UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Sistema Multimodal de Interacción Humano Robot basado en técnicas de IA

AI-based Multimodal System for Human-Robot Interaction

Realizado por
Eulogio Quemada Torres

Tutorizado por
Cipriano Galindo Andrades
Francisco Ángel Moreno Dueñas

Departamento
Ingeniería de Sistemas y Automática

MÁLAGA, junio de 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA DEL SOFTWARE

Sistema Multimodal de Interacción Humano Robot basado en técnicas de IA

AI-based Multimodal System for Human-Robot Interaction

Realizado por
Eulogio Quemada Torres

Tutorizado por
Cipriano Galindo Andrades
Francisco Ángel Moreno Dueñas

Departamento
Ingeniería de Sistemas y Automática

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2025

Fecha defensa: julio de 2025

Resumen

La interacción humano-robot sigue siendo un desafío decisivo para la robótica de servicio: entablar un diálogo natural exige que la máquina perciba, interprete y responda con agilidad a través de la vista, el lenguaje y el sonido. Este trabajo presenta un sistema de interacción multimodal que integra visión por computador, comprensión del lenguaje natural y síntesis de voz expresiva en una arquitectura ROS 2 cohesiva, dotando a un robot móvil de la capacidad de saludar, informar y asistir a los usuarios de forma contextual y fluida.

La propuesta se articula en servicios especializados para transcripción y síntesis del habla, modelado de intenciones, análisis facial y control corporal. Una capa de coordinación vela por la sincronía conversacional y deja margen para que cada módulo evolucione de manera independiente. Los ensayos cualitativos llevados a cabo en entornos controlados de laboratorio han evidenciado que la integración de múltiples canales sensoriales incrementa significativamente la percepción de cercanía, confianza y eficacia en comparación con sistemas basados exclusivamente en la voz.

Más allá del prototipo, el trabajo ofrece un plano realista para robots socialmente competentes: demuestra que la IA avanzada puede ejecutarse a bordo sin hardware especializado y subraya la importancia de mantener la modularidad, la transparencia y el diseño ético en los sistemas de interacción. Las conclusiones extraídas aspiran a guiar futuros desarrollos hacia robots que sean aliados empáticos y proactivos en los entornos humanos.

Palabras clave: Interacción humano-robot, interacción multimodal, inteligencia artificial, ROS 2, modelos de lenguaje

Abstract

Human–robot interaction (HRI) remains a pivotal challenge on the road to genuinely helpful service robots. Building such an interaction demands that the machine *perceive*, *interpret* and *respond* through several sensory and cognitive channels as effortlessly as a human interlocutor. This work introduces a comprehensive multimodal system that unifies computer vision, natural-language understanding and expressive speech within a cohesive ROS 2 framework, enabling a mobile robot to greet, guide and collaborate with users in a fluid, context-aware manner.

The platform is organised as loosely coupled services for speech transcription and synthesis, intent modelling, face analysis and embodiment control. A dedicated orchestration layer balances conversational timing with computational efficiency, ensuring that each module can evolve independently while preserving a coherent dialogue flow. Qualitative studies conducted in controlled laboratory environments have shown that the integration of multiple sensory channels significantly enhances users' perception of approachability, trustworthiness, and effectiveness, compared to systems relying solely on voice.

Beyond the immediate prototype, the work offers a pragmatic blueprint for socially capable robots: it demonstrates that advanced AI can be brought on-board without specialised hardware, and it highlights best practices for maintaining modularity, transparency and ethical design in HRI pipelines. The lessons distilled here aim to inform future efforts toward robots that are not merely reactive tools but active, empathetic partners in human environments.

Keywords: Human–Robot Interaction, multimodal interaction, artificial intelligence, ROS 2, large language models

Índice

Índice de figuras

Índice de tablas

1

Introducción

La robótica social representa uno de los campos más dinámicos y prometedores de la inteligencia artificial aplicada, con aplicaciones crecientes en educación, asistencia a personas mayores, museos interactivos, *retail* o robótica de compañía (Belpaeme et al., 2018; Broadbent et al., 2023; Dautenhahn, 2007; Kernbach, 2011). La interacción humano-robot (HRI, por sus siglas en inglés) se ha consolidado como uno de los pilares esenciales de esta disciplina, ya que define cómo los robots se comunican, interpretan y responden a las acciones e intenciones humanas en contextos compartidos.

Durante las últimas dos décadas, los avances en HRI han estado impulsados por mejoras en percepción visual, procesamiento del habla y modelos de comportamiento, pero la interacción seguía siendo, en muchos casos, rígida, predecible y carente de verdadera comprensión semántica. Los sistemas tradicionales se basaban en reglas, árboles de decisión o bases de conocimiento limitadas, lo que restringe severamente su capacidad para mantener conversaciones naturales y adaptativas (Lambert et al., 2020).

La irrupción de los grandes modelos de lenguaje (LLMs), como GPT (Brown et al., 2020), BERT (Devlin et al., 2019) o LLaMA (Touvron et al., 2023), ha supuesto un punto de inflexión para la HRI. Estos modelos, entrenados sobre corpus masivos y capaces de comprender y generar lenguaje de forma contextual, han permitido una mejora radical en la fluidez, flexibilidad y riqueza semántica de las interacciones entre humanos y robots. Estudios recientes han demostrado que integrar LLMs en sistemas robóticos no solo mejora la comprensión de intenciones, sino que también permite respuestas adaptadas al contexto emocional y situacional del usuario (Atuhurra, 2024; B. Zhang & Soh, 2023).

Por otro lado, la percepción artificial mediante visión por computador también ha avanzado significativamente, con sistemas capaces de detectar rostros, inferir identidades, emociones e incluso rasgos demográficos (Mellouk & Handouzi, 2020; Schroff et al., 2015). Estas

capacidades permiten que los robots reconozcan y recuerden a sus interlocutores, adaptando su comportamiento en función del historial previo de interacciones.

Estos avances han propiciado el desarrollo de múltiples robots sociales en escenarios reales. Un ejemplo representativo se muestra en la Figura 1, donde se aprecia el robot Pepper, uno de los robots sociales más conocidos y utilizados en contextos de interacción humano-robot.

Figura 1: Robot social Pepper durante una demostración pública, empleando percepción visual, voz y expresividad. [Fuente: https://commons.wikimedia.org/wiki/File:Pepper_the_Robot.jpg]

A pesar de estos avances, persisten numerosos retos técnicos: la integración efectiva de múltiples modalidades (voz, texto, imagen), la sincronización entre percepción y respuesta, la expresividad física del robot y la capacidad de supervisar y configurar estos sistemas de forma accesible para operadores humanos no técnicos. Este Trabajo de Fin de Grado propone una arquitectura que aborda de forma integral estos desafíos, integrando modelos de IA de última generación con control físico y una interfaz web moderna, validada sobre una plataforma real como el robot Sancho.

1.1. Objetivos

El objetivo principal de este trabajo es el desarrollo de un sistema multimodal de interacción humano-robot basado en técnicas de inteligencia artificial, capaz de operar de forma fluida, robusta y natural en entornos reales, combinando visión, audio, lenguaje y expresión física.

A continuación, se detallan los objetivos específicos, cada uno de los cuales corresponde a una funcionalidad clave del sistema:

- **Diseñar una arquitectura modular y desacoplada:** La base del sistema debe estar compuesta por módulos independientes y reutilizables, facilitando la escalabilidad, la integración con otros robots y la depuración durante el desarrollo. La arquitectura está basada en ROS 2 (*Robot Operating System*, (Quigley et al., 2009)) y debe permitir la comunicación eficiente entre sus nodos, así como su supervisión desde herramientas externas como APIs REST o interfaces web.

- **Implementar un sistema de reconocimiento facial adaptativo:** Se desarrollará un sistema de detección y reconocimiento de rostros que permita identificar a personas conocidas y aprender nuevas identidades en tiempo real. Este sistema debe ser capaz de operar en condiciones de iluminación variable, con expresiones faciales distintas, y debe integrarse con una base de datos ligera, exportable y editable desde la interfaz web.

- **Desarrollar un módulo de entrada por voz:** El sistema de entrada auditiva debe ser capaz de detectar de forma inteligente los fragmentos de audio que contienen voz humana relevante, descartando ruido o pausas innecesarias. Para ello, se integrará un mecanismo de detección automática de voz (*Voice Activity Detection*, VAD), junto con una palabra de activación configurable que asegure que la interacción comienza de forma intencionada por parte del usuario.

- **Integrar modelos de transcripción de voz a texto:** Una vez capturado el audio válido, se procesará mediante modelos de transcripción automática del habla (*Speech-to-Text*, STT) de última generación —como Whisper (Radford et al., 2023)—, tanto en versión local como en la nube, con el objetivo de obtener una transcripción precisa, rápida y en tiempo real de la entrada del usuario.

- **Desarrollar un sistema de comprensión de intenciones basado en LLMs:** La entrada textual del usuario será interpretada por modelos de lenguaje (LLMs) como GPT (Brown et al., 2020), Mistral (Mistral AI, 2024) o Gemini (Google DeepMind, 2024), capaces de identificar la intención del mensaje y extraer los argumentos necesarios para

su ejecución. Esta capa de razonamiento conversacional es clave para dotar al robot de flexibilidad lingüística y adaptabilidad contextual.

- **Sintetizar respuestas auditivas con modelos TTS expresivos:** Las respuestas generadas por el sistema serán convertidas en audio mediante modelos de TTS (Text-to-Speech), seleccionando voces naturales, expresivas y, en la medida de lo posible, multilingües. El sistema debe permitir seleccionar modelos locales o *cloud*, así como cambiar de voz dinámicamente.

- **Hacer la interacción más amigable mediante expresividad visual:** Se implementará un sistema de animación visual que simule el movimiento de la boca del robot, utilizando una tira LED RGB cuya intensidad y patrón estarán sincronizados con el volumen y la cadencia del audio. Este módulo se desarrollará en C++ para optimizar su rendimiento e integrarse eficientemente con el flujo de audio del sistema.

- **Diseñar una interfaz web de control y supervisión:** La web debe permitir al usuario visualizar la percepción del robot (quién habla, qué dice, a quién ve), gestionar la base de datos de identidades, cambiar modelos de IA, y consultar estadísticas de interacción, todo ello en tiempo real mediante tecnologías como WebSockets (Fette & Melnikov, 2011).

- **Asegurar la compatibilidad con arquitecturas distribuidas:** Dado que muchos modelos IA requieren un alto rendimiento computacional, se diseñará el sistema para delegar el procesamiento pesado a un servidor externo cuando sea necesario, garantizando así que el robot pueda operar con autonomía sin comprometer sus recursos internos.

1.2. Tecnologías empleadas

El sistema ha sido desarrollado combinando herramientas de desarrollo profesional, librerías de inteligencia artificial y frameworks específicos de robótica, visión por computador y procesamiento de audio. La descripción técnica detallada de estas herramientas se encuentra en la Sección 4.1, dentro del capítulo de implementación. No obstante, a continuación se muestra un resumen general de las tecnologías más relevantes utilizadas en este trabajo (Tabla 1):

Tecnología	Descripción breve
ROS 2 (Humble)	*Middleware* robótico para comunicación distribuida entre nodos.
Python	Lenguaje principal del sistema: nodos, *backend*, lógica de control.
FastAPI	*Framework backend* para exponer la API REST del sistema.
React	Librería JavaScript para construir la interfaz web.
WebSockets	Canal en tiempo real entre la web y ROS 2.
Arduino IDE	Control embebido de la boca y ojos del robot.
FaceNet	Extracción de *embeddings* faciales para reconocimiento.
Whisper	Transcripción de voz mediante red neuronal.
SQLite	Base de datos local para sesiones, *logs* y métricas.
PyQt6	Interfaz local para el registro facial.
Git y GitHub	Control de versiones y gestión colaborativa del repositorio.
Trello y Notion	Organización de tareas, ideas y documentación técnica.

Tabla 1: Resumen de tecnologías principales utilizadas en el sistema.

1.3. *Hardware* del sistema

La plataforma robótica utilizada en este TFG es el robot **Sancho**, un robot social de propósito general orientado a la interacción humano-robot. Su arquitectura física está basada en la base móvil *AgileX Ranger Mini V3* y combina capacidades de navegación autónoma con mecanismos expresivos para la interacción empática. Cabe destacar que este robot es el mismo que aparece en el trabajo presentado en (Gonzalez et al., 2009), pero años después tras distintas mejoras en sus componentes.

Entre sus componentes principales destacan:

- **Base AgileX Ranger Mini V3**: plataforma robusta para entornos interiores y semiexteriores, con control diferencial y suspensión independiente.
- **Láseres Hokuyo**: dos escáneres láser 2D situados en esquinas opuestas (frontal derecha y trasera izquierda), girados 45° respecto al eje principal, que permiten una percepción amplia del entorno.

- **Cámara RGB-D Orbbec Astra**: montada en la parte superior, permite estimar la distancia y posición de los usuarios, ajustando trayectorias de navegación.
- **Cámara RGB frontal**: ubicada en la cabeza robótica, enfocada a la interacción directa con el usuario.
- **Tira LED RGB con 12 segmentos como boca**: conectada a un microcontrolador **ESP32**, simula una boca expresiva con control independiente de color (valores RGB de 0 a 255) e intensidad.
- **Pantallas RGB para los ojos**: permiten mostrar expresiones animadas y refuerzan la capacidad empática del robot. Esta parte fue desarrollada en otro Trabajo de Fin de Grado.

Esta configuración dota al robot de capacidades completas tanto para navegación reactiva como para interacción social, combinando sensores para percepción y elementos visuales para la expresión emocional. Una visión general del robot y sus componentes se muestra en la Figura 2.

Figura 2: Vista general del robot social Sancho y sus componentes físicos. [Fuente: propia.]

El sistema computacional que controla los módulos de inteligencia artificial y la interacción con el usuario está basado en un miniPC de bajo consumo, montado directamente en el chasis del robot. Este ordenador ejecuta los nodos principales de ROS 2 y permite la operación autónoma del sistema sin necesidad de conexión a un servidor externo. Sus características técnicas se recogen en la Tabla 2, y una vista del dispositivo físico se muestra en la Figura 3.

Componente	Especificación
Procesador	AMD Ryzen 7 4800U (8 núcleos / 16 hilos, hasta 4.2 GHz)
Memoria RAM	16 GB DDR4 a 3200 MHz
GPU	AMD Radeon integrada (8 núcleos, 1750 MHz)
Almacenamiento	512 GB NVMe SSD
Conectividad	Wi-Fi 6, Bluetooth 5.1, Ethernet 1 GbE + 2.5 GbE
Puertos	1× HDMI, 1× DisplayPort, USB-C, 6× USB-A
Sistema operativo	Ubuntu 22.04 LTS (reemplazando Windows 10 Pro preinstalado)

Tabla 2: Especificaciones del miniPC integrado en el robot Sancho para la ejecución local de los módulos.

Figura 3: Ordenador integrado en el robot Sancho utilizado para ejecutar los módulos en tiempo real. [Fuente: propia.]

Aunque el robot cuenta con funcionalidades avanzadas de movimiento, navegación y percepción espacial, en este TFG no se profundiza en dichas capacidades. El enfoque principal de este trabajo se centra en el desarrollo y mejora de un sistema de interacción humano-robot multimodal, con especial atención a la comprensión de intenciones, el reconocimiento de usuarios y la expresividad del robot.

1.4. Estructura del documento

Este documento está estructurado en seis capítulos principales y tres apéndices, organizados de forma que el lector pueda comprender tanto la motivación general como el diseño metodológico, así como los detalles técnicos y los resultados obtenidos.

- **Capítulo 1: Introducción.** Se exponen la motivación del trabajo, los objetivos generales y específicos, y una visión global de la estructura del documento.
- **Capítulo 2: Estado del arte.** Se analiza la literatura y tecnologías existentes en las áreas clave del proyecto: visión artificial, síntesis y reconocimiento de voz, modelos de lenguaje, HRI y arquitecturas robóticas modernas.
- **Capítulo 3: Diseño del sistema.** Se detalla la metodología seguida durante el desarrollo, las decisiones arquitectónicas tomadas y la organización general de los módulos del sistema.
- **Capítulo 4: Implementación.** Se describe en profundidad el funcionamiento interno de los paquetes ROS 2 desarrollados, las tecnologías utilizadas, el flujo de información entre módulos y su integración con *hardware*, la interfaz web y la interfaz de usuario del robot.
- **Capítulo 5: Validación y pruebas.** Se presentan diversos escenarios de uso del sistema, los resultados obtenidos, ejemplos reales de interacción y una evaluación cualitativa de su funcionamiento.
- **Capítulo 6: Conclusiones y trabajos futuros.** Se resumen las principales contribuciones del proyecto, las lecciones aprendidas y las posibilidades de mejora o extensión a medio y largo plazo.

Asimismo, se incorporan cuatro apéndices finales:

- **Apéndice A: Manual de instalación y despliegue.** Explica los pasos necesarios para instalar y poner en marcha el sistema completo, incluyendo dependencias, parámetros y configuraciones específicas.

- **Apéndice B: Prompts utilizados en la lógica conversacional.** Reúne y documenta los sistemas de *prompt* diseñados para el uso de modelos LLM, incluyendo la extracción de intenciones, argumentos, diálogos y generación de respuestas.
- **Apéndice C: Protocolos de comunicación.** Detalla los formatos de mensaje empleados entre nodos ROS y entre el sistema y la web, incluyendo el protocolo *ros2web*, mensajes de control, eventos y respuestas estructuradas.
- **Apéndice D: Definición de mensajes ROS2.** Para la comunicación entre nodos ROS2 se han definido una serie de mensajes y servicios que se usan en la implementación. En este anexo, se especifican estos mensajes y su estructura.

2

Estado del arte

Este capítulo presenta una revisión del estado del arte en las tecnologías que conforman la base del sistema propuesto. Se describen los principales avances en síntesis y reconocimiento de voz, modelos de lenguaje, *embeddings* semánticos, visión por computador, y comunicación web distribuida, así como las tendencias actuales en percepción multimodal. El objetivo es contextualizar las decisiones tecnológicas adoptadas en el desarrollo del sistema, destacando las capacidades, limitaciones y retos de cada componente.

2.1. Síntesis de voz

La síntesis de voz (*Text-to-Speech*, TTS) es una tecnología fundamental en interfaces conversacionales, al permitir la generación de habla artificial a partir de texto. Su evolución ha estado marcada por tres grandes paradigmas: los enfoques concatenativos (basados en la unión de unidades pregrabadas), los modelos articulatorios (que simulan el tracto vocal humano) y, en la última década, los modelos neuronales de síntesis paramétrica.

Entre los avances más destacados se encuentran Tacotron2 (Shen et al., 2018), que introdujo una arquitectura encoder-decoder con atención para generar espectrogramas mel[1], y FastSpeech (Ren et al., 2019), que aceleró el proceso mediante predicción paralela. Estos espectrogramas se transforman en audio mediante vocoders como WaveNet (Van Den Oord et al., 2016) o HiFi-GAN (Kong et al., 2020). La combinación de ambos modelos logra una síntesis natural con alta fidelidad y entonación fluida.

Una comparativa entre los principales sistemas de síntesis de voz y sus características se recoge en la Tabla 3.

[1]Los espectrogramas mel son representaciones del espectro de frecuencias del audio que imitan la percepción auditiva humana. Véase (Shen et al., 2018) para una explicación técnica.

Modelo	Naturaleza	Expresividad	Multilingüe
Tacotron2	Neuronal (autoregresivo)	Media	Limitada
FastSpeech	Neuronal (no autoregresivo)	Media	Parcial
WaveNet	Vocoder neuronal	Alta	No
HiFi-GAN	Vocoder neuronal	Alta	Parcial
XTTS	Neuronal (multilingüe)	Alta	Sí
YourTTS	Zero-shot	Alta	Sí
Piper	Neuronal optimizado	Media	Sí

Tabla 3: Comparativa entre distintos sistemas de síntesis de voz.

Recientemente, la investigación ha girado hacia modelos expresivos y multilingües, como XTTS (Casanova et al., 2024) o VITS (Kim et al., 2021), con capacidad para emular prosodia emocional, cambios de idioma y estilos de habla. Modelos como YourTTS (Casanova et al., 2022) o VALL-E (C. Wang et al., 2023) han impulsado la síntesis *zero-shot*, donde es posible clonar una voz a partir de pocos segundos de referencia. Paralelamente, se han desarrollado soluciones como Piper (rhasspy, 2023), que priorizan eficiencia para su uso en dispositivos embebidos o aplicaciones de robótica con recursos limitados.

Los desafíos actuales incluyen la reducción de la latencia, la mejora de la expresividad controlable, la adaptación multilingüe con pocos datos, y la integración natural con señales no verbales, como gestos o expresiones faciales.

2.2. Reconocimiento automático del habla

El reconocimiento automático del habla (*Speech-to-Text*, STT) permite transcribir audio hablado en texto. Es una pieza crítica en interfaces habladas, ya que constituye el canal de entrada para tareas de comprensión y respuesta. Su desarrollo ha transitado desde modelos HMM-GMM hasta redes neuronales profundas basadas en arquitecturas como CTC, atención o transductores.

Modelos como DeepSpeech, Wav2Vec2 y HuBERT han supuesto hitos importantes. Sin embargo, ha sido Whisper de OpenAI (Radford et al., 2023) quien ha marcado un cambio de paradigma. Entrenado con cientos de miles de horas multilingües y ruidosas, Whisper presenta

una robustez notable frente a ruido, acentos y condiciones adversas. Su arquitectura encoder-decoder basada en Transformers (Vaswani et al., 2017) lo convierte en un modelo versátil tanto para STT como para tareas auxiliares como detección de idioma o segmentación.

Una comparativa de los principales modelos y servicios de reconocimiento de voz, se muestra en la Tabla 4.

Sistema	**Naturaleza**	**Multilingüe**	**Robustez al ruido**
DeepSpeech	Neuronal (CTC)	Limitada	Media
Wav2Vec2	Neuronal (auto-supervisado)	Sí	Alta
HuBERT	Neuronal (preentrenado)	Sí	Alta
Whisper	Neuronal (Transformer)	Sí	Muy alta
Google STT	API comercial	Sí	Alta
Amazon Transcribe	API comercial	Sí	Alta
Microsoft Azure	API comercial	Sí	Alta
Silero-VAD	Detección VAD neuronal	No	Alta

Tabla 4: Comparativa entre distintos sistemas de reconocimiento automático del habla.

Frente a estas soluciones *open-source*, proveedores como Google, Amazon o Microsoft ofrecen APIs comerciales con baja latencia, modelos entrenados con datos masivos, y capacidad de personalización por usuario o dominio. Además, técnicas como la detección de actividad vocal (VAD), mediante modelos como Silero-VAD, y los sistemas de activación por palabra clave (*hotword detection*) han mejorado la eficiencia y la precisión en sistemas siempre activos.

Las líneas actuales de investigación incluyen la diarización automática, el reconocimiento de habla espontánea o afectiva, la transcripción multilingüe en tiempo real y la combinación con modelos de lenguaje para corrección contextual.

2.3. Modelos de lenguaje

Los modelos de lenguaje de gran tamaño (*Large Language Models*, LLMs) representan uno de los avances más transformadores en inteligencia artificial. Basados en arquitecturas Transformer, estos modelos son capaces de generar texto coherente, comprender instrucciones complejas, realizar razonamiento semántico y adaptarse a contextos conversacionales variados.

Desde GPT-2 hasta GPT-4 (Brown et al., 2020), se ha evidenciado una escalada exponencial en parámetros, capacidades y versatilidad. Modelos *open-source* como LLaMA (Touvron et al., 2023), DeepSeek, Mistral, Qwen o Yi han democratizado el acceso a estas capacidades, facilitando su uso local en sistemas especializados. Además, modelos como Claude (Anthropic, 2023) o Gemini han sido diseñados con énfasis en el alineamiento conversacional y la seguridad.

