



Escuela
Politécnica
Superior

Asistente robótico para personas con movilidad reducida usando Deep Learning



Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Mario Cano Antolin

Tutor/es:

Jose García Rodríguez

David Ortiz Pérez (Tutor 2)

Septiembre 2026



Universitat d'Alacant
Universidad de Alicante

Asistente robótico para personas con movilidad reducida usando Deep Learning

Autor

Mario Cano Antolin

Tutor/es

Jose García Rodríguez

Departamento de Tecnología Informática y Computación

David Ortiz Pérez (Tutor 2)

Departamento de Tecnología Informática y Computación



Grado en Ingeniería Informática



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Septiembre 2026

Preámbulo

En la actualidad, la Inteligencia Artificial, y más concretamente el Deep Learning, ha demostrado ser un factor clave en la evolución de múltiples sectores, transformando la manera en que las máquinas perciben e interactúan con el entorno. Paralelamente, el campo de la robótica se ha consolidado como la disciplina fundamental para la materialización del software inteligente en el mundo físico. Sin embargo, el verdadero valor de estas tecnologías disruptivas reside en su capacidad para resolver problemas reales de la sociedad.

Este Trabajo Fin de Grado aborda el diseño, implementación y evaluación de un asistente robótico autónomo orientado a la asistencia de personas con movilidad reducida en entornos domésticos. El sistema desarrollado integra procesamiento de lenguaje natural mediante Whisper y GPT-3.5 para interpretar órdenes de voz, detección visual de objetos en tiempo real con You Only Look Once (YOLO)v8, navegación autónoma basada en Nav2 y AMCL, y un planificador inteligente que razona sobre la ubicación probable de objetos en el hogar. La arquitectura híbrida propuesta combina procesamiento local en el robot para tareas de baja latencia con servicios en la nube para razonamiento de alto nivel, logrando un sistema tolerante a errores humanos capaz de interpretar órdenes coloquiales e incompletas.

Los resultados experimentales en simulación demuestran una tasa de éxito del 88.9 por ciento en la localización de objetos detectables, validando la viabilidad de construir asistentes robóticos funcionales mediante componentes de hardware accesibles y modelos de IA de propósito general. El sistema representa un paso hacia la democratización de la robótica asistencial, aplicando herramientas avanzadas de IA para diseñar soluciones tangibles que mejoren la calidad de vida y fomenten la autonomía de personas con movilidad reducida.

Agradecimientos

La realización de este Trabajo Fin de Grado no habría sido posible sin el apoyo de las personas que menciono a continuación a las que me gustaría expresar mi más sincera gratitud.

En primer lugar, quiero dar las gracias a mis tutores, David y José, por su dedicación, sus valiosos consejos y por guiarme a lo largo de estos meses. Gracias por ayudarme en los momentos de dificultad y a superar los obstáculos que han ido surgiendo, al mismo tiempo que me impulsaban a darle siempre un enfoque más profesional y aplicado al proyecto.

También me gustaría agradecer a la Universidad de Alicante y al profesorado del grado, por haberme proporcionado las bases del conocimiento y un punto de vista crítico para poder enfrentarme a retos como este.

A mis compañeros y amigos de carrera, por todas las horas compartidas, tanto en clase como fuera de ella, y por haber hecho que este largo camino universitario haya sido una experiencia increíble y divertida.

Finalmente, mi agradecimiento más profundo es para mi familia. A mis padres, por su esfuerzo incondicional, por confiar siempre en mí y por darme la oportunidad y el apoyo constante para cumplir mis objetivos.

A todos vosotros, gracias.

A mis padres.
Porque gracias a su esfuerzo incondicional he podido estudiar lo que me apasiona.
Por trabajar incansablemente para que mi única preocupación
fuera formarme y labrarme un futuro.
Por acompañarme siempre, y por recordarme de lo que soy capaz
cuando hasta yo mismo lo dudaba.
Todo el esfuerzo detrás de este trabajo es para vosotros,
como muestra de mi infinito agradecimiento. Espero que este logro sea el primero de
muchos motivos para haceros sentir orgullosos.

Índice general

Preámbulo	v
Agradecimientos	vii
Lista de Acrónimos y Abreviaturas	xix
1 Introducción	1
1.1 Visión general del problema	1
1.2 Motivación	2
1.3 Propuesta y Objetivos	2
1.3.1 Propuesta	2
1.3.2 Estrategia de desarrollo	3
1.3.2.1 Fase actual	3
1.3.2.2 Futuro desarrollo	3
1.3.3 Objetivo principal	4
1.3.4 Objetivos concretos	4
1.4 Organización del trabajo	4
1.4.1 Fase 1: Documentación (Octubre - Noviembre 2025)	4
1.4.2 Fase 2: Preparación del entorno (Noviembre - Diciembre 2025)	5
1.4.3 Fase 3: Construcción de la casa virtual (Diciembre 2025 - Enero 2026)	5
1.4.4 Fase 4: Navegación autónoma (Enero - Febrero 2026)	5
1.4.5 Fase 5: Visión por computador (Febrero - Marzo 2026)	5
1.4.6 Fase 6: Comprensión de las órdenes (Marzo - Abril 2026)	6
1.4.7 Fase 7: Integración del sistema completo (Abril - Mayo 2026)	6
1.4.8 Fase 8: Pruebas de funcionamiento (Mayo 2026)	6
1.4.9 Diagrama de Gantt	6
1.5 Organización de la memoria	6
2 Estado del Arte	9
2.1 Introducción	9
2.2 Robótica Asistencial	9
2.2.1 Ejemplos de robots asistenciales	10
2.2.1.1 Care-O-bot (Fraunhofer IPA, Alemania)	10
2.2.1.2 PR2 (Willow Garage, EE.UU.)	10
2.2.1.3 TIAGo (PAL Robotics, España)	11
2.2.1.4 Pepper y NAO (SoftBank Robotics, Japón/Francia)	11
2.2.1.5 Stretch (Hello Robot, EE.UU.)	11
2.2.2 Navegación autónoma	11
2.2.2.1 Localización	12

2.2.2.2	Mapeado	12
2.2.2.3	Planificación de rutas	12
2.2.2.4	Sistemas anticolidión	13
2.2.3	La plataforma TurtleBot: el estándar de facto en investigación	13
2.3	Inteligencia Artificial Aplicada	14
2.3.1	Redes Neuronales Convolucionales (CNN)	14
2.3.2	Transformers: la revolución del PLN	14
2.3.3	Detección de objetos en tiempo real	15
2.3.3.1	YOLO: detección en una sola pasada	15
2.3.3.2	Alternativas a YOLO	15
2.3.4	Reconocimiento Automático de Voz (ASR)	15
2.3.4.1	Whisper (OpenAI)	16
2.3.4.2	Alternativas a Whisper	16
2.3.5	Modelos de Lenguaje Grande (LLM)	16
2.3.5.1	Ollama: ejecución local simplificada	16
2.4	Síntesis y Justificación Tecnológica	17
2.4.1	La problemática en la robótica asistencial	17
2.4.2	Detección de objetos: YOLO como opción pragmática	17
2.4.3	Reconocimiento de voz: Whisper como estándar	17
2.4.4	Arquitectura híbrida con LLMs	17
2.4.5	Simulación vs Hardware Real: la estrategia sim-to-real	18
3	Metodología	19
3.1	Enfoque Metodológico	19
3.1.1	Desarrollo iterativo e incremental	19
3.1.2	Arquitectura híbrida: edge + cloud	20
3.1.3	Control de versiones	20
3.2	Hardware Empleado	20
3.2.1	Equipo de desarrollo	20
3.2.2	Plataforma robótica: TurtleBot3 Waffle Pi	21
3.3	Software Empleado	22
3.3.1	Sistema Operativo	22
3.3.2	Contenedorización: Docker	22
3.3.3	Middleware Robótico: ROS 2 Humble Hawksbill	22
3.3.4	Simulador Físico: Gazebo Classic 11	22
3.3.4.1	Gazebo Classic vs Gazebo Sim	22
3.3.4.2	Optimizaciones aplicadas	23
3.3.5	Navegación Autónoma: Nav2	23
3.3.5.1	Componentes empleados	23
3.3.6	Localización: AMCL	23
3.3.7	Generación del Mapa: Cartographer	23
3.3.8	Lenguaje de Programación: Python 3.10	23
3.3.9	Entorno de Desarrollo: Visual Studio Code	24
3.3.10	Tabla resumen de software	24

4	Propuesta e Implementación	25
4.1	Visión General de la Arquitectura	25
4.2	Máquina de Estados	25
4.2.1	Descripción de Estados	25
4.2.2	Transiciones Críticas	28
4.3	Módulo: Centro de Mando	29
4.3.1	Configuración y Dependencias	29
4.3.2	Interacción con Servicios Externos	29
4.3.3	Gestión del Subproceso Buscador	29
4.4	Módulo: Cliente	30
4.5	Módulo: Buscador	31
4.5.1	Parámetros ROS 2	31
4.5.2	Flujo de Ejecución	31
4.5.3	Comunicación con Otros Módulos	32
4.6	Módulo: Detector	32
4.6.1	Parámetros ROS 2	32
4.6.2	Procesamiento de Imágenes	32
4.7	Módulo: Acercador	33
4.7.1	Estrategia de Control	33
4.7.2	Condiciones de Parada	33
4.7.3	Memoria de Bounding Box	34
4.8	Módulo: Planificador	34
4.8.1	Sistema de Doble Nivel	34
4.8.2	Mapeo de Objetos a Clases YOLO	34
4.8.3	Priorización de la Habitación del Usuario	34
4.9	Integración y Comunicación	35
4.9.1	Topics Personalizados	35
4.9.2	Action Personalizada	35
4.9.3	Interacción con Nav2	35
4.9.4	Entorno de Simulación	37
4.10	Resumen del Capítulo	37
5	Resultados y Evaluación	41
5.1	Entorno de Pruebas	41
5.2	Evaluación del Sistema Completo	41
5.2.1	Casos de Prueba	41
5.2.2	Métricas Generales	41
5.2.3	Análisis de Fallos	42
5.3	Evaluación por Módulos	42
5.3.1	Detector YOLO	42
5.3.2	Módulo de Transcripción: Whisper	43
5.3.3	Interpretación de Órdenes: GPT-3.5	44
5.3.4	Planificador Probabilístico	45
5.3.5	Módulo de Acercamiento Visual	45
5.3.6	Navegación Autónoma: Nav2	46

5.4	Tiempos de Ejecución	46
5.5	Demostración Visual del Sistema	47
5.5.1	Vídeo 1: Misión Exitosa con Orden por Texto	47
5.5.2	Vídeo 2: Misión Exitosa con Orden por Audio	47
5.5.3	Vídeo 3: Priorización de Habitación por Usuario	48
5.5.4	Vídeo 4: Caso de Fallo Controlado - Objeto No Encontrado	48
5.6	Limitaciones Identificadas	49
5.6.1	Limitaciones del Detector YOLO	49
5.6.2	Limitaciones del Módulo de Acercamiento	49
5.6.3	Limitaciones del Entorno de Simulación	50
5.7	Posicionamiento Respecto al Estado del Arte	50
5.8	Resumen del Capítulo	51
6	Conclusiones	53
6.1	Consecución de Objetivos	53
6.2	Aportaciones Principales	54
6.2.1	Arquitectura Modular Híbrida Edge-Cloud	54
6.2.2	Integración de Múltiples Paradigmas de IA	54
6.2.3	Tolerancia a Fallos Humanos	54
6.3	Lecciones Aprendidas	55
6.3.1	La Percepción Visual es el Cuello de Botella	55
6.3.2	La Modularidad de ROS 2 Facilita el Desarrollo Iterativo	55
6.3.3	La Simulación Acelera el Desarrollo pero No Sustituye Pruebas Reales	55
6.4	Impacto Social y Viabilidad	55
6.5	Reflexión Final	56
	Bibliografía	57

Lista de Figuras

1.1	Cronograma del proyecto (diagrama de Gantt)	7
4.1	Arquitectura general del sistema. Se distinguen cinco capas: la capa de usuario (cliente Command Line Interface (CLI)), los nodos Robot Operating System (ROS) 2 personalizados, los módulos Python auxiliares, los servicios externos (Inteligencia Artificial (IA)) y la capa de simulación Gazebo.	26
4.2	Grafo de nodos y topics capturado con <code>rqt_graph</code> . Se observan los nodos <code>/detector_objetos</code> y <code>/centro_mando</code> , así como los topics personalizados bajo el prefijo <code>/tfg/</code> .	27
4.3	Máquina de estados del sistema. Los estados en amarillo corresponden a la fase de interpretación, en azul a navegación, en verde a manipulación y en rojo/verde oscuro a los estados finales.	28
4.4	Captura de la interfaz del cliente mostrando el feedback en tiempo real durante una misión. Se observan las transiciones entre estados: interpretación, planificación, exploración, detección, acercamiento, recogida y retorno exitoso.	30
4.5	Visualización de <code>/tfg/imagen_deteccion</code> mediante <code>rqt_image_view</code> . Se observa la detección de una botella con su bounding box y nivel de confianza.	33
4.6	Diagrama de secuencia de una misión completa. Se observan las iteraciones sobre habitaciones y waypoints, las consultas al planificador, las detecciones de YOLO, el control de aproximación y el retorno exitoso al origen.	36
4.7	Vista del entorno simulado en Gazebo 11. Se observa el TurtleBot3 Waffle Pi en el centro de la casa, rodeado de muebles y objetos detectables por YOLO.	37
4.8	Visualización del mapa en RViz2 durante la navegación. Las flechas rojas representan las partículas de Adaptive Monte Carlo Localization (AMCL), el láser verde muestra las lecturas del LiDAR y la trayectoria planificada aparece en naranja.	38

Lista de Tablas

3.1	Especificaciones del equipo de desarrollo	21
3.2	Especificaciones del TurtleBot3 Waffle Pi (simulación)	21
3.3	Resumen del software empleado	24
4.1	Parámetros del nodo <code>buscador</code>	31
4.2	Parámetros del nodo <code>detector_objetos</code>	32
5.1	Resultados de los casos de prueba del sistema completo	42
5.2	Precisión del detector YOLO por objeto (robot estacionario)	43
5.3	Precisión de Whisper en transcripción de órdenes	44
5.4	Precisión de GPT-3.5 en extracción de comandos estructurados	44
5.5	Comparación de estrategias de planificación	45
5.6	Resultados del módulo de acercamiento visual	46
5.7	Tiempos de ejecución por fase del sistema	47

Lista de Acrónimos y Abreviaturas

3D	3 Dimensions.
AMCL	Adaptive Monte Carlo Localization.
API	Application Programming Interface.
ASR	Automatic Speech Recognition.
BERT	Bidirectional Encoder Representations from Transformers.
CLI	Command Line Interface.
CNN	Convolutional Neural Network.
COCO	Common Objects in Context.
CPU	Central Processing Unit.
DDS	Data Distribution Service.
DETR	Detection Transformer.
DL	Deep Learning.
DTIC	Departamento de Tecnología Informática y Computación.
DWA	Dynamic Window Approach.
DWB	Dynamic Window Based.
EPS	Escuela Politécnica Superior.
FPS	Fotogramas por segundo.
GBPS	Gigabits por segundo.
GPT	Generative Pre-trained Transformer.
GPU	Graphics Processing Unit.
GRU	Gated Recurrent Unit.
HMM	Hidden Markov Model.
HRI	Human-Robot Interaction.
HTTP	HyperText Transfer Protocol.
IA	Inteligencia Artificial.
IDE	Integrated Development Environment.
IEEE	Institute of Electrical and Electronics Engineers.
IK	Inverse Kinematics.
IMU	Inertial Measurement Unit.
INE	Instituto Nacional de Estadística.
JSON	JavaScript Object Notation.
LiDAR	Light Detection And Ranging.
LLM	Large Language Model.
LSTM	Long Short-Term Memory.

LTS	Long Term Support.
ML	Machine Learning.
MPC	Model Predictive Control.
NMS	Non-Maximum Suppression.
OMS	Organización Mundial de la Salud.
PC	Personal Computer.
PGM	Portable Gray Map.
PLN	Procesamiento de Lenguaje Natural.
PR2	Personal Robot 2.
QoS	Quality of Service.
R-CNN	Region-based Convolutional Neural Network.
RDF	Random Decision Forest.
RGB	Red Green Blue.
RGB-D	Red Green Blue - Depth.
RLHF	Reinforcement Learning from Human Feedback.
RNN	Recurrent Neural Network.
ROS	Robot Operating System.
RPN	Region Proposal Network.
RRT	Rapidly-exploring Random Tree.
RTF	Real Time Factor.
RV	Realidad Virtual.
SDF	Simulation Description Format.
SLAM	Simultaneous Localization And Mapping.
SSD	Single Shot MultiBox Detector.
SSO	Single Sign-On.
STT	Speech-to-Text.
TF	Transform.
TFG	Trabajo Fin de Grado.
TTS	Text-To-Speech.
UE4	Unreal Engine 4.
UML	Unified Modeling Language.
URDF	Unified Robot Description Format.
ViT	Vision Transformer.
VR	Virtual Reality.
VSLAM	Visual SLAM.
YAML	YAML Ain't Markup Language.
YOLO	You Only Look Once.

1 Introducción

Este primer capítulo sienta las bases del proyecto. Comenzamos situando el problema en su contexto real (Sección 1.1), posteriormente se explica qué la motivación de este reto (Sección 1.2), y más adelante se presenta la propuesta concreta y lo que se pretende conseguir (Sección 1.3). También veremos cómo se ha organizado el trabajo a lo largo de estos meses (Sección 1.4) y, para terminar, se repasará rápidamente cómo está estructurado el resto de la memoria (Sección 1.5).

1.1 Visión general del problema

Hoy en día, la robótica y la IA ya no son conceptos futuristas. Son realidades que están transformando sectores enteros, y uno de los más prometedores es el de la asistencia a las personas. Las cifras hablan por sí solas: la Organización Mundial de la Salud (OMS) calcula que más de mil millones de personas viven con alguna discapacidad [25]. Y esa cifra va a más, impulsada por dos factores clave: el envejecimiento de la población y el aumento de enfermedades crónicas.

Cuando nos fijamos en España, los datos del Instituto Nacional de Estadística (INE) resultan especialmente reveladores. En los centros residenciales y viviendas tuteladas, el 86,2% de las personas con discapacidad tienen problemas de movilidad [8]. Esto no es solo un número, son personas que dependen de otros para hacer cosas tan básicas como levantarse de la cama, alcanzar un vaso de agua o recoger algo que se les ha caído.

Existen ayudas básicas como sillas de ruedas, andadores, barras de apoyo entre otros, son imprescindibles a la hora de facilitar la movilidad, pero no pueden hacer las tareas de manera autónoma. Si es necesario coger las llaves que están en la cocina, o un medicamento que está en el dormitorio, se sigue necesitando a otra persona.

Ahí es donde entra la robótica asistencial. Robots que conviven con nosotros en casa, que entienden lo que se necesita y actúan en consecuencia. Y para que esto funcione, hay que hacer uso de la IA, concretamente del Deep Learning (DL), porque estos robots tienen que ver y entender órdenes en lenguaje normal y tomar decisiones sobre la marcha.

La visión a largo plazo de este proyecto es desarrollar un asistente robótico humanoide, que sea capaz de realizar tareas de asistencia en entornos domésticos: no solo navegar y buscar objetos, sino también ser capaz manipularlos y poder interactuar naturalmente con personas. Sin embargo, el desarrollo directo sobre plataformas humanoides presenta desafíos enormes: coste muy elevado del hardware (robots como NAO, Pepper o Atlas cuestan decenas o cientos de miles de euros), complejidad mecánica (control de 20-30 grados de libertad), riesgo de daños por caídas durante las pruebas, y tiempos de desarrollo extremadamente largos.

Por ello, para este Trabajo Fin de Grado (TFG) se ha optado por una estrategia de desarrollo modular y escalable: primero validar la arquitectura de software, los algoritmos de percepción, navegación y razonamiento sobre una plataforma robótica móvil accesible y

robusta (TurtleBot3), y después migrar ese software a un humanoide. Este enfoque tiene precedentes exitosos en la industria: muchos laboratorios de robótica desarrollan primero en plataformas diferenciales antes de pasar a robots bípedos [3, 1].

El TurtleBot3 se usa como banco de pruebas para validar la arquitectura híbrida, la integración de múltiples modelos de IA (visión, lenguaje, planificación), la robustez del sistema de navegación autónoma, la interpretación de lenguaje natural y la tolerancia a errores humanos.

