and optionally publishes a zero `Twist`. Even without that, if `/cmd_vel` stops updating, the bridge timeout sends a stop. In all cases, the robot does not continue moving on stale commands.

5. **Shutdown.** When the session ends or the user presses `Ctrl+C`, `SpotControl` stops the timer, sends a few zero velocity commands, and then calls `sit`, `power_off`, and `release`. The robot returns to a stable sitting pose and is powered down.

This integration keeps the ROS 2 and robot sides clearly separated. Nav2, the semantic mapper, and the object-goal selector only ever see standard ROS topics and actions (`/cmd_vel`, TF, `NavigateToPose`). The Spot-specific details (trigger services, robot namespace, and command expectations) are hidden inside a small, testable bridge.

3.5 Goal Selection Interfaces and Human–Robot Interaction

The object–goal navigation loop described above only needs a single input: a pose in the `map` frame with a standoff and facing constraint. How that pose is chosen is a front–end design choice. In this work we expose two interfaces that produce exactly the same type of goal for Nav2:

- a menu–based command–line tool (`semantic_nav_cli`), where the user picks an object instance by number; and
- a small language and voice interface that lets the user say commands such as “go to the nearest chair” or “go back to the second person”.

Both interfaces operate on the same semantic database produced by the mapper and recorder. They do not change the underlying mapping or navigation stack; they only decide which object in the database should be turned into a standoff goal. In the experiments we still use the original menu interface for repeatable trials, and we treat the language interface as a lightweight human–robot interaction layer on top of the same pipeline.

3.5.1 Menu-Based Goal Selection

Semantic database as a shared backend. The semantic mapper and recorder described in Section 3.3 maintain a YAML file on the NUC with one entry per confirmed object:

{`class, x, y, z, yaw, score, ...`}.

For goal selection we only need the object class and its pose in the `map` frame. A small helper node publishes the current contents of this file as a `std_msgs/String` on `/semantic_objects_yaml` whenever the file changes.

The two goal–selection front ends both read from this shared stream. The CLI tool parses the same YAML file directly from disk, while the language interface subscribes to the topic and keeps an in–memory copy of the object list. In both cases the semantic database is treated as read-only; the front ends never modify object positions.

Menu–based goal selection (baseline). The baseline interface is the Python node `semantic_nav_cli`. It loads the YAML database, prints a numbered list of objects (for example, “1. chair at $x = 1.2$, $y = 0.5$”), and lets the operator choose a target by typing an index. Index 0 is reserved for a manually defined “initial point”. Once an object is selected, the node computes a standoff pose in front of it using the geometry in Section 3.4.2 and sends a `NavigateToPose` action goal to Nav2.

This tool is simple but useful for debugging and repeatable evaluations: each run can be started from the same terminal, with the same index, on the same map. It also provides a clear fallback if the language interface is unavailable or mishears a command.

3.5.2 Voice Command Interface

To enable spoken commands without changing the underlying navigation logic, we add a small speech frontend made of two ROS 2 nodes:

- a speech-to-text node (`voice_stt_node`) that records a short audio clip on demand and transcribes it; and
- a text-to-speech node (`tts_feedback_node`) that reads back short feedback messages so the user knows what the robot understood.

Both nodes communicate with the rest of the system only through simple text topics. This keeps the speech components optional and easy to replace.

Speech-to-text with Whisper. The `VoiceSTTNode` exposes a simple "push-to-talk" interaction on the NUC console. Every 0.1 s it polls `stdin`. When the user presses Enter, the node records a fixed-duration audio clip (by default 5 s) from the system microphone at 16 kHz using `sounddevice`. The audio is then passed to a local Whisper model (Radford *et al.*, 2022) via the `faster_whisper` library (Klein, 2023).

Parameters configure the node:

- `model_size`: which Whisper size to load (default `small`);
- `device`: `cpu` or `cuda` if a GPU is available;
- `compute_type`: quantisation mode (for example `int8`) to reduce CPU load;
- `sample_rate`: audio sampling rate (16 kHz in our setup);
- `record_seconds`: recording length per command; and
- `language`: expected language of the commands (here English).

We use Whisper because it is robust to background noise, works well on short commands, and is available in efficient variants that run locally on the NUC. The `small` model, quantised to `int8`, offers a good trade-off between recognition quality and CPU usage; it avoids streaming audio to an external service and keeps all speech data on board (Radford *et al.*, 2022).

Whisper returns one or more text segments. The node concatenates these into a single command string, logs it, and publishes it as a `String` on the `/nav_query_text` topic. If the user types "q" or "quit" instead of pressing Enter, the node shuts down cleanly. No audio is stored to disk; only the recognised text is passed on.

Feedback via text-to-speech. The language goal node (Section 3.5.3) publishes short feedback messages on `/nav_feedback_text`, for example:

- "Navigating to: chair at (1.20, 0.50)";
- "I could not match your request to any mapped object"; or
- "Navigation finished."

The `tts_feedback_node` subscribes to this topic and passes each message to a local text-to-speech engine, which reads the text aloud on the lab speakers. We use a lightweight offline TTS backend so that the node remains responsive on the CPU-only NUC and does not depend on network connectivity. The exact synthesis quality is not critical here; the goal is simply to confirm what the robot understood without requiring the user to read the console.

3.5.3 Language-Based Goal Selection

The central piece of the language interface is the `VLMLanguageGoalNode`. It turns the transcribed text from `/nav_query_text` and the current semantic database from `/semantic_objects_yaml` into a standard `NavigateToPose` goal for Nav2. At runtime we only use language signals, but the model behind this node is a vision–language model: CLIP (Radford *et al.*, 2021). We rely on CLIP's text encoder because it has been trained on image–text pairs and therefore captures object-level semantics in its language space.

Initialisation and model choice. On startup, the node:

- creates a TF buffer and listener, to query the robot pose in the `map` frame;
- initialises an `ActionClient` for the Nav2 `NavigateToPose` action;
- subscribes to `/semantic_objects_yaml` and `/nav_query_text`; and
- loads a CLIP text encoder (Radford *et al.*, 2021) using the `open_clip` library (Cherti *et al.*, 2023).

We use the text branch of a CLIP ViT-B/32 model with OpenAI weights. This variant is widely used, small enough to run in real time on the NUC (especially when only the text tower is used), and provides good-quality text embeddings for common object classes. The node loads the model on GPU if available, otherwise on CPU, and keeps it in evaluation mode. A small publisher on `/nav_feedback_text` is used to send textual feedback to the TTS node.

Unlike a full VLM-based planner, we never run CLIP on images during navigation. All images have already been processed by the detector and semantic mapper. Here CLIP serves only as a language embedding model that helps match free-form commands ("take me to the other chair", "go to the person near the table") to the discrete set of objects in the semantic map.

Maintaining an in-memory semantic database. Whenever a new YAML message arrives on `/semantic_objects_yaml`, the node parses it with `yaml.safe_load` and extracts a list of objects in the form

{`class: str, x: float, y: float, yaw: float`}.

Other fields in the YAML (e.g. height or a confidence score) are ignored here. The class labels are stored in a list `self.labels`. The node then calls a helper `_encode_text` that tokenises all labels, runs them through CLIP's text encoder, and L2-normalises the resulting vectors. The result is a matrix of label embeddings `label_embs` of size $N \times D$, with N the number of objects.

This step is cheap and only runs when the YAML changes. It gives a compact representation of the semantic map in CLIP's language space. A small dictionary `last_idx_for_class` stores, for each class label, which object index was chosen last, so that commands like "another chair" can cycle through instances.

Understanding a text command. When a new `/nav_query_text` message arrives, the node processes the string in several stages:

1. **Stop and home commands.** The text is lowercased and scanned for keywords:
   - If it contains "stop", "cancel", or "abort", the node cancels any active Nav2 goal and sends a feedback message.
   - If it mentions "initial", "start", or "home", the node sends a goal to a predefined initial pose $(x_{\text{init}}, y_{\text{init}}, \psi_{\text{init}})$, using the same convention as the CLI tool.
2. **Ordinal extraction.** The node then looks for ordinal expressions such as "second", "third", or "6th" using a small dictionary and a regular expression. If found, it records the desired index (for later use when there are multiple instances of the same class).
3. **Direct string match for class labels.** The simplest case is when the user explicitly mentions a class name that appears in the semantic database (for example "chair" or "person"). The node scans the command for each known

class label and collects indices of matching objects. If more than one class appears in the text, it picks the one whose name occurs first in the sentence.

If at this point one or more objects of the chosen class are available, the node has a set of candidate indices. If no class name can be matched at all, it falls back to CLIP.

CLIP similarity as a fallback. When there is no direct string match, the node embeds the whole command text with the same CLIP text encoder used for labels and computes cosine similarities between the query embedding and each label embedding:

$$s_i = \langle \hat{e}_i, \hat{q} \rangle, \qquad i = 1, \dots, N.$$

The index with the highest similarity is selected. If the best similarity is below a configurable threshold (default 0.25), the node refuses the command and replies that it could not match the request to any mapped object.

Otherwise, the chosen label defines a class (for example, "office chair" and "chair" would both be treated as chairs), and all objects of that class become candidates. This CLIP-based step allows simple paraphrases or longer sentences to work even when the exact label string is not present in the command.

Choosing one instance among candidates. Given a class and a set of candidate object indices, the node applies a small set of rules to choose one index:

- If the text contains "closest" or "nearest" and there are at least two candidates, the node queries the robot pose in `map` using TF and picks the candidate with the smallest Euclidean distance.
- If an ordinal was extracted ("second chair", "third person"), it uses that to index into the sorted list of candidates. If the requested index is out of range (for example, "third chair" when only two chairs exist), the node reports this explicitly.
- If the text contains words such as "another", "other", or "next", and there is more than one candidate, the node cycles through instances of that class. It stores the last index used for that class and returns the next one in the list, wrapping around if needed.
- If none of the above applies, it simply picks the first candidate.

These rules cover simple but common patterns in human commands ("nearest chair", "the second person", "another chair") without requiring a full natural language parser.

Goal generation and Nav2 interaction. Once an object index has been chosen, the node retrieves the stored map coordinates (x_o, y_o) and yaw ψ_o from the semantic database. It then computes a standoff pose in the same way as the CLI tool:

- If the current robot pose in `map` is available from TF, it places the goal on the line from the robot to the object, at a fixed distance `standoff_m` in front of the object, and oriented to face it.
- If the robot pose is not available, it falls back to a pose at a fixed offset in front of the object using the stored object yaw.

This pose is wrapped into a `PoseStamped` in the global frame (usually `map`) and sent as a `NavigateToPose` goal to Nav2. Before sending a new goal, the node cancels any active goal to avoid overlap. Feedback messages such as "Navigating to: chair at (x, y)" and "Navigation finished" are published on `/nav_feedback_text` and read aloud by the TTS node.

From Nav2's point of view, there is no difference between a goal triggered by the CLI and one triggered by the language node: both are standard `NavigateToPose` actions in the same frame, using the same standoff distance and tolerances.

Role in this thesis. This language and voice interface is deliberately limited in scope. It does not change the mapping or semantic modules, and it does not introduce new perception. Instead, it sits on top of the confirmed-only semantic map and gives the user a second way to choose goals:

- the menu-based CLI for repeatable, scriptable experiments; and
- the language interface for small human–robot interaction demos, where the operator can talk to the robot using short English commands.

In the experimental chapter we mainly use the CLI interface to keep conditions controlled. The language interface is included as a proof-of-concept front end that shows how the same modular semantic map can support both classical and conversational goal selection without changing the underlying navigation stack.

Chapter 4

Experimental Results

This chapter reports how the system behaves on a real robot across three indoor environments. In this first part, the focus is only on the geometric backbone: building 2D occupancy maps with SLAM Toolbox using the 2D LiDAR and the T265 odometry. Semantic mapping and object–goal navigation experiments are described in the following sections.