En el contexto de interacción humano-robot (HRI), los LLMs permiten interpretar intenciones, generar respuestas naturales y mantener contextos de diálogo prolongados (B. Zhang & Soh, 2023). No obstante, su integración directa en sistemas robóticos plantea retos como la latencia, el control sobre la salida generada, el uso eficiente de recursos y la explicabilidad de las decisiones. Por ello, se investiga el uso de *prompts* estructurados, planificación híbrida neurosimbólica, y adaptación mediante *fine-tuning* o técnicas como RAG (*Retrieval-Augmented Generation*) (Lewis et al., 2020).

2.4. *Embeddings* semánticos de texto

Los embeddings semánticos son representaciones vectoriales densas que codifican el significado de fragmentos de texto en un espacio latente. Su principal ventaja es la capacidad de capturar relaciones semánticas más allá de coincidencias léxicas, permitiendo medir similitud entre frases, agrupar conceptos o realizar clasificación semántica.

Entre los enfoques más utilizados se encuentran Sentence-BERT (SBERT) (Reimers & Gurevych, 2019), MiniLM (W. Wang et al., 2020), E5 (L. Wang et al., 2024) y BGE-M3 (Chen et al., 2024). Muchos de estos modelos utilizan técnicas de aprendizaje contrastivo para maximizar la distancia entre frases no relacionadas y minimizarla entre aquellas con significado similar. OpenAI ha popularizado el uso de modelos de propósito general como **text-embedding-ada-002**, mientras que otros modelos como **E5-large-v2** o **BGE-M3** ofrecen alternativas de ejecución local con gran rendimiento.

En sistemas HRI, los embeddings son cruciales para la detección de intenciones, clasificación de comandos, agrupación de entradas similares y recuperación de información relevante. Además, permiten construir representaciones compartidas entre lenguaje natural y otras modalidades, facilitando la integración de contexto visual o auditivo.

2.5. Detectores de rostros

La detección facial ha evolucionado desde enfoques clásicos basados en características manuales hasta sistemas neuronales robustos entrenados con grandes volúmenes de datos. En sus inicios, métodos como las cascadas de Haar propuestas por Viola y Jones (Viola & Jones, 2001) permitieron la detección en tiempo real mediante clasificadores simples en cascada. Posteriormente, técnicas como el descriptor HOG (*Histogram of Oriented Gradients*) (Dalal & Triggs, 2005) junto con clasificadores lineales dieron lugar al detector frontal de Dlib (King, 2009), combinando simplicidad y eficiencia.

El auge del aprendizaje profundo trajo consigo detectores más precisos y versátiles. El modelo MTCNN (K. Zhang et al., 2016), basado en una arquitectura de tres etapas, logró detectar rostros y puntos clave faciales con alta fiabilidad. Paralelamente, Dlib propuso su propia red convolucional entrenada específicamente para detección de caras, con mayor robustez pero a costa de un mayor coste computacional.

En la última generación de detectores destacan los modelos basados en arquitecturas de tipo YOLO, como YOLOv5-face (Qi et al., 2022) y YOLOv8-face (Varghese & Sambath, 2024), optimizados para detectar múltiples rostros en tiempo real incluso en escenas complejas. Otros enfoques como RetinaFace (Deng et al., 2020) han logrado una alta precisión gracias a su diseño *anchor-based* y supervisión adicional con *landmarks*. Por su parte, el detector ligero incluido en el modelo *buffalo_l* de InsightFace (InsightFace, 2023) ofrece un equilibrio entre precisión y eficiencia, siendo especialmente adecuado para aplicaciones embebidas o en tiempo real.

Una comparativa entre los principales detectores faciales y sus características se muestra en la Tabla 5.

Detector	Tipo	Precisión	Velocidad
OpenCV (Haar)	Clásico cascada	Baja	Muy alta
Dlib frontal	HOG + SVM	Baja	Alta
Dlib CNN	CNN personalizada	Alta	Baja
MTCNN	CNN multietapa	Alta	Media
YOLOv5-face	One-stage CNN	Alta	Alta
YOLOv8-face	One-stage CNN	Muy alta	Muy alta
RetinaFace	Anchor-based CNN	Muy alta	Media
InsightFace-lite	CNN optimizado	Alta	Muy alta

Tabla 5: Comparativa entre distintos detectores faciales.

2.6. Codificadores faciales

Una vez detectado un rostro en una imagen, el siguiente paso clave en el reconocimiento facial es extraer una representación vectorial del mismo, conocida como embedding. Estos vectores permiten comparar rostros de manera eficiente mediante métricas de similitud. A lo largo de los años, los codificadores faciales han evolucionado desde redes entrenadas sobre datasets relativamente pequeños hasta modelos de gran escala con capacidades generales de representación.

Entre los primeros modelos ampliamente utilizados se encuentra VGG-Face (Parkhi et al., 2015), una red convolucional profunda entrenada sobre un conjunto masivo de imágenes de celebridades. Posteriormente, FaceNet (Schroff et al., 2015) introdujo un enfoque basado en triplet loss, generando embeddings normalizados de dimensión 512 especialmente robustos para verificación y reconocimiento.

Modelos como OpenFace (Amos et al., 2016) y SFace (J. Wang et al., 2018) surgieron con la intención de ofrecer alternativas más ligeras y rápidas, adecuadas para sistemas embebidos o entornos con recursos limitados. OpenFace, inspirado en FaceNet, genera vectores de 512 dimensiones con buena eficiencia, mientras que SFace optimiza tanto precisión como velocidad con embeddings compactos también de 128 dimensiones.

Uno de los codificadores más precisos en la actualidad es ArcFace (Deng et al., 2019), que

emplea una función de pérdida aditiva angular para lograr una separación óptima entre identidades faciales. Sus embeddings de 512 dimensiones han establecido nuevos estándares en benchmarks de reconocimiento facial.

Finalmente, cabe destacar el uso de modelos visuales generales como DINOv2 (Oquab et al., 2023), que, a pesar de no estar entrenado específicamente para reconocimiento facial, resulta interesante probar un modelo de este tipo para intentar realizar clasificación. Su enfoque autosupervisado lo hace especialmente atractivo en escenarios sin anotaciones.

La Tabla 6 resume las características más relevantes de los principales codificadores faciales utilizados en este trabajo.

Modelo	**Tipo**	**Dimensión**	**Velocidad**
FaceNet	Triplet CNN	512	Alta
VGG-Face	CNN clásica	4096	Media
OpenFace	CNN ligera	128	Muy alta
SFace	CNN eficiente	128	Alta
ArcFace	Angular margin loss	512	Alta
DINOv2	ViT auto-supervisado	768	Baja

Tabla 6: Comparativa entre distintos codificadores faciales.

2.7. Comunicación web y arquitecturas distribuidas

ROS 2 (*Robot Operating System*) es *middleware* para robots moderno y modular que permite el desarrollo distribuido mediante la ejecución de múltiples nodos que se comunican de forma asíncrona a través de *topics*, servicios y acciones (Quigley et al., 2009).

Diseñado con foco en la escalabilidad, interoperabilidad y soporte a tiempo real, se ha convertido en un estándar en robótica académica e industrial.

La robótica moderna tiende hacia arquitecturas distribuidas en las que múltiples nodos cooperan mediante redes locales o internet. La integración entre ROS 2 y sistemas web requiere mecanismos eficientes y seguros para exponer datos y controlar el robot desde interfaces gráficas.

La solución más extendida es ROSBridge (Crick et al., 2016), que permite exponer *topics*,

servicios y acciones ROS mediante WebSockets. Sin embargo, su complejidad y latencia no siempre son óptimas para aplicaciones ligeras o especializadas. Por ello, han emergido soluciones más personalizadas que implementan sus propios protocolos JSON sobre *WebSockets*, con suscripción dinámica, control granular y envío estructurado de eventos.

Estas arquitecturas permiten desacoplar el frontend del backend robótico, facilitar el desarrollo de interfaces interactivas en React o Vue, y habilitar el control remoto, la monitorización en tiempo real y la visualización de datos complejos (audio, vídeo, estado interno, etc.).

2.8. Percepción auditiva y visual multimodal

La percepción multimodal busca integrar diferentes fuentes sensoriales para construir una representación más rica y robusta del entorno y los interlocutores. En HRI, esto incluye combinar audio, vídeo, texto y contexto temporal para comprender la intención, el estado emocional o la identidad del usuario.

En audio, herramientas como Silero-VAD permiten detectar con precisión la presencia de habla. Sistemas como Pyannote.audio permiten la identificación de hablantes, estimar la dirección del sonido y segmentar turnos de conversación. Estos avances permiten gestionar interacciones con múltiples usuarios simultáneamente, algo clave en entornos sociales.

En visión artificial, se utilizan modelos CNN o ViT (*Visual Transformers*) (Dosovitskiy et al., 2020) para el reconocimiento de expresiones faciales, detección de gestos o análisis de la atención visual. Combinando ambas modalidades, es posible adaptar la interacción según el tono emocional, el interés del usuario o la dinámica grupal, lo que abre el camino hacia robots empáticos y contextualmente conscientes.

3

Diseño del sistema

Este capítulo describe en detalle el proceso de diseño del sistema desarrollado, abarcando desde la definición de requisitos funcionales y técnicos hasta la arquitectura final implementada. El objetivo es proporcionar una visión clara y estructurada de cómo se ha concebido, modelado e integrado cada uno de los componentes que conforman el sistema de interacción humano-robot.

A lo largo del capítulo se expone la metodología de desarrollo empleada, el modelado arquitectónico del sistema y su integración multimodal en tiempo real. También se presentan los prototipos de interfaz desarrollados, el diseño del modelo de datos unificado, y las decisiones adoptadas para garantizar la escalabilidad, extensibilidad y compatibilidad con entornos en la nube. Por último, se incluyen consideraciones técnicas y éticas que han guiado el diseño global del sistema.

3.1. Especificación de requisitos

A continuación, se enumeran los requisitos funcionales y no funcionales del sistema, agrupados por categorías lógicas. Estos requisitos han sido definidos a partir del análisis de objetivos y características técnicas del sistema, siguiendo buenas prácticas de ingeniería del software.

Requisitos Funcionales

1. Entrada y salida por voz

- **RF01:** El sistema deberá transcribir en tiempo real la voz del usuario utilizando modelos STT locales o remotos.
- **RF02:** El sistema deberá generar respuestas habladas utilizando síntesis de voz.

- **RF03:** El sistema deberá permitir la selección dinámica del modelo TTS y STT a utilizar.
- **RF04:** El sistema deberá detectar automáticamente la activación por palabra clave (*hotword*).
- **RF05:** El sistema deberá utilizar un sistema de detección de actividad vocal (VAD) para activar y desactivar el micrófono.

2. Comprensión semántica e interpretación

- **RF06:** El sistema deberá interpretar la intención del usuario a partir del texto transcrito mediante un modelo LLM.
- **RF07:** El sistema deberá extraer los argumentos relevantes asociados a cada intención reconocida.
- **RF08:** El sistema deberá permitir cambiar entre múltiples modelos LLM disponibles, tanto locales como por API.

3. Reconocimiento facial y gestión de usuarios

- **RF09:** El sistema deberá detectar rostros en la imagen capturada por la cámara frontal.
- **RF10:** El sistema deberá generar *embeddings* faciales y compararlos con una base de datos de usuarios registrados.
- **RF11:** El sistema deberá registrar nuevos usuarios a partir de su rostro, con consentimiento explícito.
- **RF12:** El sistema deberá permitir editar, renombrar y eliminar usuarios desde una interfaz web.

4. Visualización y control desde interfaz web

- **RF13:** El sistema deberá mostrar en la web los usuarios reconocidos y su historial de interacciones.
- **RF14:** El sistema deberá mostrar la transcripción, intención y respuesta generada para cada interacción.

- **RF15:** El sistema deberá permitir seleccionar el modelo de cada componente (STT, TTS, LLM, *Embedding*) desde la interfaz web.
- **RF16:** El sistema deberá mostrar eventos del sistema en tiempo real mediante WebSockets.

5. Control embebido y expresión robótica

- **RF17:** El sistema deberá controlar en tiempo real una tira de LEDs conectada a un microcontrolador para simular el movimiento de boca.
- **RF18:** El sistema deberá modificar la expresión visual (ojos y boca) en función del estado del robot (escuchando, hablando, pensando).

Requisitos No Funcionales

Usabilidad

- **RNF01:** El sistema deberá incluir una interfaz web con diseño responsive, accesible desde dispositivos móviles y de escritorio.
- **RNF02:** El sistema deberá proporcionar retroalimentación clara al usuario sobre las acciones realizadas, incluyendo confirmaciones, mensajes de error y estado del sistema.

Seguridad y privacidad

- **RNF03:** El sistema deberá almacenar todos los datos biométricos y personales de forma local, evitando su envío a servicios externos.
- **RNF04:** El sistema deberá tratar de forma segura las claves API de los operadores que usen el sistema.

Portabilidad y mantenibilidad

- **RNF05:** El sistema deberá ser compatible con arquitecturas Linux x86_64 y ARM64, con instrucciones de instalación documentadas.
- **RNF06:** El sistema deberá estructurarse en módulos ROS 2 independientes, reutilizables y con interfaces bien definidas.

3.2. Metodología de desarrollo

El desarrollo del sistema se ha llevado a cabo siguiendo una metodología **iterativa e incremental**, en la que cada componente funcional ha evolucionado a través de ciclos sucesivos de diseño, validación e integración. Esta estrategia ha permitido construir el sistema de forma progresiva, adaptándose a los cambios en los requisitos, integrando nuevas tecnologías conforme se evaluaban, y asegurando una arquitectura modular y escalable desde sus primeras etapas.

Enfoque modular e incremental

Cada bloque funcional —voz, visión, razonamiento, interfaz gráfica— se ha concebido como una unidad independiente, con interfaces bien definidas y sin dependencias implícitas. Este diseño modular ha permitido desarrollar, probar y refactorizar cada componente de forma aislada antes de integrarlo progresivamente en el sistema completo.

Como se ilustra en la Figura 4, cada iteración del desarrollo se ha centrado en:

- Definir el comportamiento esperado de cada módulo, sus entradas y salidas.
- Implementar una versión inicial funcional y validarla con datos de prueba.
- Integrar el módulo en el sistema completo y ajustar su interoperabilidad.
- Refactorizar y documentar tras cada integración para mantener consistencia y calidad.

Este enfoque ha facilitado la incorporación paulatina de componentes complejos como modelos LLM, sistemas STT neuronales o control embebido, permitiendo que el diseño evolucionase en respuesta a necesidades reales.

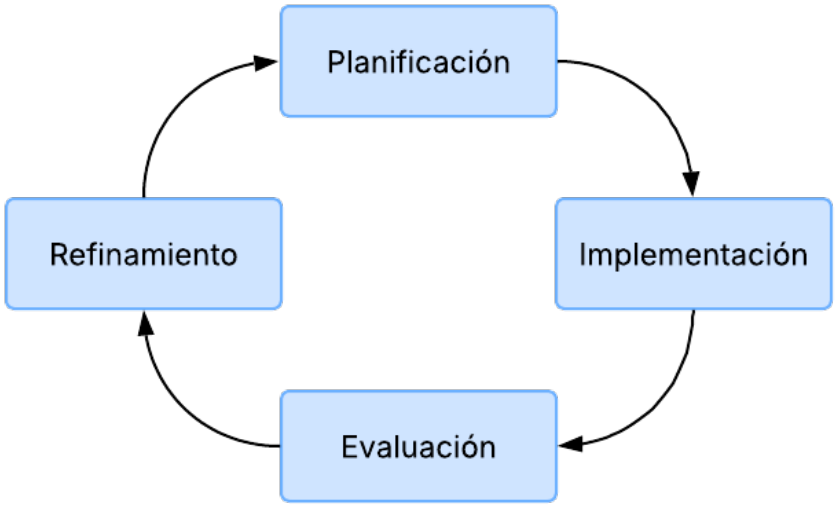


Figura 4: Ciclo seguido en cada iteración. [Fuente: propia.]

Gestión técnica y planificación del desarrollo

El proyecto se ha gestionado mediante una herramienta visual tipo Kanban, que ha permitido organizar tareas por funcionalidades, prioridad y estado. Cada funcionalidad se ha dividido en subtareas de desarrollo, prueba e integración, manteniendo así un control continuo sobre el avance del sistema. Esta dinámica puede observarse en la Figura 5, donde se muestra el tablero de trabajo con las tareas categorizadas por columnas.

El uso de herramientas de control de versiones ha permitido gestionar adecuadamente la evolución del sistema, facilitando el trabajo incremental y la trazabilidad de los cambios. Las funcionalidades principales han estado organizadas en ramas independientes, integradas de forma progresiva sobre una versión estable común.

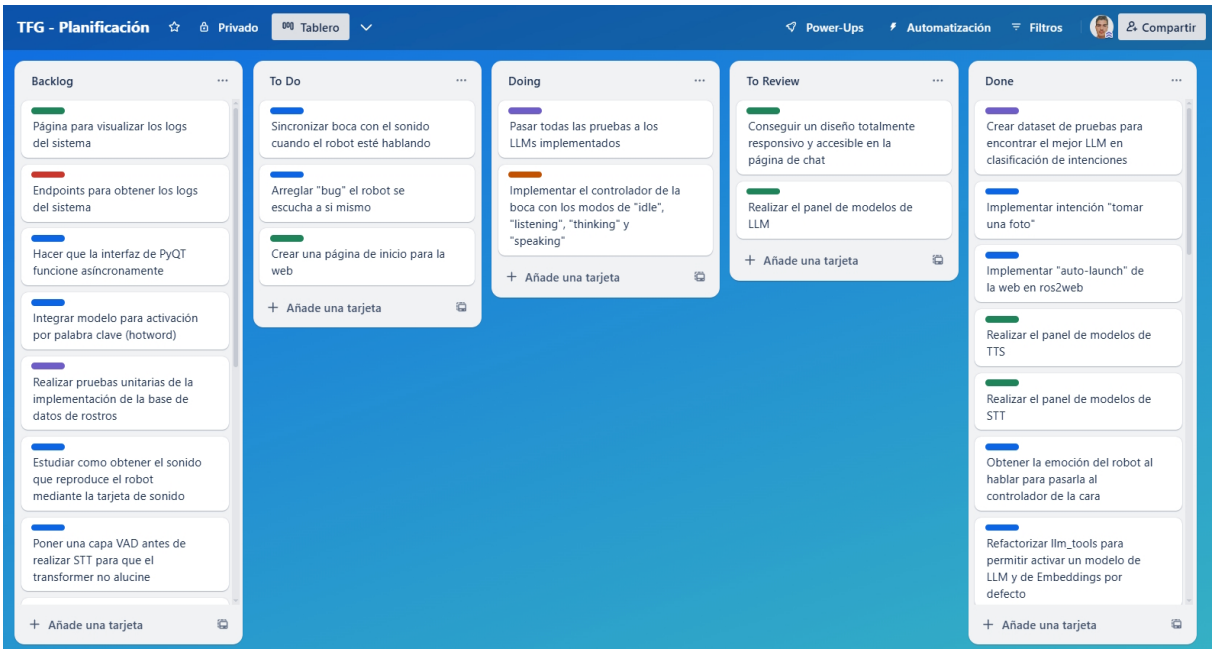


Figura 5: Tablero de planificación con tareas organizadas por funcionalidad y prioridad. [Fuente: propia.]

Buenas prácticas de diseño

Durante el proceso se han seguido principios de ingeniería que han guiado el diseño general:

- **Separación de responsabilidades:** cada módulo realiza una función concreta, con mínima dependencia del resto.
- **Interfaces claras y estables:** comunicación entre módulos mediante tópicos, servicios ROS 2 o canales web bien definidos.
- **Reutilización y extensibilidad:** el diseño favorece la incorporación de nuevos modelos o nodos sin modificar la estructura existente.
- **Documentación continua:** se ha generado documentación funcional y técnica paralelamente al desarrollo para garantizar la mantenibilidad.

Ciclo de desarrollo progresivo

A pesar de la flexibilidad del enfoque, el desarrollo ha seguido una progresión estructurada que ha permitido avanzar desde el concepto general hasta la implementación completa:

- **Fase 1:** Definición de requisitos y objetivos funcionales.
- **Fase 2:** Diseño modular de la arquitectura general.
- **Fase 3:** Desarrollo aislado de módulos clave (voz, visión, razonamiento).
- **Fase 4:** Integración de subsistemas en un flujo conversacional único.
- **Fase 5:** Pruebas funcionales en escenarios reales, ajuste de tiempos y sincronización.
- **Fase 6:** Documentación técnica, validación final y preparación para despliegue.

Esta organización ha hecho posible construir un sistema complejo de forma progresiva, sin perder la cohesión entre componentes ni comprometer la calidad del resultado final.

3.3. Modelado del sistema

El modelado del sistema constituye una herramienta clave para comprender su organización interna, su arquitectura de componentes, la estructura de datos y las relaciones funcionales entre módulos. A lo largo de esta sección se presentan distintos niveles de modelado: desde la arquitectura lógica y física del sistema, hasta la representación de datos, flujos de interacción y prototipos de interfaz gráfica.

3.3.1. Arquitectura general del sistema

El sistema desarrollado presenta una arquitectura modular, escalable y desacoplada, diseñada para integrar múltiples modalidades de interacción —voz, visión, lenguaje y expresión facial— en un entorno distribuido, flexible y controlable de forma remota. Esta arquitectura se basa en el *middleware* **ROS 2**, que actúa como núcleo de comunicación entre nodos funcionales, y se complementa con una **interfaz web avanzada** que permite gestionar, supervisar y extender el comportamiento del sistema desde cualquier dispositivo conectado a la red.

Visión funcional por capas

La arquitectura general se organiza en tres capas funcionales bien diferenciadas, que permiten entender el flujo completo de información desde la entrada sensorial hasta la salida expresiva:

1. **Capa de percepción y entrada:** agrupa los subsistemas encargados de capturar y analizar información sensorial, incluyendo audio y vídeo.
2. **Capa de procesamiento y razonamiento:** integra los modelos de comprensión semántica, clasificación de intenciones, generación de respuestas y gestión de embeddings.
3. **Capa de salida y expresión:** se encarga de materializar las respuestas mediante voz y señales físicas, y de comunicarse con el exterior.

Este diseño se representa de forma esquemática en la Figura 6, donde puede observarse cómo cada capa cumple una función específica dentro del flujo interactivo.