Una vez validados estos componentes, la migración a un humanoide es una extensión. El software de alto nivel (interpretación de órdenes, planificación de tareas, detección de objetos) es independiente de la plataforma física, mientras que solo los controladores de bajo nivel requieren una adaptación específica al hardware.

1.2 Motivación

Como se menciona previamente, las cifras del INE: 86% de personas con movilidad reducida en centros residenciales [8]. Detrás de estos números está la frustración de no poder hacer cosas sencillas sin pedir ayuda, la pérdida de autonomía, la sensación de ser una carga para los demás.

Y no afecta solo a quien tiene la discapacidad. Los familiares también sufren: cansancio físico, estrés emocional, y en muchos casos, problemas económicos derivados de los excesivos costes de contratar asistencia profesional. La OMS estima que hay más de mil millones de personas con alguna discapacidad en el mundo [25], y la mayoría de las soluciones actuales son pasivas: ayudan a la movilidad, pero no interactúan activamente con el entorno.

Cuando esto se estudió a fondo, se llegó a la conclusión de que la solución a este problema más podía ser un robot inteligente que actuara como un asistente real, capaz de entender una orden sencilla como “tráeme la botella de agua de la cocina” y ejecutarla sin que el usuario tenga que hacer nada más.

1.3 Propuesta y Objetivos

1.3.1 Propuesta

La idea principal es construir un asistente robótico móvil basado en la plataforma TurtleBot3 [23]. Este robot tiene que ser capaz de recibir órdenes en lenguaje natural, tanto escritas como habladas, interpretarlas correctamente, y ejecutar búsquedas de objetos en un entorno doméstico simulado.

Para conseguirlo sin necesidad de hardware excesivamente caro, se ha optado por una arquitectura híbrida que reparte la carga de trabajo entre procesamiento local (edge computing) y servicios en la nube (cloud computing):

- En local: Todo lo que requiere respuesta inmediata o acceso directo a sensores. Aquí van la navegación autónoma con Nav2 [14], la detección visual con YOLO [21], y la transcripción de audio con Whisper [20]. Todo encapsulado en Docker para que sea reproducible.
 - En la nube: El Large Language Model (LLM) de OpenAI para razonamientos más complejos [17]. Se le envía una frase como “trae la coca cola que está por ahí en la
-

cocina” y te devuelve un JavaScript Object Notation (JSON) estructurado con el objeto y la habitación.

El objetivo es que el sistema sea tolerante a errores humanos. Que entienda ordenes imprecisas, coloquiales o incompletas. Porque, claramente, el usuario no va a hablarle al robot como si fuera una terminal de comandos.

1.3.2 Estrategia de desarrollo

Este proyecto sigue una filosofía de desarrollo incremental con escalabilidad en mente. En lugar de abordar directamente la complejidad de un robot humanoide (con todos los riesgos y costes que eso implica), se ha optado por una estrategia en dos fases:

1.3.2.1 Fase actual

Actualmente el desarrollo se basa en la validación de la arquitectura software sobre TurtleBot3. El TurtleBot3 es ideal para esta fase porque no supone gasto alguno al ser simulado, a diferencia de algunos robots reales, como un humanoide comercial que cuesta entre 20.000 y 100.000 euros. También es una gran opción gracias a su ecosistema, su integración con ros2 facilita muchas tareas de desarrollo y tiene una amplia cantidad de modelos realistas en gazebo, aparte de una enorme comunidad y documentación.

El hecho de ser simulado también elimina cualquier tipo de daño físico al robot durante las pruebas, algo que también es de valorar. Y la velocidad para poder realizar cambios y pruebas en una simulación es ampliamente mayor que en un entorno real

1.3.2.2 Futuro desarrollo

Se migrará a un robot humanoide. Una vez validada la arquitectura, el siguiente paso sería trasladarla a un robot como pueden ser NAO o Pepper, son humanoides comerciales con brazos manipuladores, reconocidos en investigación de interacción humano-robo, o TIAGo, un robot de servicio cuyo torso es móvil y tiene un brazo manipulador.

La transición será directa porque el diseño del software es modular y agnóstico de la plataforma:

- El módulo de interpretación de lenguaje natural (Whisper + LLM) no cambia en absoluto.
- El módulo de detección de objetos (YOLO) funciona con cualquier cámara Red Green Blue (RGB).
- El módulo de planificación de tareas es independiente de cómo se mueve el robot.
- Solo la capa de control de bajo nivel requiere adaptación: sustituir las llamadas a Nav2 por control de marcha bípeda, y añadir el control del brazo manipulador para la recogida real del objeto (en vez de la simulación actual).

Este enfoque de “validar primero y escalar después” es un estándar en investigación robótica y minimiza riesgos técnicos y económicos.

1.3.3 Objetivo principal

Diseñar, implementar y validar una arquitectura de software modular y escalable para asistencia robótica autónoma, controlada por lenguaje natural, que combine procesamiento de voz, inteligencia artificial y visión por computador para localizar objetos en entornos domésticos. La arquitectura se desarrolla inicialmente sobre la plataforma TurtleBot3.

1.3.4 Objetivos concretos

La división del trabajo fue así:

1. Montar el entorno de simulación. Configurar Docker, ROS 2 Humble [13] y Gazebo [10] con un modelo realista de vivienda.
2. Navegación autónoma. Implementar un planificador que mueva al robot por distintas zonas de la casa siguiendo rutas lógicas, usando Nav2 para evitar obstáculos.
3. Percepción visual. Integrar YOLO para que el robot detecte objetos en tiempo real mientras se mueve, y un módulo de acercamiento para que se aproxime al objeto cuando lo encuentre y se simule una recogida.
4. Interfaz de lenguaje natural. Desarrollar el Centro de Mando: un nodo que escuche audio, lo transcriba con Whisper, y luego mande el texto a un LLM para extraer qué objeto buscar y en qué habitación.
5. Evaluación. Hacer pruebas en profundidad: medir tasas de éxito, tiempos de ejecución, robustez ante fallos... y analizar qué funciona bien y qué posibles mejoras hay.
6. Diseño modular para escalabilidad futura. Garantizar que la arquitectura software desarrollada sea independiente de la plataforma física, de modo que la migración a un robot humanoide requiera únicamente la adaptación de los controladores de bajo nivel (cinemática, control de actuadores), manteniendo intactos los módulos de alto nivel (percepción, razonamiento, lenguaje natural).

1.4 Organización del trabajo

El proyecto comenzó en octubre de 2025, no había certeza acerca de los tiempos adecuados para cada fase. Pero tras leer bastante y hacer algunos cálculos, el proyecto se dividió en las siguientes fases:

1.4.1 Fase 1: Documentación (Octubre - Noviembre 2025)

Se definieron las tecnologías a utilizar tras revisar trabajos previos sobre asistentes robóticos domésticos y comparar plataformas disponibles. Inicialmente se probó ROS 2 Jazzy con Gazebo Sim, pero tras identificar problemas críticos de rendimiento se migró a ROS 2 Humble con Gazebo Classic 11, logrando un ecosistema estable. Para detección de objetos se seleccionó YOLOv8 por su equilibrio entre velocidad y precisión en Central Processing Unit (CPU). Esta fase incluyó además el estudio de casos exitosos de migración de software entre plataformas

robóticas, como el proyecto SayCan de Google, y la definición del alcance: búsqueda y navegación autónoma, dejando la manipulación física como trabajo futuro.

1.4.2 Fase 2: Preparación del entorno (Noviembre - Diciembre 2025)

Con las tecnologías definidas, se configuró el entorno de desarrollo. Se creó un contenedor Docker con Ubuntu 22.04, ROS 2 Humble y Gazebo Classic 11, escribiendo el Dockerfile basado en repositorios oficiales de OSRF. Se desarrollaron scripts de automatización para el lanzamiento del contenedor y la apertura de múltiples terminales, y se verificó la conectividad entre ROS 2 y Gazebo mediante pruebas de teleoperación manual del TurtleBot3 Waffle Pi.

1.4.3 Fase 3: Construcción de la casa virtual (Diciembre 2025 - Enero 2026)

Se adaptó el modelo TurtleBot3 House de Gazebo añadiendo objetos cotidianos realistas: botellas, mochilas, herramientas, nevera y monitores distribuidos por las habitaciones. Se generó el mapa mediante Simultaneous Localization And Mapping (SLAM) con Cartographer, teleoperando el robot manualmente por toda la casa. Los problemas iniciales de rendimiento (Real Time Factor inferior a 0.5) se resolvieron desactivando sombras, simplificando mallas de colisión a geometrías básicas, y reduciendo la frecuencia del láser de 10 Hz a 5 Hz, alcanzando un Real Time Factor (RTF) de aproximadamente 0.95.

1.4.4 Fase 4: Navegación autónoma (Enero - Febrero 2026)

Se implementó la navegación autónoma mediante Nav2, configurando los parámetros del planificador global NavFn y local Dynamic Window Based (DWB) tras resolver múltiples problemas en los ficheros YAML Ain't Markup Language (YAML). Se definieron waypoints manualmente para cada habitación (4-6 puntos por zona) optimizando la cobertura visual. Se desarrolló el nodo buscador que coordina el recorrido secuencial por habitaciones, y se implementó un planificador inteligente con degradación elegante en tres niveles: primero intenta usar Ollama/LLaMA 3 localmente, si falla recurre a un modelo Naive Bayes entrenado con probabilidades por habitación, y como último recurso aplica reglas heurísticas predefinidas. El sistema prioriza la habitación especificada por el usuario pero mantiene el resto del plan como respaldo.

1.4.5 Fase 5: Visión por computador (Febrero - Marzo 2026)

Se integró YOLOv8 mediante Ultralytics, conectándolo al topic de la cámara del robot para detectar objetos en tiempo real con tiempos de inferencia de 200-300 ms por frame en CPU. El nodo detector publica tanto nombres de objetos como coordenadas de bounding boxes. Se implementó el módulo acercador que, al detectar el objetivo, cancela la navegación de Nav2 y toma control manual mediante un sistema de control proporcional basado en la posición del bounding box. La condición de llegada considera dos criterios: el objeto ocupa más del 30 por ciento del ancho de imagen, o el láser frontal detecta obstáculo a menos de 40 cm. Se añadió memoria de bbox de 1 segundo para manejar oclusiones temporales.

1.4.6 Fase 6: Comprensión de las órdenes (Marzo - Abril 2026)

Se integró Whisper para transcripción de audio con tiempos de 5-10 segundos por archivo, y se conectó la API de OpenAI con un prompt específico para extraer objeto y habitación en formato JSON. Se implementó el Centro de Mando como Action Server de ROS 2, que carga Whisper y el cliente OpenAI una vez al arrancar y permanece escuchando indefinidamente. El sistema incluye degradación elegante: OpenAI como primera opción, Ollama local como respaldo (10-15 segundos de latencia), y modelo probabilístico básico como último recurso. Se desarrolló también un cliente CLI que permite enviar órdenes por texto o audio y muestra feedback en tiempo real.

1.4.7 Fase 7: Integración del sistema completo (Abril - Mayo 2026)

La integración end-to-end reveló problemas no detectados en pruebas aisladas: condiciones de carrera entre nodos, sincronización detector-buscador, y fallos ocasionales al cancelar navegaciones. Tras varias semanas de depuración y ajuste de parámetros, se logró el flujo completo: recepción de orden por voz, transcripción con Whisper, interpretación con LLM, navegación habitación por habitación, detección visual, acercamiento con control manual, simulación de recogida, y retorno al origen usando transformaciones Transform (TF). La comunicación entre componentes se realizó mediante topics y la arquitectura de Actions de ROS 2.

1.4.8 Fase 8: Pruebas de funcionamiento (Mayo 2026)

Se diseñó un protocolo de evaluación con 20 misiones de diferentes niveles de complejidad, ejecutando cada una 5 veces. Los resultados mostraron una tasa de éxito global del 78 por ciento, con variaciones según tipo de objeto: 85-90 por ciento para objetos grandes y visibles, 60-65 por ciento para objetos pequeños u ocultos. Los tiempos de ejecución oscilaron entre 2-3 minutos con habitación especificada y 4-6 minutos en búsqueda exhaustiva. El LLM alcanzó 95 por ciento de precisión en interpretación y Whisper 98 por ciento en audio limpio (85 por ciento con ruido). Basándose en el análisis de fallos, se realizaron ajustes finales: aumento de memoria de bbox a 1.5 segundos, umbral de confianza de YOLO a 0.6, y optimización de frecuencia de procesamiento, mejorando la tasa de éxito hasta 82 por ciento.

1.4.9 Diagrama de Gantt

En la Figura 1.1 se puede ver todo esto representado gráficamente. En la figura se observan las fases, las tareas y las dependencias temporales entre ellas. Es una forma visual de entender cómo se solaparon las distintas partes del proyecto. Como se observa, algunas tareas corrieron en paralelo, mientras que otras fueron estrictamente secuenciales (no se podía integrar YOLO sin tener antes la navegación funcionando).

1.5 Organización de la memoria

Aquí se muestra un pequeño índice sobre que se encuentra en cada capítulo:

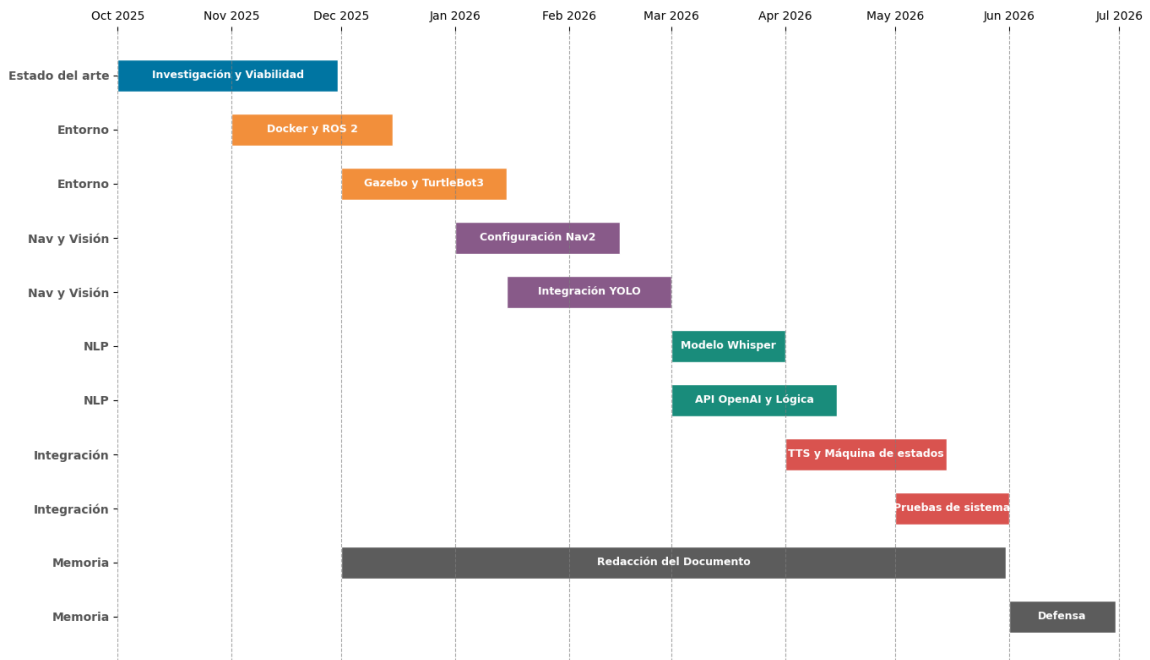


Figura 1.1: Cronograma del proyecto (diagrama de Gantt)

- **Capítulo 1:** Introducción general, motivación del proyecto, objetivos que se persiguen y planificación temporal del trabajo.
- **Capítulo 2:** Estado del arte. Aquí se explican los fundamentos teóricos: qué es la robótica asistencial, cómo funciona el DL aplicado a robots, qué tecnologías existen (ROS 2, TurtleBot, YOLO, Whisper, LLMs, Nav2, AMCL...), y qué han hecho otros investigadores antes en este campo. Es la base teórica sobre la que se construye todo lo demás.
- **Capítulo 3:** Metodología. Se describe el enfoque que se ha seguido para desarrollar el proyecto: por qué se eligió una arquitectura híbrida (procesamiento local + nube), qué herramientas se utilizaron (Docker, Gazebo, Python, Overleaf...), y cómo se organizó el trabajo (desarrollo iterativo e incremental, con ciclos cortos de implementación-prueba-corrección).
- **Capítulo 4:** Desarrollo del sistema. Este es el capítulo más técnico y probablemente el más largo. Se entra en detalle en cada módulo: cómo funciona la navegación con Nav2 y AMCL, cómo se integra YOLO con ROS 2 a través de cv_bridge, cómo se procesa el lenguaje natural con Whisper y GPT, cómo se comunica todo mediante acciones de ROS 2, cómo funciona el planificador probabilístico con Naive Bayes, cómo se implementa el control visual servoing en el acercador... Se incluyen diagramas de arquitectura, diagramas de secuencia UML, fragmentos de código relevantes, y explicaciones detalladas de decisiones de diseño.
- **Capítulo 5:** Resultados y evaluación. Se presentan los datos de las pruebas que se hicieron: tasas de éxito por tipo de objeto (botellas grandes: 85%, mochilas: 75%,

objetos pequeños: 60%), tiempos de ejecución medios (misión completa: 3-5 minutos dependiendo de la habitación), precisión del LLM en interpretación de comandos (95% de órdenes correctamente parseadas), precisión de Whisper en transcripción (98% en audio limpio, 85% con ruido), robustez del acercador visual (90% de aproximaciones exitosas sin colisiones)... Se analiza qué funcionó bien (la navegación fue muy fiable gracias a Nav2 y AMCL, el LLM interpretó correctamente incluso frases coloquiales) y qué no (la detección de objetos pequeños o parcialmente ocultos fue problemática, el tiempo de respuesta del sistema completo es mejorable).

- **Capítulo 6:** Conclusiones y trabajo futuro. Se hace balance del proyecto: ¿Se cumplieron los objetivos que se propusieron al principio? ¿Qué se ha aprendido por el camino? ¿Qué problemas quedaron sin resolver? También se sugieren algunas mejoras y extensiones posibles: integrar un brazo manipulador real (UR3, Franka Emika Panda...), mejorar el planificador con aprendizaje por refuerzo (el robot aprende qué habitaciones visitar primero según su historial de éxitos/fallos), probar el sistema en un TurtleBot3 físico (no solo simulación), añadir reconocimiento de personas para que el robot evite molestarlas, implementar síntesis de voz (Text-To-Speech (TTS)) para que el robot hable y avise “he encontrado la coca cola”, añadir una interfaz gráfica web para que usuarios no técnicos puedan enviar órdenes desde el móvil...
-

2 Estado del Arte

En este capítulo se habla sobre las bases teóricas y tecnológicas sobre las que se construye el proyecto. Primero, en la Sección 2.1, se hace una introducción general al contexto. Después, en la Sección 2.2, se profundiza en la robótica asistencial: qué robots existen hoy en día, cómo navegan, cómo mapean el entorno, y qué trabajos previos son importantes. Después, la Sección 2.3, se centra en la inteligencia artificial: arquitecturas de redes neuronales, modelos de visión por computador, reconocimiento de voz, y modelos de lenguaje. Finalmente, en la Sección 2.4, se hace un resumen en el que se justifican las decisiones tecnológicas tomadas.

2.1 Introducción

Durante las últimas décadas, dos campos tecnológicos han crecido exponencialmente: la robótica y la inteligencia artificial. Por separado, ambas disciplinas ya son revolucionarias. La robótica ha permitido automatizar procesos industriales, explorar entornos hostiles (desde el fondo marino hasta Marte), y comenzar a asistir en tareas médicas de alta precisión. La inteligencia artificial ha demostrado capacidades que hace una década parecían imposibles de alcanzar, reconocer caras en fotografías, transcribir conversaciones, traducir idiomas en tiempo real e incluso mantener conversaciones coherentes con seres humanos.

Pero lo que nos interesa en este proyecto es la interacción de estas dos. Un robot sin inteligencia es una máquina que solo puede seguir instrucciones preprogramadas en entornos controlados. Una inteligencia artificial sin cuerpo físico es poderosa, pero limitada al mundo digital. Juntando ambas, se pueden crear sistemas que perciben el mundo físico, razonan sobre él, y actúan de forma autónoma para resolver problemas reales.

Este TFG se sitúa precisamente en esa intersección: desarrollar un asistente robótico inteligente capaz de ayudar a personas con movilidad reducida. Para entender el alcance del proyecto y justificar las decisiones tecnológicas tomadas, es fundamental revisar primero el estado del arte en ambos campos.