4.1 2D Mapping with SLAM Toolbox

All the maps used in this thesis are built with the same setup described in Chapter 3. The robot carries a 2D LiDAR and an Intel RealSense T265 tracking camera. The T265, running through the `realsense2_camera` driver, publishes the odometry that we treat as the `odom` frame. The 2D LiDAR publishes a planar scan on `/scan` from a frame rigidly attached to `base_link`. SLAM Toolbox runs in online asynchronous mode and fuses `/scan` and the T265-based `/odom` into a 2D occupancy grid.

During mapping, the robot is driven entirely with the official Spot tablet and joystick interface. No velocity commands are sent from ROS in this phase, and the `spot_nav` bridge is not used. From the ROS side, the only information used for mapping is the T265 odometry and the LiDAR scans. The Spot ROS 2 driver may be running for convenience and safety, but its internal odometry output is ignored in the SLAM pipeline.

Once coverage of the area looks sufficient in RViz, the current SLAM Toolbox map is saved to disk and reused for later experiments in that environment. We do not update the map during navigation; each environment uses a single fixed 2D occupancy grid as the geometric backbone for semantic mapping and object–goal navigation.

We built maps in three real environments:

- a small church, as an example of mapping outside the laboratory;
- the main DLS lab space;
- a large open indoor room at IIT used for experiments with Spot.

The same mapping procedure and SLAM configuration are used in all three cases.

4.1.1 Mapping Procedure

The mapping procedure is intentionally simple so that it can be repeated without fine-tuning:

1. Launch the system in *mapping* mode (SLAM Toolbox active, Nav2 off).
2. Check in RViz that the LiDAR scans, T265 pose, and TF tree are correct.
3. Drive the robot slowly through the environment with a joystick.
4. Close at least one loop so that SLAM Toolbox has a chance to correct drift.
5. When the map looks stable, save it to disk and stop the mapping run.

Maps are saved using the standard Nav2 map saver. The tool writes a `.pgm` file with the occupancy grid and a `.yaml` file with the resolution, origin, and threshold parameters. Later, these files are loaded by the navigation stack and by the semantic mapping node. We keep one map per environment and reuse it for all later experiments.

4.1.2 Church Environment

The church environment is a compact public space with a central hall and side areas. The structure is mostly open, with long straight walls, columns, and benches. This environment lets us test the mapping pipeline outside the usual laboratory setting: the space is larger than the DLS lab, the ceiling is higher, and there is more open volume with fewer nearby obstacles.

During mapping, Spot starts near one of the entrances and is driven along the main corridor of the hall. Using the Spot tablet and joystick, the operator first follows the long central aisle, then traces the perimeter of the accessible area, and finally crosses the centre again to close a loop. The total path is on the order of a few hundred metres, driven at walking speed.

As in the other environments, SLAM Toolbox runs online and fuses the 2D LiDAR scans on `/scan` with odometry from the Intel RealSense T265, which defines

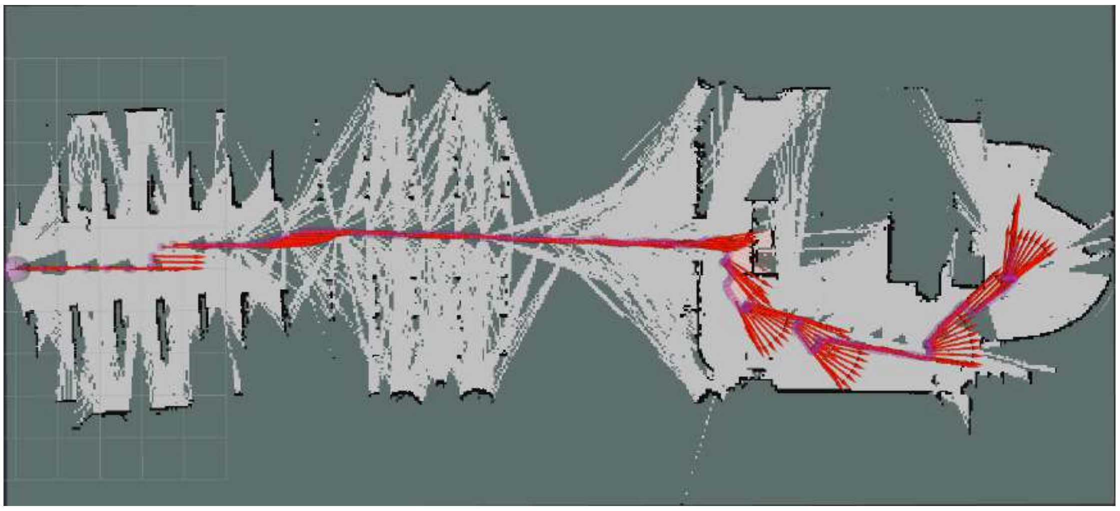

Figure 4.1: 2D occupancy grid of the church environment built with SLAM Toolbox, with a long navigation path overlaid. Grey cells show the map, magenta circles mark successive robot poses, and red arrows indicate local heading samples from the planner based on T265 odometry. The map captures the main rectangular hall and side areas as free regions separated by straight walls and obstacles.

the `odom` frame. The result is a 2D occupancy grid in the `map` frame together with a consistent transform chain `map` → `odom` → `base_link`. No AMCL is used in this experiment; localisation is entirely handled inside SLAM Toolbox.

Figure 4.1 shows an RViz snapshot of the final map and a long navigation run over it. The grey background is the occupancy grid built by SLAM Toolbox. The magenta circles mark a sequence of robot poses along a global path, and the red arrows are local heading samples from the planner and controller, expressed from the T265–based odometry.

Qualitatively, the map captures the main structure well. The long straight walls of the nave appear as straight segments without obvious double–wall artefacts, and the side areas are correctly aligned with the central hall. The entrance region and interior obstacles (columns, benches, and small furniture) show up where expected from the physical layout.

The overlaid path and arrows illustrate that localisation stays consistent over the full length of the hall. The T265–based poses follow the central corridor without visible drift into the walls, and the local headings remain aligned with the corridor direction except at deliberate turns near the far end. This suggests that, in this type of open indoor environment, the combination of T265 odometry and 2D LiDAR is sufficient to maintain a usable global pose for long straight runs.

In this thesis, the church map is mainly used as an example of mapping and long–range localisation outside the laboratory. We do not run the semantic layer

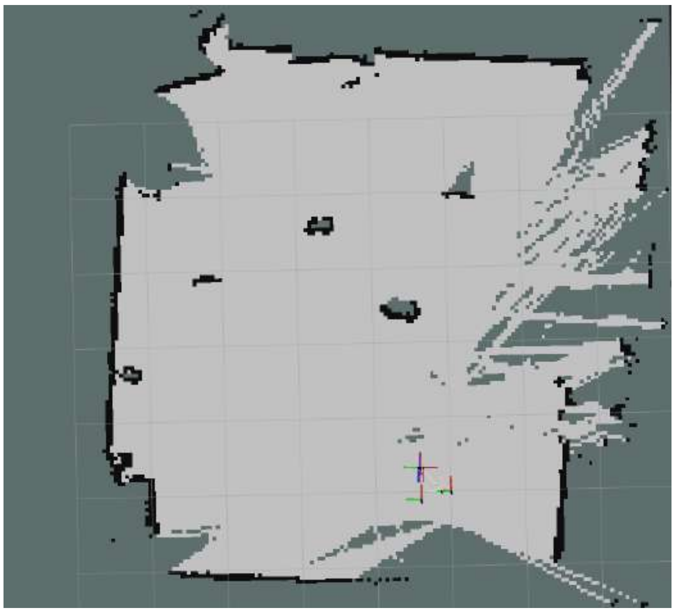

Figure 4.2: 2D occupancy grid of the main DLS lab space. The room boundary and large internal obstacles, such as desks and benches, are visible. This map is later reused for semantic mapping and navigation experiments in the lab.

or object–goal navigation here, but the experiment shows that the same SLAM Toolbox configuration that we use in the DLS lab and IIT room can also produce stable maps in a public indoor space with different geometry and lighting conditions.

4.1.3 DLS Lab Environment

The second environment is the main DLS laboratory space. This is a medium-sized room with desks, workbenches, shelves, and free floor area for experiments. Compared to the church, the geometry is more cluttered, with more small obstacles, but the overall footprint is smaller and more regular.

The mapping run starts near one corner of the lab. The robot is driven along the perimeter of the room, then through the central area between desks and equipment. The aim is to see the walls from different angles and to observe the main pieces of furniture at least once with the LiDAR. Because the environment is indoors and relatively controlled, the mapping session is shorter than in the church.

Figure 4.2 shows the occupancy grid produced by SLAM Toolbox.

The map shows a clear rectangular outline of the room. There are no apparent large-scale inconsistencies: walls close correctly, and corners align. Some minor details, such as chair legs or small boxes, are either blurred into larger occupied

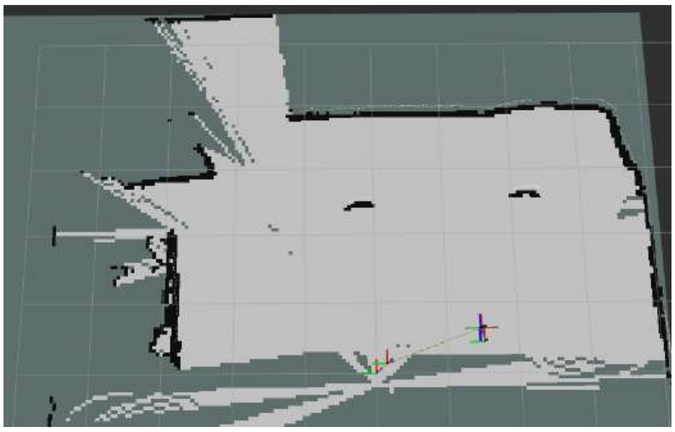

Figure 4.3: 2D occupancy grid of the mapped portion of the large IIT room. The open central area appears as free space, with occupied regions corresponding mainly to walls and two elements. This map is later used for semantic mapping and object–goal trials in a less constrained layout.

regions or disappear entirely, as expected at the chosen resolution.

This map is essential because it is the main testbed for later experiments: semantic mapping and object–goal navigation are both run in this lab environment. The quality of the 2D grid seen in Figure 4.2 is sufficient for that purpose. Nav2 can plan paths through the free areas, and the semantic layer can be aligned with the same map to store object locations.

4.1.4 Large IIT Room

The third environment is a large indoor room at IIT, generally used for robot testing. For our mapping run, the room was essentially empty: most furniture and equipment had been cleared, leaving the space mostly open. Compared to the DLS lab, there is less clutter and fewer small obstacles, but the overall footprint is larger.

The mapping procedure is again the same, but we only cover part of the environment. Starting from one side of the room, the operator drives the robot along a section of the perimeter, then across the interior in a loose pattern, without completing a full loop of the room. Because the space is open and empty, the LiDAR often sees long unobstructed ranges, which helps SLAM Toolbox align scans over distance, but there are fewer strong geometric features than in the lab. To compensate, we repeat some passes over the same area and monitor the live map in RViz to ensure the partial map remains stable.

Figure 4.3 shows the resulting occupancy grid for the mapped portion of the room.

The map shows a large free central area, with occupied regions around the desks and structural elements. As in the lab, small objects are either merged into larger blocks or filtered out by the map resolution and SLAM configuration. For navigation, this is acceptable: the main structure (walls, significant obstacles, free corridors) is cleanly captured, and Nav2 can use the map to navigate through the space while maintaining a safe distance from occupied cells.

This environment is used later to test how the semantic layer and object–goal navigation behave when the robot has more room to maneuver and when object locations are not constrained to narrow corridors.

4.1.5 Discussion

Across the three environments, the mapping behaviour of SLAM Toolbox with the T265 odometry and the 2D LiDAR is consistent. Using a single parameter set and the same driving procedure, we obtain occupancy grids that align well with the actual geometry for planning and attaching semantic information.

A few points are worth noting:

- The mapping pipeline does not rely on Spot's internal odometry. All maps are built from the RealSense T265 odometry and the LiDAR scans.
- We do not perform any post-processing or manual clean-up of the maps. The grids shown in Figures 4.1–4.3 are the direct outputs of SLAM Toolbox, saved via the standard map saver.
- Small artefacts exist (extra occupied cells near moving people, slight softening of sharp corners). Still, they do not prevent the planned use of the maps as a fixed backbone for semantic mapping and object–goal navigation.