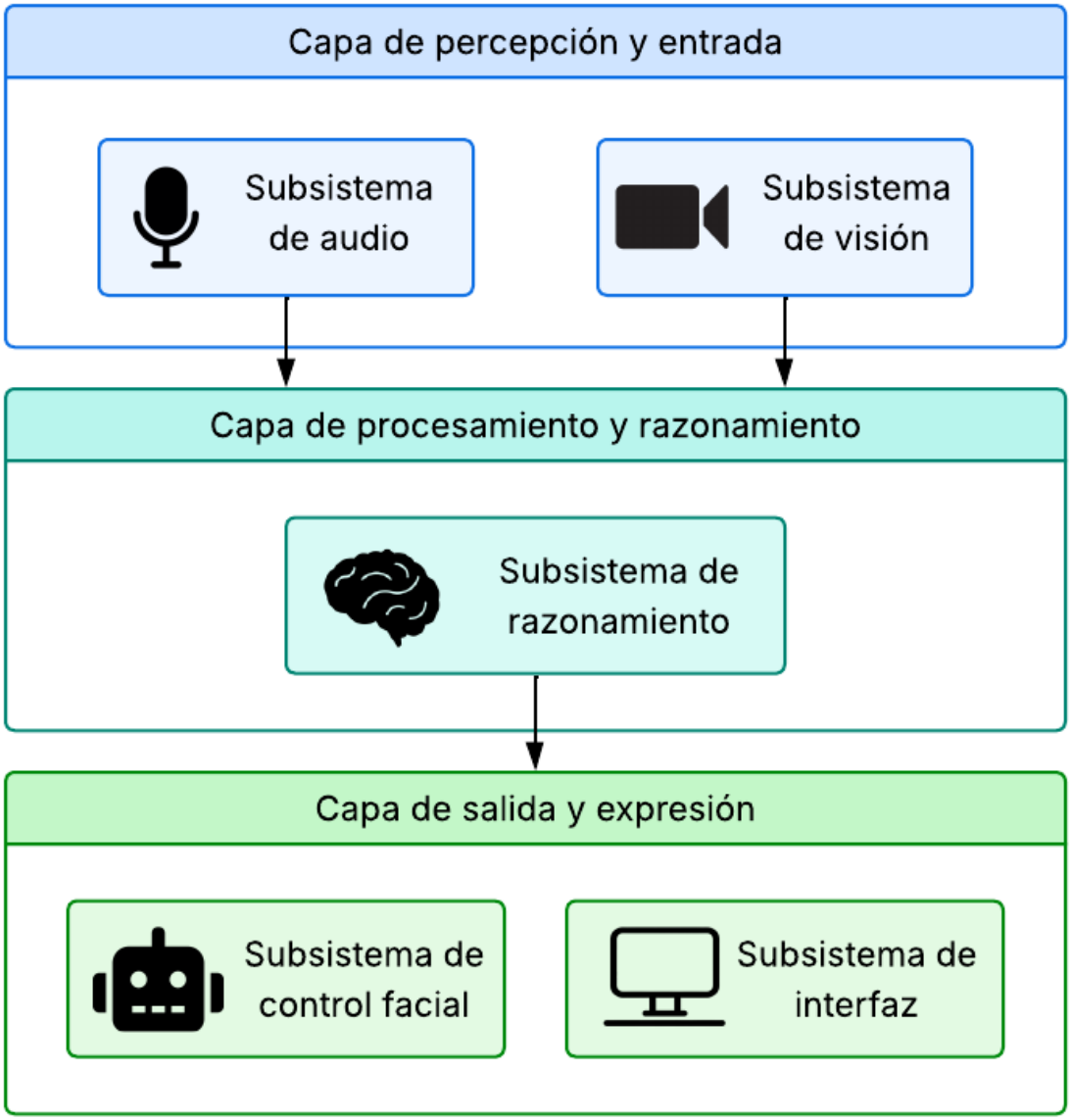


Figura 6: Diagrama lógico de la arquitectura del sistema por capas funcionales. [Fuente: propia.]

División en subsistemas

Cada capa funcional del sistema se implementa a través de subsistemas especializados que operan de forma coordinada sobre la infraestructura ROS 2. A continuación se describen los cinco principales subsistemas, junto con una figura esquemática de su estructura interna:

- **Subsistema de entrada por audio**

El subsistema de entrada por audio permite al robot captar, procesar y responder al habla del usuario de forma eficiente y contextual. Está compuesto por varios módulos interconectados que cooperan a través de ROS 2 para transformar la señal de voz en texto, generar respuestas y gestionar la interacción mediante una máquina de estados.

El flujo general se inicia con un nodo de micrófono que publica audio sin procesar. Esta señal pasa por un módulo de detección de actividad vocal (VAD), que permite ignorar segmentos silenciosos o irrelevantes. En paralelo, un detector de *hotword* activa el sistema solo cuando se pronuncia una palabra clave. Una vez activado, se graba y transcribe el audio mediante un modelo STT (como Whisper), y la transcripción resultante se analiza semánticamente con un modelo LLM para generar una respuesta. Esta respuesta se sintetiza por TTS y se acompaña de animación visual en la boca del robot.

Todo este flujo se resume en la Figura 7, que recoge los pasos y decisiones clave del procesamiento de entrada por audio.

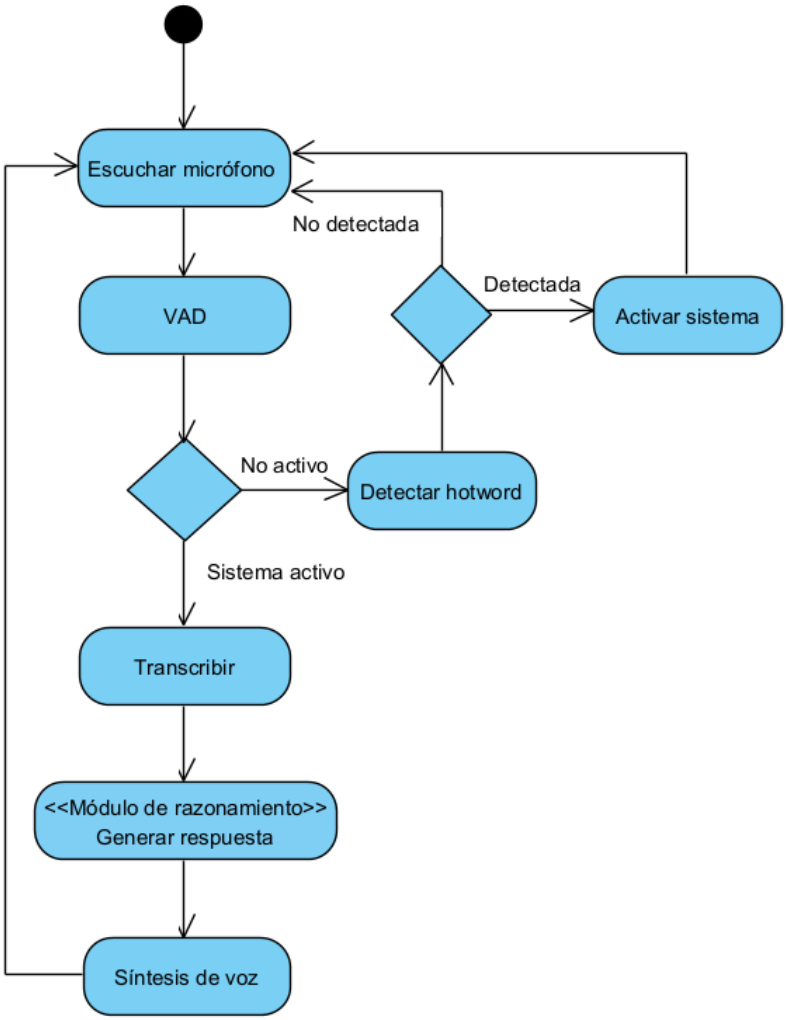


Figura 7: Diagrama de actividad del subsistema de entrada por audio. [Fuente: propia.]

Gestión de modos mediante máquina de estados

Para evitar interferencias y adaptar el comportamiento del sistema según el contexto, se ha implementado una máquina de estados finita (FSM) en el nodo de gestión de audio. Esta FSM regula qué módulos están activos y cómo se interpreta el audio recibido en función del estado actual. Los estados definidos son:

- ***Name*:** Estado inicial. El sistema está inactivo a la espera de una palabra clave.
- ***Command*:** Activado tras detectar la *hotword*. Todo lo que se diga en este estado será transcrito y procesado como una petición o comando.
- ***Speaking*:** Mientras se emite una respuesta sintetizada, el sistema desactiva la es-

cucha para evitar retroalimentación.

- ***Asking*:** Activado cuando el subsistema de visión detecta un rostro desconocido y espera una respuesta verbal del usuario (por ejemplo, “me llamo Juan”, “sí soy yo”, “no soy esa persona”) o entrada por teclado.

La FSM permite cambiar de estado de forma explícita, garantizando una interacción natural y sin ambigüedades. Su lógica se muestra en la Figura 8, donde se esquematizan las transiciones entre modos en función del contexto y los eventos del sistema.

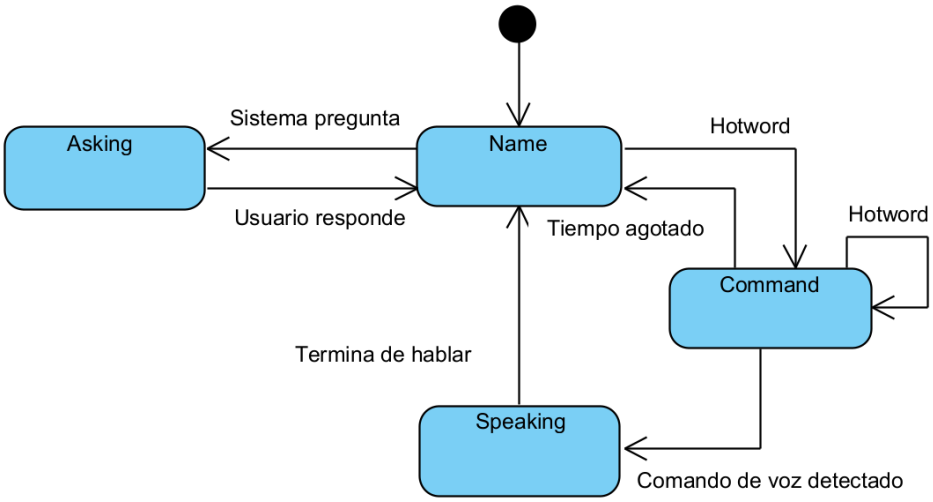


Figura 8: Máquina de estados del gestor de audio. [Fuente: propia.]

- **Subsistema de visión:**

  Procesa imágenes capturadas por la cámara del robot para detectar rostros humanos en tiempo real. Por cada rostro que se encuentre se extrae un vector de características mediante un modelo de codificación facial y se compara contra una base de datos local para identificar a la persona. Este subsistema permite personalizar la interacción en función del interlocutor, asociar sesiones previas y desencadenar flujos de registro cuando se detectan usuarios nuevos.

  El funcionamiento completo de este subsistema se resume en la Figura 9, donde se representa el flujo de procesamiento desde la captación de imagen hasta la identificación.

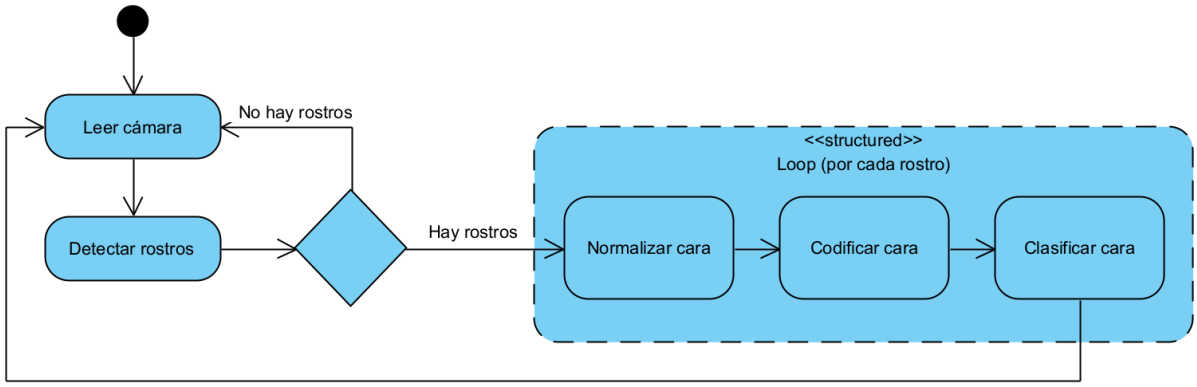


Figura 9: Diagrama de actividad del subsistema de visión. [Fuente: propia.]

- **Subsistema de razonamiento:**

  Este subsistema constituye el núcleo cognitivo del sistema, responsable de interpretar las intenciones del usuario y generar respuestas coherentes. Recibe como entrada el texto transcrito procedente del subsistema de audio y lo procesa mediante modelos de lenguaje (LLMs) configurables en tiempo de ejecución.

  El proceso se estructura en dos fases diferenciadas mediante el uso de *prompts* encadenados:

  1. En primer lugar, se aplica un *prompt* de clasificación que determina la intención del usuario a partir del mensaje recibido.
  2. A continuación, según el resultado de la clasificación, se genera un segundo *prompt* específico:
     - Si la intención está predefinida, se ejecuta la acción correspondiente y el *prompt* transforma su resultado en una respuesta natural orientada al usuario.
     - Si la intención es desconocida, el sistema construye un *prompt* general que incorpora el contexto conocido por el robot, buscando mantener una conversación fluida y abierta.

  Este subsistema está diseñado para ser altamente flexible: permite intercambiar modelos LLM *on-line*, se adapta dinámicamente a distintos contextos conversacionales y sopor-

ta tanto entradas de voz como interacciones escritas desde la interfaz web. La lógica completa se muestra en la Figura 10.

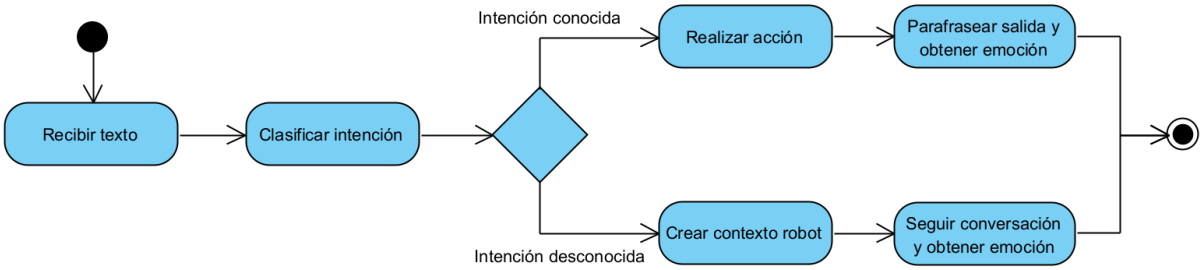


Figura 10: Diagrama de actividad del subsistema de razonamiento e interpretación. [Fuente: propia.]

- **Subsistema de salida expresiva:**

  Su función principal es transformar la respuesta textual generada por el subsistema de razonamiento en una salida física multimodal, coordinando voz y expresión visual. Está compuesto por un módulo de síntesis de voz (TTS) que convierte el texto en audio, y por un conjunto de actuadores visuales: una tira LED animada que representa la boca del robot y un sistema de iluminación RGB que controla la expresión de los ojos.

  Además de reflejar los distintos estados funcionales del robot (hablando, escuchando, pensando), este subsistema incorpora un componente emocional. Al generar una respuesta, se analiza también la emoción subyacente en el contenido del texto, y dicha emoción se expresa visualmente mediante combinaciones de color y patrones de movimiento en ojos y boca. Por ejemplo, emociones como alegría, sorpresa o neutralidad se traducen en colores y animaciones específicas que enriquecen la expresividad del robot y mejoran la percepción empática de la interacción.

  El subsistema asegura una sincronización precisa entre la salida de audio y los elementos visuales, garantizando una respuesta coherente, fluida y emocionalmente significativa. Todo el flujo de funcionamiento puede observarse en la Figura 11.

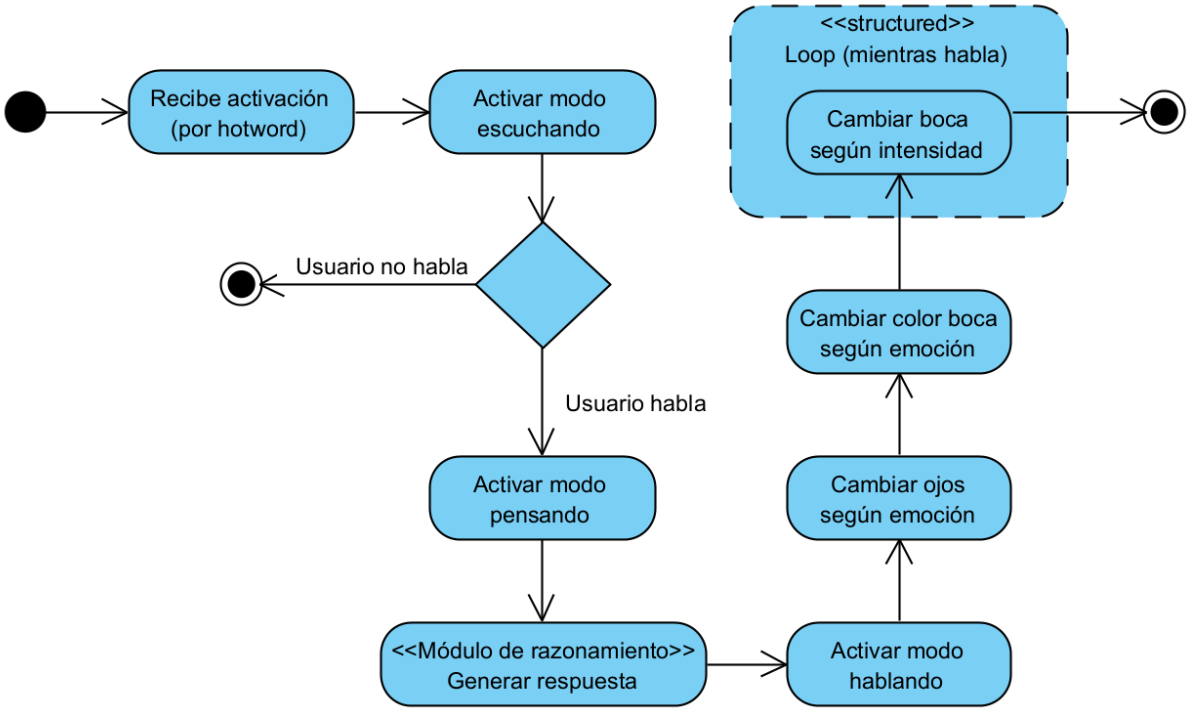


Figura 11: Diagrama de actividad del subsistema de salida expresiva. [Fuente: propia.]

- **Subsistema de interfaz web y control remoto:**

  Este subsistema proporciona una interfaz gráfica avanzada accesible desde cualquier navegador, permitiendo al usuario interactuar con el sistema en tiempo real y de forma remota. Se basa en dos canales principales de comunicación:

  - Una **API REST**, utilizada para consultar o modificar datos estructurados, como usuarios, sesiones o modelos activos.
  - Un **canal WebSocket**, gestionado por 'ros2web', que permite la recepción de eventos generados por el sistema en tiempo real, así como la transmisión de mensajes tipo chat desde el usuario hacia el robot.

  Aunque este subsistema no sigue un flujo de procesamiento secuencial como otros módulos (audio o visión), su funcionamiento general puede representarse mediante un diagrama de actividad que recoge las acciones más representativas del usuario (consultas, modificaciones, mensajes) y las reacciones del sistema (procesamiento backend, envío de eventos, respuesta del robot), como se muestra en la Figura 12.

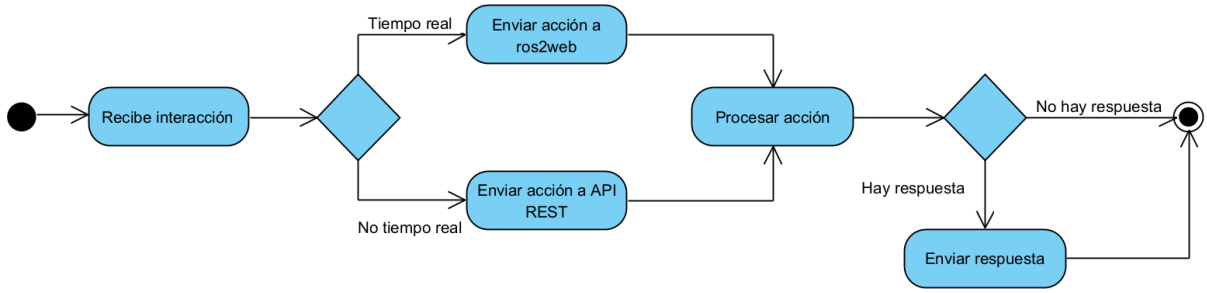


Figura 12: Diagrama de actividad del subsistema de interfaz web y control remoto. [Fuente: propia.]

3.3.2. Diagrama de despliegue general del sistema

Además de la visión funcional modular, se ha incluido un diagrama de despliegue que ofrece una representación estructural de alto nivel del sistema. Esta vista permite entender cómo se distribuyen físicamente y lógicamente los principales componentes —*hardware*, nodos ROS 2, servicios web y módulos en la nube— así como los canales de comunicación que los interconectan.

El diagrama, mostrado en la Figura 13, refleja la disposición real del sistema en tres entornos diferenciados: el robot (con sensores y actuadores físicos), el servidor ROS 2 que orquesta la lógica del comportamiento, y los servicios externos desplegados en la nube (CSAR), donde residen los modelos de lenguaje, herramientas de síntesis/reconocimiento de voz y los módulos de visión artificial. También se muestran las interfaces de usuario —web y escritorio— que interactúan con el sistema a través de la API REST y ros2web.

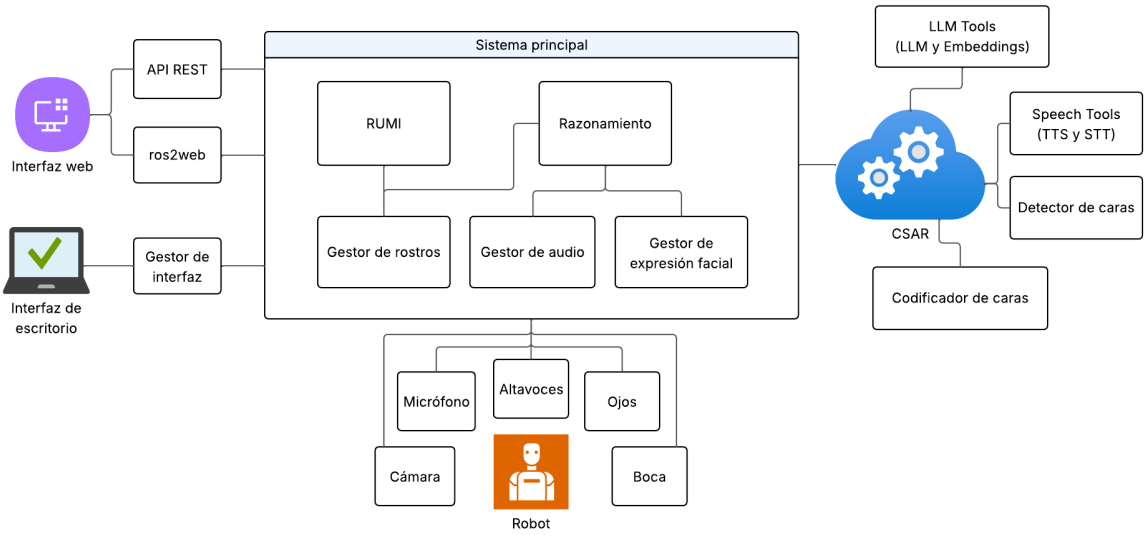


Figura 13: Diagrama de despliegue del sistema. Interacción entre el robot, los módulos ROS 2, la infraestructura en la nube (CSAR) y las interfaces de usuario. [Fuente: propia.]

Entre los elementos representados se incluyen herramientas desarrolladas específicamente en este proyecto, como **ros2web**, **speech_tools**, **llm_tools** y **rumi**. Estos módulos han sido concebidos como componentes reutilizables, con interfaces bien definidas y una arquitectura desacoplada que facilita su integración en otros sistemas. Su diseño independiente y potencial de escalabilidad se aborda en detalle en la subsección 3.4.

3.3.3. Integración multimodal en tiempo real

El sistema combina procesamiento visual y auditivo en tiempo real mediante un conjunto de nodos ROS 2 especializados que cooperan de forma asincrónica. A continuación se describen por separado las dos secuencias principales de interacción: la visual (centrada en el reconocimiento de personas) y la auditiva (centrada en la conversación natural).

Secuencia de percepción visual y reconocimiento

La interacción visual comienza cuando el sistema detecta presencia humana en la escena. La cámara RGB captura imágenes en tiempo real, que son enviadas a un nodo de detección facial. Este módulo localiza los rostros en la imagen y, para cada uno de ellos, ejecuta el siguiente flujo:

- Se extrae un *embedding* facial con un encoder entrenado (por ejemplo, FaceNet).