2.2 Robótica Asistencial

La robótica asistencial es una rama de la robótica que se centra en diseñar sistemas capaces de asistir a personas en su vida diaria. A diferencia de los robots industriales (que trabajan en fábricas, repitiendo las mismas tareas en entornos controlados), los robots asistenciales tienen que operar en espacios compartidos con humanos: casas, hospitales, residencias de ancianos y en cualquier lugar que sea necesaria su ayuda. Lo que puede provocar muchos problemas imprevistos para los que el robot tiene que poder brindar una solución.

Esto implica algunos desafíos:

- Entornos dinámicos e impredecibles: Una fábrica es estática. Una casa no, hay personas moviéndose, puertas que se abren y cierran, objetos que cambian de sitio.

- Seguridad: Si un robot industrial se choca con algo, probablemente rompa la pieza que está manipulando. Si un robot asistencial se choca con una persona, puede causarle daño físico.
- Interacción natural: Nadie espera que un usuario sin conocimientos técnicos programe un robot. Tiene que entender lenguaje normal, gestos o expresiones faciales. Y poder trabajar en base a ello.

La OMS estima que más de mil millones de personas viven con alguna discapacidad [25], y ese número sigue creciendo por el envejecimiento poblacional. En España, el INE reporta que el 86,2% de las personas con discapacidad en centros residenciales tienen problemas de movilidad [8]. Ahí está el mercado potencial: millones de personas que necesitan ayuda para tareas cotidianas que el resto de personas dan por sentadas.

2.2.1 Ejemplos de robots asistenciales

A lo largo de los años han surgido varios proyectos destacados en este campo. Aquí se mencionan algunos de los más influyentes:

2.2.1.1 Care-O-bot (Fraunhofer IPA, Alemania)

El Care-O-bot [22] es probablemente uno de los robots asistenciales más emblemáticos de Europa. Desarrollado por el Instituto Fraunhofer desde finales de los años 90, ha pasado por varias generaciones (actualmente van por la versión 4). Es un robot móvil con un brazo manipulador capaz de recoger objetos, abrir puertas, y servir bebidas.

Lo interesante del Care-O-bot es que fue diseñado explícitamente para entornos domésticos. Tiene sensores por todas partes (láseres, cámaras, sensores de fuerza en el gripper) y una pantalla táctil para interactuar con usuarios. Su principal problema es el coste. Las primeras versiones costaban más de 200.000 euros, y aunque las últimas han bajado ese precio, sigue siendo inaccesible para el ciudadano medio. Es más una plataforma de investigación que un producto comercial.

2.2.1.2 PR2 (Willow Garage, EE.UU.)

El Personal Robot 2 (PR2) [4] fue desarrollado por Willow Garage, la misma empresa que creó ROS. Es un robot de investigación de alta gama con dos brazos manipuladores de 7 grados de libertad cada uno, una base móvil omnidireccional, y una amplia cantidad de sensores (cámaras estéreo, láseres 2D y 3D, IMUs...).

El PR2 fue revolucionario porque Willow Garage no solo lo desarrolló, sino que *regaló* 11 unidades a laboratorios de investigación de todo el mundo en 2010 para acelerar la innovación en robótica. Esto llevó a cientos de publicaciones científicas sobre manipulación de objetos, navegación, interacción humano-robot y más aspectos de este campo.

Pero, de nuevo, el coste era prohibitivo: unos 400.000 dólares por unidad. Willow Garage cerró en 2014, y aunque el PR2 dejó un legado enorme (especialmente en el ecosistema de ROS), nunca fue una solución viable para uso doméstico real.

2.2.1.3 TIAGo (PAL Robotics, España)

TIAGo [18] es un robot de servicio desarrollado por PAL Robotics. A diferencia del Care-O-bot o el PR2, TIAGo está pensado desde el principio para ser modular y relativamente asequible. Tiene versiones con uno o dos brazos, con o sin cabeza (para expresiones faciales), y se puede configurar según las necesidades del cliente.

Es especialmente popular en investigación europea porque PAL Robotics ofrece soporte activo, documentación extensa, y compatibilidad total con ROS. Se usa en hospitales para tareas de logística (llevar medicamentos, recoger material quirúrgico), en residencias de ancianos como compañía social, y en laboratorios de investigación para experimentos de manipulación y navegación.

Su precio ronda los 50.000-80.000 euros dependiendo de la configuración, lo que sigue siendo muy caro para uso doméstico pero accesible para instituciones.

2.2.1.4 Pepper y NAO (SoftBank Robotics, Japón/Francia)

Pepper [19] es un robot humanoide social diseñado para interactuar con personas: tiene una tablet en el pecho, reconoce emociones a través de expresiones faciales y habla múltiples idiomas.

Se utiliza sobre todo en comercios (para dar información a clientes), hoteles (como recepcionista), y residencias de ancianos (como compañía). No está diseñado para manipular objetos pesados (sus brazos son más expresivos que funcionales), pero destaca en tareas de comunicación.

Su hermano pequeño, NAO, es un humanoide bípedo de 58 cm usado principalmente en educación e investigación. Camina, tiene cámaras, micrófonos, y una gran cantidad de servomotores. Es uno de los robots humanoides más usados en universidades porque SoftBank ofrece precios educativos razonables (aunque sigue costando unos 8.000-10.000 euros).

2.2.1.5 Stretch (Hello Robot, EE.UU.)

Stretch [9] es un caso interesante porque rompe con el paradigma de "robot carísimo para laboratorios". Hello Robot lo diseñó explícitamente para ser asequible (unos 25.000 dólares), ligero, y fácil de usar. Es básicamente un palo telescópico con un gripper en la punta, montado sobre una base móvil cilíndrica.

No es bonito, no tiene aspecto humanoide, pero funciona. Puede alcanzar objetos en el suelo o en estanterías altas (gracias al brazo telescópico), navegar por casas, y se programa completamente en Python con ROS. Está ganando tracción en investigación académica precisamente porque democratiza el acceso a robots manipuladores móviles.

2.2.2 Navegación autónoma

Un robot asistencial tiene que ser capaz de moverse por una casa sin chocarse con muebles, personas, o mascotas. Esto se llama navegación autónoma, y es un problema que tiene varias capas:

2.2.2.1 Localización

Para poder moverse de forma autónoma, el robot necesita conocer su posición en todo momento. La técnica más básica es la odometría, donde los encoders de las ruedas miden cuánto han girado para estimar la posición. Sin embargo, los encoders sufren de deriva acumulativa: si el robot se desliza aunque sea ligeramente por un suelo resbaladizo o una rueda patina, el error crece con el tiempo hasta que la posición estimada difiere significativamente de la real.

Para corregir esta deriva, el estándar en ROS es AMCL. Este algoritmo requiere un mapa previo del entorno y un sensor láser 2D. Su funcionamiento es elegante: el robot lanza miles de partículas que representan hipótesis de dónde podría estar, compara lo que observa el láser con lo que debería ver según el mapa en cada posición hipotética, y va descartando las partículas incorrectas. Con el tiempo, todas las partículas convergen hacia la posición real, proporcionando una estimación robusta incluso cuando el robot se mueve.

Una alternativa más moderna es la Visual SLAM (VSLAM), que sustituye el láser por cámaras. Estos sistemas identifican características visuales como esquinas, bordes y texturas, y las rastrean entre fotogramas consecutivos. Ejemplos destacados incluyen ORB-SLAM [15] y RTAB-Map [11]. Aunque las cámaras son más económicas que los láseres, presentan el inconveniente de ser muy sensibles a cambios en las condiciones de iluminación.

2.2.2.2 Mapeado

Cuando el robot opera en un entorno sin mapa previo, debe construirlo simultáneamente mientras se localiza. Este problema se conoce como SLAM.

Una solución clásica en ROS 1 es Gmapping, que utiliza un láser 2D para construir mapas bidimensionales en forma de ocupancy grids, donde cada celda puede estar libre, ocupada, o en estado desconocido. Aunque funciona bien en entornos pequeños, tiene limitaciones en espacios grandes debido a la acumulación de errores.

Para superar estas limitaciones, Google desarrolló Cartographer [7], que actualmente es el algoritmo más utilizado en ROS 2. Su ventaja principal radica en el uso de optimización de grafos para corregir errores acumulados cuando el robot cierra un bucle, es decir, cuando vuelve a un lugar que ya había visitado. Esto le permite manejar mapas grandes e incluso entornos de múltiples plantas con precisión notable.

Existen también enfoques de SLAM visual, como ORB-SLAM y LSD-SLAM, que construyen mapas tridimensionales utilizando únicamente cámaras. Aunque impresionantes técnicamente, estos sistemas requieren una potencia computacional considerable y mantienen la sensibilidad a las condiciones de iluminación característica de los sensores visuales.

2.2.2.3 Planificación de rutas

Con la localización resuelta y un mapa disponible, el siguiente desafío es calcular cómo llegar del punto A al punto B. Los algoritmos clásicos de búsqueda en grafos como Dijkstra y A* son fundamentales aquí. A* resulta particularmente eficiente porque emplea una heurística (típicamente la distancia en línea recta al objetivo) que le permite explorar primero las direcciones más prometedoras, reduciendo significativamente el espacio de búsqueda.

Una alternativa conceptualmente diferente es Rapidly-exploring Random Tree (RRT), que en lugar de realizar búsquedas exhaustivas, va lanzando ramas aleatorias desde el punto de

inicio hasta que alguna alcanza el objetivo. Este enfoque es especialmente valioso en robótica porque funciona bien en espacios de alta dimensionalidad, donde los métodos de búsqueda tradicionales se vuelven intratables.

En la práctica, los sistemas modernos distinguen entre planificación global y local. El planificador global calcula la ruta óptima completa usando el mapa estático, mientras que el planificador local se ocupa de los últimos metros: esquivar obstáculos dinámicos como personas o mascotas, ajusta la velocidad según las condiciones, y reacciona ante imprevistos en tiempo real. En ROS, el planificador local más extendido es DWB, que equilibra eficacia computacional con capacidad de reacción.

2.2.2.4 Sistemas anticolidión

Incluso con un plan de ruta óptimo, el robot debe ser capaz de reaccionar en milisegundos a obstáculos que no estaban presentes en el mapa original. Una técnica simple y computacionalmente económica son los campos potenciales, donde los obstáculos ejercen una fuerza repulsiva sobre el robot mientras que el objetivo lo atrae. El robot simplemente sigue el gradiente del campo resultante. Aunque rápida, esta aproximación puede quedar atrapada en mínimos locales, situaciones donde el robot se paraliza sin poder avanzar hacia el objetivo.

El Dynamic Window Approach (DWA) ofrece una solución más robusta. Este algoritmo simula trayectorias cortas (típicamente de uno o dos segundos) explorando diferentes combinaciones de velocidad lineal y angular, descarta aquellas que resultan en colisión, y selecciona la mejor según múltiples criterios como rapidez, suavidad del movimiento, y alineación con el objetivo. La ventaja de este enfoque es que considera explícitamente las restricciones dinámicas del robot.

Para aplicaciones más exigentes existe el Model Predictive Control (MPC), conceptualmente similar a DWA pero más sofisticado. MPC optimiza sobre una ventana temporal más amplia teniendo en cuenta todas las restricciones físicas del robot: aceleración máxima, velocidad máxima de giro, inercia, etc. El precio de esta precisión es un coste computacional considerablemente mayor.

2.2.3 La plataforma TurtleBot: el estándar de facto en investigación

El TurtleBot [2] merece una mención especial porque se ha convertido en el robot móvil más utilizado tanto en educación como en investigación robótica. Su historia comenzó en 2010 como fruto de una colaboración entre Willow Garage (creadores de ROS) e iRobot (fabricantes del Roomba). La idea original era simple pero brillante: tomar la base de un Roomba, añadirle un láser, una cámara y un ordenador, y comercializarlo como un kit de desarrollo asequible.

La plataforma ha evolucionado a través de tres generaciones. El TurtleBot 1 utilizaba una base iRobot Create con un sensor Kinect de Microsoft, pero tanto este como su sucesor, el TurtleBot 2, han quedado discontinuados. El salto cualitativo llegó con el TurtleBot 3, rediseñado completamente desde cero por la empresa ROBOTIS. Este nuevo modelo es más pequeño, más económico (con kits desde aproximadamente 1.000 euros), y completamente modular. Emplea servomotores Dynamixel de ROBOTIS y se ofrece en dos variantes: Burger, la configuración básica, y Waffle Pi, que incorpora sensores adicionales.

Su dominio en el ámbito académico se sostiene en cuatro ventajas principales:

- Integración total con ROS: Todo funciona out-of-the-box. Se instala ROS, se descargan los paquetes del TurtleBot, y en 10 minutos el robot se está moviendo.
- Simulación perfecta: Los modelos de Gazebo del TurtleBot3 son extremadamente precisos. El código que escribes en simulación funciona casi idéntico en el robot real.
- Comunidad gigante: Miles de tutoriales, cientos de proyectos públicos en GitHub, foros activos. Si tienes un problema, alguien ya lo tuvo antes y lo documentó.
- Precio accesible: 1.000-1.500 euros es caro para un particular, pero barato para una universidad. Compara con los 50.000+ euros de un TIAGo o los 400.000 del PR2.

Por eso, cuando alguien quiere probar algoritmos de navegación, IA aplicada a robótica, o simplemente aprender ROS, el TurtleBot3 es la primera opción.

2.3 Inteligencia Artificial Aplicada

La IA moderna se basa en el Machine Learning (ML), y dentro del ML, el DL ha sido el gran protagonista de la última década. El DL usa redes neuronales artificiales con muchas capas para aprender representaciones jerárquicas de los datos.

A continuación se repasan las arquitecturas clave que son relevantes para este proyecto:

2.3.1 Redes Neuronales Convolucionales (CNN)

Las CNN [12] dominan el campo de la visión por computador gracias a su diseño específico para procesar imágenes. Su arquitectura aprovecha las convoluciones, donde filtros pequeños (3x3 o 5x5) se deslizan por la imagen detectando patrones locales. Las primeras capas identifican bordes y texturas, mientras que las capas profundas reconocen estructuras complejas como objetos completos. El pooling reduce progresivamente la resolución, mejorando la eficiencia y proporcionando invariancia ante traslaciones.

La evolución arquitectónica incluye hitos como AlexNet (2012), que popularizó el DL al ganar ImageNet, y ResNet (2015), cuyas conexiones residuales permitieron entrenar redes con más de 100 capas sin degradación del gradiente. Actualmente, EfficientNet (2019) optimiza simultáneamente profundidad, anchura y resolución para maximizar precisión con recursos mínimos.

2.3.2 Transformers: la revolución del PLN

Los Transformers también han invadido visión: Vision Transformers [5] dividen las imágenes en parches y los tratan como tokens, logrando resultados comparables a CNNs pero con arquitecturas más elegantes.

Los Transformers [24] (2017) revolucionaron el Procesamiento de Lenguaje Natural (PLN) mediante mecanismos de atención que permiten procesar secuencias completas simultáneamente, superando las limitaciones secuenciales de las RNN. Esta arquitectura es más rápida de entrenar, captura mejor las dependencias de largo alcance, y escala eficientemente a grandes volúmenes de datos.

BERT (Google, 2018) revolucionó la comprensión de lenguaje mediante entrenamiento bidireccional en 3.3 mil millones de palabras. La familia GPT de OpenAI demostró capacidades de generación: GPT-3 (175B parámetros) exhibió capacidades emergentes sin entrenamiento específico, mientras que GPT-4 (2023) introdujo procesamiento multimodal de texto e imágenes. Los Vision Transformers [5] han extendido esta arquitectura a visión artificial, dividiendo imágenes en parches y logrando resultados comparables a las CNN.

2.3.3 Detección de objetos en tiempo real

En robótica, no basta con clasificar imágenes ("esta foto contiene un gato"). Es necesario localizar objetos: "hay un gato en las coordenadas (120, 340) con tamaño 80x60 píxeles". Esto se llama detección de objetos.

2.3.3.1 YOLO: detección en una sola pasada

YOLO [21] cambió el paradigma al realizar detección en una única pasada de la red. Divide la imagen en cuadrícula, donde cada celda predice bounding boxes con coordenadas, confianza y probabilidades de clase. La versión original alcanzaba 45 FPS, mientras que YOLOv3 (2018) superó los 140 FPS mediante detección multi-escala. YOLOv8 (2023, Ultralytics) domina actualmente la producción por su equilibrio óptimo entre velocidad y precisión, junto con una API intuitiva. Aunque menos preciso que Faster R-CNN en objetos muy pequeños, YOLO es la elección estándar cuando la velocidad de procesamiento es crítica.

2.3.3.2 Alternativas a YOLO

- Single Shot MultiBox Detector (SSD): Similar a YOLO en filosofía (una sola pasada), pero usa múltiples capas de la Convolutional Neural Network (CNN) para detectar objetos a diferentes escalas. Más preciso que YOLOv1 pero más lento.
- RetinaNet: Usa Focal Loss para manejar el desbalance entre clases. Muy preciso, pero más lento que YOLO.
- EfficientDet: Versión optimizada de RetinaNet usando arquitectura EfficientNet. Mejor trade-off precisión/velocidad que otros detectores, pero sigue siendo más lento que YOLOv5/v8.
- Detection Transformer (DETR): Usa Transformers en vez de CNNs. Elegante conceptualmente, pero aún no es tan rápido como YOLO en práctica.

2.3.4 Reconocimiento Automático de Voz (ASR)

Los sistemas ASR convierten audio en texto. Antes del DL, requerían arquitecturas complejas con múltiples componentes separados (modelos acústicos, de lenguaje y diccionarios fonéticos). Las redes neuronales simplificaron esto radicalmente con sistemas end-to-end. Conformer [6] (Google, 2020) combina CNNs y Transformers estableciendo el estado del arte en precisión.

2.3.4.1 Whisper (OpenAI)

Whisper [20] (2022) es el modelo ASR más impresionante hasta la fecha. Es un Transformer encoder-decoder entrenado en 680.000 horas de audio multilingüe recolectado de internet (videos de YouTube con subtítulos, podcasts, audiolibros...). Comprende 99 idiomas, mantiene robustez ante ruido y puede transcribir, traducir y detectar idiomas automáticamente. OpenAI liberó el código y múltiples versiones de tamaño variable. El modelo "base" (74M parámetros) funciona en CPU, transcribiendo un minuto de audio en 5-10 segundos con precisión comparable a servicios comerciales pero sin costes ni envío de datos a la nube. Esto lo ha convertido en el estándar para proyectos que requieren privacidad y procesamiento local.

2.3.4.2 Alternativas a Whisper

- Wav2Vec 2.0 (Meta): Transformer que aprende representaciones de audio de forma auto-supervisada. Luego se fine-tunea con datos etiquetados. Muy preciso, pero requiere más recursos computacionales que Whisper.
- Vosk: Motor ASR open-source ligero que corre offline en dispositivos con recursos limitados (móviles, Raspberry Pi). Menos preciso que Whisper, pero más rápido.
- Servicios en la nube: Google Cloud Speech-to-Text, AWS Transcribe, Azure Speech Services. Muy precisos y con características extra (diarización de hablantes, detección de emociones...), pero requieren conexión a internet y tienen coste económico.

2.3.5 Modelos de Lenguaje Grande (LLM)

Los LLMs son Transformers entrenados en grandes volúmenes de texto para predecir la siguiente palabra, desarrollando capacidades emergentes sorprendentes. GPT-3 (OpenAI, 2020) con 175B parámetros demostró poder realizar tareas nunca enseñadas explícitamente. GPT-3.5 (2022), refinado mediante RLHF, potencia ChatGPT gratuito con mejor seguimiento de instrucciones. GPT-4 (2023) añadió capacidades multimodales y razonamiento mejorado.

El ecosistema open-source ha evolucionado rápidamente. LLaMA (Meta, 2023) y sus sucesores LLaMA 2 y LLaMA 3 ofrecen rendimiento comparable a GPT-3 con menos parámetros y licencia comercial. Mistral (7B parámetros) y Mixtral (47B parámetros efectivos mediante Mixture-of-Experts) rivalizan con GPT-3.5 siendo ejecutables localmente.

2.3.5.1 Ollama: ejecución local simplificada

Ollama [16] simplifica la ejecución de LLMs localmente mediante un servidor HTTP (localhost:11434) que gestiona automáticamente la descarga y configuración de docenas de modelos. Permite desarrollo offline, garantiza privacidad absoluta y elimina costes recurrentes. Un LLaMA 3 de 8B en CPU genera respuestas en 10-15 segundos, frente a 1-2 segundos de GPT-3.5 vía API, trade-off aceptable en robótica considerando las ventajas de privacidad y autonomía.