From here on, we treat these 2D occupancy grids as given. The following sections place a lightweight semantic layer on top of the two occupancy grid maps and evaluate how well the system can record object locations and navigate to object–goal poses in these same environments.

4.2 Semantic Mapping in Real Environments

This section evaluates the semantic layer on top of the 2D maps produced by SLAM Toolbox. The SLAM node, RealSense driver, and `semantic_mapper` are launched together, and the robot is driven by the operator using the Spot tablet and joystick.

While SLAM Toolbox fuses the 2D LiDAR scans with odometry from the Intel RealSense T265 and maintains the transform between `map`, `odom`, and `base_link`, the semantic mapper subscribes to 3D detections that are already expressed in the `map` frame. Confirmed objects are saved to a small YAML database and later used for object–goal navigation.

In all runs, we restrict the classes to *chair* and *person*. The detector may output other classes, but the semantic mapper discards them early, so the static database stays compact and easy to inspect.

We report results from two environments:

- the DLS lab, where we place two chairs and two people in a cluttered lab space; and
- a large open test room at IIT, with two chairs, and in one run, an extra person.

For each environment, we show the final occupancy map with semantic markers, a photo of the real scene, and, for the lab, timing plots of the main streams.

4.2.1 DLS Lab: Two Chairs and Two People

Environment and setup. The first experiment takes place in the main DLS lab. Geometrically, this is a typical research lab: benches along the walls, robots and carts parked in different corners, shelves, cables, and various objects on the floor. For the semantic test, we focus on a smaller part of the lab to control the classes of interest.

Two office chairs are placed in front of a table, roughly one metre apart and facing the same direction. Two people stand near the chairs, at slightly different distances from the table. The rest of the background (monitors, computers, other robots) is left untouched. We do not try to simplify the scene; we want to see how the mapping and semantic stack behave in a realistically messy lab.

On the software side, we start:

- SLAM Toolbox in online asynchronous mode, subscribed to the 2D LiDAR topic `/scan` and to the T265 odometry, publishing the `map` frame and the transform `map` → `odom`;
- the RealSense driver, publishing RGB and aligned depth from the D435 and a tracking pose from the T265;
- the YOLO-based detector on the RGB stream, with outputs restricted to chairs and people. An example detector output from the robot's viewpoint is shown in Figure 4.4;

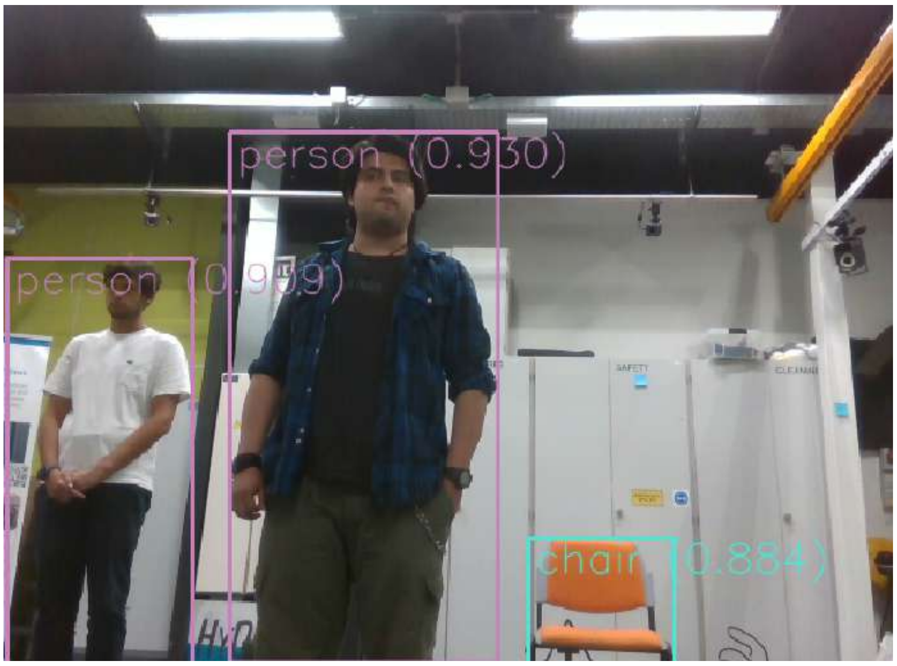

Figure 4.4: Example YOLO output from the robot's RGB camera during the DLS lab experiment. The detector runs on the D435 colour stream and overlays class labels and confidence scores, here identifying two *person* instances and one *chair*. This is the same scene used for the semantic mapping experiment in Figures 4.5 and 4.6.

- the 2D–3D projection node, which back-projects detections with depth and transforms them into `map`;
- the `semantic_mapper` and `semantic_recorder`; and
- RViz for live visualisation.

The robot is driven solely by a joystick and the Spot app; low-level locomotion and the internal Spot state estimator remain within the manufacturer's stack. From ROS 2 we only consume T265 odometry and the LiDAR scans.

Trajectories and mapping behaviour. The operator teleoperates Spot around the chairs and people in several loose loops. At first the robot does a wider loop to give SLAM Toolbox enough structure (walls and benches), then closer passes are made around the small cluster. Speed is kept moderate to avoid motion blur on the camera and to maintain smooth LiDAR scans.

As the robot moves, SLAM Toolbox incrementally fills the occupancy grid. The part of the lab we use for this experiment, which is roughly the area around the table and chairs, is covered multiple times. The final map for this experiment

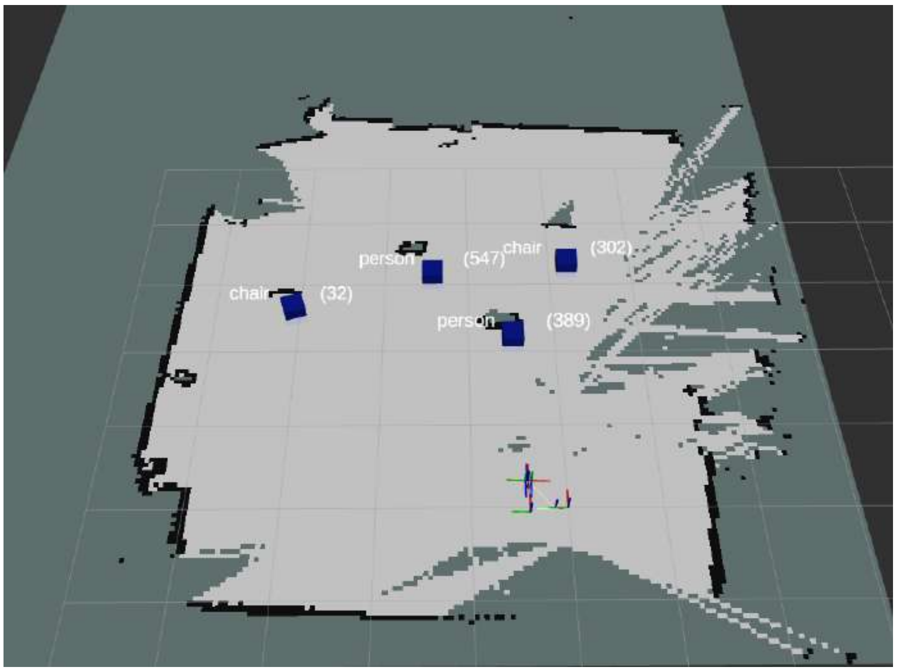

Figure 4.5: Occupancy grid of the DLS lab with the semantic layer overlaid. The two chairs and two people used in the experiment appear as four semantic markers anchored in the `map` frame, on top of the 2D LiDAR map.

is shown in Figure 4.5 (left): walls appear as thick continuous lines, benches and large objects show up as solid blocks, and free space around the chairs is clearly visible.

To make the correspondence clear, Figure 4.6 shows a photograph of the same area from a similar viewpoint. The two chairs are in front of the table, while the two people stand close to them. We rely on these photos only for qualitative inspection; they are not used by the system itself.

Evolution of the semantic map. During the first seconds of the run, the semantic layer is empty: the detector has not yet seen any chairs or people from a good angle, and the proposal buffer contains at most a few short-lived candidates. As Spot completes the first loop and passes in front of the cluster, the detector starts to output confident bounding boxes for the two chairs and both people.

For each detection, the projection node produces a 3D point in the D435 frame using the aligned depth image, then transforms it into `map`. These points are passed to `semantic_mapper` as a stream of `Detection3DArray` messages. Inside each callback, the mapper:

1. filters detections to keep only chairs and people, discarding other labels;

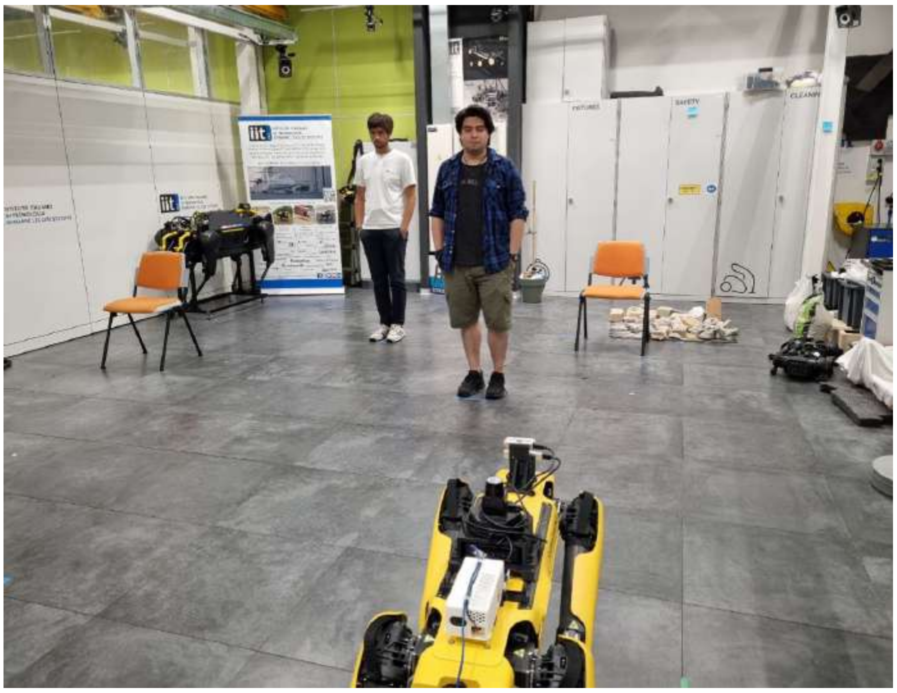

Figure 4.6: Real DLS lab scene used for the semantic mapping experiment. Two chairs are placed in front of a table and two people stand nearby. This layout corresponds to the semantic markers in Figure 4.5.

2. merges detections of the same class that lie within a small radius in the same frame (frame-level de-duplication);
3. looks for a nearby existing static object within the guard radius; if found, it simply updates that static's hit count and confidence; otherwise
4. matches or creates a proposal in the proposal buffer, updating its smoothed position, yaw, hit count, and score history.

Each proposal is temporary. Only when it has been seen enough times, within a fixed confirmation window and radius, and with an average score above the threshold, is it promoted to the static database. Promotion creates one entry in `static_db` and one cube+label marker in RViz, both anchored in `map`.

In the DLS lab run, this mechanism behaves as intended:

- Two chair proposals appear near the front of the table and quickly accumulate hits as the robot passes by from different angles. After several observations, they satisfy the confirmation conditions and are promoted. From this point onward, new chair detections near those positions update the existing static rather than creating additional objects.

- Two-person proposals follow a similar pattern, though with slightly more motion as the people occasionally shift their stance. The exponential update in the proposal state smooths this motion. Once confirmed, the person static remains anchored in roughly the correct places on the map.
- Occasional misdetections (for example, a monitor briefly tagged as "person" or a partial view of a chair) either fail the per-class score threshold or are not re-observed within the confirmation window. They die out from the proposal buffer without ever becoming static.