- El *embedding* se compara con los almacenados en la base de datos para clasificar la identidad del usuario.
- Si se reconoce con suficiente confianza, se registra la detección y se continúa la interacción.
- Si no se reconoce o la confianza es baja, el sistema solicita al usuario su nombre mediante una interfaz de entrada (por voz o desde pantalla).
- Tras recibir el nombre, el sistema registra la nueva identidad en la base de datos, asociando el *embedding* al nombre proporcionado.

Este flujo garantiza un aprendizaje incremental de identidades, permitiendo que el sistema amplíe su base de conocimiento de forma autónoma y adaptativa. La figura 14 resume esta secuencia mediante un diagrama UML detallado.

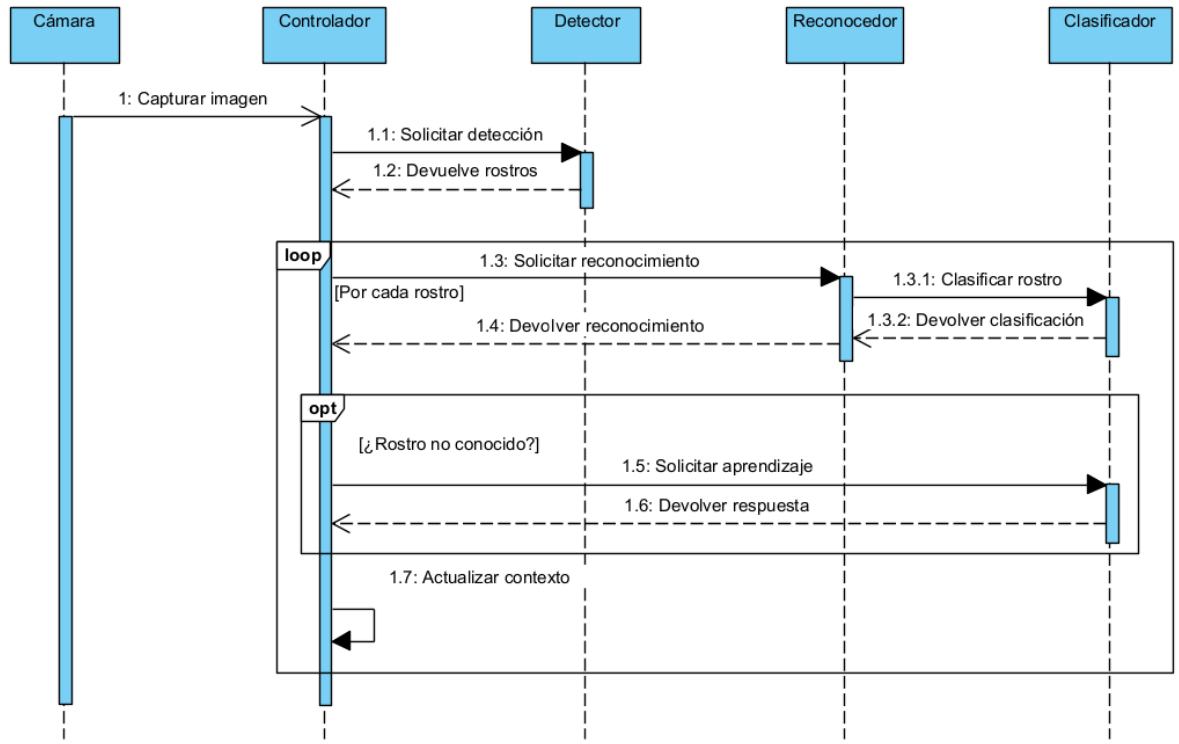


Figura 14: Diagrama de secuencia de la interacción visual: detección, codificación, clasificación y registro de nuevos rostros. [Fuente: propia.]

Secuencia de interacción auditiva y respuesta

En paralelo al procesamiento visual, el sistema está continuamente atento a la entrada de audio con el fin de responder a interacciones habladas. La secuencia auditiva se activa cuando se recibe una señal de voz y se ejecuta el siguiente flujo:

- El micrófono capta el audio del entorno, que es recibido por el controlador auditivo.
- El controlador solicita una transcripción del contenido al módulo STT (Speech-to-Text), encargado de convertir el audio en texto.
- Una vez obtenida la transcripción, se envía al módulo de lenguaje (LLM) para generar una respuesta en texto, interpretando la intención del usuario.
- La respuesta generada se transmite al módulo TTS (Text-to-Speech), que la convierte en una señal de audio sintetizada.
- Finalmente, el controlador envía la síntesis al sistema de altavoces para su reproducción física. En paralelo, se puede activar la animación sincronizada de la boca del robot.

Este flujo representa una interacción completa desde la percepción auditiva hasta la respuesta vocal, permitiendo una conversación fluida entre el usuario y el robot. La arquitectura distribuida de ROS 2 permite que todos los módulos implicados trabajen de forma concurrente y coordinada. Todo el proceso se detalla en la Figura 15.

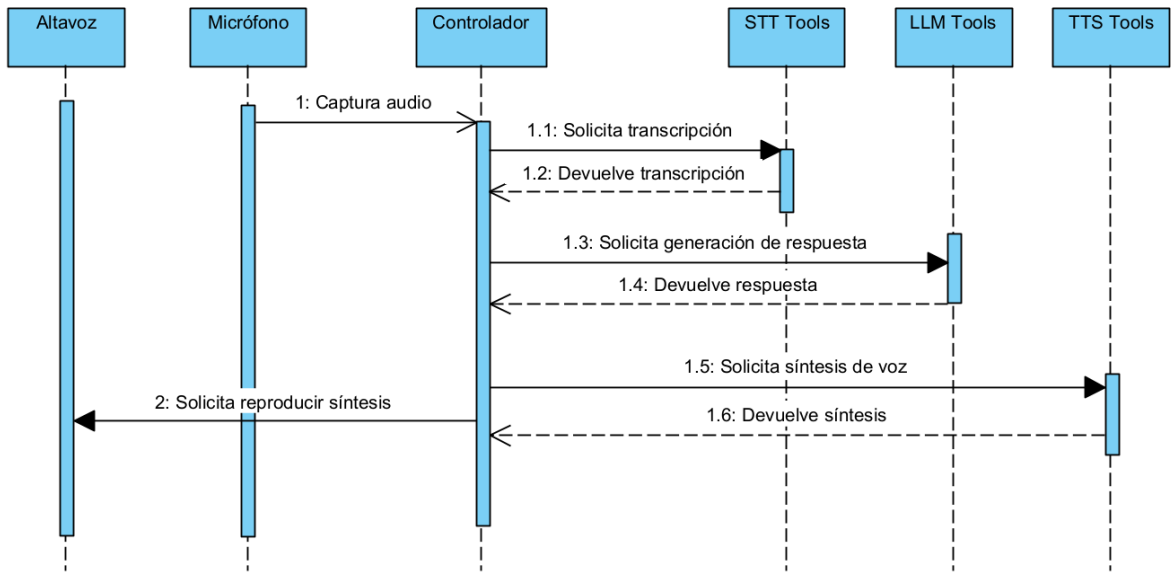


Figura 15: Diagrama de secuencia de la interacción auditiva: transcripción, generación de respuesta y síntesis de voz. [Fuente: propia.]

3.3.4. Modelo de datos unificado

El sistema gestiona tres bases de datos diferenciadas, cada una especializada en un tipo concreto de información y separadas por motivos funcionales y de diseño. Esta división permite optimizar el rendimiento, la mantenibilidad y la escalabilidad de cada subsistema.

- La base de datos de identidades faciales almacena, para cada usuario, un identificador único, su nombre y una serie de vectores de características (*embeddings*) que codifican su rostro. Como se detallará en el capítulo de implementación (Capítulo 4), esta base de datos se ha diseñado para ofrecer consultas extremadamente rápidas, por lo que no se utiliza una base relacional tradicional, sino una estructura más eficiente adaptada a búsquedas por similitud.

- La base de datos de sesiones recoge las interacciones entre los usuarios y el sistema: fecha, transcripción, intención detectada, respuesta generada y métricas asociadas. Esta base de datos está gestionada directamente por la herramienta **RUMI**, desarrollada en el marco de este proyecto, lo que justifica su separación como componente autónomo y especializado.

- La base de datos de *logs* del sistema almacena eventos técnicos como errores, tiempos de respuesta, modelos utilizados, cambios de estado, entre otros. Está orientada a la monitorización y mantenimiento del sistema, y se mantiene separada de los datos de usuario por motivos de encapsulamiento funcional.

Aunque estas bases de datos están físicamente separadas en la implementación, desde el punto de vista lógico forman parte de un mismo modelo de datos. La Figura 16 representa de forma unificada todas las entidades gestionadas por el sistema —usuarios faciales, sesiones, detecciones y eventos técnicos— incluyendo sus atributos clave y las relaciones conceptuales entre ellas. Por ejemplo, cada sesión se vincula a un identificador facial, lo que permite asociar las conversaciones a los rostros reconocidos.

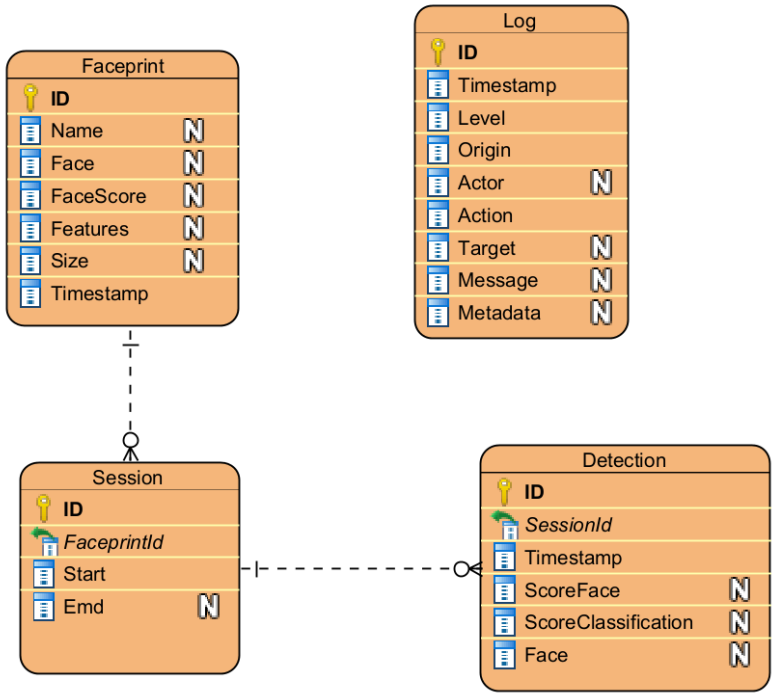


Figura 16: Modelo entidad-relación unificado del sistema. [Fuente: propia.]

Además, la separación de estas bases de datos responde a una decisión de diseño fundamentada en principios de **responsabilidad única**, **aislamiento funcional** y **resiliencia del sistema**. Al mantener dominios de datos desacoplados, se facilita la modularidad, se reduce el riesgo de errores cruzados y se garantiza que un fallo en uno de los servicios (por ejemplo, la base de logs o la de sesiones) no afecte al funcionamiento del resto del sistema, como el

reconocimiento facial en tiempo real. Esta estructura también permite escalar o migrar cada conjunto de datos de forma independiente en función de las necesidades futuras.

3.3.5. Interfaz web

El sistema incorpora una aplicación web con el objetivo de permitir la supervisión, visualización y gestión remota del robot. Esta aplicación está concebida como una herramienta de control accesible desde cualquier dispositivo conectado a la red, orientada tanto a operadores como a usuarios técnicos. Desde el punto de vista del diseño, se ha priorizado una estructura modular y coherente que permita una navegación fluida entre vistas, un acceso directo a la información más relevante y una experiencia adaptativa en todo tipo de dispositivos.

La estructura funcional de la aplicación se ha modelado utilizando el lenguaje IFML (*Interaction Flow Modeling Language* (Brambilla & Fraternali, 2014)). Este lenguaje permite representar gráficamente los flujos de interacción entre vistas, así como los eventos de usuario y las transiciones que desencadenan. La Figura 17 muestra el modelo general de navegación de la aplicación web, donde se aprecia el flujo central de conversación, la galería de identidades, el panel de modelos, la sección de sesiones de interacción y el visor de logs. Este modelo sirve como guía conceptual para el diseño de la interfaz final y garantiza una estructura lógica y extensible.

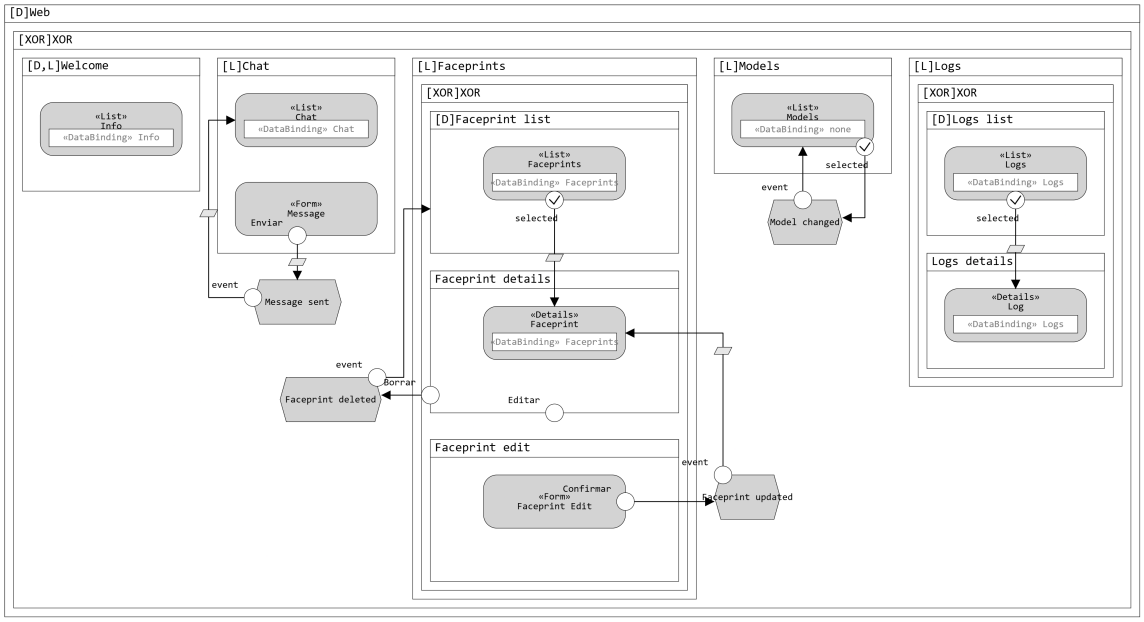


Figura 17: Modelo IFML de la aplicación web. [Fuente: propia.]

Mockups

La Figura 18 muestra la pantalla de bienvenida de la aplicación, que actúa como punto de entrada al sistema. Esta vista ofrece un resumen inicial del estado del robot, botones de acceso directo a las funcionalidades principales y un diseño simplificado que facilita la orientación inicial del usuario.

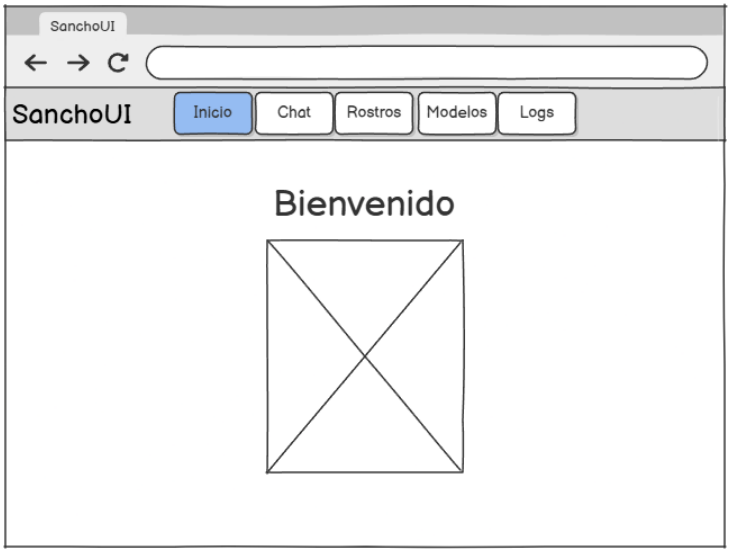


Figura 18: Mockup de la pantalla de bienvenida. [Fuente: propia.]

La Figura 19 presenta la vista de conversación, diseñada para mantener una interacción fluida con el robot. Esta vista incluye los mensajes enviados y recibidos, botones para grabar o adjuntar audios, y un acceso rápido a los ajustes de configuración. Se trata del centro de la experiencia conversacional y admite entrada por texto o por voz.

Figura 19: Mockup de la vista de conversación con el robot. [Fuente: propia.]

La Figura 20 representa la galería de identidades faciales reconocidas por el sistema. Cada rostro se muestra como una tarjeta que incluye nombre, ID y fecha de registro. Desde esta vista es posible añadir, editar o eliminar identidades de forma remota, y acceder a los detalles asociados a cada persona.

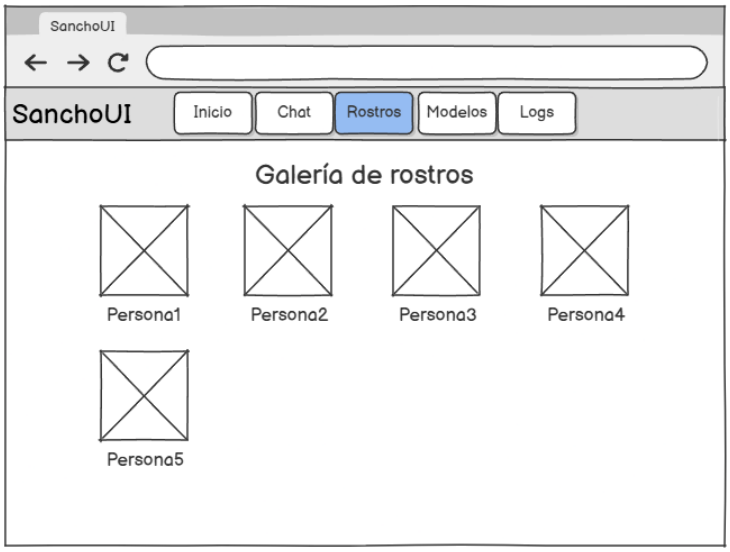


Figura 20: Mockup de la galería de identidades faciales. [Fuente: propia.]

En la Figura 21 se observa la vista de detalle de una identidad facial, que muestra información extendida como metadatos del rostro, sesiones de interacción asociadas y una galería de imágenes capturadas. Desde aquí se pueden realizar acciones como editar el nombre o eliminar la identidad del sistema.

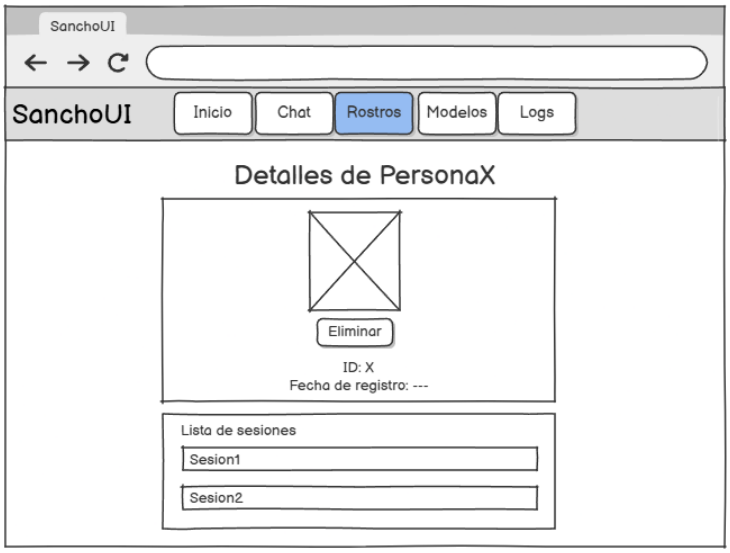


Figura 21: Mockup de detalles de una identidad facial. [Fuente: propia.]

La Figura 22 representa la sección de sesiones de interacción, donde se muestra el historial de encuentros con cada usuario. Cada sesión incluye gráficas con la evolución temporal de la puntuación de detección y clasificación, junto con estadísticas globales como duración,

número de detecciones o nivel de fiabilidad media.

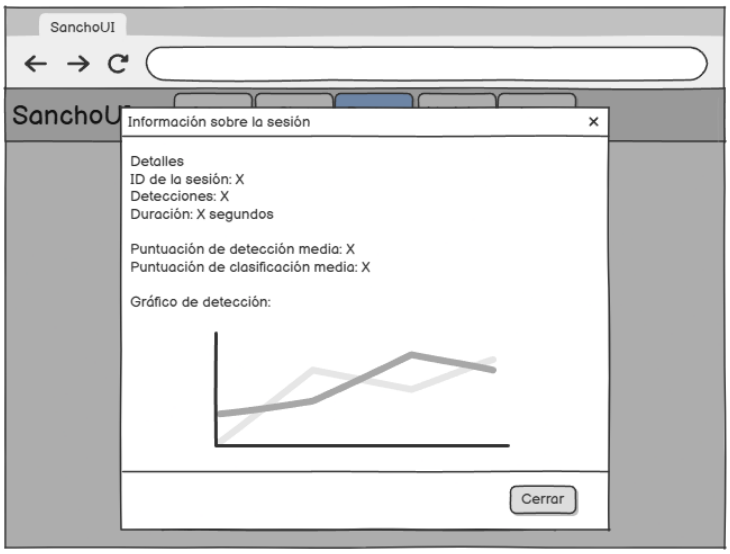


Figura 22: Mockup del historial de sesiones de interacción. [Fuente: propia.]

La Figura 23 muestra el panel centralizado para la gestión de modelos de IA. Desde aquí es posible cargar y activar modelos de lenguaje (LLM), transcripción (STT), síntesis (TTS) y embeddings. También se pueden configurar claves API y seleccionar voces específicas en los modelos de TTS compatibles.

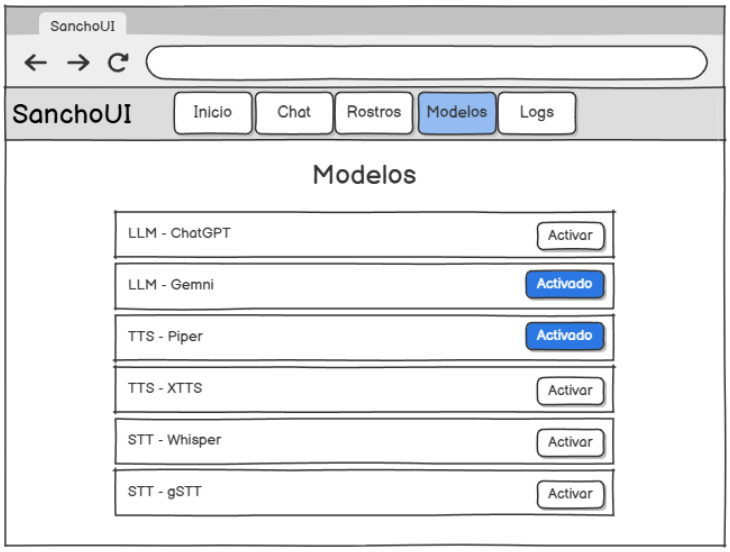


Figura 23: Mockup del panel de gestión de modelos. [Fuente: propia.]

La Figura 24 presenta la vista de *logs* del sistema, donde se registran eventos internos organizados por tipo, severidad, componente y timestamp. Esta vista es útil para depuración,

trazabilidad y supervisión técnica durante el funcionamiento del robot.