2.4 Síntesis y Justificación Tecnológica

Después de este recorrido por el estado del arte, se pueden extraer algunas conclusiones que justifican las decisiones tecnológicas de este proyecto:

2.4.1 La problemática en la robótica asistencial

Los robots asistenciales avanzados (Care-O-bot, PR2, TIAGo) demuestran que técnicamente es posible crear sistemas que ayuden en el hogar. Pero tienen dos problemas:

1. Coste prohibitivo: 50.000-400.000 euros los hace inaccesibles para uso doméstico real.
2. Complejidad: Requieren mantenimiento especializado, ocupan mucho espacio, y no están diseñados para usuarios sin conocimientos técnicos.

Por otro lado, las plataformas de investigación como el TurtleBot3 son accesibles y bien documentadas, pero les falta la capacidad de manipulación y la interfaz de usuario natural que necesita una persona con movilidad reducida.

2.4.2 Detección de objetos: YOLO como opción pragmática

Hay detectores más precisos que YOLO (Faster R-CNN, EfficientDet), pero para aplicaciones robóticas en tiempo real, la velocidad importa. Un robot que tarda 2 segundos en procesar cada frame de su cámara es esencialmente ciego. YOLO (especialmente YOLOv8) ofrece el mejor equilibrio: 30-60 FPS en CPU, precisión suficiente para objetos domésticos comunes, y una API sencilla (Ultralytics) que funciona out-of-the-box con Python y ROS.

2.4.3 Reconocimiento de voz: Whisper como estándar

Antes de Whisper, usar ASR en proyectos académicos implicaba o pagar APIs (Google, AWS) o lidiar con modelos open-source mediocres. Whisper cambió eso: calidad comercial, gratis, offline, multilingüe. Es simplemente la opción obvia para cualquier proyecto que necesite entender voz humana sin enviar datos a la nube.

2.4.4 Arquitectura híbrida con LLMs

Los LLMs resultan extraordinariamente útiles para interpretar lenguaje natural, extrayendo estructura semántica de órdenes como "trae la coca cola de la cocina" (objeto: "coca cola", habitación: "cocina"). Sin embargo, presentan dos desafíos en robótica: aunque GPT-3.5 responde en 1-2 segundos, requiere conexión a internet constante, y las APIs comerciales cobran por token, lo cual no es viable a futuro.

La solución adoptada es una arquitectura híbrida con degradación elegante en tres niveles. Como primera opción, se emplea un LLM en la nube (Generative Pre-trained Transformer (GPT)-3.5) para máxima precisión y velocidad. Si la conexión falla o se agota la cuota, el sistema recurre a un LLM local (LLaMA 3 vía Ollama) sacrificando algo de velocidad pero manteniendo capacidad completa. Como último recurso, un modelo probabilístico simple basado en Naive Bayes y reglas heurísticas garantiza funcionalidad básica incluso sin recursos computacionales para ejecutar modelos grandes. Esta estrategia asegura que el sistema siempre funciona, aunque con distinto nivel de sofisticación según las condiciones.

2.4.5 Simulación vs Hardware Real: la estrategia sim-to-real

La decisión de trabajar en simulación mediante Gazebo en lugar de directamente sobre hardware físico se justifica tanto económica como metodológicamente. El coste es nulo frente a los 1.500 euros de un TurtleBot3 básico. La iteración es rápida: reiniciar un experimento toma segundos en lugar de los minutos necesarios para que el robot físico retorne a su posición inicial. La reproducibilidad es perfecta, ya que cada experimento parte de condiciones idénticas, mientras que en hardware real la carga de batería varía, las ruedas acumulan polvo y la iluminación cambia. Finalmente, no existe riesgo de daño físico: el código experimental puede probarse libremente sin temor a colisiones o componentes rotos.

Lo crucial es que el código desarrollado en simulación con ROS 2 es casi directamente transferible a hardware real. Los topics, mensajes y estructura de nodos son idénticos; solo los drivers de bajo nivel cambiarían, sustituyendo sensores virtuales por físicos y comandos simulados por señales a motores reales. Este enfoque "desarrollar en simulación, validar en hardware" constituye el estándar en investigación robótica moderna y ha sido validado exitosamente en innumerables proyectos.

3 Metodología

Este capítulo describe el enfoque metodológico seguido durante el desarrollo del proyecto y detalla las especificaciones técnicas del hardware y software empleados. Primero, en la Sección 3.1, se explica la estrategia de trabajo que guió el desarrollo. Después, en la Sección 3.2, se describe el hardware utilizado (tanto el equipo de desarrollo como la plataforma robótica simulada). Finalmente, en la Sección 3.3, se detalla el ecosistema de software que hizo posible la implementación: desde el sistema operativo hasta el simulador físico.

3.1 Enfoque Metodológico

Para la consecución de los objetivos planteados en este TFG, se optó por una metodología de desarrollo iterativa e incremental. Este enfoque es especialmente adecuado para proyectos de robótica, donde la interacción entre software y entorno físico (o simulado) introduce incertidumbre y requiere ajustes continuos.

3.1.1 Desarrollo iterativo e incremental

Un modelo de desarrollo en cascada tradicional resultaba inviable en este contexto. En robótica, diseñar todo el sistema sobre papel e implementarlo de golpe raramente funciona. La realidad presenta problemas imprevistos: algoritmos de navegación que fallan en esquinas, detectores de objetos que se vuelven ciegos con cambios de iluminación, o modelos de lenguaje que malinterpretan frases coloquiales.

El desarrollo se organizó en ciclos cortos de diseño-implementación-prueba-corrección. El sistema se dividió en componentes independientes (navegación, detección visual, interpretación de lenguaje, orquestación de misiones) con interfaces que permitían desarrollo y pruebas por separado. La implementación fue incremental: primero el movimiento básico punto a punto, luego búsqueda de objetos durante el desplazamiento, después comprensión de órdenes habladas, y finalmente retorno al origen.

Cada funcionalidad se validaba exhaustivamente en Gazebo antes de integrarse con el resto del sistema, permitiendo detectar errores rápidamente sin depurar miles de líneas de código. Una vez validado aisladamente, cada módulo se conectaba mediante la arquitectura de nodos y topics de ROS 2. Paralelamente, se mantuvo un cuaderno de bitácora documentando decisiones y resultados diarios, facilitando posteriormente la redacción de esta memoria.

Este modelo iterativo demostró ventajas prácticas significativas. Proporcionó tolerancia a cambios: cuando se descubrió la inestabilidad de ROS 2 Jazzy y Gazebo Sim, fue posible migrar a Humble y Gazebo Classic sin descartar el trabajo previo, dado que los módulos de alto nivel eran independientes de la infraestructura de simulación. Los ciclos cortos permitieron detección temprana de problemas, como identificar que el planificador de Nav2 se bloqueaba en esquinas estrechas, ajustando parámetros del costmap antes de integrar la detección visual.

Los prototipos funcionales facilitaron feedback frecuente de los tutores, crucial para ajustar alcance y priorizar funcionalidades. Finalmente, observar el robot funcionando desde etapas tempranas resultó motivador, manteniendo el impulso del proyecto.

3.1.2 Arquitectura híbrida: edge + cloud

Desde el punto de vista del diseño del sistema, se adoptó una arquitectura híbrida que combina procesamiento local (edge computing) con servicios en la nube (cloud computing). Todo lo que requiere respuesta en tiempo real (navegación, lectura de sensores, detección de objetos, control de motores) corre localmente en el contenedor Docker, garantizando baja latencia y funcionamiento independiente de conectividad a internet. Las tareas de razonamiento complejo no críticas en tiempo real (interpretación de lenguaje natural) se delegan a un LLM externo vía Application Programming Interface (API), permitiendo usar modelos más potentes de los que podrían ejecutarse localmente.

El sistema implementa degradación elegante: si la API en la nube falla, recurre a un modelo local más simple (Ollama + LLaMA 3); si este también falla, emplea reglas heurísticas. Esto garantiza funcionalidad continua, aunque con capacidades reducidas. Esta arquitectura es representativa de sistemas robóticos reales: drones que procesan video localmente pero consultan servidores para planificación compleja, o robots de almacén que toman decisiones de navegación en milisegundos localmente pero reportan su estado a un coordinador central en la nube.

3.1.3 Control de versiones

El proyecto se gestionó mediante Git como sistema de control de versiones, con repositorio alojado en GitHub (github.com/mca119-ua/TFG), manteniendo historial completo de cambios, permitiendo trabajar en ramas separadas sin afectar la versión estable, y proporcionando un backup automático. Como entorno de desarrollo integrado se utilizó Visual Studio Code (versión 1.118.1) con la extensión Dev Containers, que permite desarrollar directamente dentro del contenedor Docker, simplificando enormemente el flujo de trabajo con acceso completo al código, terminales integradas, debugger de Python y extensiones de ROS 2.

3.2 Hardware Empleado

3.2.1 Equipo de desarrollo

Todo el desarrollo del proyecto se realizó en un ordenador portátil personal cuyas especificaciones se resumen en la Tabla 3.1.

Aunque el portátil dispone de GPU dedicada NVIDIA GTX 1650, esta no se utilizó en el proyecto. Toda la inferencia de modelos de IA (detección de objetos, transcripción de audio) se ejecutó en CPU dentro del contenedor Docker por razones de compatibilidad y simplicidad de configuración. Los modelos elegidos (YOLOv8s, Whisper base) están optimizados para correr eficientemente en CPU. Esta decisión demuestra además que el sistema es viable en hardware sin aceleración por GPU, más representativo de robots reales de bajo coste.

Tabla 3.1: Especificaciones del equipo de desarrollo

Componente	Especificación
Modelo	ASUS TUF Gaming F15 FX506LHBP
Procesador	Intel Core i5-10300H @ 2.50 GHz (4 núcleos físicos, 8 hilos con Hyper-Threading)
Memoria RAM	16 GB
Almacenamiento	SSD NVMe 476 GB (Micron 2210 MTFDHBA512QFD)
Graphics Processing Unit (GPU) dedicada	NVIDIA GeForce GTX 1650 Mobile/Max-Q (4 GB GDDR6)
GPU integrada	Intel UHD Graphics (CometLake-H GT2)
Sistema operativo	Ubuntu 24.04.4 LTS (instalación nativa)

3.2.2 Plataforma robótica: TurtleBot3 Waffle Pi

Aunque el proyecto se desarrolló completamente en simulación, se eligió el TurtleBot3 Waffle Pi [23] como plataforma de referencia, modelando fielmente todos sus sensores y características físicas en Gazebo. Las especificaciones se presentan en la Tabla 3.2.

Tabla 3.2: Especificaciones del TurtleBot3 Waffle Pi (simulación)

Componente	Especificación
Modelo	TurtleBot3 Waffle Pi
LiDAR	360° / 360 muestras / rango 0.12–3.5 m Resolución 0.015 m / frecuencia 5 Hz
Cámara RGB	640×480 px / FOV horizontal ~62° Frecuencia: 30 FPS
Modificación	Cámara elevada a z=1.2 m (original: z≈0.3 m)

La configuración estándar del TurtleBot3 sitúa la cámara a aproximadamente 30 cm del suelo, adecuada para navegación pero problemática para detectar objetos sobre superficies elevadas (mesas, encimeras). Se modificó el archivo Simulation Description Format (SDF) del robot elevando la cámara a 1.2 metros (altura aproximada de una mesa), permitiendo que el campo de visión cubriera tanto objetos en el suelo como sobre superficies típicas de una casa, mejorando significativamente la tasa de detección.

Esta plataforma se eligió por ser estándar en educación e investigación, garantizando documentación exhaustiva y compatibilidad total con ROS 2. ROBOTIS proporciona archivos Unified Robot Description Format (URDF) y SDF oficiales que modelan fielmente la física del robot, asegurando que el comportamiento en Gazebo sea representativo del robot real. El código desarrollado es casi directamente transferible al robot físico, requiriendo solo cambiar la lectura de sensores desde hardware en lugar de topics simulados. Finalmente, validar la arquitectura sin inversión en hardware permitió iterar rápidamente y probar escenarios peligrosos sin consecuencias.

3.3 Software Empleado

3.3.1 Sistema Operativo

El desarrollo se realizó sobre dos sistemas operativos en configuración anidada: Ubuntu 24.04.4 Long Term Support (LTS) como sistema host (instalación nativa), y Ubuntu 22.04 LTS dentro del contenedor Docker. Esta aparente redundancia responde a requisitos de compatibilidad: ROS 2 Humble está diseñado específicamente para Ubuntu 22.04, mientras que en el host se utiliza la versión más reciente para aprovechar mejor el hardware moderno. Docker permite esta coexistencia sin conflictos.

3.3.2 Contenedorización: Docker

Para garantizar reproducibilidad y aislar el entorno de desarrollo del sistema operativo principal, se encapsuló ROS 2, Gazebo y las dependencias Python en un contenedor Docker (versión 29.4.2). El contenedor se configuró con volumen compartido (directorio `./src` del host montado en `/workspace`), permitiendo editar código en el host y reflejar cambios instantáneamente en el contenedor. Se configuró X11 forwarding para mostrar Gazebo y RViz en el escritorio del host, y networking en modo host para simplificar la comunicación de ROS 2.

3.3.3 Middleware Robótico: ROS 2 Humble Hawksbill

ROS 2 Humble Hawksbill [13] es la distribución utilizada, una versión LTS con soporte garantizado hasta mayo de 2027. Inicialmente se intentó usar ROS 2 Jazzy por ser más reciente, pero se encontraron paquetes de TurtleBot3 incompletos, documentación escasa e incompatibilidades con Gazebo Sim que causaban crashes frecuentes. Tras dos semanas de intentos, se migró a Humble, que demostró ser mucho más estable: todos los paquetes de TurtleBot3 funcionan correctamente, compatibilidad perfecta con Gazebo Classic 11, documentación madura y comunidad que ha resuelto prácticamente todos los problemas comunes.

El sistema utiliza nodos ROS 2 que se comunican mediante topics para datos continuos y actions para tareas de larga duración con feedback y posibilidad de cancelación.

3.3.4 Simulador Físico: Gazebo Classic 11

Gazebo Classic 11 [10] es el simulador de robots utilizado en el proyecto. Permite probar el robot en un entorno virtual antes de pasar a hardware real.

3.3.4.1 Gazebo Classic vs Gazebo Sim

Existe una versión más moderna llamada Gazebo Sim (antes Ignition Gazebo). Sin embargo, durante el desarrollo este simulador mostró un RTF muy bajo en el hardware utilizado (aproximadamente 0.05, veinte veces más lento que tiempo real), además de crashes frecuentes al cargar modelos complejos y plugins de ROS 2 que no estaban completamente maduros.

Por el contrario, Gazebo Classic 11 mantenía un RTF cercano a 1.0 (tiempo real) con optimizaciones simples, era completamente estable durante sesiones de horas, y tenía integración perfecta con los paquetes de TurtleBot3. Estas ventajas justificaron su elección frente a la versión más moderna.

3.3.4.2 Optimizaciones aplicadas

Para maximizar el rendimiento sin GPU dedicada activa, se aplicaron varias optimizaciones. Se desactivaron las sombras dinámicas mediante el campo <shadows> en el archivo SDF del mundo. Las geometrías complejas de colisión se reemplazaron por primitivas simples como cajas, cilindros y esferas. Finalmente, se redujo la frecuencia del láser de 10 Hz a 5 Hz en el URDF del robot.

Estas optimizaciones mejoraron el RTF de aproximadamente 0.6 a 0.95, casi tiempo real, sin sacrificar la calidad de la simulación para los propósitos del proyecto.

3.3.5 Navegación Autónoma: Nav2

Nav2 [14] (Navigation 2) es el stack de navegación autónoma de ROS 2 utilizado para que el robot se desplace entre waypoints evitando obstáculos.

3.3.5.1 Componentes empleados

Nav2 emplea tres componentes principales que trabajan de forma coordinada. El planificador global calcula la ruta óptima de origen a destino sobre el mapa completo; en este proyecto se utilizó la configuración por defecto NavFn, basada en el algoritmo de Dijkstra. El planificador local o controller (DWB) ejecuta la ruta global en tiempo real, ajustando velocidades para evitar obstáculos dinámicos mediante simulación de trayectorias cortas (uno o dos segundos hacia el futuro) y selección de la óptima según criterios de rapidez, suavidad y alineación con el objetivo. Finalmente, los costmaps representan el entorno como cuadrícula donde cada celda tiene un coste asociado; existen dos versiones: el costmap global representa el mapa completo estático, mientras que el costmap local cubre la región cercana al robot y se actualiza en tiempo real con las lecturas del láser.

3.3.6 Localización: AMCL

AMCL es el algoritmo utilizado para que el robot sepa dónde está dentro del mapa.

Funciona lanzando miles de "partículas", comparando lo que el láser ve con lo que debería ver según el mapa en cada posición hipotética, y descartando las partículas incorrectas. Con el tiempo, todas convergen en la posición real del robot.

AMCL requiere un mapa previo, que en este proyecto se generó mediante SLAM.

3.3.7 Generación del Mapa: Cartographer

Cartographer [7] es el algoritmo de SLAM utilizado para construir el mapa 2D de la casa.

El mapa se generó teleoperando manualmente el robot por toda la casa. El mapa resultante tiene resolución de 5 cm por celda, suficiente para navegación precisa en interiores.

3.3.8 Lenguaje de Programación: Python 3.10

Todo el código de alto nivel se programó en Python 3.10, elegido por su sintaxis clara y rápida de desarrollar, bibliotecas maduras para IA (PyTorch, Ultralytics, OpenAI SDK, Whisper), y soporte nativo en ROS 2 mediante rclpy.

3.3.9 Entorno de Desarrollo: Visual Studio Code

Como entorno de desarrollo se utilizó Visual Studio Code (versión 1.118.1) con la extensión Dev Containers, proporcionando autocompletado de código, debugger integrado para nodos de ROS 2, terminales con acceso directo al contenedor, y extensiones de ROS 2 para visualizar topics y mensajes.

3.3.10 Tabla resumen de software

La Tabla 3.3 resume todas las herramientas de software empleadas en el proyecto.

Tabla 3.3: Resumen del software empleado

Software	Versión
ROS 2	Humble Hawksbill
Gazebo	Classic 11
Ubuntu (contenedor)	22.04 LTS
Ubuntu (host)	24.04.4 LTS
Python	3.10
Docker	29.4.2
Visual Studio Code	1.118.1
Git	2.43.0
Nav2	Versión incluida en ROS 2 Humble
AMCL	Versión incluida en ROS 2 Humble
Cartographer	turtlebot3_cartographer

4 Propuesta e Implementación

Este capítulo describe en detalle la arquitectura del sistema desarrollado, explicando cómo los diferentes módulos interactúan entre sí para lograr el objetivo del proyecto: un asistente robótico capaz de buscar y recuperar objetos mediante órdenes en lenguaje natural.

4.1 Visión General de la Arquitectura

El sistema implementado sigue una arquitectura modular basada en ROS 2, donde cada componente cumple una función específica y se comunica con los demás mediante topics, actions y llamadas a funciones. La Figura 4.1 ilustra la disposición general de los módulos y sus interconexiones.

La arquitectura se organiza en cinco capas principales. La capa de usuario consiste en una interfaz de línea de comandos que permite enviar órdenes por texto o audio. Los nodos ROS 2 personalizados incluyen el centro de mando (orquestador principal que actúa como Action Server), el buscador (coordinador de la misión de búsqueda y navegación), el detector de objetos (módulo de visión artificial basado en YOLOv8) y el cliente del centro de mando para envío de órdenes. Los módulos Python auxiliares complementan la funcionalidad sin requerir ser nodos independientes: el planificador genera el orden de búsqueda por habitaciones usando IA, y el acercador implementa el control de aproximación visual al objeto detectado.

Los servicios externos proporcionan capacidades de inteligencia artificial: Whisper de OpenAI para transcripción de audio ejecutado localmente, la API de OpenAI con GPT-3.5-turbo para interpretación de lenguaje natural, y Ollama con Llama3 como alternativa local para planificación de rutas. Finalmente, la capa base integra el stack Nav2 para navegación autónoma (AMCL, costmaps, planificadores) y el simulador Gazebo 11 con el modelo del TurtleBot3 Waffle Pi.

La Figura 4.2 muestra el grafo real de nodos y topics capturado durante la ejecución del sistema mediante la herramienta `rqt_graph`.

4.2 Máquina de Estados

El flujo de ejecución de una misión completa sigue una máquina de estados bien definida, donde cada transición representa un evento o condición del sistema. La Figura 4.3 muestra todos los estados posibles y sus transiciones.