At the end of the run, the YAML database saved by `semantic_recorder` contains exactly four entries, matching what is visible in Figure 4.5: two chairs and two people. Their (x, y) positions on the map line up with the observed layout in Figure 4.6: chairs appear just in front of the table edge, and the people are slightly offset.

Runtime behaviour in the lab. To check that the semantic pipeline can run continuously together with SLAM on the NUC, we logged the rates of the main streams during this DLS lab experiment. Figure 4.7 shows four plots:

- Panel (A) reports the final average frame rate for four topics: camera RGB, aligned depth, the detector's debug image, and the semantic map output. The approximate averages are 21.9 Hz for RGB, 25.7 Hz for depth, 8.0 Hz for the debug image, and 14.1 Hz for the semantic map.
- Panel (B) shows the running average rate of the RGB and depth streams as a function of the window size. Both streams start higher and slowly settle; depth remains slightly faster than RGB because it does not go through the detector.
- Panel (C) tracks the detector debug image topic. The rate fluctuates more at the beginning and stabilises around 8 Hz.
- Panel (D) tracks the semantic map publication rate, which climbs quickly to about 13–14 Hz and then remains stable.

These numbers are enough for our purposes. The semantic map refreshes faster than the 10 Hz local controller used later for navigation, and none of the topics show long gaps or drops to zero that would indicate overload. Mapping and semantic processing comfortably share the CPU budget.

In this scene (two chairs, two people), we logged the semantic detections on `/semantic/detections_map` for about 70 s. The semantic map contained at least

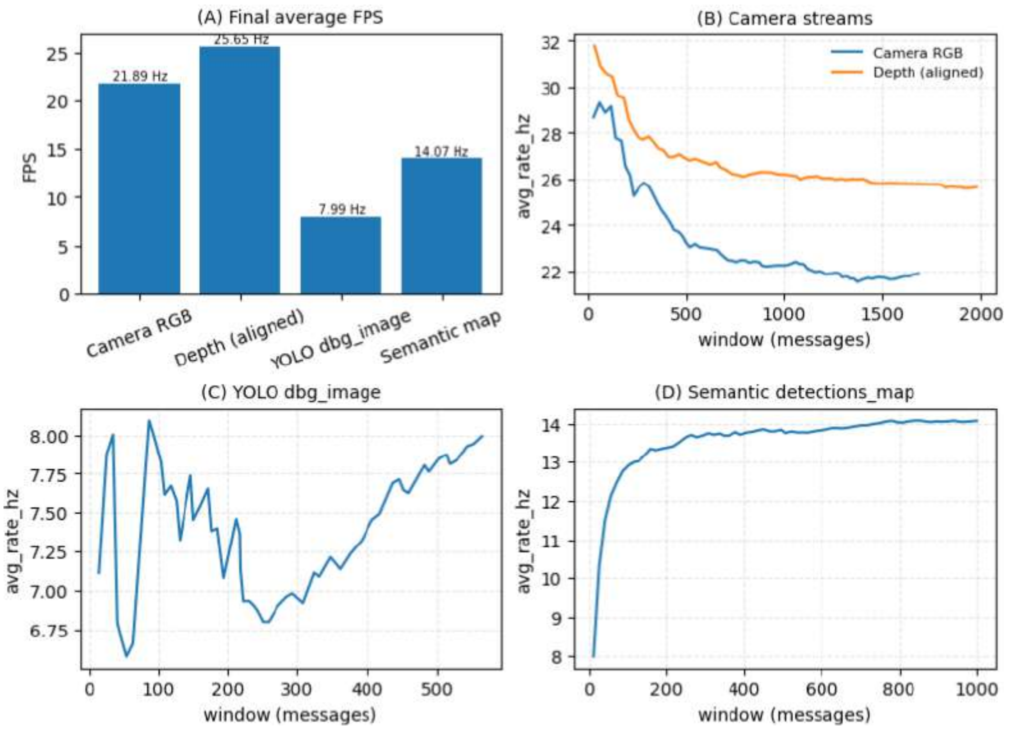

Figure 4.7: Runtime behaviour of the semantic stack during the DLS lab experiment. (A) Final average frame rate of the four main topics. (B) Running average frame rate for camera RGB and aligned depth. (C) Running average for the detector debug image. (D) Running average for the semantic map output (`/semantic/detections_map`).

one chair and at least one person in 99.6% of the recorded frames, with the longest gap without any detection of approximately 0.9 s. For people, both individuals were present in the map in the same fraction of frames. In practice, these short gaps correspond to moments when the robot turns or an object is briefly occluded. We did not observe spurious extra chairs or people in this sequence. These results suggest that, in this indoor setting, the overall detection–and–mapping pipeline is robust enough for our object–goal navigation experiments.

4.2.2 IIT Test Room: Two Chairs and One Person

The second environment is a large test room at IIT. Compared to the DLS lab, it is more open and less cluttered. Most heavy equipment and furniture are pushed to the sides, leaving a wide central area for experiments. This gives a different stress test for the semantic mapper: fewer strong geometric features, but long free corridors and clear views of the objects.

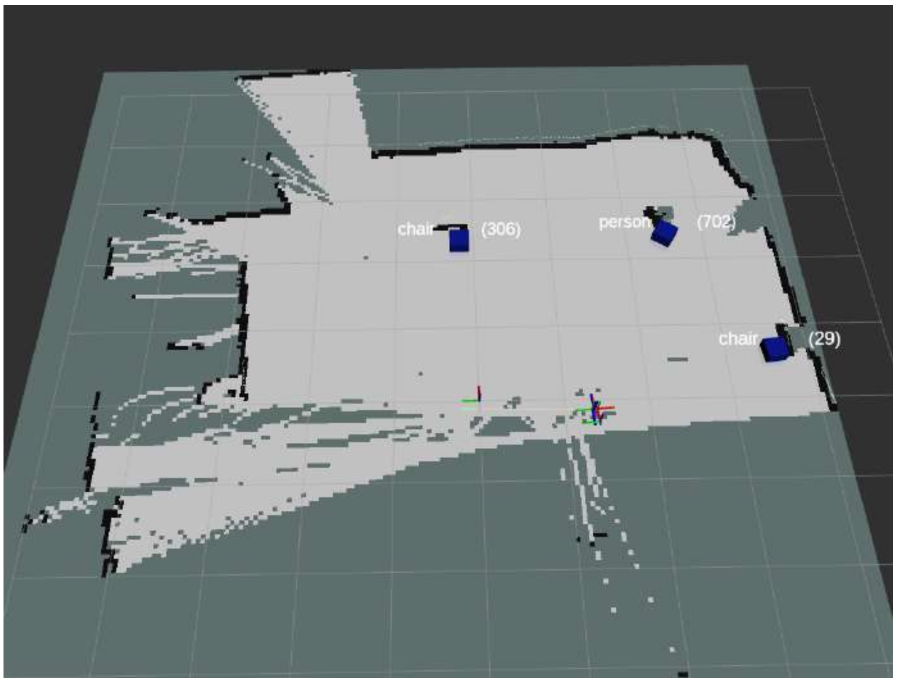

Figure 4.8: Occupancy grid of the IIT test room with semantic markers for two chairs and one person. The additional person instance appears next to the right–hand chair in the otherwise open central area.

Environment and trajectories. We use the same hardware and software configuration as in the lab: SLAM Toolbox, the RealSense driver, the detector, the projection node, the semantic mapper, and the recorder all run together while the operator drives the robot with the Spot tablet and joystick. The T265 odometry again defines the `odom` frame, the 2D LiDAR publishes `/scan`, and SLAM fuses them into the `map` frame.

For this experiment we place two identical chairs in the open part of the room, a few metres apart and facing roughly the same direction. One person stands close to the right–hand chair. The rest of the room is left as–is so that the background remains realistic.

The operator drives Spot in wide loops that cover the central area and, more densely, the region around the chairs and the person. Because the room is open, the LiDAR often sees long stretches of walls and corners, which helps SLAM Toolbox keep the map consistent even when the robot moves far from the objects. The resulting occupancy map, with the semantic layer overlaid, is shown in Figure 4.8: a large open space with obstacles mainly along the perimeter and three semantic markers in the centre.

To make the layout clear, Figure 4.9 shows a photograph of the corresponding

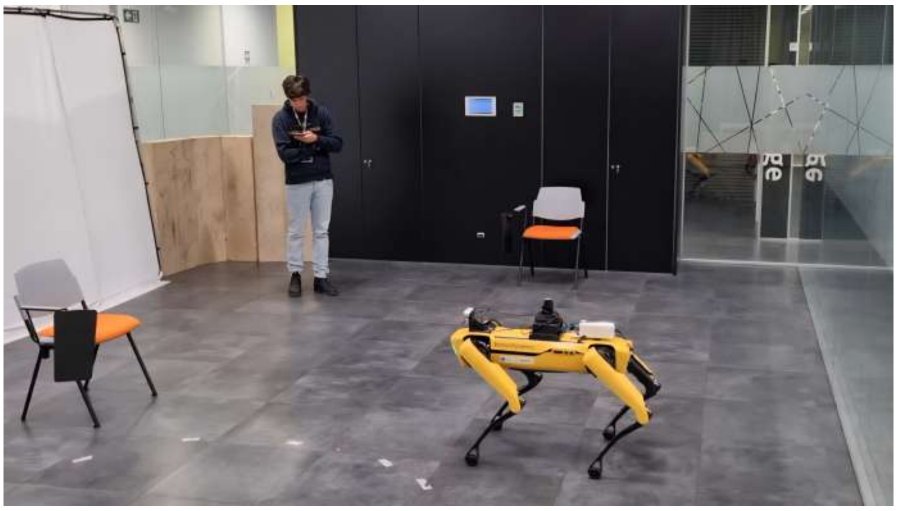

Figure 4.9: Real IIT test-room scene for the IIT experiment. Two chairs are placed in the central open area, and one person stands next to the right–hand chair, matching the semantic markers in Figure 4.8.

real scene. The two identical chairs are placed in the open area, and the person stands near the right–hand chair. We use this photo only for qualitative comparison; it is not part of the online pipeline.

Semantic layer behaviour. From the D435 point of view, the scene is simple: the chairs are well separated and appear against a mostly uniform background, and the person is clearly visible near one of them. As the robot loops around the cluster, the detector repeatedly finds both chairs and the person whenever they are in view. For each detection, the projection node back–projects the centre of the 3D bounding box using the aligned depth image and transforms the resulting point into the `map` frame. The `semantic_mapper` then applies the same confirmation logic as in the DLS lab experiment.

The internal behaviour is as follows:

- Detections of each chair form tight clusters in the proposal buffer. Because the chairs are static and the viewpoints are varied, their proposals reach the required number of hits very quickly and are promoted to static objects early in the run.
- Person detections are slightly more noisy in position and orientation, since the person occasionally shifts weight or moves an arm. The exponential smoothing in the proposal state absorbs most of this motion. After several consistent observations, a single person static is confirmed next to the right–hand chair.

- When the detector briefly confuses parts of the person and chair (for example, a leg of the chair tagged as "person" or a piece of clothing tagged as "chair"), those boxes fall within the guard radius of an existing static instance and only update its statistics instead of creating a new object. This prevents the map from filling with near–duplicate objects.

By the end of the run, the YAML database saved by `semantic_recorder` contains exactly three entries: two chairs and one person. Their (x, y) positions align with the layout seen in Figure 4.9. In RViz, the three markers in Figure 4.8 appear in the open area of the map, at distances and relative angles that are consistent with the physical scene. The residual error is small compared to the standoff distance that we later use for navigation.

One limitation of the current design is also visible here. Once the person leaves the room and we reload the semantic database along with the static map, the person instance remains. The semantic layer, by design, does not automatically remove statics when objects disappear; it assumes that once an object is confirmed it is part of the long–term environment. This is appropriate for furniture and doors, but dynamic classes such as "person" should be used with care when building maps intended for repeated reuse.