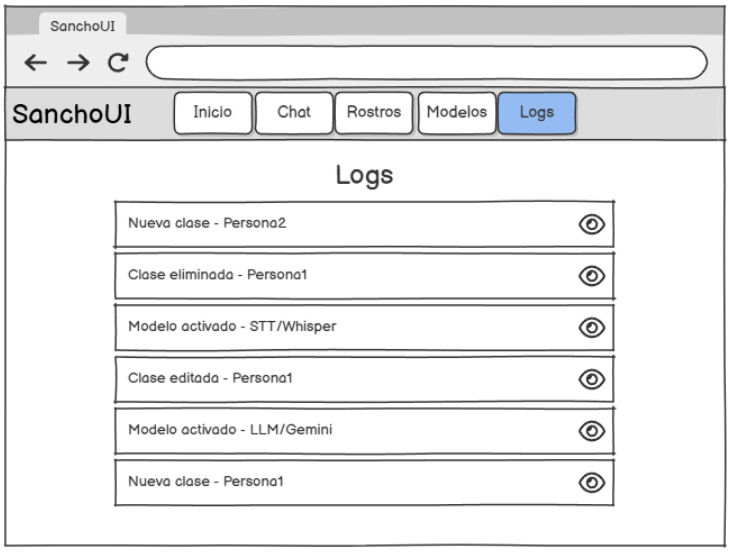


Figura 24: Mockup de la vista de logs del sistema. [Fuente: propia.]

Por último, la Figura 25 muestra el modal de detalles que se despliega al seleccionar un evento concreto. Esta vista permite consultar información ampliada como el mensaje completo, el nodo ROS responsable, datos complementarios y la hora exacta de emisión.

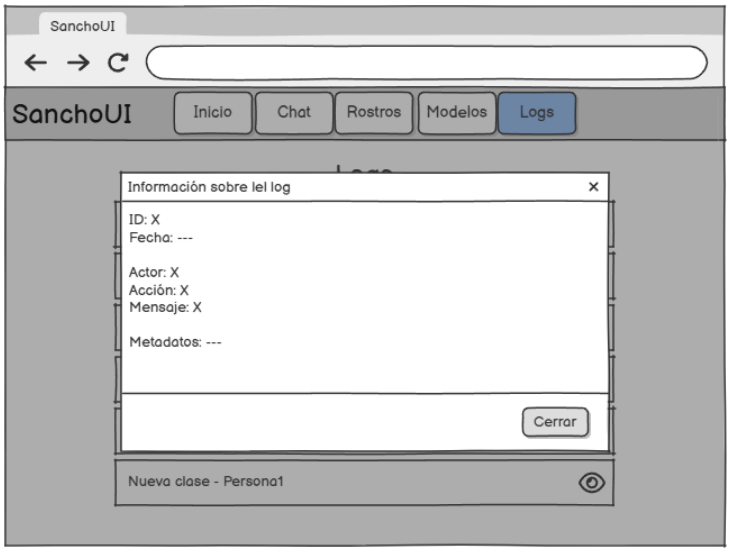


Figura 25: Mockup del modal de detalles de un log del sistema. [Fuente: propia.]

3.3.6. Interfaz de escritorio

El sistema cuenta con una interfaz gráfica sencilla, pensada para facilitar la interacción con el usuario. Lo interesante de esta interfaz es que en el caso de ejecutar el sistema en un robot físico como **Sancho**, es que el robot la presente al usuario para mejorar la interacción humano robot. En caso de que se ejecute sin un robot físico, simplemente tiene la misma funcionalidad pero como interfaz de escritorio.

Estas pantallas se activan automáticamente cuando se detecta un rostro no reconocido, permitiendo asociar un nombre o verificar identidades de forma directa. Cumple un papel fundamental durante el registro facial o en situaciones donde se requiere una intervención inmediata.

Actualmente, se han diseñado cuatro pantallas principales, cada una adaptada a un estado concreto del sistema:

Mockups

La **pantalla de bienvenida** se muestra cuando el sistema está en reposo, en modo *idle*. Presenta un mensaje simple como *"¡Hola! Soy Sancho"*, ofreciendo un punto de partida visual claro para el usuario. Esta vista se muestra en la Figura 26.

Figura 26: Mockup de la pantalla de bienvenida. [Fuente: propia.]

La **pantalla de registro de nombre** aparece cuando se detecta un rostro desconocido. Muestra la imagen capturada junto con un campo de texto para introducir el nombre del usuario, asociando así una nueva identidad a la base de datos. Se ilustra en la Figura 27.

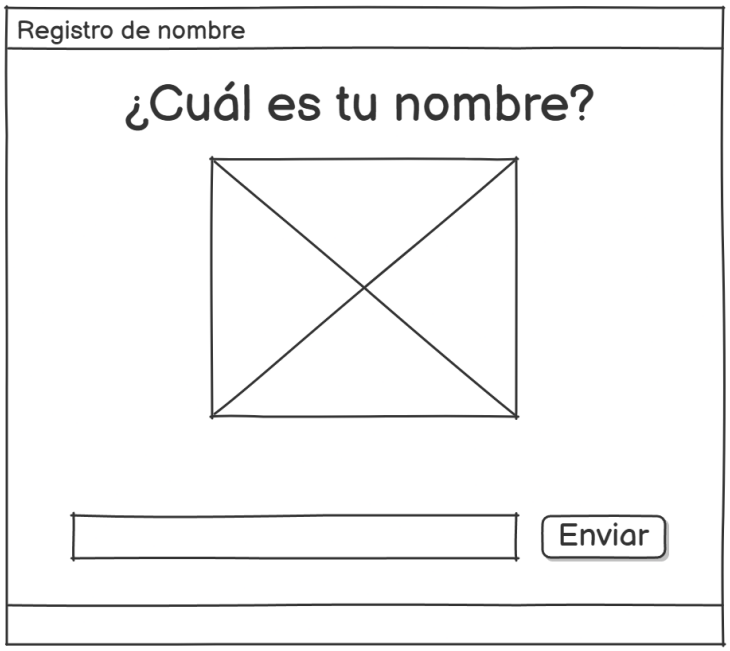


Figura 27: Mockup de la pantalla de registro de nombre. [Fuente: propia.]

La **pantalla de confirmación de nombre** se utiliza para verificar si un rostro reconocido ha sido correctamente identificado. Presenta la imagen detectada y dos botones (*"Sí"* y *"No"*) que permiten al usuario confirmar o rechazar la identificación propuesta. Puede verse en la Figura 28.

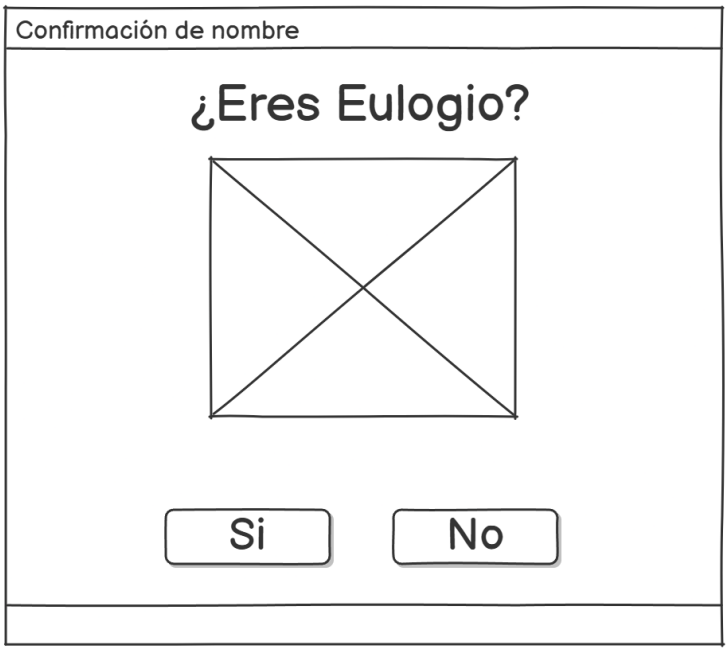


Figura 28: Mockup de la pantalla de confirmación de identidad. [Fuente: propia.]

Por último, la **pantalla de visualización temporal** permite mostrar imágenes durante unos segundos. Está pensada para que el robot pueda enseñar contenidos visuales como parte de una explicación o interacción con el usuario. Se muestra en la Figura 29.

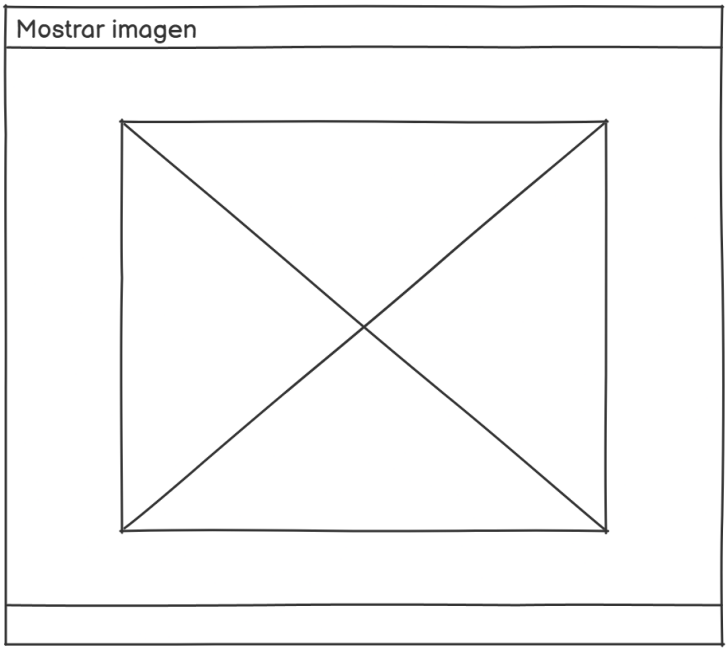


Figura 29: Mockup de la pantalla de visualización temporal. [Fuente: propia.]

3.4. Escalabilidad y arquitectura extensible

El sistema ha sido concebido como una plataforma extensible, modular y desacoplada, capaz de adaptarse a distintos entornos y requerimientos. Lejos de constituir una solución cerrada y específica, su diseño se apoya en componentes independientes y genéricos, fácilmente reutilizables en otros proyectos de interacción humano-robot, asistentes embebidos o aplicaciones de inteligencia artificial conversacional.

Diseño modular y orientado a componentes

La mayoría de las piezas del sistema se han diseñado de forma que sean totalmente reutilizables, con el acoplamiento mínimo, lo que permite que estos módulos se puedan emplear en cualquier tipo de proyecto en el que sea necesario, sin necesidad de tener que pasar por un proceso de desacoplamiento ni rediseño de código. Los módulos más útiles que se generan a raíz de este trabajo son los siguientes:

- **Módulo de comunicación web (ros2web):** actúa como puente entre ROS 2 y la interfaz web. Está basado en eventos estructurados vía WebSocket y ofrece un canal en tiempo real para emitir detecciones, resultados o configuraciones desde el robot hacia el navegador. Este diseño evita dependencias innecesarias y permite desplegar visualizaciones o paneles externos sin necesidad de modificar los nodos ROS.

- **Módulo de síntesis y transcripción de voz (speech_tools):** encapsula las funcionalidades de Text-to-Speech (TTS) y Speech-to-Text (STT), permitiendo seleccionar modelos locales o cloud en tiempo de ejecución. Soporta múltiples proveedores y configura el uso de *hotword* detection y VAD, gestionando las sesiones de audio de manera autónoma.

- **Módulo de grandes modelos de lenguaje (llm_tools):** permite gestionar múltiples modelos LLM de forma dinámica, incluyendo su carga, activación y liberación. Facilita la ejecución de prompts personalizados con historial, parámetros y distintos sistemas de interacción, y abstrae tanto modelos generativos como de embeddings, ofreciendo una interfaz unificada reutilizable desde otros módulos (como la web o los comandos de voz).

- **Herramienta de gestión de usuarios (RUMI):** proporciona un panel web para administrar identidades detectadas por reconocimiento facial. Permite añadir, renombrar, eliminar y revisar sesiones de cada usuario, todo ello desde una interfaz intuitiva conectada a ROS 2. Esta herramienta ha sido formalizada y publicada como contribución científica independiente (Quemada-Torres et al., 2025).

Estos módulos han sido diseñados como bloques funcionales reutilizables, independientes del dominio particular del sistema y preparados para ser integrados en otras soluciones robóticas o sistemas de IA distribuidos.

Extensibilidad en múltiples niveles

La arquitectura contempla mecanismos explícitos para facilitar su crecimiento futuro:

- **Carga dinámica de modelos:** los módulos de voz, lenguaje y *embeddings* permiten cargar y activar nuevos modelos sin reiniciar el sistema ni recompilar código. La arquitectura interna separa claramente la lógica de proveedor de la lógica de uso.
- **Interfaz desacoplada y reactiva:** la interfaz web no depende del ciclo de vida interno de ROS 2 y puede actualizarse en caliente mediante mensajes emitidos por ros2web. Esto permite modificar, probar o extender componentes sin afectar al resto del sistema.
- **Prompts reconfigurables:** los sistemas de razonamiento permiten redefinir prompts, ajustar parámetros o introducir tareas nuevas sin modificar el código del modelo. Esto facilita la adaptación a nuevos dominios de aplicación o modos de interacción.
- **Encapsulamiento funcional:** cada módulo se comunica mediante servicios y mensajes ROS 2 bien definidos, lo que permite sustituir componentes (como STT local por uno en la nube) sin que el sistema global se vea afectado.
- **Gestión remota:** las operaciones del sistema —activación de modelos, edición de identidades, exploración de sesiones— pueden realizarse desde la web, eliminando la necesidad de intervención en terminales o cambios en tiempo de desarrollo.

Reutilización en otros entornos

Cómo hemos comentado, el diseño orientado a componentes permite que los módulos del sistema puedan integrarse como bloques independientes en otros proyectos de HRI, interfaces multimodales o sistemas embebidos:

- La interfaz ros2web permite construir dashboards visuales para robots móviles, simuladores o servidores ROS en la nube.
- El módulo de voz puede usarse como *backend* de síntesis y transcripción en dispositivos sin pantalla.
- El módulo de grandes modelos de lenguaje puede ser usado para realizar tareas de procesamiento de lenguaje con cualquier modelo.
- La herramienta RUMI es útil para proyectos con usuarios identificables por cámara, como robots en aulas o museos.

Este enfoque de diseño potencia la mantenibilidad, la extensibilidad a futuro, y la portabilidad a distintos dominios. Más allá del caso de uso desarrollado en este TFG, el sistema representa un conjunto de herramientas de propósito general aplicables a cualquier proyecto que busque interacción humano-robot natural, configurable y modular.

3.5. Computación en la nube

La arquitectura del sistema ha sido diseñada desde sus bases para operar en entornos distribuidos, combinando capacidades locales de percepción y expresión con servicios de inteligencia artificial desplegados en la nube. Esta separación permite que el robot actúe como interfaz sensorial y física, mientras que el razonamiento, la generación de lenguaje o la síntesis de voz expresiva se delegan a servidores externos, maximizando la eficiencia sin comprometer la experiencia interactiva.

Motivación y planteamiento arquitectónico

La ejecución local de modelos de gran tamaño —como LLMs tipo GPT-4 o Mixtral, o TTS neuronales expresivos como XTTS o Bark— requiere recursos computacionales no disponi-

bles en plataformas embebidas como Raspberry Pi o en ordenadores de a bordo típicos en robots móviles. Estos modelos demandan GPUs dedicadas, gran cantidad de memoria RAM y procesamiento paralelo intensivo.

Para abordar este reto sin renunciar a un sistema responsivo y natural, se ha adoptado un enfoque arquitectónico híbrido, donde los módulos de alta demanda se ejecutan en contenedores Linux desplegados en servidores remotos, dentro del entorno **CSAR (*Cloud System Architecture for Robotics*)** (Ambrosio-Cestero et al., 2024), mientras que el resto del sistema opera localmente sobre ROS 2.

CSAR como solución de ejecución distribuida

CSAR es una infraestructura ligera y flexible para desplegar sistemas robóticos en contenedores persistentes (LXC), manteniendo compatibilidad nativa con ROS 2. Cada contenedor actúa como nodo computacional autónomo, con su propio entorno de ejecución, y puede comunicarse directamente con el robot a través del *middleware* DDS.

Entre sus ventajas técnicas destacan:

- Acceso transparente a *hardware* especializado (GPUs, redes de alta velocidad).
- Aislamiento completo de entornos, versiones y dependencias.
- Descubrimiento automático de nodos ROS 2 mediante multicast.
- Monitorización, reinicio y actualización independientes de cada contenedor.

Cada contenedor puede incluir imágenes personalizadas con ROS 2, librerías de IA, servidores de modelo, intérpretes Python o herramientas de depuración, permitiendo adaptar el entorno a las necesidades de cada módulo funcional.

Distribución lógica del sistema

El sistema ha sido diseñado para funcionar de forma híbrida, distribuyendo sus componentes entre una unidad local (robot físico o estación de control) y un entorno de computación en la nube basado en contenedores (CSAR). Esta división se basa en una separación clara entre las tareas sensoriales, de control y visualización —que requieren baja latencia y acceso directo al

hardware— y las tareas de procesamiento intensivo —que exigen gran capacidad computacional y acceso a modelos avanzados—.

En la parte local se ejecutan los módulos encargados de:

- Captura de señales sensoriales (audio y vídeo).
- Control directo de *hardware* (tira LED, ojos del robot, botones físicos).
- Interfaces de usuario (web y Qt).
- Servicios web, API REST y el sistema de publicación de eventos en tiempo real (ros2web).
- Nodos ROS 2 ligeros y de coordinación local.

En la parte remota, desplegada sobre contenedores CSAR, se ejecutan los módulos más exigentes a nivel de cómputo:

- Modelos de lenguaje natural (LLMs) para razonamiento y generación de respuestas.
- Modelos de síntesis y transcripción de voz (TTS y STT).
- Modelos de embeddings semánticos y de texto.
- Modelos de detección y reconocimiento facial.

Esta separación permite mantener la reactividad del sistema, minimizar la carga local y aprovechar servidores de alto rendimiento sin alterar el flujo de interacción natural con el usuario.

Ventajas del enfoque híbrido distribuido

El uso de CSAR como plataforma de computación en la nube aporta beneficios estratégicos al diseño del sistema:

- **Rendimiento optimizado:** los modelos pesados no cargan el dispositivo embebido, que puede centrarse en tareas sensoriales y de control físico.
- **Escalabilidad horizontal:** se pueden añadir nuevos contenedores con nuevos modelos sin alterar la lógica local ni reiniciar nodos.

- **Mantenibilidad y pruebas aisladas:** cada módulo puede actualizarse, duplicarse o sustituirse sin riesgo para el sistema global.
- **Compatibilidad con múltiples robots:** varios dispositivos pueden conectarse al mismo conjunto de servidores CSAR, compartiendo recursos o manteniendo sesiones independientes.

Esta arquitectura no solo garantiza un rendimiento robusto en condiciones reales, sino que sienta las bases para extender el sistema a escenarios multiusuario, redes de robots o servicios cognitivos compartidos, cumpliendo los principios de flexibilidad, escalabilidad y reutilización definidos en el diseño global del proyecto.

3.6. Consideraciones técnicas y éticas

El sistema ha sido diseñado teniendo en cuenta aspectos críticos relacionados con la privacidad, la seguridad y el uso responsable de tecnologías basadas en inteligencia artificial. Estas consideraciones han influido directamente en la arquitectura, la forma en la que se almacenan los datos y el modo en que interactúa con los usuarios.

Privacidad y protección de datos personales

Dado que el sistema opera con datos sensibles como rostros, nombres e historiales de interacción, se han adoptado medidas específicas para garantizar la privacidad de los usuarios:

- La base de datos de rostros es local, editable y accesible únicamente desde la interfaz de gestión autorizada.
- No se almacenan grabaciones de audio ni imágenes, salvo decisión explícita del operador.
- Todo el sistema puede funcionar sin conexión a Internet, garantizando que ningún dato personal se transmite a terceros.

Seguridad en el acceso a servicios externos

Para utilizar servicios como OpenAI, Google o Gemini, el sistema requiere credenciales privadas. Se han implementado medidas para evitar riesgos asociados a su gestión:

- Las claves se introducen desde la web y se mantienen únicamente en la memoria del navegador del cliente.
- Nunca se almacenan en disco ni se embeben en el código.
- La transmisión se realiza mediante canales autenticados.

Uso controlado de modelos generativos

Los modelos LLM utilizados para generar respuestas están integrados en flujos estructurados que limitan su autonomía:

- Las respuestas generadas siguen formatos predefinidos y los sistemas de prompt incentivan a ser un robot amigable.
- El sistema evita deliberadamente generar contenidos ambiguos o atribuir al robot capacidades cognitivas no reales.

Sostenibilidad y transparencia técnica

Todo el sistema se basa en tecnologías abiertas, con módulos encapsulados y documentación detallada. Esta estrategia facilita su mantenimiento, auditoría y transferencia a otros proyectos:

- Se utilizan contenedores, entornos virtualizados y versiones fijas de bibliotecas críticas.
- La trazabilidad de configuraciones y dependencias permite replicar la instalación de forma controlada.

4

Implementación

Este capítulo describe en profundidad cómo se ha llevado a cabo la implementación del sistema multimodal propuesto, abordando tanto los aspectos técnicos como las decisiones arquitectónicas clave. Se detallan las tecnologías empleadas, la organización del código, los paquetes ROS 2 desarrollados, los componentes embebidos, la interfaz gráfica y los mecanismos de comunicación entre módulos.

El sistema ha sido diseñado con un enfoque modular, distribuido y escalable, integrando múltiples modalidades de entrada y salida (voz, visión, texto, expresión física) en una arquitectura basada en ROS 2 y tecnologías web modernas. A lo largo de las secciones siguientes se explica con precisión cómo se ha implementado cada subsistema, así como las herramientas utilizadas en el desarrollo, despliegue y documentación del proyecto.

La primera subsección presenta las tecnologías y herramientas empleadas, agrupadas según su propósito dentro del flujo de trabajo, y sirve como base para entender los componentes descritos en el resto del capítulo.

4.1. Tecnologías y herramientas utilizadas

El desarrollo del sistema ha requerido una integración coherente de tecnologías que cubren todo el ciclo de vida del software: diseño, implementación, prueba, despliegue y documentación. A continuación se detallan las herramientas y marcos más relevantes agrupados por su propósito.

4.1.1. Herramientas de desarrollo y gestión

- **Git y GitHub:** utilizados para el control de versiones y la colaboración. Git ha permitido trabajar con ramas funcionales, mantener trazabilidad precisa de los cambios y realizar integraciones seguras. GitHub ha servido como plataforma central para alojar

el código, coordinar tareas mediante issues y gestionar pull requests, además de incluir documentación técnica y automatización mediante GitHub Actions.

- **Visual Studio Code:** entorno de desarrollo integrado (IDE) utilizado para programar los diferentes componentes del sistema, tanto en Python como en C++ y JavaScript. Sus extensiones específicas para ROS 2, Python, C++ o JavaScript han permitido una experiencia de desarrollo fluida y adaptada a cada lenguaje, todo en un único entorno unificado.

- **Trello y Notion:** herramientas utilizadas para la organización del proyecto. Trello ha facilitado la planificación mediante tableros Kanban, mientras que Notion se ha empleado para estructurar ideas, documentación técnica y listas de tareas durante las distintas fases del desarrollo.

- **Visual Paradigm y Balsamiq:** Visual Paradigm ha permitido diseñar los diagramas UML y entidad-relación necesarios para documentar la arquitectura y el modelo de datos. Por su parte, Balsamiq se ha utilizado para prototipar rápidamente las interfaces de usuario, centrando el diseño en la usabilidad y funcionalidad sin distraer con detalles gráficos.

- **Markdown y LaTeX:** Markdown se ha empleado para la documentación técnica dentro del repositorio, mientras que LaTeX ha sido la herramienta principal para redactar esta memoria, permitiendo estructurar el contenido con precisión, incorporar bibliografía y mantener una presentación académica de alta calidad.