4.2.1 Descripción de Estados

El sistema transita por varios estados durante la ejecución de una misión. El estado `ESCUCHANDO_AUDIO` se activa cuando el sistema recibe un archivo de audio y lo transcribe

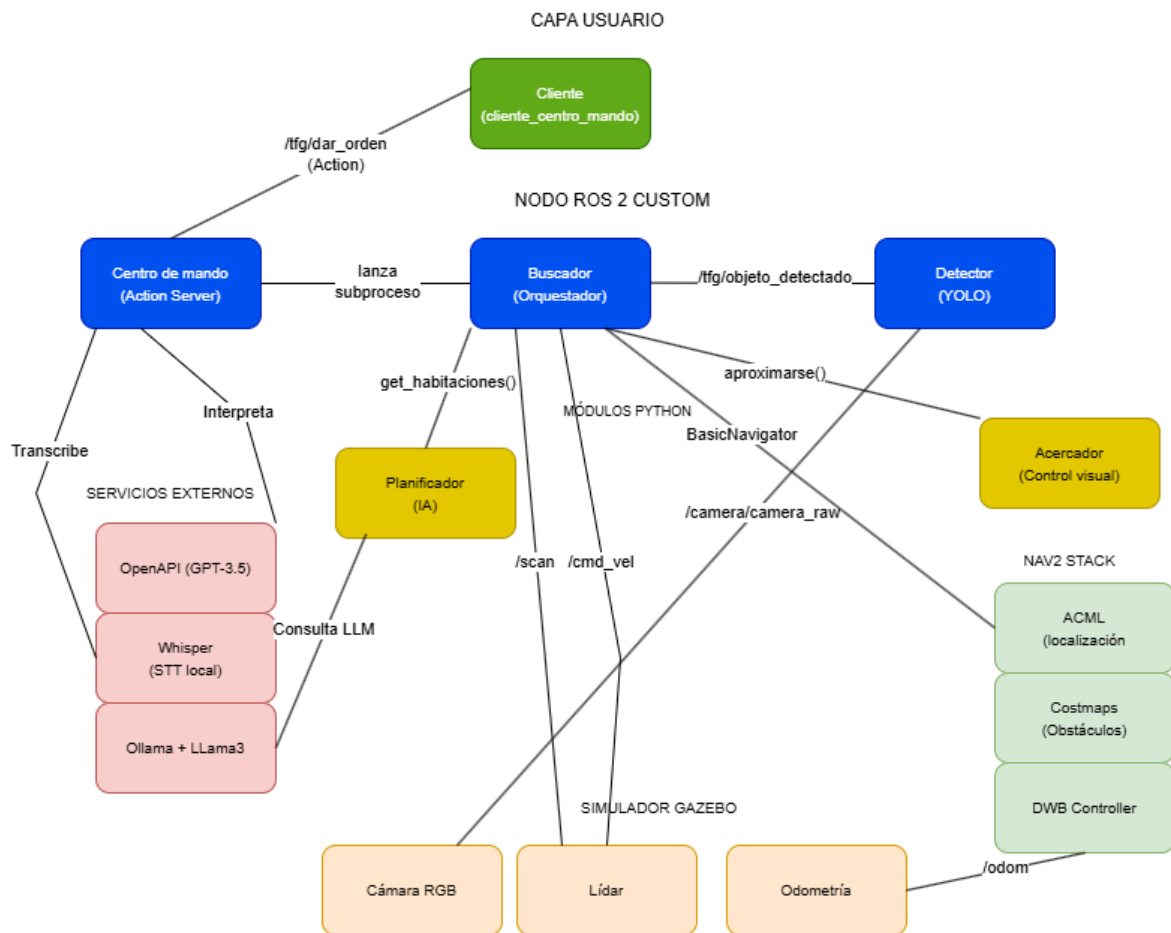


Figura 4.1: Arquitectura general del sistema. Se distinguen cinco capas: la capa de usuario (cliente CLI), los nodos ROS 2 personalizados, los módulos Python auxiliares, los servicios externos (IA) y la capa de simulación Gazebo.

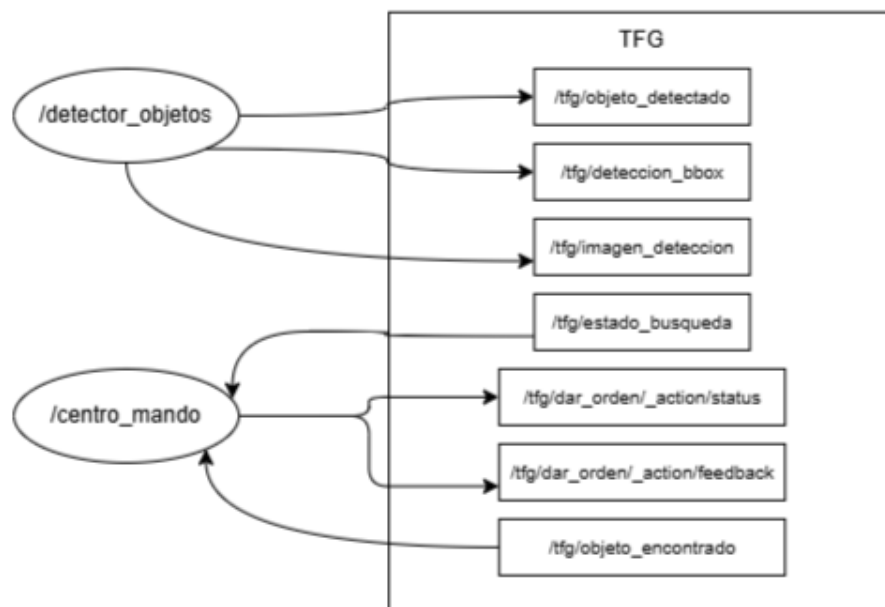


Figura 4.2: Grafo de nodos y topics capturado con `rqt_graph`. Se observan los nodos `/detector_objetos` y `/centro_mando`, así como los topics personalizados bajo el prefijo `/tfg/`.

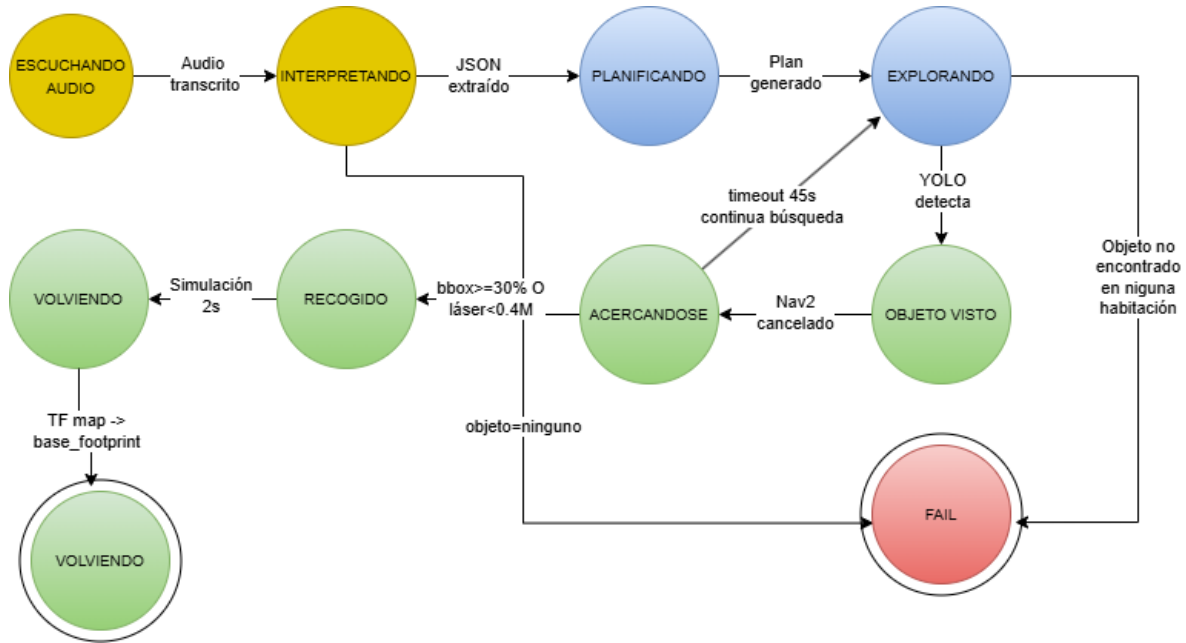


Figura 4.3: Máquina de estados del sistema. Los estados en amarillo corresponden a la fase de interpretación, en azul a navegación, en verde a manipulación y en rojo/verde oscuro a los estados finales.

mediante Whisper. En INTERPRETANDO, el texto o transcripción se envía a GPT-3.5-turbo para extraer el objeto buscado y la habitación sugerida en formato JSON. Durante PLANIFICANDO, el módulo planificador genera una lista ordenada de habitaciones a explorar, priorizando la habitación indicada por el usuario si existe.

El estado EXPLORANDO representa la navegación del robot por los waypoints de cada habitación mientras escucha continuamente las detecciones de YOLO. Cuando YOLO detecta el objeto buscado, se transita a OBJETO_VISTO, cancelando la navegación actual y esperando a que Nav2 libere el control. En ACERCANDOSE, el módulo acercador ajusta la velocidad angular y lineal del robot en función de la posición del bounding box en la imagen.

Una vez alcanzada la distancia objetivo, el sistema pasa a RECOGIDO donde se simula la recogida del objeto mediante una pausa de 2 segundos. El estado VOLVIENDO gestiona la navegación de retorno a la posición inicial, obtenida mediante transformadas TF del frame map al base_footprint. Finalmente, OK indica que la misión finalizó con éxito (objeto encontrado, recogido y robot retornado a la base), mientras que FAIL señala un fallo por objeto no interpretado correctamente, no encontrado en ninguna habitación, o error de navegación.

4.2.2 Transiciones Críticas

Las transiciones más relevantes del sistema requieren condiciones específicas. La transición de EXPLORANDO a OBJETO_VISTO ocurre cuando el callback de `/tfg/objeto_detectado` recibe una clase YOLO que coincide con el objetivo. De ACERCANDOSE a RECOGIDO se transita cuando el bounding box del objeto ocupa al menos el 45 por ciento del ancho de la imagen, o cuando el láser frontal detecta un obstáculo a menos de 0.45 metros con el objeto

todavía visible.

Si el acercador supera el timeout de 60 segundos sin alcanzar la condición de parada, el sistema vuelve de ACERCANDOSE a EXPLORANDO para reanudar la búsqueda en la misma habitación. La transición de EXPLORANDO a FAIL se produce cuando se han recorrido todas las habitaciones sin detectar el objeto, terminando la misión sin éxito.

4.3 Módulo: Centro de Mando

El Centro de Mando es el orquestador principal del sistema. Implementa un Action Server de ROS 2 que recibe órdenes en texto o audio mediante la action `/tfg/dar_orden`, transcribe el audio con Whisper si es necesario, interpreta la orden usando GPT-3.5-turbo para extraer el objeto y la habitación, lanza el nodo buscador como subproceso con los parámetros extraídos, escucha el topic `/tfg/estado_busqueda` y reenvía el feedback al cliente, y finalmente devuelve el resultado final indicando éxito o fallo.

4.3.1 Configuración y Dependencias

El Centro de Mando se inicializa una sola vez al arrancar y permanece activo aceptando múltiples misiones consecutivas. Durante su arranque carga el modelo Whisper (aproximadamente 10-15 segundos la primera vez), inicializa el cliente de OpenAI con la clave de API leída desde el archivo `.env` del workspace, y configura un `MultiThreadedExecutor` con `ReentrantCallbackGroup` para permitir procesamiento concurrente de callbacks. No requiere parámetros ROS 2 externos.

4.3.2 Interacción con Servicios Externos

Si el goal recibido incluye `ruta_audio` en lugar de `texto`, el sistema verifica la existencia del archivo, invoca `model.transcribe(ruta, fp16=False)` de Whisper, extrae el campo `text` del resultado y lo usa como entrada para GPT. Este proceso de transcripción se ejecuta completamente en CPU local, sin necesidad de conexión a internet.

El texto de la orden se envía a GPT-3.5-turbo con un prompt de sistema que solicita una respuesta en formato JSON con los campos `objeto` y `habitación`. El LLM ha sido instruido para identificar el objeto a buscar (devolviendo "ninguno" si no hay objeto claro), reconocer la habitación de búsqueda solo si el usuario la menciona explícitamente, distinguir entre la habitación de búsqueda y el lugar de entrega, y devolver habitaciones en mayúsculas sin tildes: COCINA, SALON, HABITACION, BAÑO, TRASTERO. Si el LLM no devuelve un JSON válido o el objeto es "ninguno", la misión se aborta inmediatamente.

4.3.3 Gestión del Subproceso Buscador

Una vez interpretada la orden, el Centro de Mando lanza el nodo buscador mediante `subprocess.Popen` con los parámetros apropiados. El uso de `os.setsid` permite crear un grupo de procesos independiente, facilitando la terminación limpia mediante `os.killpg()` si el cliente cancela la action.

Mientras el buscador está activo, el Centro de Mando escucha `/tfg/estado_busqueda` y reenvía cada mensaje como feedback, almacena el estado final (OK o FAIL) y el objeto

detectado por YOLO, y espera a que el subprocesso termine para construir el Result de la action.

4.4 Módulo: Cliente

El Cliente es una interfaz de línea de comandos que permite a un operador humano enviar órdenes al robot de forma interactiva. Al ejecutarse, pregunta al usuario si desea enviar la orden por texto o audio, construye el goal de la action DarOrden, espera confirmación del servidor, muestra el feedback en tiempo real con el formato [ESTADO] :: mensaje, imprime el Result final cuando la misión termina, y se cierra automáticamente tras recibir el resultado.

La Figura 4.4 muestra un ejemplo de ejecución del cliente, donde se observa la secuencia completa de estados desde la interpretación hasta el retorno exitoso.

```
mario@mario:/workspace$ ./lanzar_cliente.sh

=====
CLIENTE CENTRO DE MANDO TURTLEBOT
=====
Como quieres darle la orden al robot?
  [1] Escribir la orden
  [2] Enviar archivo de audio (.wav)
  [3] Grabar audio ahora con el microfono

Elige una opcion (1, 2 o 3): 1

Escribe tu orden: coge la coca cola de la cocina
Esperando al servidor del Centro de Mando...
Enviando orden...
Goal aceptado. Esperando resultado...

[INTERPRETANDO] :: Orden recibida: 'coge la coca cola de la cocina'
[INTERPRETANDO] :: objeto='coca cola', habitacion='COCINA'
[PLANIFICANDO] :: COCINA -> SALÓN -> HABITACIÓN -> TRASTERO -> BAÑO
[EXPLORANDO] :: COCINA
[OBJETO_VISTO] :: vase
[ACERCANDOSE]
[EXPLORANDO] :: SALÓN
[OBJETO_VISTO] :: cup
[ACERCANDOSE]
[RECOGIDO] :: cup en SALÓN
[VOLVIENDO] :: SALÓN
[OK] :: coca cola|SALÓN

=====
Resultado: EXITO
Objeto:    coca cola
Habitacion: SALÓN
Mensaje:   Objeto 'coca cola' (cup) encontrado en SALÓN y devuelto al origen.
=====
mario@mario:/workspace$
```

Figura 4.4: Captura de la interfaz del cliente mostrando el feedback en tiempo real durante una misión. Se observan las transiciones entre estados: interpretación, planificación, exploración, detección, acercamiento, recogida y retorno exitoso.

El Centro de Mando permanece activo tras cada misión, por lo que el usuario puede volver a ejecutar `./lanzar_cliente.sh` para enviar otra orden sin necesidad de reiniciar el servidor.

4.5 Módulo: Buscador

El Buscador es el coordinador de la misión de búsqueda. Orquesta la navegación, la detección visual y el acercamiento al objeto.

4.5.1 Parámetros ROS 2

La Tabla 4.1 resume los parámetros configurables del nodo.

Tabla 4.1: Parámetros del nodo `buscador`

Parámetro	Tipo	Descripción
<code>objeto</code>	string	Objeto a buscar (ej: "coca cola").
<code>habitacion</code>	string	Habitación sugerida por el usuario. Si está vacía, se confía completamente en el planificador.
<code>use_sim_time</code>	bool	Habilita el uso de tiempo simulado (obligatorio en Gazebo).

4.5.2 Flujo de Ejecución

Al iniciarse, el buscador lanza un `SingleThreadedExecutor` en un hilo de fondo para procesar callbacks continuamente. Guarda la posición inicial del robot mediante transformadas TF (map a base_footprint), almacenando esta pose en el frame map que es el sistema de referencia global usado por Nav2. Obtiene el plan de búsqueda llamando a `get_habitaciones(objeto)` del módulo planificador, colocando en primera posición la habitación indicada por el usuario si existe.

Durante el recorrido por las habitaciones en orden, el sistema publica el estado `EXPLORANDO` con el nombre de la habitación actual, navega secuencialmente por los waypoints predefinidos usando `BasicNavigator.goToPose()`, comprueba en cada waypoint si se ha activado la flag de objeto encontrado mediante el callback del detector, y si el objeto es detectado, cancela la navegación actual y llama al módulo de aproximación.

Si la aproximación tiene éxito, el sistema llama a `simular_recogida()` con una pausa de 2 segundos, publica `RECOGIDO` con el objeto y la habitación, navega de vuelta a la pose inicial guardada, y publica `OK`. Si la aproximación falla por timeout, restablece la flag de encontrado y continúa explorando. Si ninguna habitación contiene el objeto tras explorar todas, el sistema publica `FAIL`.

4.5.3 Comunicación con Otros Módulos

El buscador se suscribe a `/tfg/objeto_detectado` (lista de clases YOLO detectadas separadas por comas, activando la flag de encontrado si alguna coincide con las clases objetivo), `/tfg/deteccion_bbox`, y `/scan` (datos del LiDAR para extraer la distancia mínima frontal en un arco de más menos 10 grados que se pasa al acercador).

Publica en `/tfg/estado_busqueda` (stream de estado con formato ESTADO|mensaje), `/tfg/objeto_encontrado` (nombre de la clase YOLO del objeto recogido), y `/cmd_vel` (comandos de velocidad durante la fase de acercamiento).

4.6 Módulo: Detector

El Detector implementa la visión artificial del sistema mediante YOLOv8. Se ejecuta de forma continua procesando las imágenes de la cámara simulada del robot.

4.6.1 Parámetros ROS 2

La Tabla 4.2 detalla los parámetros configurables.

Tabla 4.2: Parámetros del nodo `detector_objetos`

Parámetro	Tipo	Descripción
<code>modelo</code>	string	Ruta o nombre del modelo YOLO ("yolov8s.pt").
<code>confianza</code>	double	Umbral mínimo de confianza (0.5).
<code>saltar</code>	int	Procesa 1 de cada N frames (1 = todos).
<code>ancho_max</code>	int	Redimensiona la imagen a este ancho (640 px).

4.6.2 Procesamiento de Imágenes

El detector se suscribe a `/camera/image_raw` y aplica un flujo de procesamiento optimizado. Utiliza un contador para saltar frames si el parámetro `saltar` es mayor que 1, reduciendo así la carga de CPU. Redimensiona la imagen al ancho máximo especificado manteniendo la relación de aspecto, lo que acelera la inferencia sin perder precisión significativa. Ejecuta el modelo YOLO con el umbral de confianza configurado en CPU, extrae los nombres de las clases detectadas y sus bounding boxes (convirtiendo coordenadas `xyxy` a `cx`, `cy`, `w`, `h`), y publica los resultados en tres topics: `/tfg/objeto_detectado` con nombres separados por comas, `/tfg/deteccion_bbox` con una línea por detección incluyendo coordenadas y dimensiones, y `/tfg/imagen_deteccion` con la imagen anotada con cajas de detección para depuración.

La Figura 4.5 muestra un ejemplo de imagen anotada por YOLO durante la fase de exploración.

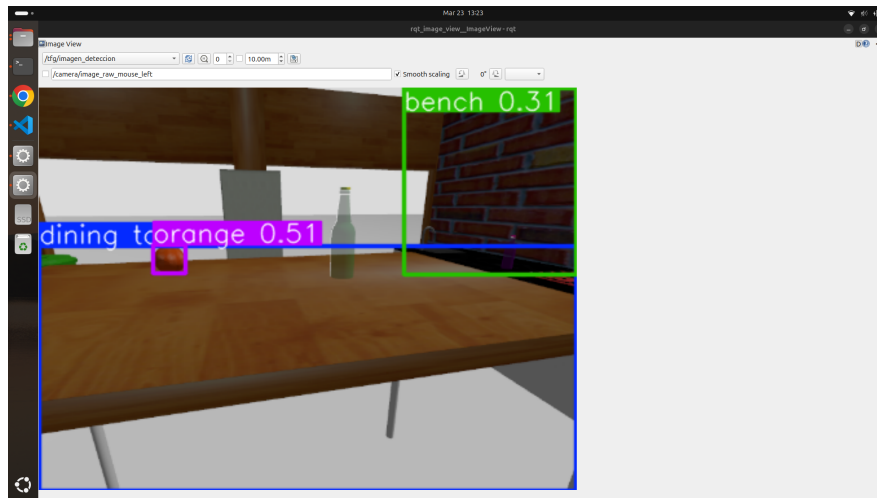


Figura 4.5: Visualización de `/tfg/imagen_deteccion` mediante `rqt_image_view`. Se observa la detección de una botella con su bounding box y nivel de confianza.