Summary of semantic mapping results. Across both environments, with very different layouts, the behaviour of the semantic layer is consistent:

- The combination of SLAM Toolbox, T265 odometry, 2D LiDAR, and the RealSense cameras provides enough stability to run semantic mapping in parallel with geometric mapping.
- The confirmed-only design keeps the semantic database small: in the DLS lab we end up with two chair and two person instances; in the IIT room we obtain two chairs and one person, matching the real scene.
- False detections are mostly filtered out by the confirmation logic. Only objects that are re-observed at similar positions over a short time are stored.
- The runtime of the semantic stack, measured in the DLS lab, stays comfortably within real-time bounds on the CPU-only NUC.

These semantic maps are the starting point for the next part of the evaluation, where we use them to drive object–goal navigation with Nav2.

4.3 Semantic Object–Goal Navigation in Real Environments

In the previous sections, we described the whole pipeline: a 2D occupancy grid built with SLAM Toolbox, a confirmed-only semantic layer aligned to that grid, and an object–goal interface that turns a selected instance into a navigation goal for Nav2. Here we test this pipeline in real environments, using only saved maps and object poses. SLAM is not running during these experiments; the system reloads a previously built map and semantic database, localises with AMCL, and relies on Nav2 for planning and control.

We report three real-world experiments on two different maps. Two of them reuse the same IIT test room with different object layouts (four chairs; two chairs and one person). The third experiment uses a second map with a different geometry and object placement. In all cases, the user selects targets at the semantic level, for example, by asking for a specific chair index through a voice interface, and the system converts that choice into a PoseStamped goal in the map frame.

The following subsections describe each experiment in detail, starting from the four-chair scenario in the IIT test room.

4.3.1 Object–Goal Navigation with Four Chairs

For the first object–goal navigation experiment, we use the same large test room at IIT that was used in the mapping trials, but we rearrange the furniture so that four identical chairs are placed in the central open area. The rest of the room remains empty primarily, with walls and heavier furniture concentrated near the perimeter. The intention here is not to fill the room with obstacles but to create several similar targets in a relatively uncluttered space, and then test whether the system can reliably visit them one after another using only the saved map and object poses.

The underlying 2D occupancy grid is generated with the SLAM Toolbox using the same configuration as in the previous mapping experiments. Once a satisfactory map is obtained, we save it to disk and stop SLAM. For this object–goal experiment, we do not attempt to remap the whole room: we reload the stored occupancy grid and the semantic database and use them as a static world model for navigation. In this way, the test shows that, once the semantic pass has been completed, the system can be restarted in a navigation mode that depends only on the saved map and the list of objects.

Figure 4.10 shows the occupancy grid together with the confirmed semantic layer. Each chair appears as a blue cube in RViz, positioned at the estimated 3D centre of the seat and projected into the map frame. The text overlay next

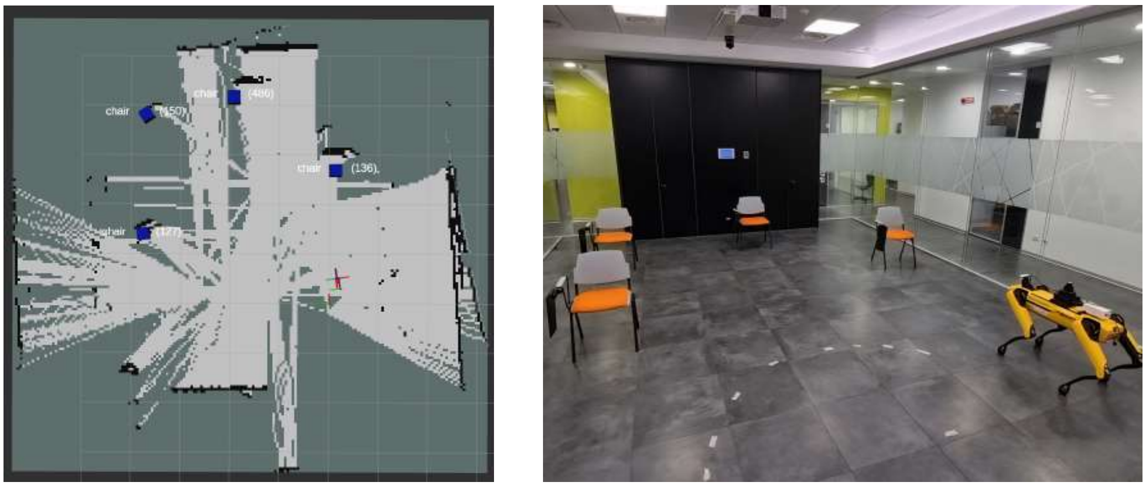

Figure 4.10: Semantic map of the IIT test room used for the four–chair experiment (left) and the corresponding real environment (right). The grey cells represent free and occupied space in the 2D occupancy grid, while the photo shows the actual placement of the four chairs in the central area of the room.

Table 4.1: Positions of the four chairs in the map frame used for the four–chair experiment. Coordinates are expressed in metres and rounded to two decimal places.

Chair	x [m]	y [m]
Chair 1	1.68	-0.04
Chair 2	3.65	1.38
Chair 3	3.39	2.68
Chair 4	0.70	2.74

to each cube shows the class name and the number of detections collected during the semantic pass. No dynamic objects are present in this experiment; the only semantic instances in the room are the four chairs.

To make the geometry of the setup explicit, Table 4.1 reports the map–frame coordinates of the four chairs as exported by the semantic mapper. Values are rounded to two decimal places for readability.

In addition to the frame sequence, we also visualise the robot trajectory in the map frame using the pose estimates from `/amcl_pose`. During the experiment we log this topic and, offline, we plot the (x, y) positions together with the chair centres and a circle of radius 0.30 m around each chair, corresponding to the chosen standoff distance. The resulting trajectory is shown in Figure 4.11: the robot starts near the origin, passes through the standoff regions around chairs 2 and 3, and finally reaches the region around chair 1.

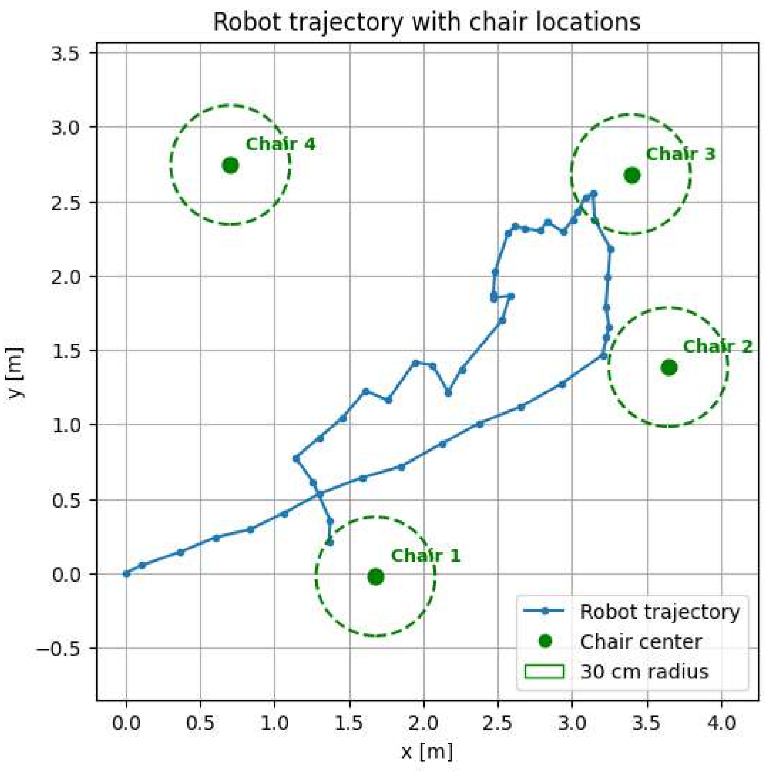

Figure 4.11: Robot trajectory for the four–chair experiment, reconstructed from the `/amcl_pose` estimates. The blue line shows the robot path in the map frame. Green dots mark the chair, and dashed circles of radius 0.30 m indicate the standoff region around each chair.

Object ordering and interface. During the semantic mapping pass, the `semantic_mapper` node records each confirmed instance to a YAML file in the order in which the object was first confirmed. When we later reload this database for navigation, the object–selection CLI assigns a simple index to every entry in the file: "chair 1", "chair 2", "chair 3", and "chair 4". The system does not introduce any additional notion of importance or distance; the ordering is exactly the confirmation order.

For this experiment, we connect the object–selection CLI to the speech interface used elsewhere in the system. The user can therefore issue voice commands such as "go to the second chair" or "now go to the third chair", which are transcribed into text and parsed into the corresponding menu choices. In this way, the user never interacts directly with coordinates or pose messages: they only refer to the chairs by their index.

Goal generation and stopping distance. When a chair is selected, the CLI looks up its stored pose in the map frame and combines it with the current robot pose from TF. A standoff–and–facing goal is then computed along the line connecting the robot to the object. For the four–chair experiment, we chose a standoff distance of

$$d_{\text{off}} = 0.30 \text{ m},$$

so the navigation goal is placed thirty centimetres in front of the chair, and the yaw is chosen so that the robot faces the seat. The resulting pose is sent to Nav2 as a standard `NavigateToPose` action goal; from this point on, the usual global planner, local planner, and costmaps take over. No additional tuning of Nav2 is explicitly introduced for object–goal navigation.

Experiment protocol. The robot starts from an initial pose close to one side of the mapped area. After the system is brought up in navigation mode (map server, AMCL, costmaps, BT navigator, and the semantic services), the following sequence is executed:

1. The user issues a voice command: "go to the second chair".
2. The speech recogniser converts the audio into text and passes the parsed index (2) to the object–selection CLI.
3. The CLI looks up "chair 2" in the semantic database, computes the corresponding standoff–and–facing pose, and sends it as a goal to Nav2.
4. Nav2 plans and executes the path. The robot starts moving immediately and stops when it reaches the neighbourhood of the goal, that is, roughly 30 cm in front of chair 2.
5. Without moving the robot manually, the user then asks for "the third chair", and the same process is repeated for chair 3.
6. Finally, the user asks for "the first chair", and the robot navigates from chair 3 to chair 1 using the same pipeline.