4.1.2. Lenguajes de programación

- **Python:** lenguaje principal del sistema. Se ha utilizado para la mayoría de los nodos de ROS 2, en especial los módulos de audio, lógica conversacional, *backend* web y gestión de embeddings. Su compatibilidad con ROS 2 mediante **rclpy**, junto con librerías como librosa, FastAPI, NumPy o Whisper, ha hecho posible un desarrollo ágil, expresivo y altamente funcional.

- **C++:** empleado en aquellos módulos donde se requería rendimiento y acceso a hardware de bajo nivel, como el nodo de animación de la boca del robot y el firmware del ESP32.

Dentro de ROS 2 se ha utilizado **rclcpp** para garantizar baja latencia y sincronización precisa con la señal de audio.

- **JavaScript:** utilizado en la construcción del frontend web mediante React. Ha permitido una experiencia de usuario dinámica, con componentes reutilizables, actualizaciones en tiempo real y gestión eficiente del estado de la aplicación.

4.1.3. Frameworks y tecnologías principales

- **ROS 2 (Humble):** ROS 2 es un *framework* para el desarrollo de software robótico modular y distribuido. Se basa en DDS (Data Distribution Service), lo que permite una comunicación eficiente, escalable y tolerante a fallos entre múltiples nodos que pueden ejecutarse en paralelo, incluso en distintas máquinas.

  La versión utilizada ha sido **Humble**, una LTS estable y ampliamente soportada. Gracias a su arquitectura, ha sido posible diseñar el sistema como un conjunto de nodos independientes comunicados mediante tópicos, servicios y acciones, favoreciendo el desacoplamiento, la reutilización y la evolución modular del sistema.

  ROS 2 permite configurar los componentes mediante archivos **launch.py** y **params .yaml**, facilitando su despliegue flexible. También ofrece soporte nativo para Python (**rclpy**) y C++ (**rclcpp**), así como herramientas de visualización, grabación y depuración que han sido clave durante el desarrollo.

  En conjunto, ROS 2 ha actuado como columna vertebral del sistema, proporcionando una base sólida para la integración de componentes complejos de percepción, razonamiento e interacción.

- **FastAPI:** *framework* moderno y de alto rendimiento para construir APIs REST en Python. Se ha utilizado en el backend del sistema para exponer datos como sesiones, usuarios, *embeddings* o configuraciones activas. Su documentación automática con OpenAPI ha facilitado el mantenimiento y pruebas.

- **React:** librería utilizada para el desarrollo del frontend. Gracias a su modelo basado en componentes y su sistema de hooks y contextos, ha permitido construir una interfaz

modular, reactiva y conectada tanto al backend REST como a los eventos en tiempo real vía WebSockets.

- **WebSockets:** se ha implementado un canal bidireccional entre ROS 2 y la interfaz web usando el *framework* **ros2web**. Este sistema permite recibir eventos en tiempo real (como detecciones faciales o transcripciones de voz) y enviar comandos de forma directa, todo con baja latencia y sin depender de rosbridge.

- **PyQt6:** utilizado para la creación de una interfaz local que permite introducir el nombre del usuario cuando se detecta un rostro desconocido. Su arquitectura basada en el patrón MVC facilita la organización del código y su mantenimiento.

- **Arduino IDE:** el microcontrolador gestiona la tira LED (boca) y anillos RGB (ojos), recibiendo comandos por puerto serie desde ROS 2. El firmware en C++ ejecuta animaciones en función del estado del sistema (hablando, escuchando, pensando...).

4.1.4. Bases de datos y almacenamiento

- **SQLite:** base de datos ligera empleada para almacenar logs, sesiones y estadísticas. Ha sido utilizada por el backend web y por la herramienta RUMI, permitiendo un registro persistente y accesible sin necesidad de servidores externos.

- **Base de datos facial optimizada:** se ha implementado un motor propio en Python para almacenar y consultar *embeddings* faciales en memoria. Esta solución permite búsquedas instantáneas, operaciones concurrentes seguras (mediante RLock) y persistencia asincrónica en JSON. Se ha optado por esta alternativa frente a bases de datos estándar por su mayor rendimiento en tiempo real.

4.2. Estructura general del proyecto

El sistema desarrollado se ha estructurado siguiendo una arquitectura modular, distribuida y escalable, basada en ROS 2 Humble como núcleo de comunicación. Cada funcionalidad se ha encapsulado en un paquete ROS 2 independiente, lo que ha permitido aislar responsabilidades, facilitar el desarrollo paralelo de distintos subsistemas y reducir el acoplamiento entre componentes.

A nivel interno, se sigue una organización clara que separa los paquetes por responsabilidad, permite el uso de componentes reutilizables y garantiza que cada módulo pueda desplegarse y configurarse de forma autónoma. Esta estructura modular ha sido clave para mantener la coherencia del sistema a medida que crecía en complejidad.

Los paquetes se agrupan en las siguientes categorías:

- **Mensajes personalizados:** definidos en paquetes como **ros2web_msgs**, **rumi_msgs**, **speech_msgs**, **llm_msgs** y **hri_msgs**, contienen las estructuras específicas necesarias para representar rostros, transcripciones, intenciones o modelos activos, y permiten una comunicación eficiente y adaptada entre nodos.

- **Infraestructura de comunicación:** el paquete **ros2web** actúa como pasarela entre ROS 2 y la interfaz web, permitiendo enviar y recibir eventos en tiempo real a través de WebSockets sin acoplar el backend al frontend.

- **Paquetes reutilizables:** Además de **ros2web**, los paquetes **RUMI**, **speech_tools** y **llm_tools** proporcionan funcionalidades genéricas y desacopladas, como el registro de sesiones, el control de voz, o el acceso a modelos LLM, TTS y STT. Están diseñados para integrarse en cualquier sistema ROS 2 sin dependencias externas innecesarias.

- **Paquetes funcionales de alto nivel: hri_audio**, **hri_vision**, **sancho_ai**, **sancho_web** y **face_controller** implementan las funcionalidades específicas del sistema, incluyendo la gestión de la interacción por voz, reconocimiento facial, razonamiento basado en intenciones y control embebido de expresividad física.

- **Control embebido:** se incluye un firmware en C++ para el ESP32 encargado de animar la boca y los ojos del robot, que se comunica mediante puerto serie con el nodo **face_controller**.

Cada paquete sigue una estructura coherente y orientada a la mantenibilidad, centrada en una organización clara del código y el uso explícito de argumentos o variables internas en lugar de ficheros de configuración externos. Los nodos están diseñados para funcionar de forma autónoma y modular, con sus propias funciones de inicialización, lógica de control y manejo de errores.

Además, muchos de los paquetes incluyen scripts de lanzamiento que automatizan el arranque conjunto de varios nodos funcionales, facilitando las pruebas y el despliegue del sistema completo en entornos locales o remotos. La estructura típica de los paquetes distingue claramente entre los directorios que contienen los nodos, las utilidades internas y los archivos asociados a la definición de mensajes y servicios.

A continuación, se describe en detalle el funcionamiento de cada uno de los paquetes y componentes que integran el sistema, incluyendo los nodos desarrollados, su lógica interna, sus dependencias y los flujos de comunicación que mantienen entre sí.

4.3. Despliegue distribuido mediante contenedores CSAR

Durante el desarrollo e integración del sistema, se ha utilizado la infraestructura CSAR para desplegar los distintos módulos del robot en contenedores Linux ligeros, de forma distribuida y aislada. Esta solución ha permitido organizar de manera práctica los diferentes componentes del sistema sin interferencias entre ellos, manteniendo entornos limpios, estables y reproducibles.

Concretamente, se ha trabajado con al menos dos entornos de ejecución diferenciados. Por un lado, un contenedor desplegado en el propio robot, dedicado a tareas de percepción sensorial, visualización y control físico. Este contenedor es completamente independiente de los de otros usuarios que también utilizan el mismo robot, lo que ha evitado conflictos de configuración, versiones o recursos compartidos. Gracias al aislamiento por contenedor, múltiples desarrolladores pueden ejecutar sus sistemas en paralelo sobre una misma máquina sin interferencias, lo cual resulta especialmente útil en entornos académicos o laboratorios compartidos.

Por otro lado, los módulos de inteligencia artificial que requieren una gran capacidad de cómputo se han ejecutado en un contenedor desplegado sobre **uedge**, un servidor de altas prestaciones conectado a la red del laboratorio MAPIR. Este equipo está especialmente preparado para tareas de computación intensiva, y sus especificaciones se resumen en la Tabla 7.

Componente	Especificación
Procesador	AMD Ryzen Threadripper PRO 7975WX (32 núcleos, 4.0 GHz)
Memoria RAM	512 GB DDR5 ECC a 4800 MHz
GPU	3 × NVIDIA RTX6000 ADA (48 GB GDDR6 cada una)
Almacenamiento	6 TB NVMe (2 TB + 4 TB) + 18 TB SATA
Red	2 × Ethernet 10 GbE
Sistema operativo	Ubuntu 22.04 con CUDA 12.6 y drivers NVIDIA actualizados

Tabla 7: Especificaciones del servidor *uedge* utilizado para ejecutar los módulos de inteligencia artificial.

La comunicación entre este servidor y el robot se gestiona a través de la red virtual proporcionada por CSAR, que permite visibilidad mutua entre contenedores distribuidos y descubrimiento automático de nodos ROS 2.

Esta arquitectura ha hecho posible separar claramente los subsistemas del sistema: la parte local se encarga de la percepción, control físico y visualización; la parte remota ejecuta modelos LLM, STT y TTS de gran tamaño, así como procesamiento semántico y generación de respuestas, aprovechando al máximo los recursos de **uedge**.

La organización basada en contenedores ha permitido lanzar de forma independiente los diferentes subsistemas, reiniciarlos o actualizarlos sin afectar al resto del sistema, así como probar nuevas funcionalidades en entornos controlados. Además, al estar encapsulados, los contenedores se pueden duplicar fácilmente para realizar pruebas paralelas o regresiones sin comprometer la instalación principal.

Desde el punto de vista del desarrollo, CSAR ha proporcionado una infraestructura robusta, estable y profesional, que ha facilitado enormemente la gestión del sistema y ha permitido desplegar una arquitectura distribuida realista, modular y escalable.

4.4. Dimensión del proyecto

La magnitud del proyecto desarrollado justifica la extensión del presente capítulo de implementación. A lo largo del desarrollo, se ha producido una considerable base de código, abarcando múltiples lenguajes y tecnologías: alrededor de 300 archivos en Python con más

de 16.000 líneas, unos 50 archivos JavaScript/JSX con aproximadamente 5.000 líneas, y varios archivos C++ que suman más de 500 líneas. Esta envergadura no solo refleja la complejidad técnica del sistema, sino también el esfuerzo dedicado a crear una arquitectura sólida, modular y escalable que pueda sostener una interacción multimodal en tiempo real entre humanos y robots.

El sistema integra componentes para procesamiento de lenguaje natural, síntesis y transcripción de voz, reconocimiento facial, control embebido, gestión de sesiones, comunicación WebSocket, interfaces web reactivas y API RESTful, entre otros. Por ello, la implementación se organiza en múltiples paquetes ROS 2, componentes web y controladores físicos, cada uno con responsabilidades claramente delimitadas. En las siguientes secciones se detalla esta implementación con el objetivo de documentar de forma clara y rigurosa cómo se han materializado los distintos módulos del sistema.

4.5. Calidad del código y patrones de diseño

Durante todo el desarrollo se ha puesto especial atención en mantener una alta calidad del código. Para ello se han aplicado buenas prácticas de programación aprendidas a lo largo de la titulación, prestando atención a la legibilidad, modularidad, mantenibilidad y reutilización del código. Se ha hecho un uso intensivo de la programación orientada a objetos, y se ha intentado seguir los principios **SOLID**, ampliamente reconocidos como base para un diseño de *software* robusto:

- **S – Single Responsibility Principle (SRP)**: cada clase o módulo tiene una única responsabilidad bien definida.
- **O – Open/Closed Principle (OCP)**: el diseño permite extender funcionalidades sin modificar el código ya existente.
- **L – Liskov Substitution Principle (LSP)**: las subclases pueden sustituir a sus clases base sin alterar el funcionamiento.
- **I – Interface Segregation Principle (ISP)**: las interfaces se han mantenido específicas, evitando imponer métodos innecesarios.

- **D – Dependency Inversion Principle (DIP)**: los módulos de alto nivel no dependen directamente de los de bajo nivel, sino de abstracciones.

Estos principios fueron definidos y popularizados por primera vez en *Design Principles and Design Patterns* (Martin, 2000).

Además, a lo largo del proyecto se han utilizado diversos **patrones de diseño**, tanto estructurales como de comportamiento, que han facilitado la construcción de un sistema escalable y desacoplado. Algunos de los más empleados incluyen:

- **MVC (Model-View-Controller)**: como patrón arquitectónico base de la interfaz web, separando lógica, presentación y estado.
- **Observer**: utilizado en el sistema de eventos y publicación/suscripción tanto en la web (EventBusContext) como en ROS 2.
- **Factory Method**: presente en la carga dinámica de modelos TTS, STT y LLM desde distintas implementaciones.
- **Strategy**: usado para seleccionar dinámicamente comportamientos como la clasificación de intenciones o la síntesis de voz.
- **Adapter**: empleado para estandarizar el acceso a modelos heterogéneos a través de una interfaz común.
- **Facade**: la interfaz web actúa como fachada unificada para múltiples funcionalidades complejas del sistema ROS 2.
- **Command**: visible en la arquitectura del protocolo HRI, donde cada mensaje representa una orden o acción bien definida.
- **State**: reflejado en la gestión del estado global en los contextos de React y en el comportamiento de interacción del robot.

El uso consciente de estos patrones ha contribuido a mejorar la claridad, robustez y capacidad de extensión del sistema. Aunque no todos los patrones han sido aplicados de forma estricta o académica, su presencia en el diseño y su aplicación práctica reflejan una comprensión sólida de los principios de ingeniería de software moderna.

4.6. Entorno ROS 2 y arquitectura de paquetes

El sistema se construye sobre ROS 2 Humble como infraestructura base de comunicación entre nodos. Cada subsistema (voz, visión, razonamiento, interfaz, control físico) se ha implementado como un conjunto de nodos ROS 2 distribuidos, que se comunican de forma desacoplada mediante mensajes y servicios personalizados. Esta arquitectura permite escalar, reutilizar y adaptar cada componente por separado.

Una de las claves de este diseño ha sido la definición de un conjunto de mensajes adaptados a las necesidades específicas del proyecto. En lugar de reutilizar tipos genéricos, se han creado paquetes independientes de definición de mensajes para cada uno de los módulos reutilizables, asegurando su portabilidad e independencia del dominio. Además, se ha creado un paquete general para representar eventos y estructuras comunes del sistema.

4.6.1. Paquetes de definición de mensajes

A continuación se listan los paquetes que definen los tipos de mensajes y servicios empleados en el sistema, acompañados de una breve descripción de su funcionalidad. La definición completa puede consultarse en el Apéndice D.

- **ros2web_msgs:**
  - **R2WMessage.msg**: mensaje genérico usado para transmitir datos entre ROS 2 y la web.
  - **R2WSubscribe.srv**: permite suscribirse desde la web a un topic de ROS 2 de forma dinámica.
- **rumi_msgs:**
  - **SessionMessage.msg**: informa sobre una detección facial durante una sesión de interacción.
  - **GetString.srv**: servicio utilitario que devuelve texto a partir de argumentos en JSON.
  - **SetSessionParams.srv**: configura parámetros como el tiempo entre detecciones o el timeout de sesión.

- **speech_msgs:**
  - **ModelItem.msg**: representa un modelo de TTS o STT disponible, indicando si necesita API key.
  - **ModelSpeaker.msg**: asocia un modelo TTS con las voces disponibles.
  - **LoadUnloadResult.msg**: informa del resultado de carga o descarga de modelos.
  - **LoadModel.srv**: carga uno o varios modelos de voz (TTS/STT).
  - **UnloadModel.srv**: libera modelos de la memoria del sistema.
  - **TTSGetModels.srv**: lista los modelos TTS disponibles y sus voces.
  - **STTGetModels.srv**: devuelve los modelos STT disponibles en el sistema.
  - **TTSGetActiveModel.srv**: devuelve el modelo y la voz TTS actualmente activos.
  - **STTGetActiveModel.srv**: devuelve el modelo STT actualmente activo.
  - **TTSSetActiveModel.srv**: establece qué modelo y voz TTS usar por defecto.
  - **STTSetActiveModel.srv**: establece el modelo STT a usar por defecto.
  - **TTS.srv**: sintetiza voz a partir de un texto, devolviendo el audio generado.
  - **STT.srv**: transcribe un audio a texto usando un modelo STT.
- **llm_msgs:**
  - **ProviderItem.msg**: define una petición de carga de modelos desde un proveedor.
  - **LoadUnloadResult.msg**: resultado detallado de carga o descarga de modelos.
  - **LoadModel.srv**: carga uno o varios modelos LLM o de embeddings.
  - **UnloadModel.srv**: descarga modelos previamente cargados.
  - **GetModels.srv**: obtiene la lista de modelos disponibles y cargados.
  - **GetActiveModels.srv**: devuelve los modelos activos de LLM y embeddings.
  - **SetActiveModel.srv**: establece un modelo específico como activo por defecto.
  - **Prompt.srv**: genera texto como respuesta a una entrada del usuario.
  - **Embedding.srv**: calcula un *embedding* vectorial a partir de una frase.

- **hri_msgs:**
  - **ChunkMono.msg**: contiene un fragmento de audio mono PCM.
  - **ChunkStereo.msg**: contiene un fragmento de audio estéreo con ambos canales.
  - **FaceNameResponse.msg**: devuelve el nombre introducido por el usuario para una cara.
  - **FaceQuestionResponse.msg**: respuesta binaria a una pregunta sobre identidad facial.
  - **FacePosition.msg**: define la posición y tamaño de una cara detectada (bounding box).
  - **FaceprintEvent.msg**: informa sobre la creación, edición o eliminación de una identidad facial.
  - **Log.msg**: registra eventos internos del sistema con nivel, origen y metadatos.
  - **Detection.srv**: ejecuta la detección de caras sobre una imagen.
  - **Recognition.srv**: realiza el reconocimiento facial de una detección dada.
  - **SanchoPrompt.srv**: envía una petición a la IA del robot, devolviendo intención y respuesta.
  - **Training.srv**: lanza instrucciones de entrenamiento del clasificador facial.
  - **GetString.srv**: servicio auxiliar que transforma argumentos JSON en una cadena de texto.
  - **TriggerUserInteraction.srv**: fuerza la interacción del robot con el usuario (preguntas, avisos).

4.6.2. Paquete ros2web

El paquete **ros2web** es un módulo desarrollado como parte de este Trabajo de Fin de Grado para habilitar la comunicación en tiempo real entre ROS 2 y una interfaz web moderna. Su objetivo es proporcionar una solución ligera, segura y extensible para exponer información desde el sistema robótico a la web, y recibir comandos o eventos desde el navegador, sin necesidad de recurrir a herramientas complejas o sobrepotenciadas como **rosbridge** (Crick et al., 2016).

A diferencia de **rosbridge**, que expone toda la API de ROS (incluyendo publicación arbitraria, ejecución de servicios, modificación de parámetros, etc.), **ros2web** restringe la funcionalidad a lo estrictamente necesario para habilitar una interacción web segura y controlada. Esta decisión de diseño se basa en una filosofía de "mínima exposición y máxima utilidad", especialmente indicada en escenarios donde la web actúa como interfaz supervisora o colaborativa, pero no como un nodo ROS completo.

Nodos.

- **Server:** Nodo único del sistema **ros2web**. Se encarga de orquestar toda la comunicación entre ROS 2 y la web. Recibe mensajes procedentes de la interfaz web mediante el topic **ros2web/ros** y los reenvía a los nodos de ROS 2. A su vez, publica mensajes generados desde ROS 2 en el topic **ros2web/web**, que se envían automáticamente a los clientes web conectados mediante WebSocket. La lógica del nodo permite transformar y reenviar datos de forma dinámica, y se apoya en tres módulos auxiliares: el servidor *WebSocket*, el servidor HTTP y el gestor de fragmentación.

  - **Topics de comunicación:**
    - **ros2web/ros (R2WMessage.msg)**: canal de entrada desde la web. Todo lo publicado aquí se procesa y se reenvía a los nodos correspondientes dentro de ROS 2.
    - **ros2web/web (R2WMessage.msg)**: canal de salida hacia la web. Todo lo que se publique aquí se envía automáticamente a los clientes conectados vía WebSocket.
  - **Parámetros:**
    - **topics (string)**: permite declarar una lista de topics ROS 2 que se deben reenviar automáticamente a la web al iniciar el nodo. Cada entrada puede incluir un alias opcional para facilitar su identificación en la interfaz.
    - **open_on_start (bool)**: si está activado (valor por defecto **true**), abre automáticamente el navegador web del usuario al arrancar el nodo, cargando la interfaz.

- **Servicios:**
  - **ros2web/subscribe** (**R2WSubscribe.srv**): permite añadir dinámicamente nuevos topics que serán reenviados a la web. El cliente puede especificar tanto el nombre del topic ROS 2 como un *alias* opcional, que se usará en la interfaz web para identificar el flujo de datos correspondiente.

Este diseño permite aislar y controlar de forma precisa la entrada y salida de datos del sistema, garantizando seguridad, modularidad y bajo acoplamiento.

Funcionalidades principales.

Las capacidades de **ros2web** incluyen, entre sus funcionalidades principales:

- **Comunicación bidireccional** entre ROS 2 y la web mediante WebSockets.
- **Reenvío de topics ROS 2** a la web, con opción de alias personalizado.
- **Publicación web de eventos y estructuras** de alto nivel desde ROS 2.
- **Infraestructura HTTP** embebida para servir la interfaz web en cualquier entorno ROS.
- **Segmentación de mensajes grandes** mediante un protocolo propio para evitar los límites de WebSocket.
- **Modularidad y extensibilidad**, compatible con cualquier nodo ROS y personalizable por el usuario.

Arquitectura de clases.

El sistema **ros2web** está compuesto por un conjunto de clases bien definidas que encapsulan cada una de las responsabilidades del framework.

- **WebSocketServer:** gestiona la conexión con los clientes web y la transmisión de mensajes *WebSocket*, tanto de forma directa como fragmentada.
- **HTTPServer:** servidor HTTP embebido para servir el *frontend* web directamente desde el entorno ROS 2.
- **WebSocketThreadMixer:** permite ejecutar en paralelo los servidores HTTP y *WebSocket* en hilos separados, asegurando la reactividad del sistema sin bloquear ROS.

- **ServerNode:** nodo ROS 2 que actúa como puente entre el sistema ROS y los componentes web. Recibe mensajes desde la web mediante un topic ROS, los reemite según alias definidos y almacena temporalmente tanto los mensajes entrantes como los cambios de topics dinámicos. También gestiona la conversión entre tipos de datos ROS y el formato del protocolo *ros2web* mediante la clase *R2WBridge*.

- **Server:** clase principal que integra y coordina los distintos componentes del sistema. Se encarga de procesar la cola de mensajes ROS, construir mensajes web en formato fragmentado (chunked) y enviarlos a través de WebSocket. También maneja la lógica de publicación/broadcast, gestiona las conexiones web, y utiliza el nodo *ServerNode* como interfaz ROS. Incluye un servidor HTTP y una arquitectura multihilo para ejecutar todos los servicios en paralelo.

- **R2WBridge:** convierte estructuras ROS (mensajes, imágenes, textos) a formatos serializables para la web, como JSON o base64, y viceversa.

- **DynamicSubscribableNode:** clase base que extiende nodos ROS para permitirles suscribirse dinámicamente a topics y reenviarlos a la web sin alterar su implementación original.

- **StoppableNode:** clase abstracta que encapsula la lógica de ejecución controlada del sistema, permitiendo iniciar y detener el ciclo de vida de un nodo de forma segura.