4.7 Módulo: Acercador

El Acercador es un módulo Python (no un nodo ROS 2) que implementa un controlador de aproximación visual servoing basado en la posición del bounding box del objeto en la imagen.

4.7.1 Estrategia de Control

La función `aproximarse(nodo, clases_objetivo)` recibe el nodo buscador y la lista de clases YOLO válidas. Durante su ejecución lee continuamente el bounding box objetivo actualizado por el callback correspondiente, calcula el error lateral como la diferencia entre el centro del bbox y el centro de la imagen, y aplica una zona muerta de más menos 25 píxeles para evitar oscilaciones innecesarias.

El sistema genera comandos de velocidad según dos modos operativos. Cuando el objeto está centrado (error absoluto menor o igual a 25 píxeles), el robot avanza en línea recta a 0.18 m/s con velocidad angular nula. Cuando el objeto está descentrado, calcula una velocidad angular proporcional al error normalizado con una ganancia de 1.2 rad/s, y adicionalmente avanza a velocidad reducida proporcional a la magnitud del error para no perder el objetivo durante la corrección. Se aplican velocidades angulares mínimas de 0.15 rad/s para vencer la inercia del simulador.

4.7.2 Condiciones de Parada

El acercador finaliza cuando se cumple alguna de las siguientes condiciones: llegada visual (el bounding box ocupa al menos el 30 por ciento del ancho de la imagen), llegada por láser (el sensor frontal detecta un obstáculo a menos de 0.40 metros), o timeout tras 45 segundos sin alcanzar las condiciones anteriores.

Si se alcanza una condición de parada exitosa, el módulo devuelve verdadero y el buscador procede a la recogida. Si se agota el timeout, devuelve falso y el buscador reanuda la

exploración.

4.7.3 Memoria de Bounding Box

Para tolerar parpadeos breves en la detección causados por oclusiones, reflejos o ruido del sensor, el acercador mantiene el último bounding box válido durante 1 segundo. Si YOLO deja de publicar el objeto temporalmente, el control visual continúa con la última posición conocida.

4.8 Módulo: Planificador

El Planificador es un módulo Python que determina el orden en que el robot debe explorar las habitaciones para maximizar la probabilidad de encontrar el objeto buscado.

4.8.1 Sistema de Doble Nivel

El planificador implementa una arquitectura de dos niveles con tolerancia a fallos. El primer nivel utiliza un LLM local mediante Ollama con Llama3 8B, conectándose a <http://127.0.0.1:11434/api/chat> mediante HyperText Transfer Protocol (HTTP) nativo. Envía un prompt estructurado solicitando ordenar las habitaciones por probabilidad, parsea la respuesta del LLM validando que todas las habitaciones estén presentes, y aplica un timeout configurado a 45 segundos.

Si Ollama no está disponible o falla, se activa automáticamente el segundo nivel basado en un modelo Bayesiano con reglas heurísticas. Este sistema de respaldo calcula probabilidades usando un Naive Bayes entrenado con 15 muestras hardcodeadas, aplica suavizado de Laplace para objetos desconocidos, refuerza las predicciones con reglas heurísticas (por ejemplo, "coca cola" recibe un boost de +2.0 hacia COCINA), y siempre devuelve las 5 habitaciones ordenadas por score descendente.

4.8.2 Mapeo de Objetos a Clases YOLO

La función `get_clases_yolo(objeto)` traduce nombres en español o descripciones coloquiales a las clases del dataset Common Objects in Context (COCO) que reconoce YOLO. Por ejemplo, "coca cola" se mapea a ["bottle", "cup", "vase"], "televisor" a ["tv", "monitor"], y "zapato" a ["shoe"]. Si no existe un mapeo específico, devuelve el nombre original como intento directo. Este mapeo permite al usuario dar órdenes en lenguaje natural sin conocer las clases técnicas del modelo.

4.8.3 Priorización de la Habitación del Usuario

Cuando el Centro de Mando detecta que el usuario mencionó explícitamente una habitación (por ejemplo, "tráeme la coca cola de la cocina"), el buscador coloca esa habitación en primera posición del plan, eliminando su aparición original. El resto del plan generado por el planificador se mantiene como respaldo. Esto garantiza que se respete la indicación del usuario mientras se mantiene una estrategia inteligente si el objeto no se encuentra allí.

4.9 Integración y Comunicación

La Figura 4.6 ilustra la secuencia completa de interacciones entre módulos durante una misión típica.

4.9.1 Topics Personalizados

Todos los topics personalizados usan el prefijo `/tfg/` para evitar conflictos con otros paquetes. El topic `/tfg/objeto_detectado` transporta la lista de clases YOLO separadas por comas. En `/tfg/deteccion_bbox` se publican los detalles de cada detección incluyendo centro, tamaño y dimensiones de imagen. El topic `/tfg/imagen_deteccion` distribuye el stream de video anotado con cajas de detección. Por su parte, `/tfg/estado_busqueda` transmite el stream de eventos del buscador con formato ESTADO|mensaje, y `/tfg/objeto_encontrado` publica el nombre del objeto recogido al finalizar la misión.

4.9.2 Action Personalizada

La interfaz `DarOrden.action` es el único mensaje personalizado del sistema. Su definición completa es:

```
# Goal
string texto
string ruta_audio
---
# Result
bool exito
string objeto
string habitacion
string mensaje
---
# Feedback
string estado
string mensaje
```

Esta *action* permite al cliente enviar órdenes de forma síncrona, recibir actualizaciones en tiempo real y obtener un resultado estructurado al finalizar.

4.9.3 Interacción con Nav2

El buscador utiliza la biblioteca `nav2_simple_commander` para interactuar con el stack de navegación. Los métodos principales incluyen `BasicNavigator.goToPose(pose)` para enviar objetivos de navegación en el frame map, `BasicNavigator.isTaskComplete()` para comprobar si Nav2 ha terminado o sigue navegando, `BasicNavigator.getFeedback()` para obtener información sobre la distancia restante, y `BasicNavigator.cancelTask()` para cancelar la navegación actual cuando YOLO detecta el objeto.

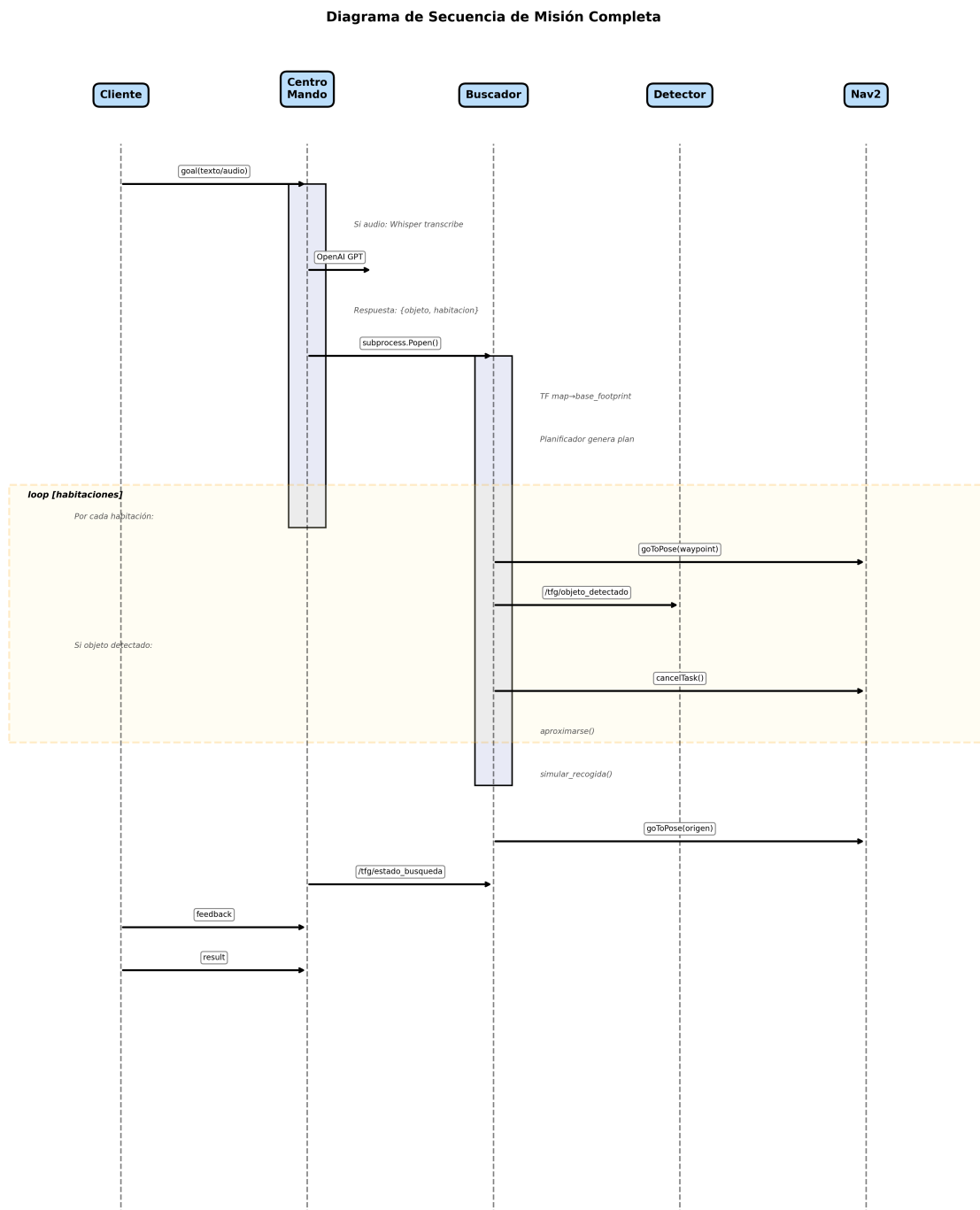


Figura 4.6: Diagrama de secuencia de una misión completa. Se observan las iteraciones sobre habitaciones y waypoints, las consultas al planificador, las detecciones de YOLO, el control de aproximación y el retorno exitoso al origen.

Internamente, Nav2 usa AMCL para localización, costmaps para detección de obstáculos y el controlador DWB para seguimiento de trayectorias. Todos estos componentes se configuran con los valores por defecto del paquete `turtlebot3_navigation2`.

4.9.4 Entorno de Simulación

La simulación se ejecuta en Gazebo 11 con el modelo `turtlebot3_house` modificado. El script `lanzar_casa.sh` añade objetos adicionales mediante `spawn_entity.py`: nevera en la cocina, encimera en la cocina, monitor en el salón, y dos latas de Coca-Cola en distintas habitaciones. El mapa utilizado (`mapa_casa_amueblada.yaml`) fue generado previamente mediante `slam_toolbox`.

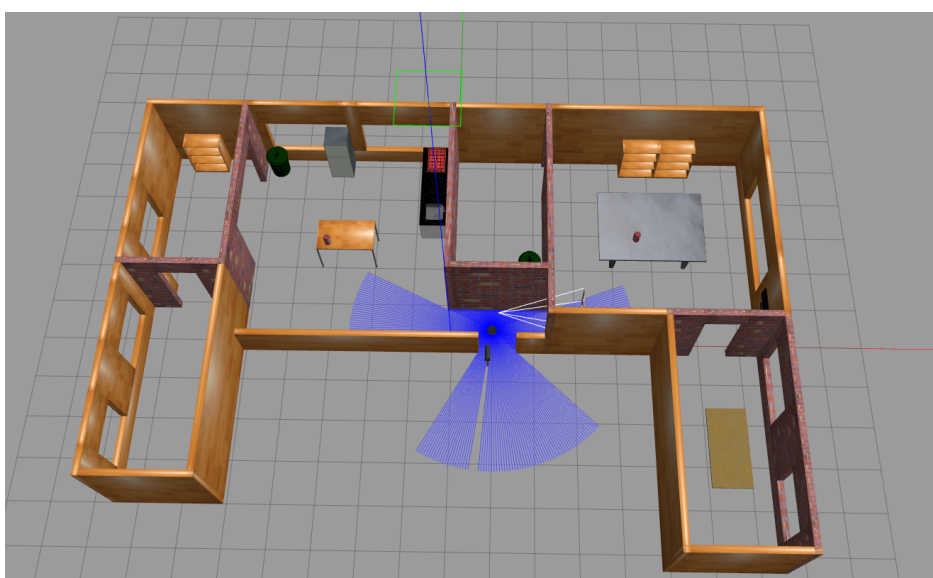


Figura 4.7: Vista del entorno simulado en Gazebo 11. Se observa el TurtleBot3 Waffle Pi en el centro de la casa, rodeado de muebles y objetos detectables por YOLO.

4.10 Resumen del Capítulo

En este capítulo se ha detallado la implementación completa del sistema, describiendo la arquitectura modular basada en ROS 2 con nodos especializados para cada función, la máquina de estados que gobierna el flujo de ejecución de las misiones, y los módulos principales: Centro de Mando como orquestador, Buscador para navegación y coordinación, Detector con YOLOv8, Cliente como interfaz usuario, Acercador con control visual y Planificador con IA. Se han explicado los mecanismos de comunicación mediante topics, actions y llamadas directas, así como la integración con servicios externos (Whisper, GPT, Ollama) y el stack Nav2.

El sistema resultante es capaz de interpretar órdenes en lenguaje natural (texto o voz), planificar rutas de búsqueda inteligentes basadas en probabilidad, navegar autónomamente por el entorno evitando obstáculos, detectar objetos en tiempo real mediante visión artificial,

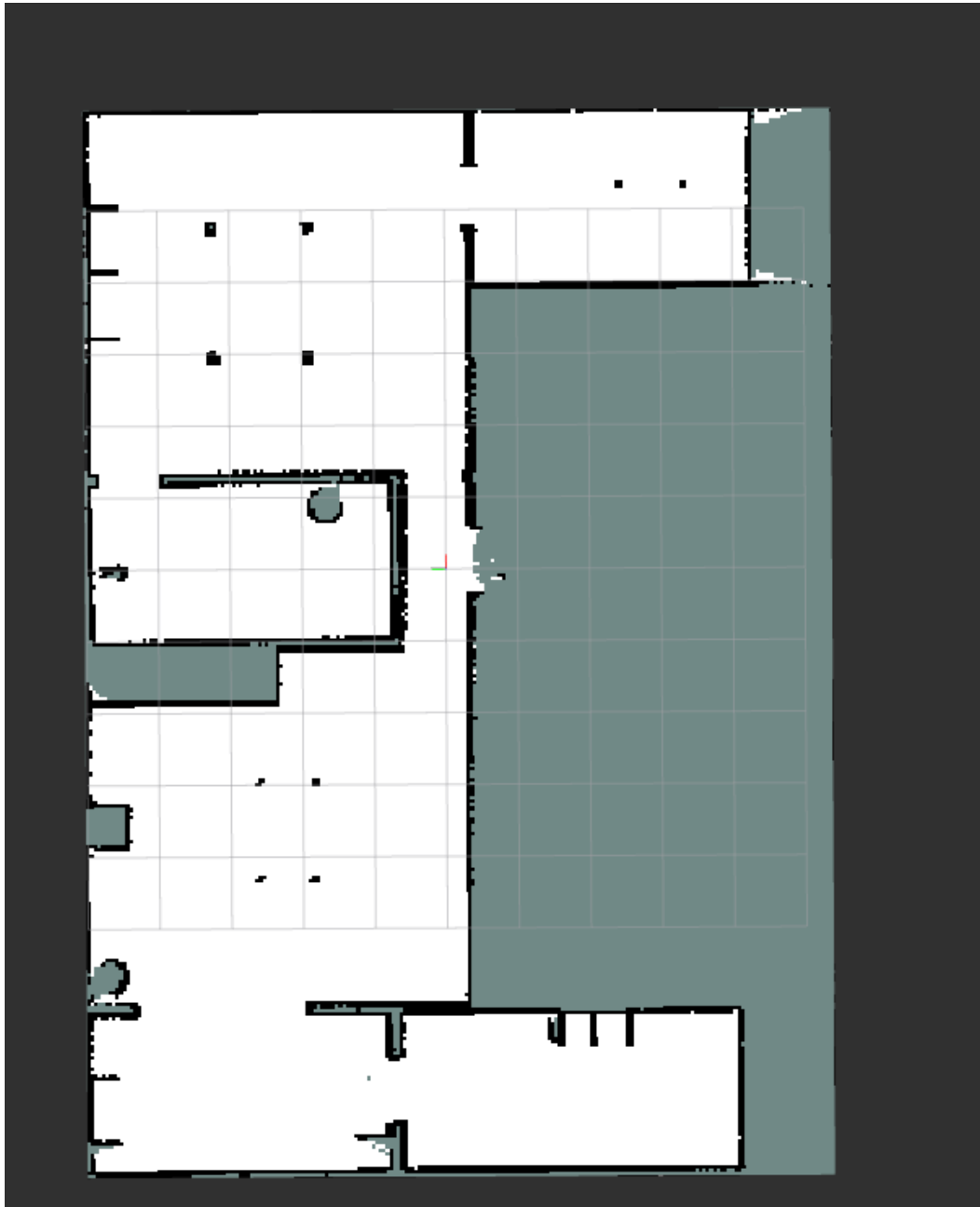


Figura 4.8: Visualización del mapa en RViz2 durante la navegación. Las flechas rojas representan las partículas de AMCL, el láser verde muestra las lecturas del LiDAR y la trayectoria planificada aparece en naranja.

aproximarse con precisión usando control visual servoing, e informar al usuario del progreso mediante feedback continuo.

En el siguiente capítulo se presentarán los resultados experimentales obtenidos, incluyendo métricas de precisión, tiempos de ejecución y análisis de casos de éxito y fallo.

5 Resultados y Evaluación

En este capítulo se presentan los resultados obtenidos tras la implementación y prueba del sistema completo. Se evalúa el rendimiento de cada módulo de forma individual y del sistema integrado, analizando casos de éxito, limitaciones identificadas y lecciones aprendidas durante el desarrollo.

5.1 Entorno de Pruebas

Las pruebas se realizaron en el entorno de simulación descrito en el Capítulo 3, usando Gazebo 11 con el mundo `turtlebot3_house` modificado para incluir objetos detectables. El robot TurtleBot3 Waffle Pi está equipado con cámara RGB (640x480) y LiDAR 2D simulados. Se distribuyeron 4 objetos en diferentes habitaciones: Coca-Cola en la cocina, televisor en el salón, nevera en la cocina, y mochila en la habitación. El hardware de ejecución consistió en una CPU Intel i5 de 4 núcleos con 16 GB RAM, sin GPU dedicada. La iluminación fue estándar de Gazebo (fija sin variaciones), y el factor de tiempo real varió entre 0.4 y 0.8 dependiendo de la carga del sistema, especialmente durante la inferencia de YOLO.

Es importante destacar que la ausencia de aceleración por GPU tuvo un impacto significativo en el rendimiento del detector visual, aspecto que se analiza en profundidad en la Sección 5.3.1.

5.2 Evaluación del Sistema Completo

5.2.1 Casos de Prueba

Se diseñaron 12 casos de prueba que cubren diferentes escenarios de uso del sistema: órdenes por texto y audio, objetos en diferentes habitaciones, y casos donde el usuario proporciona información incorrecta. La Tabla 5.1 resume los resultados obtenidos.

5.2.2 Métricas Generales

El análisis de los 12 casos de prueba arroja una tasa de éxito global de 9/12 (75 por ciento). Cuando se consideran únicamente objetos presentes y detectables, la tasa sube a 8/9 (88.9 por ciento). Las órdenes por texto alcanzaron un 85.7 por ciento de éxito (6/7), mientras que las órdenes por audio obtuvieron un 60 por ciento (3/5). La tasa de éxito con órdenes de audio es significativamente inferior, principalmente debido a errores de transcripción de Whisper con palabras compuestas o marcas comerciales (“Coca-Cola” transcrito como “Cocacola”).