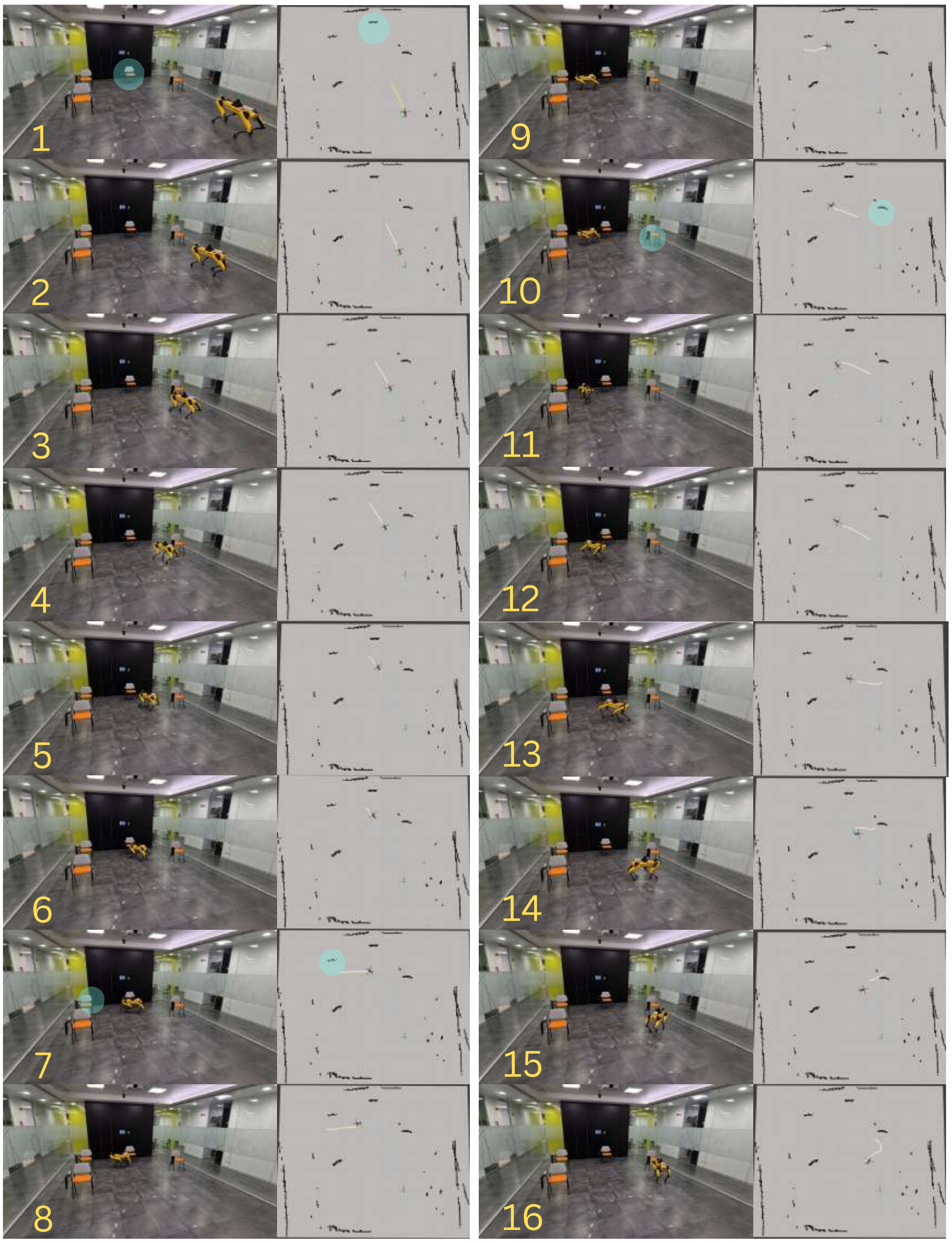

Figure 4.12: Sequence of sixteen frames from a representative four–chair run. Each row shows a pair of images: on the left, a view of the real scene in the IIT test room; on the right, the RViz view of the occupancy grid with the robot pose, the global plan, and the local trajectory.

At no point during this sequence is teleoperation used. All motions from one chair to the next are driven by the object–goal pipeline and the Nav2 stack. The behavior is easier to follow in the numbered frames of Figure 4.12. Frames 1–6 show the robot going to chair 2; frames 7–9 correspond to the command for chair 3; and frames 10–16 show the final motion towards chair 1. In frames 1, 7, and 10, we highlight the currently selected target chair with a light blue overlay in both the real image and the RViz view; these three frames mark the moments when a new object–goal is issued, and the previous goal has been reached.

Qualitative behavior. Figure 4.12 illustrates how the robot moves between the chairs. In each row, the left image shows the physical robot in the room, while the right image shows the corresponding state of RViz. The global planner produces a path from the current robot pose to the generated standoff pose, and the local planner refines it into a smooth trajectory that keeps a safety margin from walls and other obstacles.

When the user first commands the robot to go to the second chair, the planned path bends through the central free space, avoiding the chairs that are not the current target. As the robot approaches the goal, the path shortens and aligns with the chair so that the robot arrives facing the seat and stops once the goal tolerance is satisfied. The subsequent transitions to chair 3 and chair 1 follow the same pattern: a new standoff goal is computed from the stored object pose, Nav2 replans, and the robot follows the updated path without manual intervention.

Discussion. Even though the four–chair setup is simple, it stresses several important aspects of the system:

- It shows that once an initial semantic mapping run has been completed, navigation only needs a lightweight representation: a fixed 2D occupancy grid plus a list of object poses. There is no need to maintain an online semantic map during navigation.
- Indexing objects by confirmation order and exposing that index to the user through the interface makes behavior easy to explain. When the user asks for the "second" or "third" chair, there is a clear, deterministic mapping from that command to one of the instances in Figure 4.10.
- The experiment also illustrates how the standoff parameter can be used as a simple but effective way to encode task constraints. In our case, a fixed 30 cm offset is enough to keep a comfortable distance from the chairs while still bringing the robot close enough for interaction.

Overall, the four–chair scenario confirms that the proposed pipeline can turn a small, static semantic database into a usable interface for object–goal navigation. The next experiments build on this by changing the set of objects and the room configuration while keeping the same basic interaction pattern.

4.3.2 Object–Goal Navigation with Two Chairs and One Person

We now reuse the semantic map from Section 4.2.2, where two identical chairs and one person are present in the central area of the IIT test room. As before, we load only the saved occupancy grid and the semantic YAML file: SLAM is not running, and navigation relies on AMCL for localization and on Nav2 for planning and control. The three semantic instances from the mapping experiment (two chairs and one person) are taken as potential object goals.

In this scenario we use the `nav_cli` interface directly, without the speech wrapper. The operator selects one of the semantic instances by index and issues the corresponding object–goal command from the terminal. The standoff distance is kept equal to the previous experiment, $d_{\text{off}} = 0.30\,\text{m}$.

Table 4.2 reports the map–frame centres of chair 1, chair 2, and the person as exported by the semantic mapper. Coordinates are expressed in metres and rounded to two decimal places for readability.

Table 4.2: Positions of the two chairs and the person in the map frame for the two–chairs–and–person experiment. Coordinates are expressed in metres and rounded to two decimal places.

Object	x [m]	y [m]
Chair 1	2.09	0.07
Chair 2	0.95	-2.68
Person	2.16	-2.03

During the experiment we log the robot pose from `/amcl_pose` while issuing three object goals in sequence: first the person, then chair 1, and finally chair 2. Offline, we plot the (x, y) positions together with the object centres and a circle of radius 0.30 m around each object, matching the chosen standoff distance. The resulting trajectory is shown in Figure 4.13.

The plot highlights a limitation of the current stack. The first and third goals behave as expected: the trajectory enters the standoff circles around the person and chair 2, showing that the object–goal interface, Nav2, and the costmaps are able to drive the robot into the desired neighbourhood. However, for the second goal (chair 1) the robot does not reach the corresponding 30 cm circle. Instead,

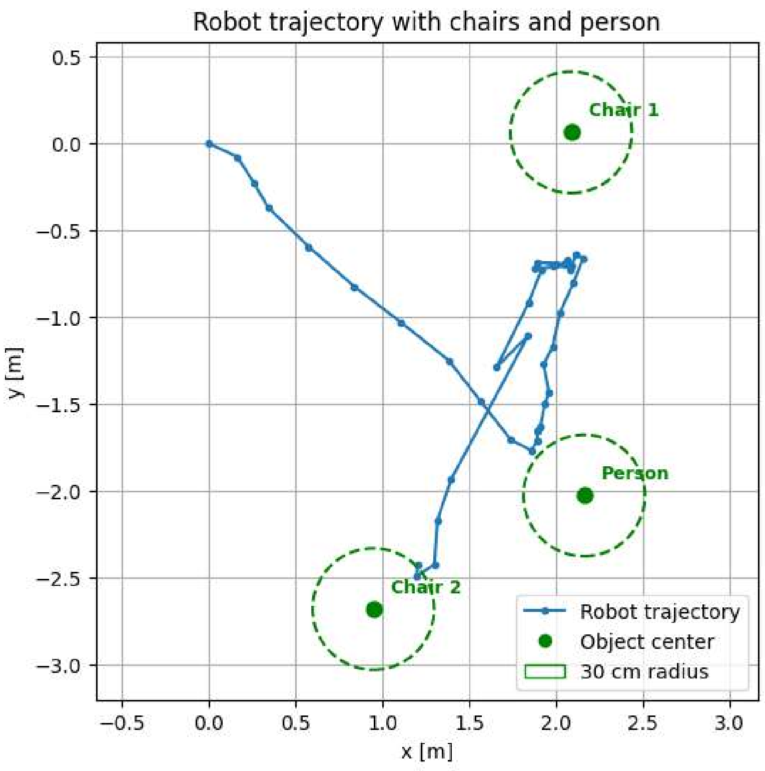

Figure 4.13: Robot trajectory for the IIT test room with two chairs and one person, reconstructed from the `/amcl_pose` estimates. The blue line shows the robot's path in the map frame. Green dots mark the centres of chair 1, chair 2, and the person, and dashed circles of radius 0.30 m indicate the standoff region around each object.

the path bends in the general direction of chair 1 but stops short of the target region.

Inspection of the logs indicates that this failure is not due to the semantic layer or the goal generation, but to inconsistencies between the Spot driver and the pose estimate from `/amcl_pose`. In this run, AMCL maintains a biased estimate of the robot pose, and the driver reports an offset between the internal odometry and the pose used for planning. As a result, Nav2 believes that the robot is closer to chair 1 than it actually is and prematurely considers the goal satisfied.

When we subsequently send a goal to chair 2 using the same `nav_cli` interface, the robot successfully reaches its standoff region despite the earlier error. Figure 4.13 thus summarises the scenario: all three objects are available as semantic goals, the person and chair 2 are reached, but chair 1 is missed because of a localization problem rather than a failure in the semantic navigation pipeline itself.

Chapter 5

Conclusions and Future Work

This thesis has presented a modular pipeline for semantic object–goal navigation on a quadruped robot in known environments. The system follows a two-stage strategy: first, building a 2D map of the environment, and then reusing that map, together with a semantic layer, for navigation. A 2D occupancy grid of the environment is built with SLAM Toolbox. On top of this, a semantic layer is constructed online from RGB–D detections, producing a small database of object instances anchored in the map frame. Finally, an object–goal interface converts a selected instance into a navigation goal for Nav2.

All components run on a ROS 2 stack on Spot, using standard packages where possible (SLAM Toolbox, Nav2, AMCL, RealSense drivers) and custom nodes where necessary (projection, semantic mapper, recorder, object–goal CLI and voice interface, safety supervisor). The pipeline was evaluated in three real environments: a church, the DLS lab, and a large IIT test room with different object layouts. The experiments covered both mapping and semantic mapping, and three object–goal navigation scenarios in the IIT room (four chairs; two chairs and one person; a second map with a different layout).

The primary outcome is not a new SLAM or detection algorithm, but the integration of existing tools into a workflow that a human operator can actually use: build a map once, record a small set of semantic objects, and later come back and ask the robot to “go to chair 2” or “go to the printer” instead of clicking poses on an occupancy grid.

5.1 Summary of Contributions

The work provides three main contributions:

- A **confirmed–only semantic mapping layer** that takes 3D detections from an RGB–D camera, projects them into the map frame, and maintains a compact database of static object instances. The layer is decoupled from SLAM: it assumes an existing 2D map and uses only pose estimates to anchor objects.

- A **semantic object–goal interface** that runs on top of Nav2 and turns a selected object instance into a PoseStamped goal in the map frame. The interface supports a standoff distance and facing constraint, and can be driven either from a terminal (`nav_cli`) or from a simple speech wrapper.
- A set of **real-world experiments** on Spot that demonstrate the full pipeline working end to end: maps built in large indoor spaces, semantic databases extracted from real runs, and object–goal navigation between multiple instances, including failure cases where localization issues prevent a goal from being reached.

All components are deliberately modular. The semantic mapper does not depend on Spot; it only expects poses, depth images, and detections. The navigation layer assumes a Nav2-compatible robot base and a localization source. This modularity is important for future reuse across different robots and environments.

5.2 Observed Limitations

The experiments also make apparent several limitations of the current design.

Depth and sensor alignment

The most visible limitation on the semantic side is **depth misalignment**. The D435 depth image is not perfectly aligned with the 2D LiDAR scan plane used by SLAM Toolbox to build the map. As a result, the 3D points obtained from the detector centres and projected into the map frame carry a small but systematic bias in range and sometimes in angle. This bias shows up as chairs and other objects appearing slightly closer or further away than they are in the occupancy grid.

In many of our runs this error remained within the 30 cm standoff radius and did not prevent navigation, but it reduces the precision with which objects can be placed in the map. For tight manoeuvres or smaller standoff distances, this misalignment would become problematic.