- **ChunkManager:** responsable de fragmentar y recomponer mensajes grandes para evitar los límites de tamaño de WebSocket. Divide los mensajes en trozos (‘chunks‘) con metadatos que permiten su reconstrucción ordenada.

- **ChunkMessage:** define para el protocolo de fragmentación el tipo de mensaje, usado para transmitir porciones de datos con control de índice y finalización.

- **Message, TopicMessage y JSONMessage:** clases que definen la estructura de los mensajes del protocolo ros2web.

Reenvío de topics.

Una de las características más potentes de **ros2web** es su capacidad para subscribirse dinámicamente a topics ROS 2 y reenviarlos a la web. Para ello, el único nodo del paquete, hereda de **DynamicSubscribableNode** lo que le permite exponer mediante parámetro o servicio:

- El **topic ROS 2 real** del que quiere reenviar datos.
- Un **nombre clave** o alias con el que ese topic se identificará en la web.

Esto permite, por ejemplo, que la interfaz web reciba imágenes en directo, eventos del sistema, transcripciones parciales, cambios de estado o métricas, sin que sea necesario modificar los nodos existentes ni exponer directamente la estructura interna del sistema.

Conversión de mensajes.

El sistema de serialización de **ros2web** gestiona automáticamente los siguientes casos:

- **std_msgs/String:** se transforma directamente a texto plano.
- **sensor_msgs/Image:** se convierte a imagen JPEG y se codifica en base64.
- **Otros tipos ROS:** se serializan a diccionarios JSON ordenados, usando introspección de campos.

Esta lógica está encapsulada en la clase **R2WBridge**, que puede extenderse para soportar nuevos tipos de mensajes o transformar su representación de forma personalizada.

Segmentación de mensajes.

WebSocket impone un límite práctico en el tamaño de los mensajes, típicamente entre 1 MB y 2 MB según el navegador. Para superar esta limitación, **ros2web** implementa un **protocolo propio de fragmentación**, basado en la clase **ChunkManager**, que divide cada mensaje en bloques más pequeños (chunks) de hasta 256 KB. Cada fragmento incluye un identificador único, un índice y una marca de finalización que permite reconstruir el mensaje completo en destino.

Esta funcionalidad no solo evita errores de transmisión, sino que representa una **aportación clave del sistema**, ya que permite enviar desde ROS 2 a la web información compleja como imágenes en alta resolución, grandes estructuras JSON o archivos de audio, manteniendo una comunicación fluida y sin comprometer la arquitectura lógica del sistema.

Infraestructura embebida.

Otro aspecto destacable es que **ros2web** incorpora un **servidor HTTP integrado**, que permite servir el *frontend* web (por ejemplo, una aplicación React) directamente desde el entorno ROS 2. Basta con indicar la ruta al directorio del *frontend* para tener una solución lista y autoalojada, ideal para despliegues en robots o laboratorios.

Comparación con rosbridge.

Aunque **rosbridge** es una solución veterana y versátil, **ros2web** presenta claras ventajas en ciertos escenarios:

- **Seguridad y simplicidad:** no permite publicaciones arbitrarias desde la web, evitando interferencias accidentales o maliciosas.
- **Configuración mínima:** no requiere modificar los nodos originales; basta con exponer los topics relevantes.
- **Ligereza y latencia:** reduce el número de dependencias y alcanza retardos inferiores a 10 ms en comunicaciones ROS2↔web.
- **Encapsulamiento semántico:** permite ocultar detalles internos del sistema usando claves *alias* para los topics.
- **Diseño modular:** puede adaptarse a cualquier arquitectura ROS 2, incluyendo herramientas como RUMI (Quemada-Torres et al., 2025).

Protocolos utilizados.

El sistema de comunicación implementado se basa en una pila de protocolos jerárquicos, donde cada nivel aporta una funcionalidad específica. El paquete **ros2web** emplea dos capas fundamentales:

- **Protocolo de segmentación:** garantiza la transmisión de mensajes de gran tamaño dividiéndolos en fragmentos (chunks) compatibles con los límites de WebSocket.
- **Protocolo ros2web:** define el formato base de los mensajes intercambiados entre ROS 2 y la web, incluyendo los tipos **MESSAGE**, **TOPIC** y **CHUNK**.

Sobre esta infraestructura básica se construye, de forma independiente, el **protocolo HRI**, diseñado específicamente en este TFG para estructurar la interacción multimodal entre el usuario y el sistema. Este protocolo define los tipos de eventos, comandos, respuestas y estructuras de datos que se utilizan en la interfaz, como **PROMPT**, **RESPONSE**, **AUDIO_RESPONSE** o **FACEPRINT_EVENT**.

Dado el nivel de detalle técnico requerido, tanto los mensajes del protocolo **ros2web** como del protocolo **HRI** se describen en profundidad en el Apéndice C, incluyendo sus campos y estructuras exactas.

Impacto y reutilización.

Aunque **ros2web** ha sido utilizado intensivamente en este TFG como componente central de la arquitectura, su diseño modular, documentado y desacoplado lo convierte en una herramienta genérica, ideal para otros proyectos que deseen dotar a un sistema ROS 2 de capacidades web sin sobreexponer su lógica interna.

Además, ha sido validado en entornos reales en combinación con otros módulos como RUMI, facilitando despliegues remotos, supervisión en tiempo real, edición segura de identidades y experimentación HRI.

4.6.3. Paquete RUMI

El paquete **RUMI** (ROS User Management Interface) es una herramienta web desarrollada como parte de este TFG para facilitar la gestión de sesiones e identidades en sistemas de interacción humano-robot (HRI). Su diseño modular y arquitectura desacoplada permiten integrarla fácilmente en cualquier sistema basado en ROS 2. La herramienta fue presentada y publicada en el simposio **RBVM25** celebrado en Almería del 4 al 6 de junio de 2025 (Quemada-Torres et al., 2025).

A diferencia de **ros2web**, que actúa como puente genérico de comunicación entre ROS 2 y la web, RUMI implementa una lógica de alto nivel orientada a la gestión remota de sesiones de interacción, incluyendo visualización del estado del sistema, edición de usuarios y trazabilidad de eventos pasados.

Nodos.

- **Session Manager:** Nodo único del sistema RUMI. Recibe las detecciones faciales desde

el topic **rumi/sessions/process** y se encarga de crear, mantener y cerrar sesiones de interacción. Internamente emplea un gestor de sesiones (**SessionManager**) que aplica lógica temporal configurable para determinar la duración de las sesiones, almacenar detecciones y gestionar la base de datos.

- **Topics de comunicación:**
  - **rumi/sessions/process (SessionMessage.msg)**: canal de entrada donde se publican las detecciones faciales generadas por el sistema HRI. Estas detecciones incluyen el ID del rostro, puntuación de detección y puntuación de clasificación, y son procesadas por el nodo para crear o actualizar sesiones.
- **Servicios:**
  - **rumi/sessions/get** (**GetString.srv**): permite consultar sesiones almacenadas, ya sea globalmente, por ID de sesión o por ID de rostro.
  - **rumi/sessions/get_summary** (**GetString.srv**): devuelve estadísticas agregadas sobre las sesiones de cada usuario, incluyendo número de sesiones, detecciones totales, duración media, primer y último contacto.
  - **rumi/sessions/set_params** (**SetSessionParams.srv**): permite modificar en tiempo de ejecución los parámetros de funcionamiento del sistema: tiempo de *timeout* para cerrar sesiones inactivas y tiempo mínimo entre detecciones almacenadas.

Gestión de sesiones.

Cada sesión de interacción representa un bloque temporal de interacción de un usuario con el sistema. Cuando el nodo recibe una detección con un nuevo **faceprint_id**, inicia una nueva sesión. Las siguientes detecciones del mismo usuario se acumulan dentro de esa sesión siempre que transcurra un tiempo mínimo (**time_between_detections**) entre ellas. Si el sistema no recibe nuevas detecciones durante un intervalo superior al **timeout_seconds** definido, la sesión se cierra automáticamente y se almacena de forma persistente.

La información registrada para cada sesión incluye:

- **ID de sesión.**
- **ID del rostro (*faceprint*).**

- **Momento de inicio y fin (timestamps).**
- **Lista de detecciones**, cada una con:
    - Marca temporal (*timestamp*).
    - Puntuación de detección facial.
    - Puntuación de clasificación facial.
    - Imagen base64 (si se desea guardar una miniatura del rostro detectado para casos anómalos).

Esta arquitectura permite un análisis detallado del comportamiento del sistema con cada usuario, manteniendo un equilibrio entre precisión y eficiencia.

Proceso interno.

El componente **SessionManager**, instanciado dentro del nodo, mantiene en memoria las sesiones activas, asociadas a cada rostro detectado. Su lógica permite:

- Abrir sesiones nuevas al detectar una persona no registrada en sesiones activas.
- Agregar detecciones a sesiones activas cuando se cumplen los criterios temporales.
- Cerrar sesiones automáticamente cuando el tiempo sin nuevas detecciones supera el umbral configurado.
- Volcar las sesiones cerradas en una base de datos SQLite a través del módulo **SessionsDatabase**.

Funcionalidades principales.

La herramienta RUMI permite, entre sus funcionalidades principales:

- Visualizar en tiempo real los eventos del sistema, incluyendo detecciones faciales, sesiones activas o respuestas generadas.
- Registrar automáticamente sesiones de interacción con los usuarios.
- Editar, fusionar o eliminar identidades desde la interfaz web.
- Visualizar remotamente la percepción visual del robot (imagen de cámara frontal).

- Integrarse fácilmente en cualquier sistema HRI sin necesidad de adaptar el backend.
- Consultar y manipular los datos a través de una API REST ligera.

Arquitectura general.

La herramienta RUMI se compone de varios módulos independientes pero interconectados:

- **SessionManagerNode (ROS 2):** núcleo funcional del sistema, donde se procesan las detecciones y se crean las sesiones.
- **SessionsDatabase (SQLite):** base de datos persistente que almacena las sesiones y sus detecciones.
- **API REST (FastAPI):** permite consultar, modificar y agregar información desde la web.
- **Faceprint Interface:** interfaz abstracta para adaptar cualquier sistema de reconocimiento facial de una manera sencilla para el operador que quiera usar la herramienta con su sistema HRI.
- **Frontend web (React):** interfaz de usuario para edición, visualización y supervisión.
- **ros2web:** pasarela de comunicación en tiempo real.

La Figura 30 muestra visualmente estos componentes y su conexión lógica dentro del sistema.

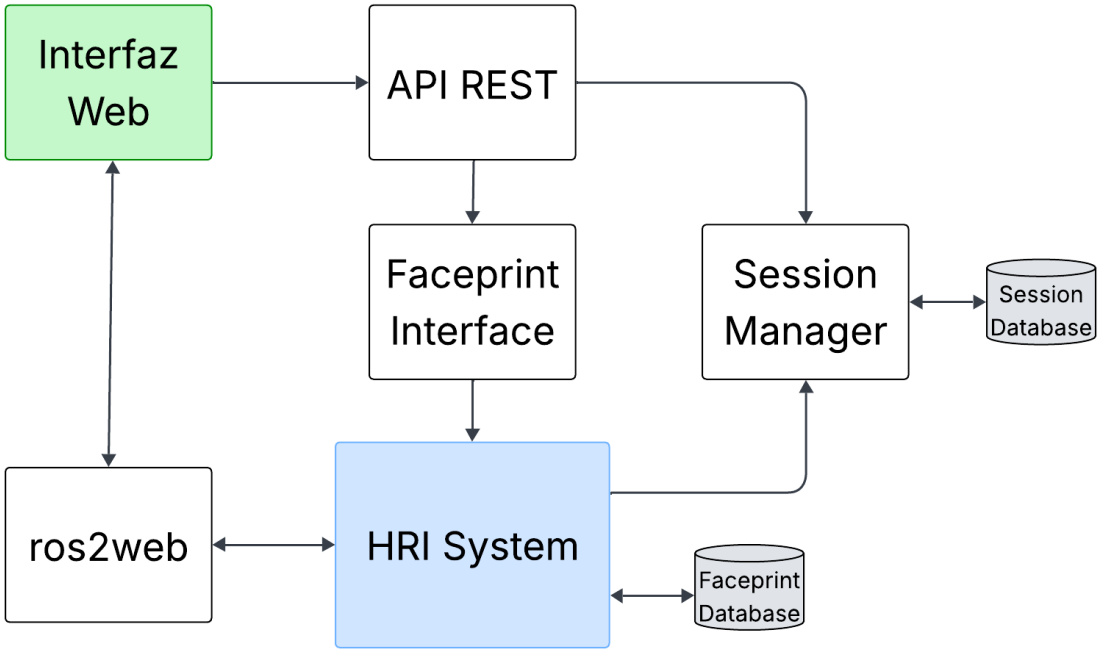


Figura 30: Arquitectura general del sistema RUMI y sus componentes ROS 2 y web. [Fuente: propia, publicada originalmente en (Quemada-Torres et al., 2025).]

Persistencia de datos.

RUMI no almacena datos biométricos como imágenes o vectores de rostro, sino únicamente la información asociada a las sesiones: hora de inicio y fin, nombre del usuario activo, eventos ocurridos durante la sesión y cualquier metainformación adicional registrada por el sistema.

Interfaz web y API REST.

Para evitar redundancias, la descripción completa de la interfaz web se detalla en la Sección 4.9, y la documentación de los endpoints REST se recoge en la Sección 4.8, ya que como hemos comentado, la herramienta se ha modificado y adaptado al resto del trabajo, por lo que la interfaz web es más amplia y la API REST se compone de más endpoints que los que se utilizan en RUMI.

Integración personalizada.

Para este TFG, RUMI se ha integrado de forma nativa con el resto del sistema multimodal, mediante los eventos definidos en el protocolo HRI y el módulo ros2web. Esta integración ha permitido:

- Registrar automáticamente las sesiones iniciadas por detección facial o comandos ver-

bales.

- Enlazar sesiones a identidades reconocidas y permitir su edición posterior.
- Visualizar el estado del sistema de forma remota desde cualquier navegador.

4.6.4. Paquete speech_tools

El paquete **speech_tools** es otra herramienta desarrollada específicamente para este proyecto que permite utilizar modelos de **Text-To-Speech (TTS)** y **Speech-To-Text (STT)** en sistemas ROS 2 de forma modular, eficiente y escalable. A diferencia de otros paquetes más rígidos o específicos de un solo modelo, **speech_tools** proporciona una interfaz común para interactuar con múltiples implementaciones de TTS y STT, tanto locales como remotas mediante API, todo ello a través de servicios ROS.

Nodos.

- **TTS:** Nodo encargado de la síntesis de voz. Permite cargar y gestionar modelos de texto a voz, configurar el modelo activo por defecto y generar audio a partir de texto. Ofrece tanto servicios comunes como servicios específicos de TTS.
  - **Servicios:**
    - **Síntesis de voz** (**TTS.srv**): convierte un texto en audio PCM utilizando el modelo y voz especificados o los que estén activos por defecto.
- **STT:** Nodo encargado de la transcripción de voz. Permite gestionar modelos de transcripción de audio a texto, configurar el modelo activo y convertir audio en texto aplicando detección de voz (VAD) previa.
  - **Servicios:**
    - **Transcripción de audio** (**STT.srv**): convierte un fragmento de audio en texto. Antes de la transcripción se aplica detección de voz (VAD) para asegurar que el audio contiene voz humana válida.

Servicios comunes a TTS y STT:

Estos servicios están disponibles en ambos nodos por separado, permitiendo una gestión completamente independiente de los modelos de TTS y STT:

- **Cargar modelos** (**LoadModel.srv**): carga uno o varios modelos indicando su nombre y, si es necesario, una clave API.
- **Liberar modelos** (**UnloadModel.srv**): descarga modelos previamente cargados para liberar memoria. No se permite descargar un modelo si está activo.
- **Obtener modelos soportados** (**TTSGetModels.srv** / **STTGetModels.srv**): devuelve todos los modelos definidos como disponibles en el sistema, incluyendo en TTS las voces asociadas.
- **Obtener modelos actualmente cargados** (**TTSGetModels.srv** / **STTGetModels.srv** con endpoint **get_available_models**): devuelve únicamente los modelos cargados y listos para usarse en ese momento.
- **Consultar el modelo activo** (**TTSGetActiveModel.srv** / **STTGetActiveModel.srv**): informa sobre el modelo que se usará por defecto si no se especifica en una petición.
- **Establecer el modelo activo por defecto** (**TTSSetActiveModel.srv** / **STTSetActiveModel.srv**): configura qué modelo (y voz en el caso de TTS) se utilizarán automáticamente si no se indican explícitamente.

La definición exacta de los mensajes utilizados en estos servicios puede consultarse en el Apéndice D.

Modelos soportados.

La herramienta **speech_tools** permite utilizar distintos modelos de **síntesis de voz** (TTS) y **transcripción de voz** (STT), cada uno con sus propias características y requisitos de ejecución. La Tabla 8 resume todos los modelos actualmente integrados en el sistema, su categoría y si requieren ejecución local o remota.

Tipo	Modelos disponibles	Ejecución
STT	whisper	Local
STT	google	Nube
TTS	bark css10 piper (voces: davefx, sharvard) tacotron2 xtts (voces: Alma María, Luis Moray, Xavier Hayasaka) your_tts	Local
TTS	google	Nube

Tabla 8: Modelos de TTS y STT soportados por speech_tools.

Algunos modelos de TTS soportan múltiples voces. Por ejemplo, **xtts** incluye voces como **Alma María**, **Luis Moray** o **Xavier Hayasaka**, mientras que **piper** soporta voces como **davefx** y **sharvard**.

Carga dinámica de modelos.

Los modelos no se cargan por defecto, ya que algunos pueden ocupar varios GB en RAM o VRAM, así que el operador que usa el paquete debe **cargar explícitamente** los modelos mediante el servicio **LoadModel**, que permite indicar qué modelos se van a utilizar y opcionalmente la **API key** en los casos que lo requieran. Internamente, cada implementación concreta de cada modelo se importa dinámicamente utilizando **importlib**, lo que evita tener que instalar todas las dependencias de todos los modelos, y permite una arquitectura modular en tiempo de ejecución.

Carga automática desde parámetros.

Además de ofrecer servicios ROS para cargar y gestionar modelos en tiempo de ejecución, la herramienta **speech_tools** permite inicializar los modelos automáticamente mediante parámetros ROS 2 al lanzar los nodos.

Este mecanismo es especialmente útil para entornos donde se desea que los modelos estén disponibles desde el inicio del sistema, sin necesidad de realizar llamadas adicionales a servicios. Los parámetros soportados son:

- **load_models**: una lista de pares **[modelo, api_key]**, donde la clave puede ir vacía si el modelo no la necesita.
- **active_model**: nombre del modelo que se desea establecer como modelo activo por defecto.
- **active_speaker**: (solo en TTS) voz activa por defecto asociada al modelo.

Para evitar el uso de cadenas mágicas, todos los modelos y voces posibles están definidos como constantes enumeradas dentro del paquete **speech_tools.models**. Estas constantes pueden ser importadas fácilmente desde cualquier archivo **launch.py**, lo que garantiza:

- Coherencia y robustez en la configuración.
- Autocompletado y validación en el entorno de desarrollo.
- Evitar errores de escritura o referencias inválidas.

Gracias a esta arquitectura, **speech_tools** permite un despliegue totalmente configurable, seguro y reproducible tanto en robots como en servidores ROS 2.

Funcionamiento.

Para utilizar un modelo, este debe estar previamente cargado. Una vez cargado, puede utilizarse:

- **Indicando el modelo explícitamente** en la petición del servicio.
- **Sin especificar modelo**, si se ha definido previamente uno activo por defecto mediante **set_active_model**.

En caso de error (modelo no cargado, inexistente o sin clave de API), el servicio devolverá una respuesta con **success** = **false** y un mensaje de error detallado. De este modo, se garantiza una trazabilidad clara y controlada del uso de los modelos.

Interfaz común y extensibilidad.

Uno de los aspectos clave de **speech_tools** es su diseño orientado a interfaces. Tanto para TTS como para STT se definen clases abstractas (**TTSModel**, **STTModel**) que encapsulan el comportamiento esperado. Las implementaciones concretas (como **WhisperSTT** o **GoogleTTS**) heredan de estas interfaces, permitiendo compatibilidad inmediata con el sistema sin necesidad de modificar los servicios, consiguiendo también así, que se puedan implementar futuros modelos sin cambiar la lógica del sistema de una manera muy sencilla.

VAD (Voice Activity Detection).

En el nodo de STT se utiliza un detector de voz humana basado en **SileroVAD** para evitar transcripciones vacías o erróneas. Si no se detecta voz humana en el audio, el sistema devuelve una transcripción vacía indicando que no se ha encontrado voz válida, lo cual mejora la robustez global del sistema.

Utilidad dentro del sistema.

Esta herramienta es utilizada dentro de este trabajo como la base de entrada y salida por voz de todo el sistema de interacción. Todos los componentes de alto nivel que gestionan diálogos, respuestas personalizadas o interacción vocal se basan en los servicios proporcionados por **speech_tools** para acceder a los modelos de TTS y STT de forma flexible, segura y eficiente.

4.6.5. Paquete llm_tools

El paquete **llm_tools** es la última de las herramientas desarrolladas en este proyecto para gestionar modelos de lenguaje (LLM) y de *embeddings* dentro de ROS 2 de forma modular, dinámica y unificada. Al igual que **speech_tools**, esta herramienta sigue una arquitectura basada en servicios, permitiendo cargar, usar y gestionar modelos de forma desacoplada y eficiente. La diferencia principal es que en lugar de separar los modelos en STT y TTS, aquí se agrupan bajo dos funcionalidades: ***Prompt*** (generación de texto) y ***Embedding*** (vectorización semántica). Aun así, muchas funcionalidades (carga dinámica, activación por defecto, extensibilidad mediante interfaces, etc.) se comparten con **speech_tools**.

Nodos.

- **LLM:** Nodo único encargado de la gestión de modelos de lenguaje (LLM) y de generación

de embeddings. Implementa una arquitectura modular que permite trabajar de forma unificada con múltiples proveedores, modelos y tipos de operación (texto y *embeddings*).

- **Servicios:**
  - **Generación de texto** (**Prompt.srv**): genera una respuesta textual a partir de un sistema de prompts, una entrada de usuario y el historial de conversación en formato JSON.
  - **Vectorización semántica** (**Embedding.srv**): convierte una frase o texto en un vector de embedding utilizando el modelo especificado (o el activo por defecto).

Servicios comunes para modelos LLM y de embeddings:

Estos servicios permiten gestionar de forma centralizada y flexible tanto modelos de LLM como de embeddings. A diferencia de otras herramientas como **speech_tools**, no se duplican por tipo: una única llamada puede actuar sobre uno u ambos tipos de modelos.

- **Cargar modelos** (**LoadModel.srv**): permite cargar uno o varios modelos de un proveedor, especificando su nombre y una clave API si es necesaria.
- **Liberar modelos** (**UnloadModel.srv**): descarga modelos previamente cargados, liberando recursos. La operación se aplica por proveedor.
- **Consultar modelos soportados o cargados** (**GetModels.srv**): devuelve los modelos definidos o actualmente disponibles para cada proveedor.
  - **get_all_models**: todos los modelos definidos.
  - **get_available_models**: solo los que están cargados.
- **Consultar modelos activos** (**GetActiveModels.srv**): indica qué modelo LLM y qué modelo de *embedding* están establecidos como predeterminados.
- **Establecer modelos activos** (**SetActiveModel.srv**): configura qué modelo (de LLM o embedding) se utilizará por defecto si no se especifica otro en las llamadas a servicios.

Proveedores y modelos soportados.

La herramienta **llm_tools** permite utilizar múltiples modelos de lenguaje natural (LLM) y de *embeddings*, agrupados por proveedor. Cada proveedor ofrece uno o varios modelos compatibles, que pueden ejecutarse de forma **local** o bien mediante **API en la nube**, en función de su implementación.