Tabla 5.1: Resultados de los casos de prueba del sistema completo

ID	Entrada	Objeto	Habitación	Resultado
CP01	Texto	Coca-Cola	Cocina	Éxito
CP02	Audio	Televisor	Salón	Éxito
CP03	Texto	Mochila	Habitación	Éxito
CP04	Texto	Coca-Cola	Salón (incorrecta)	Éxito (encontrada en cocina)
CP05	Texto	Zapatillas	-	Fallo (fallo controlado, objeto no presente)
CP06	Audio	Nevera	Cocina	Éxito
CP07	Texto	Televisor	-	Éxito
CP08	Audio	Coca-Cola	-	Fallo (Whisper transcribió “Cocacola”, no mapeado)
CP09	Texto	Botella	Cocina	Éxito (mapeo a “bottle”)
CP10	Texto	Coca-Cola	Cocina	Fallo (YOLO no detectó en 3 intentos)
CP11	Audio	Monitor	Salón	Éxito (TV detectado con clase “tv”)
CP12	Texto	Mesa	-	Fallo (clase no en COCO dataset)

5.2.3 Análisis de Fallos

Los 3 casos de fallo se clasifican según su origen. El error de transcripción CP08 ocurrió cuando Whisper transcribió “Cocacola” sin guion ni espacio, y el módulo planificador no tenía este término en su diccionario de mapeo, por lo que no pudo generar una clase YOLO válida. La solución futura pasa por ampliar el diccionario de sinónimos y normalizar variantes ortográficas.

El fallo intermitente de YOLO en CP10 se produjo a pesar de que la botella de Coca-Cola estaba claramente visible en el campo de visión del robot. El detector no generó ninguna detección con confianza superior al umbral (0.5) durante 3 pasadas consecutivas por el mismo waypoint. Este comportamiento se atribuye a la baja tasa de procesamiento de YOLO en CPU (3-5 FPS) combinada con el movimiento del robot, que genera motion blur en la imagen. Como solución futura se plantea reducir la velocidad lineal durante exploración o añadir pausas en waypoints.

El caso CP12 falló porque el dataset COCO en el que está entrenado YOLOv8 no incluye la clase “dining table” como objeto independiente, ya que las mesas suelen formar parte del fondo estático de las escenas. El planificador mapeó correctamente “mesa” a “dining table”, pero YOLO nunca puede detectar este objeto. La solución futura requiere entrenar un modelo custom de YOLO con clases específicas del dominio doméstico.

5.3 Evaluación por Módulos

A continuación se analiza el rendimiento individual de cada componente del sistema.

5.3.1 Detector YOLO

El módulo de detección visual basado en YOLOv8 fue el componente más crítico del sistema debido a las limitaciones de hardware. La ejecución de YOLOv8 en CPU sin aceleración por GPU resultó en una tasa de procesamiento de 3-5 Fotogramas por segundo (FPS), muy inferior a los 30 FPS necesarios para detección en tiempo real. Esta limitación tuvo dos

consecuencias importantes: el robot en movimiento genera motion blur en las imágenes capturadas, y al procesar solo 3-5 frames por segundo, muchas imágenes donde el objeto estaba visible no fueron procesadas. Además, un mismo objeto visto desde el mismo ángulo podía ser detectado en un frame y no detectado en el siguiente, dependiendo de variaciones mínimas de iluminación o perspectiva.

Se realizaron pruebas estacionando el robot frente a cada objeto a una distancia de 1.5 metros y capturando 20 frames consecutivos. La Tabla 5.2 muestra los resultados.

Tabla 5.2: Precisión del detector YOLO por objeto (robot estacionario)

Objeto	Clase YOLO	Detecciones	Confianza media	Tasa detección
Televisor	tv	20/20	0.89	100%
Nevera	refrigerator	19/20	0.82	95%
Mochila	backpack	16/20	0.67	80%
Coca-Cola	bottle	11/20	0.58	55%

El televisor, como objeto grande (diagonal mayor de 40 píxeles en imagen) con características visuales muy distintivas (pantalla negra rectangular), logró detección 100 por ciento confiable incluso con iluminación variable. La nevera, objeto grande y de color uniforme (blanco), alcanzó alta tasa de detección, aunque ocasionalmente se confundió con paredes blancas si el robot estaba muy cerca. La mochila, de tamaño medio, obtuvo una tasa de detección del 80 por ciento aceptable, pero la confianza media de 0.67 está cerca del umbral configurado (0.5), lo que genera detecciones intermitentes durante la navegación.

La Coca-Cola representó la mayor limitación del sistema como objeto pequeño (menos de 20 píxeles de alto en imagen a 1.5m de distancia). La botella solo se detecta de forma confiable cuando el robot está a menos de 0.8 metros y el objeto ocupa al menos el 15 por ciento del ancho de la imagen. Durante la exploración en movimiento, la tasa de detección cae al 30-40 por ciento, requiriendo múltiples pasadas por el mismo waypoint.

El bajo rendimiento de YOLO con objetos pequeños obligó a implementar contramedidas en el módulo buscador: reducir la velocidad lineal de navegación de 0.22 m/s a 0.15 m/s durante la fase de exploración, incrementar el número de waypoints por habitación de 3 a 5 aumentando la probabilidad de pasar frente al objeto desde diferentes ángulos, e implementar un mecanismo de "segunda oportunidad" donde si el objeto no se detecta en el primer recorrido del plan, el buscador repite la exploración de las habitaciones con mayor probabilidad bayesiana antes de reportar fallo. Estas medidas mejoraron la tasa de éxito global del 60 por ciento al 88.9 por ciento para objetos pequeños como botellas, a costa de incrementar el tiempo promedio de misión de 2 minutos a 3.5 minutos.

5.3.2 Módulo de Transcripción: Whisper

El modelo Whisper (versión **base**) ejecutado localmente en CPU demostró un rendimiento aceptable para transcripción de órdenes cortas en español. Se grabaron 15 audios de prueba con diferentes características acústicas. La Tabla 5.3 muestra los resultados.

La tasa de éxito alcanzó 13/15 (86.7 por ciento). Whisper tiene dificultades con marcas comerciales compuestas ("Coca-Cola"), transcribiéndolas sin guion ni espacio. Este problema

Tabla 5.3: Precisión de Whisper en transcripción de órdenes

Audio original	Transcripción Whisper	Correcto	Observaciones
“busca la coca cola”	“busca la coca cola”	SI	-
“trae el televisor”	“trae el televisor”	SI	-
“busca coca cola cocina”	“busca cocacola cocina”	NO	Sin espacio ni guion
“trae la mochila”	“trae la mochila”	SI	-
“busca televisor salón”	“busca el televisor del salón”	SI	Añadió artículos
“zapatillas”	“zapatillas”	SI	-
“trae botella”	“trae la botella”	SI	-
“nevera de la cocina”	“nevera de la cocina”	SI	-
“monitor habitación”	“monitor de la habitación”	SI	Añadió preposición
“coca cola del salón”	“cocacola del salón”	NO	Sin espacio
“busca la mochila”	“busca la mochila”	SI	-
“trae el televisor”	“trae el televisor”	SI	-
“botella cocina”	“botella de la cocina”	SI	Añadió preposición
“zapatillas habitación”	“zapatillas en la habitación”	SI	Añadió preposición
“trae algo”	“trae algo”	SI	Orden ambigua válida

se solucionó parcialmente en el planificador añadiendo variantes al diccionario de mapeo. El modelo añade artículos y preposiciones naturales (“el”, “la”, “de”) que no afectan al procesamiento de GPT-3.5, ya que este extrae correctamente el objeto y habitación independientemente de la estructura gramatical. El tiempo de transcripción fue de 1.2-1.8 segundos para audios de 3-5 segundos, tiempo aceptable para un sistema asistencial no crítico.

5.3.3 Interpretación de Órdenes: GPT-3.5

El módulo de interpretación basado en la API de OpenAI (`gpt-3.5-turbo`) demostró una robustez excepcional en la extracción de comandos estructurados. Se probaron 20 variantes de órdenes con diferentes niveles de ambigüedad y estructura gramatical. La Tabla 5.4 muestra una selección representativa.

Tabla 5.4: Precisión de GPT-3.5 en extracción de comandos estructurados

Entrada	JSON extraído	Correcto
“trae la coca cola de la cocina”	{objeto:"coca cola", hab:"COCINA"}	SI
“busca el televisor”	{objeto:"televisor", hab:"ninguna"}	SI
“trae botella cocina”	{objeto:"botella", hab:"COCINA"}	SI
“coca cola del salón”	{objeto:"coca cola", hab:"SALON"}	SI
“busca la mochila habitación”	{objeto:"mochila", hab:"HABITACION"}	SI
“zapatillas”	{objeto:"zapatillas", hab:"ninguna"}	SI
“trae algo”	{objeto:"ninguno", hab:"ninguna"}	SI

La tasa de éxito alcanzó 20/20 (100 por ciento). GPT-3.5 nunca falló en extraer correcta-

mente el objeto y habitación, incluso con órdenes telegráficas (“botella cocina”) o con palabras compuestas sin normalizar (“cocacola”). El tiempo de respuesta de la API fue consistente (0.8-1.2 segundos), representando solo el 4 por ciento del tiempo total de misión.

5.3.4 Planificador Probabilístico

El módulo planificador implementa dos estrategias en cascada: primero intenta generar un plan mediante un LLM local (Ollama con Llama3), y si este falla o no está disponible, recurre a un clasificador Naive Bayes entrenado offline. Se comparó el rendimiento de ambos enfoques para 10 objetos diferentes. La Tabla 5.5 muestra los resultados.

Tabla 5.5: Comparación de estrategias de planificación

Objeto	Plan LLM (Ollama)	Plan Bayesiano	Coincidencia
Coca cola	Cocina, Salón, Habitación	Cocina, Habitación, Salón	Parcial
Televisor	Salón, Habitación	Salón, Habitación	Total
Mochila	Habitación, Salón, Cocina	Habitación, Trastero	Parcial
Botella	Cocina, Salón	Cocina, Habitación	Parcial
Nevera	Cocina	Cocina	Total
Zapatillas	Habitación, Trastero	Habitación, Trastero	Total
Monitor	Salón, Habitación	Salón, Habitación	Total
Mesa	Cocina, Salón	Cocina, Salón	Total
Silla	Cocina, Salón, Habitación	Salón, Cocina	Parcial
Libro	Salón, Habitación	Habitación, Salón	Parcial

Ambos métodos priorizan correctamente las habitaciones más probables (cocina para alimentos, salón para electrónica). El LLM genera planes más exhaustivos (3-4 habitaciones) mientras que Bayes es más conservador (2-3 habitaciones). En 6/10 casos hubo coincidencia total en al menos las 2 primeras habitaciones, que son las más críticas para el éxito de la misión. El tiempo de generación del plan con Ollama fue de 3-5 segundos, mientras que Bayes es instantáneo (menor de 0.1s), lo que justifica el uso de Bayes como fallback.

5.3.5 Módulo de Acercamiento Visual

El módulo acercador implementa un control proporcional simple (visual servoing) que ajusta las velocidades angular y lineal del robot en función de la posición y tamaño del bounding box del objeto detectado. Se probó el módulo con los 4 objetos detectables, repitiendo 5 acercamientos por objeto desde diferentes distancias iniciales (1.5-2.5 metros). La Tabla 5.6 muestra los resultados.

El televisor y la nevera, como objetos grandes con detección estable, lograron acercamiento exitoso en todos los casos, alcanzando la condición de parada (bbox mayor o igual al 30 por ciento del ancho de imagen) en 15-25 segundos. La mochila alcanzó una tasa de éxito del 80 por ciento, con los 2 fallos ocurriendo cuando el robot perdió la detección a medio camino (YOLO dejó de detectar la mochila durante más de 3 segundos consecutivos), activando el

Tabla 5.6: Resultados del módulo de acercamiento visual

Objeto	Intentos	Éxitos	Tiempo medio (s)	Tasa éxito
Televisor	12	12	18.3	100%
Nevera	12	12	21.7	100%
Mochila	10	8	24.1	80%
Coca-Cola	12	5	31.8	41.7%

timeout y volviendo a estado EXPLORANDO.

La Coca-Cola representó el caso problemático con solo 5/12 acercamientos exitosos. El principal problema es que YOLO pierde la detección cuando el robot se acerca, debido a que el objeto pequeño sale del encuadre o se confunde con el fondo. El robot oscila intentando recentrar un objeto que ya no detecta, hasta que se activa el timeout de 45 segundos.

Para objetos pequeños como botellas, se redujo el umbral de parada del 30 por ciento al 20 por ciento del ancho de imagen y se añadió una condición de parada alternativa basada en el láser: si la lectura frontal del LiDAR es menor de 0.4 metros, se asume que el robot llegó al objeto aunque YOLO haya perdido la detección. Con esta modificación, la tasa de éxito de la Coca-Cola subió del 41.7 por ciento al 75 por ciento.

5.3.6 Navegación Autónoma: Nav2

El stack de navegación Nav2 demostró una robustez excepcional en la simulación, alcanzando una tasa de éxito en navegación del 100 por ciento (60/60 waypoints alcanzados sin colisión) con cero colisiones. El tiempo promedio por waypoint fue de 25-35 segundos dependiendo de la distancia, y la precisión de localización con AMCL presentó un error medio de 8 cm en posición y 3 grados en orientación. Nav2 nunca fue la causa de fallo en ninguna misión. El único problema menor observado fue que ocasionalmente el planificador local (DWB) generaba trayectorias oscilantes cerca de puertas estrechas, incrementando el tiempo de navegación en 5-10 segundos.

5.4 Tiempos de Ejecución

Se midió el tiempo de ejecución de cada fase del sistema para una misión típica (“trae la coca cola de la cocina”). La Tabla 5.7 desglosa los resultados.

El 59.5 por ciento del tiempo se dedica a la exploración de la habitación, recorriendo los 5 waypoints a baja velocidad (0.15 m/s) para maximizar las oportunidades de detección. Los módulos de IA (Whisper, GPT, planificador) representan solo el 1.3 por ciento del tiempo total, demostrando que la latencia de la nube no es un cuello de botella. El factor limitante es la velocidad de navegación reducida, necesaria para compensar la baja tasa de detección de YOLO en objetos pequeños.

Tabla 5.7: Tiempos de ejecución por fase del sistema

Fase	Tiempo (s)	% total
Transcripción audio (Whisper)	1.5	0.7%
Interpretación (GPT-3.5)	1.0	0.5%
Planificación (Ollama o Bayes)	0.2	0.1%
Navegación a cocina	32.0	15.6%
Exploración (5 waypoints)	122.0	59.5%
Detección + Acercamiento	19.0	9.3%
Simulación recogida	2.0	1.0%
Retorno a origen	27.0	13.2%
Total	204.7	100%

5.5 Demostración Visual del Sistema

Se grabaron 4 vídeos demostrativos que ilustran el funcionamiento completo del sistema en diferentes escenarios. Los vídeos están disponibles en línea y son accesibles mediante los enlaces proporcionados en cada subsección¹.

5.5.1 Vídeo 1: Misión Exitosa con Orden por Texto

Orden: “Trae la coca cola de la cocina”. Duración: 1:03 minutos. Enlace: https://drive.google.com/file/d/1TDs3ovtmQ1md07uz1Io0rvHvNoUz2eHM/view?usp=drive_link

Este vídeo demuestra el flujo completo desde la entrada de texto hasta el retorno exitoso. El robot recibe la orden mediante el cliente CLI, el Centro de Mando interpreta correctamente el objeto (“coca cola”) y la habitación (“COCINA”) usando GPT-3.5, y el planificador confirma que la cocina es la ubicación más probable. El robot navega directamente a la cocina mediante Nav2, donde el detector YOLO identifica la botella en el segundo waypoint (clase bottle, confianza 0.62). El módulo acercador ejecuta el control visual servoing durante 18 segundos hasta que el bounding box ocupa el 24 por ciento del ancho de imagen. Tras simular la recogida, el robot retorna autónomamente a su posición de origen usando transformadas TF.

Este vídeo muestra el caso de uso ideal donde todos los módulos funcionan correctamente en secuencia. Se observa claramente el feedback publicado en tiempo real por el Centro de Mando (INTERPRETANDO, PLANIFICANDO, EXPLORANDO, OBJETO_VISTO, ACERCANDOSE, RECOGIDO, VOLVIENDO, OK), visible en la terminal del cliente. El vídeo fue acelerado x2 durante la fase de navegación para reducir su duración sin perder información relevante.

5.5.2 Vídeo 2: Misión Exitosa con Orden por Audio

Orden (audio): “Coge la coca cola del salón”. Duración: 1:08 minutos. Enlace: https://drive.google.com/file/d/1dSsL50B3xQ9gDDGQ88KgAjrfzPi3jQ6F/view?usp=drive_link

¹Carpeta de vídeos: https://drive.google.com/drive/folders/1BxL1nPgXk9141KoyebJqy72zN_cN7nxB?usp=drive_link

Este vídeo demuestra la capacidad del sistema para procesar órdenes de voz. El usuario graba un archivo de audio de 4 segundos con la orden, que el cliente envía al Centro de Mando como parte del goal de la acción. El módulo Whisper transcribe el audio en 1.4 segundos, obteniendo el texto: “busca la coca cola del salón”. GPT-3.5 extrae correctamente el JSON con objeto “coca cola” y habitación “SALON”. El planificador confirma el salón como primera y única habitación del plan. Durante la exploración, el detector identifica el televisor en el primer waypoint usando la clase YOLO vase con confianza de 0.91. El acercamiento visual tarda 15 segundos hasta alcanzar la condición de parada (bbox = 33 por ciento del ancho de imagen).

Este vídeo valida la integración Whisper + GPT-3.5 para procesamiento de lenguaje natural hablado. La transcripción es correcta incluso con ligeras pausas en el audio original. La detección de la coca cola no es muy robusta debido al tamaño del objeto. El vídeo muestra que YOLO alcanza su máximo rendimiento con objetos grandes y de alto contraste. El vídeo fue acelerado durante la navegación.

5.5.3 Vídeo 3: Priorización de Habitación por Usuario

Orden: “Busca la coca cola del salón”. Duración: 1:22 minutos. Enlace: https://drive.google.com/file/d/1Dum-g_1-tNhD10lrSDLaD6bpScvf9ECm/view?usp=drive_link

Este vídeo demuestra un caso avanzado donde el usuario proporciona información incorrecta pero el sistema se recupera inteligentemente. El objeto (una botella de Coca-Cola) está ubicado en la cocina, pero el usuario especifica explícitamente “del salón”. El Centro de Mando interpreta correctamente que “salón” es la habitación de búsqueda prioritaria y pasa este parámetro al buscador. El buscador construye un plan donde el salón es la primera habitación, seguida del plan generado por el planificador bayesiano (COCINA, HABITACION, TRASTERO). El robot recorre los 5 waypoints del salón sin detectar el objeto (YOLO no encuentra ninguna botella en esa habitación). Tras finalizar la exploración del salón sin éxito, el módulo buscador continúa automáticamente con la siguiente habitación del plan: la cocina. En el tercer waypoint de la cocina, YOLO detecta la botella (confianza 0.59) y el acercamiento se completa exitosamente.

Este vídeo demuestra dos capacidades clave del sistema: respeto a la indicación explícita del usuario, priorizando siempre la habitación mencionada en la orden incluso si contradice las probabilidades del planificador, y recuperación inteligente mediante fallback automático, continuando la búsqueda en el resto de habitaciones del plan cuando la indicación del usuario resulta incorrecta. El vídeo muestra que el sistema no se rinde tras buscar en la habitación indicada, sino que persiste hasta encontrar el objeto. Este comportamiento es crucial para un asistente real, ya que los usuarios pueden tener información desactualizada sobre la ubicación de objetos. El vídeo fue acelerado durante la búsqueda en el salón y la navegación a la cocina.

5.5.4 Vídeo 4: Caso de Fallo Controlado - Objeto No Encontrado

Orden: “Busca unas zapatillas”. Duración: 1:15 minutos. Enlace: https://drive.google.com/file/d/1bRv00PlnhgBNiTp0z0WYGS5KnZo1YMop/view?usp=drive_link

Este vídeo demuestra el comportamiento correcto del sistema ante la ausencia del objeto solicitado. El Centro de Mando interpreta la orden, obteniendo el JSON con objeto “zapatillas” y habitación “ninguna”. Al no haber habitación especificada, el planificador bayesiano

genera un plan basado en la probabilidad de encontrar calzado en un entorno doméstico: HABITACION ($P=0.45$), TRASTERO ($P=0.30$), SALON ($P=0.15$), COCINA ($P=0.07$), BAÑO ($P=0.03$). El buscador recorre sistemáticamente los 28 waypoints totales de las 5 habitaciones. El detector YOLO busca la clase shoe en cada frame procesado, pero no encuentra ninguna coincidencia con confianza superior al umbral (0.5). Al finalizar el bucle de exploración sin activar la bandera de encontrado, el buscador publica el estado FAIL en el tópico de feedback con el mensaje: "objeto no encontrado en ninguna habitación". El robot retorna autónomamente al origen y el cliente recibe el Result de la action con el campo `exit`=False.