Object representation and association

The current semantic layer makes two simplifying assumptions:

- Each detection is represented only by the **centre of its 3D bounding box**. The system does not explicitly model object extent or orientation, nor does it use keypoints or shape information. For chairs and similar furniture, this is usually sufficient, but it ignores the functional structure that could make the pose estimate more robust.
- Association between detections and existing objects is **purely geometric**. A new detection is matched to an object if it falls within a fixed radius in the map frame. There is no appearance model, no tracking across frames, and no re-identification. This makes the implementation efficient and straightforward, but it also means that closely spaced objects of the same class may be more challenging to separate reliably.

These design choices were reasonable for the target environments and for a lightweight system, but they limit how far the approach can be pushed without additional cues.

Localization and navigation issues

On the navigation side, the main limitation observed in the experiments is **localisation quality**. In most runs, AMCL combined with the Spot driver provided sufficiently accurate poses for Nav2 to reach the standoff regions around the target objects. However, in the two–chairs–and–person scenario, one of the goals (chair 1) was not reached, even though the semantic goal was generated correctly.

The trajectory plot for this experiment shows that the robot moves in the general direction of chair 1 but stops outside the 30 cm standoff circle. Logs indicate a mismatch between the pose used by Nav2 (`/amcl_pose`) and the internal odometry reported by the Spot driver. In effect, the navigation stack believed the robot was closer to the goal than it actually was and prematurely considered the goal satisfied.

This failure underlines an important point: even with a correct semantic layer and a valid goal, the overall behavior is only as good as the localization estimate. Robust semantic navigation requires tighter integration and monitoring of the localization component, and perhaps independent sanity checks of goal satisfaction (e.g., verifying the expected relative pose of the target object in the camera image).

Evaluation scope and environment type

Another limitation is the **scope of the evaluation**. All mapping and navigation experiments were performed indoors: a church, a lab, and a large test room. These settings are practical and realistic for many service tasks, but they do not cover outdoor scenes, multi-floor buildings, or environments with significant elevation changes and rough terrain.

On the semantic side, all test objects were everyday furniture or people. The pipeline should extend to other classes supported by the detector, but this was not validated in the current work. Similarly, dynamic crowds or heavy pedestrian traffic were not considered.

Static–world assumption

The semantic layer is designed for a **static world**. Once an object is confirmed and written to the YAML database, it is treated as part of the long-term map. This fits well for doors, cabinets, printers, and other relatively static assets, but it is less appropriate for dynamic classes such as “person” or “bag”. In our experiments, the person instance remains in the semantic map even after the person leaves the room.

For navigation, this means that the robot could be asked to “go to the person” even when no one is actually standing there. In practice, this is more of a user-interface issue than a catastrophic failure, but it highlights the need for strategies to handle dynamic or short-lived objects.

5.3 Future Work

Several directions could address these limitations and enhance the system’s usefulness.

Improved depth and sensor calibration

The first step towards more accurate object placement is a better alignment between the RGB–D camera and the LiDAR. A dedicated calibration procedure that estimates the relative pose between the D435 depth frame and the laser frame could reduce the range bias observed in the experiments. Another option is to fuse depth and LiDAR information explicitly in the mapping stage, so that semantic points are anchored in a representation that accounts for both modalities.

Within the semantic mapper, representing objects by more than just their centers (e.g., storing approximate size and orientation, or using a small set of keypoints) could make pose estimates less sensitive to depth noise.

Richer association and appearance cues

To move beyond purely geometric association, future versions of the semantic layer could incorporate simple *appearance cues*. Examples include:

- tracking instance IDs from the detector, where available;
- maintaining per–object feature descriptors extracted from the RGB image;
- using a small tracker to follow detections across frames before promoting them to static.

These additions would increase complexity and compute load, but they could help separate objects that are close together and make the system more robust in cluttered scenes.

Stronger localization and safety checks

The localization failure observed in the two–chairs–and–person scenario suggests two lines of improvement:

- Tighter integration between Nav2, AMCL, and the robot's internal odometry, including monitors that detect sudden inconsistencies and trigger recovery behaviors (e.g., reinitialising AMCL or asking the operator for a new initial pose).
- Independent *goal verification* at the semantic level. When Nav2 reports that the goal has been reached, the system could check whether the target object appears in the expected region of the camera image and at roughly the expected distance. If not, the goal could be considered unsatisfied, and a small local search could be attempted.

For robots that share space with people, a safety layer that monitors both localization quality and proximity to semantic objects (such as doors and stairs) would also be important.

Dynamic and changing environments

Handling dynamic worlds requires changes in both the semantic and planning layers. On the semantic side, dynamic classes could be stored with a finite lifetime or a confidence that decays over time, so that people and other short-term objects eventually disappear from the map unless they are seen again. Another possibility is to maintain two layers: a long-term static layer for furniture and structure, and a short-term layer for dynamic objects used only for the current session.

On the planning side, integrating online perception during navigation, for example, temporary obstacles detected by the camera or LiDAR, would allow the robot to react when chairs are moved or people step into the path without having to rebuild the entire semantic map.

Broader evaluation and different robots

Because the pipeline is modular and built on ROS 2, it should be possible to deploy it on other platforms with minimal changes: wheeled bases, other quadrupeds, or even mobile manipulators, as long as they provide a TF tree, a localization source, and a Nav2-compatible controller. Testing the same semantic layer and object–goal interface on different robots would help separate robot-specific issues (such as the Spot driver behavior) from limitations of the mapping and navigation pipeline itself.

Similarly, running the system in a broader range of environments, such as offices, corridors, multi-room floors, or semi-structured outdoor spaces, would give a more complete picture of its robustness and practical utility. That would involve longer maps, more diverse object classes, and scenarios with more people moving around.

Towards task–level interaction

Finally, the current interface stops at object–goal navigation: the robot can be sent "near" a chair or a printer, but it does not understand tasks beyond reaching a standoff pose. A natural extension is to couple semantic navigation with higher-level behaviors, such as:

- patrolling a list of semantic locations;
- inspecting or scanning a region around a given object;
- handing over or picking up items near a specific piece of furniture.

Because the pipeline already exposes a compact semantic database and a simple goal API, these behaviors could be built on top without changing the underlying mapping and navigation components.

5.4 Closing Remarks

The work in this thesis shows that a relatively lightweight, modular stack can bring semantic object–goal navigation to a real quadruped robot in everyday indoor environments. The system is far from perfect: depth misalignment, simple object models, localization issues, and static world assumptions all limit what it can do. At the same time, the experiments demonstrate that once a map and a small semantic database exist, a robot like Spot can be driven by semantic commands rather than solely by coordinates.

The hope is that this kind of pre-mapped semantic navigation pipeline can serve as a practical foundation: something that is easy to deploy, transparent enough for operators to understand, and flexible enough to be reused on different robots and in various places, while leaving room for richer perception, stronger localization, and more complex tasks in future work.

Appendix A

Extra

A.1 Launch Setup and Modes

For mapping and navigation we use a single ROS 2 launch description, with a `mode` argument that can be set to `mapping` or `navigation`. This avoids maintaining separate launch files and keeps the configuration aligned between the two phases.

Some nodes are always running, independently of the mode:

- RealSense T265 node publishing pose and the `odom` transform (configured with `base_link` as base frame and `odom` as odometry frame);
- LiDAR driver;
- static TF publishers between `t265_pose`, `base_link` and `laser`;
- RViz, preconfigured to show map, robot, and laser scans.

In *mapping* mode the launch file starts SLAM Toolbox with our mapping configuration. Nav2 is not launched. The operator teleoperates the robot and SLAM Toolbox incrementally builds the 2D occupancy grid, which is later saved to disk.

In *navigation* mode SLAM Toolbox is disabled and the Nav2 stack is brought up: planner and controller servers, behaviour-tree navigator, behaviour server, map server, AMCL, and lifecycle manager. The map server loads the selected occupancy grid, and all Nav2 nodes share a common parameter file for planners, controllers, costmaps, and recoveries.

This single-launch design keeps the workflow simple: each environment is mapped once in mapping mode, the resulting grid is saved, and all experiments in that environment are then run in navigation mode using exactly the same frames and sensor configuration.

A.2 Map Storage and Management

All 2D maps are stored under the `mapping_docker/maps` directory. Each mapping run produces one occupancy-grid pair (image and YAML), typically with a name that encodes the environment (for example, `lab1`, `corridor`, `church`).

For experiments we usually select a single map per environment and normalise its name to `final_map.yaml`. The launch file always points to this name, so switching maps is done by changing which YAML file is copied or symlinked to it.

The YAML files follow the standard Nav2 format, specifying:

- path to the occupancy-grid image file;
- resolution (0.05 m in our experiments);
- origin of the map in world coordinates;
- occupancy thresholds for free and occupied cells.

After each mapping run we apply a small quality-check routine:

- load the map via `map_server` and verify alignment with live LiDAR scans;
- drive the robot along representative paths (e.g., the main corridor) and check that AMCL remains stable;
- visually confirm that walls, doors, and large obstacles have the correct shape and position.

Maps that do not pass these checks are remade, sometimes with slightly adjusted SLAM parameters or a slower teleoperation pace. We keep a small history of maps for each environment, with timestamps in the filenames, so later changes in the scene (moved furniture, new equipment) can be tracked and older maps can still be reused in simulation or for semantic-map visualisation.

A consistent directory structure and naming convention also helps other members of the lab reproduce our experiments: as long as they use the same `mapping_docker` package and `maps` directory, they can launch mapping or navigation without touching hard-coded paths.

A.3 Nav2 Configuration Summary

This section summarises the main Nav2 settings used in navigation mode. The corresponding parameters are stored in a single `nav2_params.yaml` file.

Behaviour tree and planner/controller.

- Behaviour tree navigator runs a standard navigation tree with replanning and recoveries enabled.
- Global planner is `NavfnPlanner` configured to use A* and to allow unknown cells in the map, with a goal tolerance of about 0.5 m.
- Local planner is the DWB local planner, with linear velocity limited to about 0.4 m/s and angular velocity to about 1.0 rad/s.
- Controller frequency is set to 10 Hz, with a simple progress checker (radius 0.5 m and time allowance 10 s) and a relatively loose goal checker (position tolerance around 0.6 m and yaw tolerance close to2π rad). These values favour robustness and smooth behaviour over very precise docking.

Costmaps and obstacles.

- The global costmap is defined in the `map` frame, aligned with the static occupancy grid and updated at about 1 Hz.
- The local costmap is defined in the `odom` frame, rolling with the robot, and updated at about 5 Hz.
- Both costmaps include an obstacle layer driven by the 2D LiDAR scan (`/scan`) and an inflation layer to maintain a safety margin around obstacles.
- Typical inflation radii are around 0.7 m for the local costmap and 0.4 m for the global one, with cost-scaling factors tuned to keep paths slightly away from walls and furniture.

Recoveries and AMCL.

- A recovery server provides spin, backup, and wait behaviours that can be triggered when the planner or controller stalls.
- AMCL uses the LiDAR scan topic and the T265-based odometry, with particle counts in the range 500–2000. The global frame is `map`, the odometry frame is `odom`, and the base frame is `base_link`.
- Initial pose is typically seeded near the expected start location of the robot, which speeds up convergence.

In summary, Nav2 is used in a conventional 2D configuration; the thesis-specific elements are the semantic mapping pipeline that produces object poses in `map` and the client that converts those poses into goals for Nav2.

References

Aditya, D., Homberger, T., Juraszek, J., Stachniss, C., Hutter, M. & Grynnerup, A. (2025). Robust localization and mapping for low-cost quadruped robots in indoor environments. *arXiv*. 4, 9, 10, 11, 12, 13, 14, 26

Agha, A., Otsu, K., Morrell, B., Fan, D.D., Thakker, R., Santamaria-Navarro, A., Kim, S.K., Bouman, A., Lei, X., Edlund, J., Ginting, M.F., Ebadi, K., Anderson, M., Pailevanian, T., Terry, E., Wolf, M., Tagliabue, A., Vaquero, T.S., Palieri, M., Tepsuporn, S., Chang, Y., Kalantari, A., Chavez, F., Lopez, B., Funabiki, N., Miles, G., Touma, T., Buscicchio, A., Tordesillas, J., Alatur, N., Nash, J., Walsh, W., Jung, S., Lee, H., Kanellakis, C., Mayo, J., Harper, S., Kaufmann, M., Dixit, A., Correa, G., Lee, C., Gao, J., Merewether, G., Maldonado-Contreras, J., Salhotra, G., Silva, M.S.D., Ramtoula, B., Kubo, Y., Fakoorian, S., Hatteland, A., Kim, T., Bartlett, T., Stephens, A., Kim, L., Bergh, C., Heiden, E., Lew, T., Cauligi, A., Heywood, T., Kramer, A., Leopold, H.A., Choi, C., Daftry, S., Toupet, O., Wee, I., Thakur, A., Feras, M., Beltrame, G., Nikolakopoulos, G., Shim, D., Carlone, L. & Burdick, J. (2021). Nebula: Quest for robotic autonomy in challenging environments; team costar at the darpa subterranean challenge. 1, 4, 5, 12, 13, 14

Armeni, I., He, Z.Y., Gwak, J., Zamir, A.R., Fischer, M., Malik, J. & Savarese, S. (2019). 3d scene graph: A structure for unified semantics, 3d space, and camera. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 5662–5672. 3, 4, 11, 12

Batra, D., Gokaslan, A., Kembhavi, A., Maksymets, O., Mottaghi, R., Savva, M., Toshev, A. & Wijmans, E. (2020). Objectnav revisited: On evaluation of embodied agents navigating to objects. 1

Betta, Z., Recchiuto, C.T. & Sgorbissa, A. (2024). People, cracks, stairs, and doors: vision-based semantic mapping with a quadruped robot supporting first responders in search & rescue. In *2024 33rd IEEE International Conference* *on Robot and Human Interactive Communication (ROMAN)*, 1878–1885. 5, 11, 12

Bouman, A., Ginting, M.F., Alatur, N., Palieri, M., Fan, D.D., Touma, T., Pailevanian, T., Kim, S.K., Otsu, K., Burdick, J.W. & akbar Agha-mohammadi, A. (2020). Autonomous spot: Long-range autonomous exploration of extreme environments with legged locomotion. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, dARPA SubT. 2, 3, 4, 5, 8, 9, 10

Busch, F.L., Homberger, T., Ortega-Peimbert, J., Yang, Q. & Andersson, O. (2024). One map to find them all: Real-time open-vocabulary mapping for zero-shot multi-object navigation. *arXiv*, duplicate key kept to match text; same work as `BUSCH24`. 2, 3, 5, 8, 9, 10, 11, 13

Campos, C., Elvira, R. *et al.* (2021). Orb-slam3: An accurate open-source library for visual, visual-inertial, and multi-map slam. *IEEE Transactions on Robotics*, **37**, 1874–1890. 14

Chaplot, D.S., Gandhi, D., Gupta, A. & Salakhutdinov, R. (2020). Object goal navigation using goal-oriented semantic exploration. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1, 8, 10

Chen, J., Li, G., Kumar, S., Ghanem, B. & Yu, F. (2023). How to not train your dragon: Training-free embodied object goal navigation with semantic frontiers. In *Robotics: Science and Systems (RSS)*. 5, 8

Cherti, M., Beaumont, R., Wightman, R., Wortsman, M., Ilharco, G., Gordon, C., Schuhmann, C., Schmidt, L. & Jitsev, J. (2023). Reproducible scaling laws for contrastive language-image learning. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2818–2829, IEEE. 49

Da Silva, V.D.S., Santos, M.G.S.D., Vieira, M.F.N., Matos, V.S., Queiroz, I.N. & Lima, R.T. (2023). Autonomous navigation strategy in quadruped robots for uneven terrain using 2d laser sensor. In *2023 Latin American Robotics Symposium (LARS), 2023 Brazilian Symposium on Robotics (SBR), and 2023 Workshop on Robotics in Education (WRE)*, 290–295. 9

Dudzik, T., Chignoli, M., Bledt, G., Lim, B., Miller, A., Kim, D. & Kim, S. (2020). Robust autonomous navigation of a small-scale quadruped robot in real-world environments. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3664–3671. 2

GILROY, S., LAU, D., YANG, L., IZAGUIRRE, E., BIERMAYER, K., XIAO, A., SUN, M., AGRAWAL, A., ZENG, J., LI, Z. & SREENATH, K. (2021). Autonomous navigation for quadrupedal robots with optimized jumping through constrained obstacles. 9

GINTING, M.F., KIM, S.K., FAN, D.D., PALIERI, M., KOCHENDERFER, M.J. & AKBAR AGHA-MOHAMMADI, A. (2024). Seek: Semantic reasoning for object goal navigation in real world inspection tasks. 9, 10, 11

GRINVALD, M., FURRER, F., NOVKOVIC, T., CHUNG, J.J., CADENA, C., SIEGWART, R. & NIETO, J. (2019). Volumetric instance-aware semantic mapping and 3d object discovery. *IEEE Robotics and Automation Letters*, **4**, 3037–3044. 5

GRISETTI, G., STACHNISS, C. & BURGARD, W. (2005). Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. In *IEEE ICRA*, 2432–2437. 14

HAUSER, E., CHAN, Y.C., CHONKAR, P., HEMKUMAR, G., WANG, H., DUA, D., GUPTA, S., ENRIQUEZ, E.M., KAO, T., HART, J. *et al.* (2023). ”what's that robot doing here?”: Perceptions of incidental encounters with autonomous quadruped robots. In *Proceedings of the First International Symposium on Trustworthy Autonomous Systems*, 1–15. 5

HESS, W., KOHLER, D., RAPP, H. & ANDOR, D. (2016). Real-time loop closure in 2d lidar slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 1271–1278. 11, 13, 14

JIANG, J., ZHU, Y., WU, Z. & SONG, J. (2025). Dualmap: Online open-vocabulary semantic mapping for natural language navigation in dynamic changing scenes. arXiv:2506.01950, reports sub-Hz to ∼1 Hz updates for full pipelines. 9, 11, 12, 13

JIN, R., LUO, Y. & ZHAO, J. (2024). Indoor quadruped robot navigation algorithm based on orb-slam. *International Journal of Computer Science and Information Technology*, **2**. 9

KLEIN, G. (2023). faster-whisper: Faster whisper transcription with ctranslate2. `https://github.com/SYSTRAN/faster-whisper`, python package for efficient Whisper inference. 48

KOVAL, A., KARLSSON, S. & NIKOLAKOPOULOS, G. (2022). Experimental evaluation of autonomous map-based spot navigation in confined environments.

Biomimetic Intelligence and Robotics, **2**, 100035. 2, 3, 4, 8, 10, 11, 12, 13, 14

Labbé, M. & Michaud, F. (2019). Rtab-map as an open-source lidar and visual slam library for large-scale and long-term online operation. *Journal of Field Robotics*, **36**, 416–446. 14

Liu, H. & Yuan, Q. (2024). Safe and robust motion planning for autonomous navigation of quadruped robots in cluttered environments. *IEEE Access*, **12**, 69728–69737. 9, 13

Longo, A., Chung, C., Palieri, M., Kim, S.K., Agha, A., Guaragnella, C. & Khattak, S. (2025). Pixels-to-graph: Real-time integration of building information models and scene graphs for semantic-geometric human-robot understanding. 11, 12

Macenski, S. & Jambrecic, I. (2021). Slam toolbox: Slam for the dynamic world. *Journal of Open Source Software*, **6**, 2783. 14, 18, 22

Macenski, S., Martin, F., White, R. & Ginés Clavero, J. (2020). The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 12, 13, 37

Majumdar, A., Aggarwal, G., Devnani, B., Hoffman, J. & Batra, D. (2022). Zson: Zero-shot object-goal navigation using multimodal goal embeddings. *Advances in Neural Information Processing Systems*, **35**, 32340–32352, zero-shot object-goal navigation with multimodal goal embeddings. 1, 3

Miki, T., Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2022). Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics*, **7**, eabk2822, robust perceptive locomotion for quadrupeds in the wild. 2, 3

Mur-Artal, R. & Tardós, J.D. (2017). Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, **33**, 1255–1262. 14

Newcombe, R.A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A.J., Kohli, P., Shotton, J., Hodges, S. & Fitzgibbon, A. (2011). Kinectfusion: Real-time dense surface mapping and tracking. In *IEEE ISMAR*, 127–136. 14

Oleynikova, H., Taylor, Z., Fehr, M., Siegwart, R. & Nieto, J. (2017). Voxblox: Incremental 3d euclidean signed distance fields for on-board planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1366–1373. 11

Qu, K., Tan, J., Zhang, T., Xia, F., Cadena, C. & Hutter, M. (2024). Ippon: Common sense guided informative path planning for object goal navigation. 8, 9, 10

Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G. & Sutskever, I. (2021). Learning transferable visual models from natural language supervision. 49

Radford, A., Kim, J.W., Xu, T., Brockman, G., McLeavey, C. & Sutskever, I. (2022). Robust speech recognition via large-scale weak supervision. 48

Raheema, J., Hess, M., Provost, R.C., Bilinski, M. & Christensen, H.I. (2024). Autonomous exploration and mapping payload integrated on a quadruped robot. In *International Symposium on Robotics Research (ISRR)*. 2, 12

Razavi, E., Bratta, A., Soares, J.C.V., Recchiuto, C. & Semini, C. (2025). Online object-level semantic mapping for quadrupeds in real-world environments. 15, 26

Redmon, J., Divvala, S., Girshick, R. & Farhadi, A. (2016). You only look once: Unified, real-time object detection. 26, 27

Rosinol, A., Abate, M., Chang, Y. & Carlone, L. (2020). Kimera: An open-source library for real-time metric-semantic localization and mapping. In *IEEE International Conference on Robotics and Automation (ICRA)*, 1689–1696, IEEE, real-time metric–semantic SLAM. 1, 3, 4, 13, 14

Schmidt, P., Scaife Jr., J., Harville, M., Liman, S. & Ahmed, A. (2019). Intel® RealSense™ Tracking Camera T265 and Intel® RealSense™ Depth Camera D435 - Tracking and Depth. Whitepaper Revision 001, Intel Corporation. 16, 17, 18, 23, 29, 30

Shan, T., Englot, B., Meyers, D., Wang, W., Ratti, C. & Rus, D. (2020). Lio-sam: Tightly-coupled lidar inertial odometry via smoothing and mapping. In *IEEE/RSJ IROS*, 5135–5142. 14

Unlu, H.U., Guan, T., Sathyamoorthy, A., Weerakoon, K., Manocha, D. & Chowdhary, G. (2022). Graspe: Graph-based active sensing for planning in unstructured environments. In *arXiv*. 5

Unlu, H.U., Yuan, S., Wen, C., Huang, H., Tzes, A. & Fang, Y. (2024). Reliable semantic understanding for real world zero-shot object goal navigation. *arXiv*. 3, 8, 9, 10, 11, 12, 13

Wellhausen, L. & Hutter, M. (2023). Artplanner: Robust legged robot navigation in the field. *Field Robotics*, **3**, 413–434. 5, 9, 10, 11, 12, 14

Xu, Y., Li, B., Yang, S., Zhi, Y. & Li, R. (2024). Sli-slam: Autonomous navigation and accurate mapping for quadruped robot in complex environments using lidar, stereo camera, and imu fusion. In *2024 43rd Chinese Control Conference (CCC)*, 4326–4333. 9

Yokoyama, N., Ha, S., Batra, D., Wang, J. & Bucher, B. (2023). Vlfm: Vision-language frontier maps for zero-shot semantic navigation. 1, 3, 5, 8, 9, 10, 13

Zhang, J. & Singh, S. (2017). Loam: Lidar odometry and mapping in real-time. *The International Journal of Robotics Research*, **34**, 1285–1302. 14

Zhang, L., Wang, H., Xiao, E., Zhang, X., Zhang, Q., Jiang, Z., Chen, H. & Xu, R. (2024). Multi-floor zero-shot object navigation policy. In *arXiv*. 5, 9

Zhou, K., Mu, Y., Song, H., Zeng, Y., Wu, P., Gao, H. & Liu, C. (2025). Adaptive interactive navigation of quadruped robots using large language models. 9