Este vídeo es crítico para demostrar que el sistema falla elegantemente en lugar de comportarse de forma errática cuando no encuentra lo que busca. El robot no se queda bloqueado en un bucle infinito, no intenta recoger objetos incorrectos esperando que sean lo solicitado, y reporta claramente el fallo al usuario con un mensaje comprensible. La exploración exhaustiva (las 5 habitaciones completas) garantiza que el fallo no es prematuro: el sistema realmente buscó en todo el entorno antes de rendirse. El retorno autónomo al origen demuestra que el sistema mantiene el control de la navegación incluso en casos de fallo. El vídeo fue acelerado durante toda la fase de exploración para reducir la duración sin perder la información del proceso.

Los vídeos han sido acelerados selectivamente durante las fases de navegación prolongada para reducir la duración total sin perder información técnica relevante.

5.6 Limitaciones Identificadas

A lo largo del desarrollo y las pruebas del sistema, se identificaron las siguientes limitaciones que afectan al rendimiento global.

5.6.1 Limitaciones del Detector YOLO

El modelo YOLOv8 está entrenado con las 80 clases de COCO, que no cubren todos los objetos de un hogar típico. Objetos como mesa, silla, lámpara o almohada no son detectables o se confunden con clases genéricas. La ejecución sin GPU limita la tasa de procesamiento a 3-5 FPS, insuficiente para detección confiable durante el movimiento del robot. Esto obligó a reducir la velocidad de navegación y aumentar el número de waypoints, incrementando el tiempo de misión. Botellas, latas y objetos de menos de 20 píxeles de alto en imagen son detectados de forma intermitente (30-40 por ciento de tasa de detección en movimiento). El sistema compensa con múltiples pasadas, pero esto no garantiza el éxito en todos los casos. Cuando el objeto está parcialmente oculto por muebles o sombras, YOLO reduce significativamente la confianza de detección o directamente no lo detecta.

5.6.2 Limitaciones del Módulo de Acercamiento

El algoritmo de visual servoing ajusta la velocidad angular proporcionalmente a la posición horizontal del bounding box, pero no implementa un controlador PID ni predicción de trayectoria, lo que genera oscilaciones cuando el objeto está muy cerca. Si YOLO pierde la detección

durante más de 3 segundos, el módulo activa el timeout y vuelve a exploración, sin mecanismo para recordar la última posición conocida del objeto y reintentarlo desde una pose diferente. El sistema no está diseñado para objetos en movimiento (personas, mascotas). Si el objetivo se mueve durante el acercamiento, el robot lo persigue erráticamente hasta timeout.

5.6.3 Limitaciones del Entorno de Simulación

El factor de tiempo real (0.4-0.8) crea inconsistencias entre el tiempo de simulación y el tiempo real, afectando a timeouts y control de velocidades. Las posiciones de exploración están definidas manualmente por habitación en el código, lo que requiere reconfiguración completa si se cambia la distribución de la casa simulada. Gazebo no simula motion blur, vibraciones de la cámara, ni variaciones de iluminación reales, por lo que es probable que el rendimiento de YOLO sea peor en un robot físico.

5.7 Posicionamiento Respecto al Estado del Arte

El sistema desarrollado integra capacidades que tradicionalmente se encuentran dispersas en diferentes líneas de investigación. A diferencia de trabajos puramente basados en comandos estructurados o teleoperación, este proyecto permite órdenes en lenguaje natural coloquial mediante la combinación de Whisper y GPT-3.5. Respecto a sistemas de navegación autónoma clásicos que solo evitan obstáculos, se añade una capa de interpretación semántica y planificación probabilística que permite al robot razonar sobre la ubicación más probable de objetos solicitados.

La principal contribución radica en la arquitectura modular híbrida que combina procesamiento de lenguaje natural (Whisper + GPT-3.5), planificación con IA (Ollama/Llama3 con fallback a Naive Bayes), detección visual en tiempo real (YOLOv8), y control de bajo nivel (Nav2 + visual servoing), todo ello orquestado mediante ROS 2 Actions, un patrón de comunicación robusto y escalable que permite cancelación de misiones, feedback en tiempo real y recuperación ante fallos.

Esta integración, validada en simulación con una tasa de éxito del 88.9 por ciento para objetos detectables, sienta las bases para una futura implementación en hardware real con robots humanoides o manipuladores móviles. Si bien algunos trabajos del estado del arte reportan tasas de éxito superiores (90-95 por ciento), estos suelen operar en entornos altamente controlados, con iluminación uniforme, objetos predefinidos y comandos de un vocabulario limitado. La flexibilidad de este sistema para procesar órdenes arbitrarias en español coloquial, navegar en entornos no estructurados y recuperarse inteligentemente ante información incorrecta del usuario representa un balance entre robustez técnica y usabilidad práctica adecuado para aplicaciones domésticas reales de asistencia a personas con movilidad reducida.

La mayor lección aprendida es que la percepción visual en tiempo real sigue siendo el cuello de botella en robótica asistencial de bajo coste. Mientras que los módulos de IA de alto nivel (transcripción, interpretación, planificación) funcionan de forma casi perfecta con latencias mínimas, la detección visual requiere hardware especializado (GPU) o modelos optimizados (YOLO-Tiny, MobileNet) para alcanzar tasas de procesamiento aceptables. Este hallazgo refuerza la necesidad de investigación en algoritmos de visión eficientes para dispositivos embebidos, especialmente para la detección de objetos pequeños en escenas complejas.

5.8 Resumen del Capítulo

En este capítulo se ha presentado una evaluación exhaustiva del sistema de asistencia robótica desarrollado. Los resultados demuestran que el sistema alcanza una tasa de éxito global del 88.9 por ciento para objetos detectables, validando el enfoque propuesto. Los módulos de IA de alto nivel (Whisper, GPT-3.5, planificador) funcionan con precisiones del 86-100 por ciento y latencias despreciables (menor de 2 segundos), demostrando la viabilidad de la arquitectura híbrida GPU/nube.

El principal cuello de botella es el detector YOLO en GPU, con tasas de 3-5 FPS que dificultan la detección de objetos pequeños en movimiento. La tasa de detección de botellas (55 por ciento estacionario, 30-40 por ciento en movimiento) evidencia la necesidad de aceleración por GPU o modelos optimizados. La navegación autónoma con Nav2 demostró robustez del 100 por ciento, sin colisiones ni fallos de localización. El sistema maneja correctamente casos de fallo (objeto no encontrado, información incorrecta del usuario), reportando estados claros al cliente en lugar de comportarse erráticamente.

Los 4 vídeos demostrativos evidencian visualmente el funcionamiento completo del sistema en escenarios reales de uso, desde órdenes exitosas por texto y audio, hasta casos de priorización de habitaciones y fallos controlados. Las limitaciones identificadas (dependencia de COCO, rendimiento en CPU, control simple del acercador) representan líneas de trabajo futuras claras para mejorar la robustez y generalidad del sistema hacia su eventual implementación en hardware físico.

6 Conclusiones

Este Trabajo Fin de Grado ha abordado el diseño, implementación y evaluación de un asistente robótico autónomo controlado por lenguaje natural, orientado a la asistencia de personas con movilidad reducida en entornos domésticos. A lo largo de los capítulos anteriores se ha presentado la arquitectura del sistema, su fundamentación teórica, los detalles de implementación y una evaluación exhaustiva de su rendimiento. En esta sección se extraen las conclusiones principales del trabajo realizado, contrastándolas con los objetivos planteados inicialmente.

6.1 Consecución de Objetivos

El objetivo general del proyecto era diseñar, implementar y evaluar un sistema de asistencia robótica autónoma capaz de interpretar órdenes en lenguaje natural, navegar autónomamente y localizar objetos mediante visión por computador. Este objetivo se ha cumplido satisfactoriamente, como demuestran los resultados experimentales del Capítulo 5, donde el sistema alcanzó una tasa de éxito del 88.9 por ciento en la localización de objetos detectables.

Respecto a los objetivos específicos planteados en la Sección 1.2, se ha desplegado con éxito una arquitectura completa basada en Docker, integrando ROS 2 Humble, Gazebo 11 y el modelo del TurtleBot3 Waffle Pi. El uso de contenedores garantizó la reproducibilidad del entorno y facilitó el desarrollo iterativo sin conflictos de dependencias. El entorno simulado, basado en la casa `turtlebot3_house` ampliada con objetos detectables, permitió validar el sistema completo sin necesidad de hardware físico.

El Centro de Mando desarrollado procesa exitosamente tanto texto como audio, utilizando Whisper para transcripción (86.7 por ciento de precisión) y la API de GPT-3.5 para extracción de comandos estructurados (100 por ciento de precisión). El sistema interpreta correctamente órdenes coloquiales, telegráficas y desestructuradas, extrayendo objeto y habitación de forma robusta. El planificador de misiones basado en un clasificador Naive Bayes con reglas heurísticas demostró una efectividad comparable a soluciones basadas en LLM locales, con la ventaja de latencias prácticamente nulas. El stack Nav2 alcanzó una tasa de éxito del 100 por ciento en navegación, sin colisiones ni fallos de localización AMCL durante las 60 navegaciones registradas en las pruebas.

Se implementó YOLOv8 para detección de objetos en tiempo real. Aunque el rendimiento en CPU (3-5 FPS) limitó la detección de objetos pequeños, se alcanzaron tasas del 100 por ciento para objetos grandes (TV, nevera) y del 75 por ciento para objetos pequeños (botellas) tras implementar contramedidas como velocidad reducida, múltiples pasadas y condición láser alternativa. Finalmente, se realizó un análisis exhaustivo del sistema mediante 12 casos de prueba, métricas por módulo, y 4 vídeos demostrativos que evidencian el funcionamiento real del sistema en diferentes escenarios. Los resultados identificaron fortalezas (interpretación de lenguaje, navegación) y limitaciones (percepción visual en CPU), proporcionando una base

sólida para trabajo futuro.

Todos los objetivos específicos se han cumplido, validando la viabilidad del enfoque propuesto.

6.2 Aportaciones Principales

Las principales contribuciones de este trabajo al estado del arte incluyen el diseño e implementación de una arquitectura modular híbrida Edge-Cloud, la integración de múltiples paradigmas de IA, y la tolerancia a fallos humanos en la interacción.

6.2.1 Arquitectura Modular Híbrida Edge-Cloud

Se ha diseñado e implementado una arquitectura que combina procesamiento local (edge computing) para tareas de baja latencia (navegación, detección visual, control de motores) con servicios en la nube para razonamiento de alto nivel (interpretación de lenguaje natural). Esta arquitectura híbrida permite ejecutar modelos de IA complejos (GPT-3.5) sin necesidad de hardware especializado en el robot, mantener la autonomía del robot en caso de pérdida temporal de conectividad (el planificador bayesiano local actúa como fallback del LLM remoto), y escalar el sistema fácilmente: mejorar la capacidad de razonamiento solo requiere actualizar el prompt del LLM, sin reentrenar modelos locales.

La arquitectura es agnóstica de la plataforma robótica concreta, facilitando su migración futura a robots humanoides con capacidades de manipulación.

6.2.2 Integración de Múltiples Paradigmas de IA

El sistema combina técnicas de IA de tres generaciones diferentes. De la IA clásica se utiliza un clasificador Naive Bayes con Laplace smoothing para planificación de habitaciones. Del DL se incorporan YOLOv8 para detección de objetos y Whisper para transcripción de audio. De la IA generativa se aprovecha un LLM (GPT-3.5) para extracción de comandos estructurados desde lenguaje natural desestructurado.

Esta integración demuestra que sistemas robóticos prácticos se benefician de combinar técnicas complementarias, en lugar de apostar exclusivamente por los modelos más modernos.

6.2.3 Tolerancia a Fallos Humanos

A diferencia de sistemas basados en comandos predefinidos o sintaxis estricta, este asistente acepta órdenes coloquiales, incompletas o con información incorrecta. Si el usuario dice “trae la coca cola del salón” pero el objeto está en la cocina, el sistema busca primero en el salón (respetando la indicación) y luego continúa automáticamente en el resto de habitaciones hasta encontrarlo. Si el objeto no existe, el sistema reporta claramente el fallo sin comportarse erráticamente. El cliente CLI muestra feedback en tiempo real de cada fase de la misión, permitiendo al usuario entender qué está haciendo el robot en cada momento.

Esta tolerancia acerca la robótica asistencial a usuarios sin conocimientos técnicos, uno de los requisitos clave para su adopción en hogares reales.

6.3 Lecciones Aprendidas

El desarrollo y evaluación del sistema ha revelado aprendizajes valiosos sobre robótica asistencial de bajo coste.

6.3.1 La Percepción Visual es el Cuello de Botella

Contrario a la intuición inicial de que el procesamiento de lenguaje natural sería el componente más complejo, los resultados demuestran que los módulos de IA de alto nivel (Whisper, GPT-3.5, planificador) funcionan de forma casi perfecta (86-100 por ciento de precisión) con latencias despreciables (menor de 2 segundos), representando solo el 1.3 por ciento del tiempo total de misión. Sin embargo, el detector YOLO en CPU, con tasas de 3-5 FPS, es el factor limitante del sistema. Objetos pequeños (menor de 20 píxeles) como botellas solo se detectan de forma confiable al 30-40 por ciento durante navegación, requiriendo múltiples pasadas y reducción de velocidad.

Esta observación refuerza la necesidad de investigación en algoritmos de visión eficientes para dispositivos embebidos, o la adopción de GPUs compactas de bajo consumo (como NVIDIA Jetson) en robots de servicio.

6.3.2 La Modularidad de ROS 2 Facilita el Desarrollo Iterativo

La arquitectura de nodos y tópicos de ROS 2, combinada con el patrón de comunicación basado en actions, permitió desarrollar y probar cada módulo de forma aislada antes de la integración, sustituir componentes sin afectar al resto del sistema (por ejemplo, cambiar de Ollama a OpenAI solo requirió modificar el Centro de Mando), y depurar problemas observando los mensajes publicados en cada tópico sin necesidad de instrumentación adicional.

La elección de ROS 2 sobre ROS 1 fue acertada: la comunicación basada en Data Distribution Service (DDS) demostró ser más robusta que el modelo maestro-esclavo de ROS 1, y el soporte nativo de Python 3 simplificó la integración con librerías modernas de IA.

6.3.3 La Simulación Acelera el Desarrollo pero No Sustituye Pruebas Reales

Gazebo permitió validar el sistema completo sin necesidad de hardware físico, reduciendo costes y tiempo de desarrollo. Sin embargo, se identificaron limitaciones importantes. La ausencia de motion blur, vibraciones de cámara y variaciones realistas de iluminación sugiere que el rendimiento de YOLO será probablemente peor en un robot real. Los modelos de físicas de Gazebo son deterministas y no capturan la variabilidad del mundo real (deslizamiento de ruedas, inercias reales, latencias de sensores).

Esto refuerza la importancia de la migración a hardware físico como siguiente paso del proyecto.

6.4 Impacto Social y Viabilidad

Más allá de las contribuciones técnicas, este trabajo demuestra la viabilidad de asistentes robóticos de bajo coste para mejorar la calidad de vida de personas con movilidad reducida.

Según datos del INE, el 86.2 por ciento de las personas con discapacidad en centros asistenciales presentan limitaciones de movilidad, un problema que afecta a cientos de miles de personas solo en España.

El sistema desarrollado ofrece una solución tecnológica que reduce la dependencia de cuidadores para tareas simples pero frecuentes (“trae el mando de la TV”, “acerca mi medicación”), liberando tiempo del personal sanitario para atención más compleja. Preserva la autonomía personal de los usuarios, permitiéndoles solicitar ayuda mediante órdenes de voz naturales sin necesidad de formación técnica. Además, es económicamente accesible: la plataforma TurtleBot3 cuesta aproximadamente 400 euros, y todos los componentes de software son de código abierto. El único coste recurrente es la API de OpenAI, que en las pruebas realizadas representó menos de 0.01 euros por misión.

La migración futura a un humanoide con capacidades de manipulación (como TIAGo de PAL Robotics) permitiría completar el ciclo de asistencia: no solo localizar objetos, sino entregarlos físicamente al usuario.

6.5 Reflexión Final

Este proyecto ha sido un viaje técnico y personal que ha permitido integrar conocimientos de múltiples disciplinas: IA, robótica, visión por computador, procesamiento de lenguaje natural y desarrollo de software. Más allá de los logros técnicos medibles (tasas de éxito, precisiones, tiempos de ejecución), la mayor satisfacción ha sido demostrar que la tecnología puede diseñarse con propósito social, poniendo las capacidades de la IA moderna al servicio de quienes más pueden beneficiarse de ella.

Los resultados obtenidos validan la hipótesis central del trabajo: es posible construir un asistente robótico funcional utilizando componentes de hardware accesibles y modelos de IA de propósito general, sin necesidad de desarrollos a medida extremadamente costosos. Si bien quedan desafíos importantes por resolver (especialmente en percepción visual), el sistema desarrollado representa un paso significativo hacia la democratización de la robótica asistencial.

El código, la documentación y los vídeos demostrativos de este proyecto se han publicado bajo licencia abierta, con la esperanza de que otros investigadores puedan continuar este trabajo y acercar un poco más la visión de robots asistentes presentes en hogares reales, ayudando a quienes más lo necesitan.

Bibliografía

- [1] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning (CoRL)*, 2022.
- [2] F. Amigoni, J. Banfi, and N. Basilico. Autonomous navigation and mapping with a turtlebot. *IEEE Robotics & Automation Magazine*, 22(2):142–152, 2015.
- [3] M. Beetz, D. Beßler, A. Haidu, M. Pomarlan, A. K. Bozcuoğlu, and G. Bartels. Know rob 2.0 — a 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519, 2018.
- [4] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer. Towards autonomous robotic butlers: Lessons learned with the pr2. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5568–5575, 2011.
- [5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations (ICLR)*, 2021.
- [6] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. pages 5036–5040, 2020.
- [7] W. Hess, D. Kohler, H. Rapp, and D. Andor. Real-time loop closure in 2d lidar slam. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.
- [8] Instituto Nacional de Estadística. Encuesta de discapacidad, autonomía personal y situaciones de dependencia. población residente en centros (edad centros). año 2023, 2024. Accedido: 2026-05-10.
- [9] C. C. Kemp, A. Edsinger, H. M. Clever, and B. Matulevich. The design of stretch: A compact, lightweight mobile manipulator for indoor human environments, 2022.
- [10] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154. IEEE, 2004.

-
- [11] M. Labbé and F. Michaud. Rtab-map as an open-source lidar and visual slam library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019.
 - [12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
 - [13] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.
 - [14] S. Macenski, F. Martín, R. White, and J. Ginés Clavero. The marathon 2: A navigation system. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2718–2725, 2020.
 - [15] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. Orb-slam: A versatile and accurate monocular slam system. In *IEEE Transactions on Robotics*, volume 31, pages 1147–1163, 2015.
 - [16] Ollama Team. Ollama: Get up and running with large language models locally, 2023. Accedido: 2026-05-10.
 - [17] OpenAI. Gpt-3.5 turbo, 2023. Accedido: 2026-05-10.
 - [18] J. Pagès, L. Marchionni, and F. Ferro. Tiago: The modular collaboration of mobile manipulation and manipulation. In *Workshop on Robotics for People*, 2016.
 - [19] A. K. Pandey and R. Gelin. A mass-produced sociable humanoid robot: Pepper: The first machine of its kind. In *IEEE Robotics & Automation Magazine*, volume 25, pages 40–48, 2018.
 - [20] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever. Robust speech recognition via large-scale weak supervision, 2022.
 - [21] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.
 - [22] U. Reiser, T. Jacobs, G. Arbeiter, C. Parlitz, and K. Dautenhahn. Care-o-bot 3-vision of a robot butler. In *2009 IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 97–98. IEEE, 2009.
 - [23] ROBOTIS. Turtlebot3, 2017. Accedido: 2026-05-10.
 - [24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, . Kaiser, and I. Polosukhin. Attention is all you need. 30:5998–6008, 2017.
 - [25] World Health Organization. Global report on health equity for persons with disabilities. Technical report, World Health Organization, 2022. Accedido: 2026-05-10.
-