Todos los nombres de proveedores y modelos están definidos como constantes dentro del paquete **llm_tools.models**, lo que permite evitar errores de escritura, mejorar la robustez del sistema y facilitar su importación desde archivos **launch.py**, con soporte de autocompletado y validación de código.

En la Tabla 9 se resumen todos los modelos LLM soportados por la herramienta, junto con el proveedor correspondiente y el tipo de ejecución. Del mismo modo, en la Tabla 10 se presentan los modelos específicos de *embedding* compatibles con la herramienta.

Proveedor	Modelos LLM	Tipo
OpenAI	gpt-3.5-turbo gpt-4 gpt-4o	Nube
Llama	Llama-3.1-8B-Instruct Llama-3.3-70B-Instruct	Local
Mistral	Mistral-7B-Instruct-v0.1	Local
Phi	phi-2	Local
Qwen	Qwen1.5-7B-Chat Qwen2.5-7B-Instruct Qwen2.5-14B-Instruct Qwen2.5-32B-Instruct Qwen2.5-72B-Instruct	Local
Deepseek	deepseek-llm-7b-chat deepseek-llm-67b-chat DeepSeek-R1-Distill-Llama-70B DeepSeek-R1-Distill-Qwen-32B	Local
Gemini	gemini-1.5-flash-latest gemini-2.0-flash-lite gemini-2.5-flash	Nube
Gemma	gemma-3-27b-it	Local
Falcon	Falcon3-10B-Instruct	Local
Yi	Yi-1.5-9B-Chat Yi-1.5-34B-Chat	Local

Tabla 9: Modelos LLM soportados por llm_tools.

Proveedor	Modelos de *Embedding*	Tipo
OpenAI	text-embedding-ada-002 text-embedding-3-small text-embedding-3-large	Nube
Qwen	gte-Qwen2-7B-instruct	Local
Deepseek	deepseek-coder-6.7b-instruct	Local
SBERT	all-MiniLM-L6-v2	Local
E5	e5-large-v2	Local
BAAI	BAAI/bge-m3 BAAI/bge-base-en-v1.5	Local

Tabla 10: Modelos de *embeddings* soportados por llm_tools.

Carga y ejecución modular.

Para evitar sobrecarga de dependencias, cada proveedor se importa de forma dinámica mediante **importlib**, y sus modelos se cargan de forma explícita a través del servicio **LoadModel**. Esto permite instalar únicamente las librerías necesarias para los modelos deseados, y facilita la extensión con nuevos proveedores. Todos los modelos pueden descargarse y liberarse en tiempo de ejecución sin reiniciar el nodo.

Carga automática desde parámetros.

Al igual que en **speech_tools**, la herramienta **llm_tools** permite la **carga e inicialización automática de modelos** mediante parámetros ROS 2 al lanzar el nodo. Esto permite tener disponibles desde el arranque tanto modelos de lenguaje (LLM) como de *embeddings*, sin necesidad de realizar llamadas explícitas a los servicios de carga.

Los parámetros aceptados siguen un formato estructurado y están separados para LLMs y embeddings. Son los siguientes:

- **llm_load_models**: lista de triples de la forma **[proveedor, [lista_modelos], api_key]**, que indica qué modelos cargar para cada proveedor.

- **llm_active_provider**, **llm_active_model**: indican qué proveedor y modelo se establecen como activos por defecto para realizar ***prompt*** si no se especifica uno en la petición.
- **embedding_load_models**: formato idéntico a **llm_load_models**, pero para modelos de embedding.
- **embedding_active_provider**, **embedding_active_model**: definen el proveedor y modelo activos por defecto para realizar operaciones de embedding.

Todos los valores válidos para proveedores y modelos están definidos como constantes enumeradas en el módulo **llm_tools.models**, lo que permite una configuración **segura, robusta y sin cadenas mágicas**, fácilmente integrable con los archivos **launch.py** mediante autocompletado y validación en el entorno de desarrollo.

Diseño orientado a interfaces.

Cada proveedor implementa una clase propia que hereda de **BaseProvider**, la cual define una interfaz común para operaciones de prompt y embedding. Esta arquitectura permite extender el sistema fácilmente con nuevos proveedores o funcionalidades sin modificar la lógica del nodo principal.

Utilidad dentro del sistema.

Esta herramienta permite que cualquier componente del sistema pueda interactuar con modelos LLM de forma homogénea, tanto para generación de texto como para cálculo de embeddings semánticos. Esto facilita tareas como clasificación de intenciones, recuperación de contexto, personalización de diálogo, etc., sin depender de un proveedor concreto ni modificar la lógica de alto nivel.

4.6.6. Paquete hri_audio

Para el audio, **hri_audio** es el paquete responsable de capturar, procesar y gestionar el audio de entrada en el sistema de interacción. Está compuesto por varios nodos que colaboran para detectar la presencia de voz humana, segmentar correctamente los bloques de audio relevantes, activar el asistente mediante una palabra clave (*hotword*) o por transcripción continua, y finalmente transcribir o reenviar ese audio según el estado interno del sistema.

Nodos.

- **Microphone Capturer:** Captura el audio del micrófono físico en tiempo real y publica fragmentos mono y estéreo.
  - **Topics de comunicación:**
    - **hri_audio/microphone/mono (ChunkMono.msg)**: audio monoaural real capturado.
    - **hri_audio/microphone/stereo (ChunkStereo.msg)**: audio estéreo real capturado.
- **Audio:** Nodo alternativo al anterior, que simula la entrada de audio cargando un archivo WAV en bucle. Útil para pruebas sin micrófono.
  - **Topics de comunicación:**
    - **hri_audio/microphone/mono (ChunkMono.msg)**: audio monoaural simulado.
    - **hri_audio/microphone/stereo (ChunkStereo.msg)**: audio estéreo simulado.
- **Assistant Helper:** Nodo central del procesamiento de audio. Segmenta el audio recibido en bloques válidos (voz humana) y decide cuándo enviarlos a transcripción, en función del estado del sistema. Integra VAD, detección de *hotword* (Porcupine o transcripción continua) y una lógica interna basada en una máquina de estados.
  - **Topics de comunicación:**
    - **hri_audio/microphone/mono (ChunkMono.msg)**: entrada de audio del micrófono.
    - **hri_audio/assistant_helper/mode (String.msg)**: configuración dinámica del modo de escucha (*command*, *asking*, etc.).
    - **hri_audio/assistant_helper/transcription (String.msg)**: transcripción resultante enviada al nodo asistente.

- **Assistant:** Nodo que recibe transcripciones de voz y decide cómo actuar según el estado del sistema. Puede generar respuestas conversacionales, extraer información como el nombre del usuario o confirmar respuestas afirmativas. Coordina con **sancho_ai** y **speech_tools** para generar y reproducir las respuestas.

  - **Topics de comunicación:**

    - **hri_audio/assistant_helper/transcription (String.msg)**: texto reconocido desde el helper.
    - **input_tts (String.msg)**: texto arbitrario que se quiere reproducir mediante TTS.

Funcionamiento del sistema de captura y segmentación de audio.

El nodo **Assistant Helper** es el encargado de analizar el flujo continuo de audio recibido por el sistema y segmentarlo en bloques significativos que puedan ser transcritos o utilizados como comandos. Este proceso se basa en una arquitectura de ventana deslizante con bloques de **0.5 segundos**, y puede configurarse para utilizar uno de los siguientes dos métodos de detección de voz humana:

- **Criterio por intensidad RMS:** se calcula la **raíz cuadrática media** (RMS) del fragmento de audio, y se compara con un umbral configurable. Si la intensidad supera este umbral, se considera que el bloque contiene voz. La fórmula utilizada para el cálculo del RMS es:

$$\text{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

  donde x_i representa las muestras del fragmento de audio (en formato **int16**) y N es el número total de muestras del bloque. El umbral recomendado en este sistema es de 1400, calibrado empíricamente para el micrófono del robot. Este valor puede variar dependiendo del modelo de micrófono y del entorno acústico.

- **Criterio por detección de voz (VAD):** el fragmento de audio se reescala internamente a 16 kHz y se analiza utilizando un modelo neuronal preentrenado basado en **Silero VAD**.

Este modelo, ampliamente validado en entornos reales, permite detectar la presencia de voz humana con alta precisión, incluso en entornos ruidosos o con audio de baja calidad. El sistema considera que hay voz cuando el modelo identifica algún segmento hablado dentro del fragmento analizado.

Cada bloque de audio se analiza de forma independiente según el criterio elegido, y si se determina que contiene voz, se concatena al segmento en curso. Si no contiene voz y ya se había empezado a construir un segmento, se interpreta como el final de dicho bloque significativo y se lanza el proceso de transcripción.

Este comportamiento se modela internamente mediante una **máquina de estados** definida por la enumeración **AUDIO_STATE**, con tres posibles valores:

- **NO_AUDIO**: estado inicial. No se ha detectado ningún fragmento con voz. El sistema está a la espera de bloques que superen el umbral de detección definido por el criterio elegido.
- **SOME_AUDIO**: se ha detectado al menos un bloque con voz, y se ha comenzado a construir el segmento de audio relevante. Mientras se sigan detectando bloques válidos de forma consecutiva, estos se concatenan.
- **END_AUDIO**: se ha producido una transición desde bloques con voz a bloques sin voz, lo que indica el final del segmento. El audio acumulado se transcribe y el sistema se reinicia para detectar nuevos comandos.

La transición entre estos estados ocurre de forma determinista cada vez que se acumula un nuevo fragmento de 0.5 segundos. Este fragmento se analiza con el criterio de detección de voz activado (ya sea RMS o VAD). Si se detecta voz por primera vez, se produce un cambio de estado a **SOME_AUDIO**, y se añade no solo el bloque actual sino también el bloque anterior inmediatamente anterior, almacenado temporalmente, para asegurar que no se recorta el inicio del audio útil.

Del mismo modo, cuando se deja de detectar voz y ya se había acumulado audio, se cambia a **END_AUDIO**, pero antes de lanzar la transcripción, también se añade el último bloque "silencioso", con el objetivo de no cortar de forma abrupta el final de la frase. Esta decisión se

ha tomado para asegurar una transcripción más natural y completa, ya que muchos modelos de STT pueden perder palabras si el audio se interrumpe demasiado bruscamente.

Una vez detectado el final, se lanza el proceso de transcripción sobre el bloque completo, y el sistema limpia todas las colas y buffers de entrada para evitar que fragmentos residuales contaminen la siguiente interacción.

Además, si el sistema lleva un tiempo prolongado (configurable) esperando una orden sin detectar voz y el modo actual no es **ASKING**, se considera que el usuario ha perdido el interés o que la activación fue errónea. En ese caso, se reproduce un sonido de cancelación, se publica el estado visual **idle** y se vuelve al estado **NAME**.

Todo este ciclo de funcionamiento se resume en la Figura 31, que representa gráficamente la máquina de estados utilizada para la segmentación de audio.

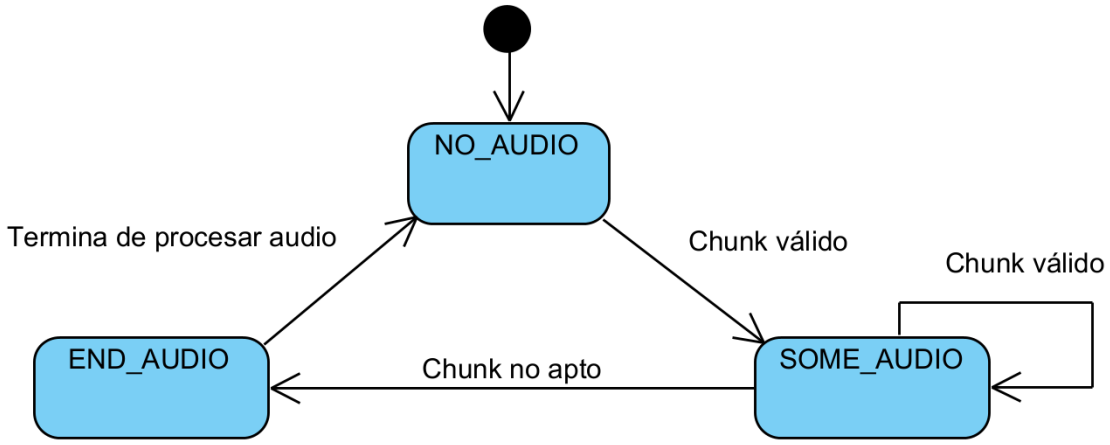


Figura 31: Máquina de estados interna utilizada para segmentar el audio en el nodo Assistant Helper. [Fuente: propia.]

Este proceso de segmentación solo se activa cuando el sistema se encuentra en los estados **COMMAND** o **ASKING**, definidos por la enumeración **HELPER_STATE** (ver Figura 8 en el apartado de diseño). Cuando el estado es **NAME**, el sistema no segmenta ni transcribe el audio, sino que escucha continuamente en busca de la palabra clave que activa al asistente (cómo veremos en el siguiente apartado de modos de detección de hotword).

Gracias a esta arquitectura, el sistema logra un equilibrio eficaz entre robustez, precisión y adaptabilidad al contexto conversacional.

Modos de detección de hotword.

Cuando el sistema se encuentra en el estado **NAME**, el nodo **Assistant Helper** activa el módulo de detección de palabra clave (*hotword*), cuya función es identificar si el usuario ha pronunciado el nombre del asistente —por defecto, "Sancho"— para iniciar la interacción. Esta detección puede realizarse mediante dos enfoques complementarios, configurables en el sistema:

- **Detección directa con Picovoice Porcupine:** este modo utiliza una librería optimizada específicamente para la detección de palabras clave en tiempo real. El modelo empleado ha sido entrenado previamente con la palabra "Sancho", y puede ejecutarse de forma local con muy bajo consumo de recursos. Internamente, cada bloque de audio recibido se reescala a la frecuencia esperada por el modelo (16 kHz) y se procesa en fragmentos del tamaño adecuado hasta que se detecta un patrón que coincide con la hotword. Este enfoque ofrece una respuesta inmediata y muy precisa, especialmente en entornos con ruido controlado.

- **Detección por transcripción continua (STT + coincidencia textual):** en este segundo modo, el audio entrante se acumula progresivamente hasta alcanzar una duración predeterminada (por ejemplo, 2 segundos), tras lo cual se transcribe utilizando el servicio de reconocimiento de voz del sistema. El texto resultante se procesa eliminando acentos, signos de puntuación y caracteres especiales, y se analiza para comprobar si contiene la palabra clave esperada. Este enfoque, si bien más costoso computacionalmente, es más robusto ante variaciones en la pronunciación y puede adaptarse a diferentes idiomas o nombres sin necesidad de reentrenar modelos específicos.

Ambos modos pueden ser intercambiados fácilmente gracias a la arquitectura modular del sistema, que define una interfaz común para todos los detectores de hotword. Una vez detectada la palabra clave en cualquiera de los dos modos, el sistema cambia automáticamente al estado **COMMAND**, reproduce un sonido de activación y vacía los buffers de audio acumulado, iniciando así un nuevo ciclo de escucha activa para capturar el comando completo del usuario.

Modos de funcionamiento del Assistant Helper.

El comportamiento del nodo **Assistant Helper** se encuentra condicionado por su estado interno, definido por la enumeración **HELPER_STATE**, cuyas transiciones se explican en detalle en la Figura 8 del apartado de diseño. En resumen:

- En el modo **NAME**, el sistema está inactivo salvo para la detección de la hotword.
- En los modos **COMMAND** y **ASKING**, se activa el sistema de segmentación y transcripción descrito anteriormente.
- En el modo **SPEAKING** el robot está físicamente hablando y se mantiene en ese estado hasta que termina de hablar.

Además, si durante el modo **COMMAND** no se detecta voz en un intervalo de tiempo determinado (configurable), el sistema interpreta que la activación ha sido accidental o que el usuario ha desistido, y revierte automáticamente al estado **NAME**, publicando un estado visual de inactividad y reproduciendo un sonido de cancelación. Esta lógica garantiza un comportamiento reactivo y resiliente a errores de activación.

Funcionamiento del nodo Assistant.

El nodo **Assistant** actúa como el centro lógico de la interacción verbal del robot. Su función principal es recibir las transcripciones generadas por el nodo **Assistant Helper** y decidir cómo procesarlas en función del contexto conversacional. Este contexto viene determinado por el parámetro **asking_mode**, que puede estar vacío (modo conversacional normal), ser **get_name** (extracción del nombre del usuario), o **confirm_name** (confirmación binaria).

En el caso de una transcripción sin modo asignado, el nodo envía el texto al servicio **sancho_ai/prompt**, que devuelve una respuesta generada por IA, junto con una emoción asociada, metadatos y una posible intención detectada. Si la intención corresponde a un comando (por ejemplo, tomar una foto), se lanza un evento específico a través del servicio **gui/request** para activar la acción correspondiente. La respuesta generada se transforma en audio mediante el servicio TTS del paquete **speech_tools** y se reproduce en el robot, controlando simultáneamente el estado facial (modo, color y emoción).

En el modo **get_name**, el sistema intenta extraer el nombre del usuario a partir del texto recibido. Si se detecta correctamente, se publica en el topic correspondiente y se revierte al modo de espera (**NAME**). En caso contrario, el robot solicita amablemente que se repita. De forma

similar, en el modo **confirm_name**, se analiza si el usuario ha respondido afirmativamente o negativamente, y se publica esta decisión en el topic correspondiente.

Además, el nodo dispone de un canal adicional (**input_tts**) que permite forzar al robot a decir una frase concreta, por ejemplo desde la interfaz web o desde el sistema visual cuando se pregunta el nombre a un rostro desconocido. Esta entrada puede ser un texto plano o un mensaje JSON que incluya, opcionalmente, la emoción deseada y el modo de interacción. En todos los casos, el sistema controla automáticamente los estados visuales (**listening**, **thinking**, **speaking**, **idle**) para garantizar una experiencia natural y coherente.

Por todo esto, el nodo **Assistant** es el componente clave para interpretar la entrada vocal, gestionar los modos de diálogo y coordinar la respuesta multimodal del robot.

Flujo de trabajo completo.

El ciclo completo de funcionamiento del sistema de entrada por voz comienza con la captura continua del audio por parte del nodo **Microphone Capturer** o, en su defecto, mediante el nodo **Audio** para reproducción de archivos de prueba. Este audio se publica en tiempo real y es consumido por el nodo **Assistant Helper**, que aplica una lógica de segmentación basada en detección de voz (mediante VAD o RMS).

Una vez completado un segmento válido, el audio se transcribe utilizando los servicios del paquete **speech_tools**, seleccionando dinámicamente el modelo STT activo. La transcripción generada se reenvía al nodo **Assistant**, que interpreta el texto según el modo de funcionamiento actual (**COMMAND**, **ASKING**, etc.).

En función del contexto, el nodo **Assistant** puede: generar una respuesta conversacional, extraer una entidad como el nombre del usuario, o confirmar una respuesta previamente solicitada. Este procesamiento se realiza mediante una llamada al sistema de lenguaje natural definido en el paquete **sancho_ai**. La respuesta generada se sintetiza con el modelo TTS activo (vía **speech_tools**) y se reproduce en el robot, coordinando a su vez la animación facial y la visualización del estado emocional mediante **face_controller**.

Durante todo este flujo, el sistema publica eventos relevantes en forma de topics ROS: cambios de modo, nombre detectado, confirmaciones, etc., lo que permite que el resto del sistema (interfaz web, lógica de sesiones, etc.) mantenga una visión coherente del estado de la interacción.

4.6.7. Paquete hri_vision

Para el la visión, **hri_vision** es el paquete responsable de gestionar el canal visual del sistema de interacción. Permite capturar imágenes en tiempo real desde una cámara, detectar y reconocer rostros, coordinar acciones con la interfaz visual del robot, y orquestar todo el *pipeline* de percepción mediante un nodo central. Está compuesto por los siguientes nodos:

Nodos.

- **Camera:** Captura imágenes en tiempo real desde una cámara física y las publica en formato **sensor_msgs/Image**.
  - **Topics de comunicación:**
    - **camera/color/image_raw (Image.msg)**: imagen capturada desde cámara física.
- **Video:** Alternativa al nodo anterior. Reproduce un archivo de vídeo en bucle simulando la cámara, útil para pruebas o demostraciones sin hardware.
  - **Topics de comunicación:**
    - **camera/color/image_raw (Image.msg)**: imagen reproducida desde archivo de vídeo.
- **HRI GUI:** Nodo que gestiona la interfaz gráfica del robot. Recibe peticiones mediante servicios y muestra una de las cuatro pantallas definidas (presentación de nombre, confirmación de identidad, muestra de imagen o espera). Implementa el patrón MVC con una arquitectura clara y desacoplada.
  - **Topics de comunicación:**
    - **gui/name_answer (String.msg)**: nombre introducido por el usuario.
    - **gui/confirm_name (Bool.msg)**: confirmación (sí/no) por parte del usuario.
  - **Servicios:**
    - **gui/request (TriggerUserInteraction.srv)**: petición para mostrar una pantalla de interacción.

- **Human Face Detector:** Nodo que ofrece un servicio de detección de rostros en imágenes. Permite elegir entre varios detectores (véase la Tabla 5), optimizados para distintos escenarios. Los resultados se devuelven en coordenadas normalizadas junto con la puntuación de confianza.

  - **Parámetros:**

    - **detector_name (string)**: permite seleccionar el detector facial que se utilizará por defecto al arrancar el nodo. Si se proporciona un nombre no válido, se utilizará automáticamente **insightface**.

  - **Servicios:**

    - **detection (Detection.srv)**: servicio de detección de rostros en imágenes RGB.

- **Human Face Recognizer:** Nodo que proporciona servicios de codificación, reconocimiento y gestión de rostros mediante *embeddings* faciales. Permite elegir entre varios codificadores (véase la Tabla 6) para obtener vectores representativos de cada rostro, y gestiona una base de datos dinámica con entrenamiento incremental, refinamiento y operaciones administrativas.

  - **Parámetros:**

    - **encoder_name (string)**: selecciona el codificador facial a utilizar por defecto al arrancar el nodo. Si se especifica un nombre no reconocido, se empleará por defecto **facenet**.
    - **db_mode (string)**: define el modo de gestión de la base de datos. Si se establece como **save**, el nodo almacena persistentemente los embeddings en disco. Si se configura como **no_save**, no se guarda ningún dato, lo que resulta útil para pruebas o entornos temporales.

  - **Servicios:**

    - **recognition (Recognition.srv)**: codificación del rostro, clasificación contra *embeddings* almacenados y evaluación de similitud.
    - **recognition/training (Training.srv)**: entrenamiento incremental y operaciones sobre la base de datos (añadir, refinar, renombrar, eliminar).

        - **recognition/get_faceprint (GetString.srv)**: consulta de información sobre los usuarios registrados, por ID o nombre.

- **HRI Logic:** Nodo principal que ejecuta y coordina todo el *pipeline* de visión del sistema. Gestiona la detección y reconocimiento facial, el entrenamiento de identidades, el almacenamiento de sesiones, la interacción con el usuario a través de la interfaz gráfica, y la generación de logs. Este nodo actúa como punto central entre los nodos ‘recognizer‘, ‘detector‘ y ‘gui‘. Toda la lógica de interacción visual se encuentra encapsulada aquí, incluyendo la gestión de respuestas del usuario y decisiones de flujo.

    - **Topics de comunicación:**
        - **camera/color/recognition (Image.msg)**: publicación del frame con resultados de reconocimiento superpuestos.
        - **logic/info/actual_people (String.msg)**: lista actual de personas reconocidas.
    - **Servicios:**
        - **logic/get/actual_people (GetString.srv)**: devuelve información de los usuarios detectados en la escena actual.
        - **logic/get/last_frame (GetString.srv)**: devuelve la última imagen procesada.

- **Test Decoder:** Nodo que permite calcular las métricas expuestas en el Capítulo 5 para los detectores, permite verificar que los resultados son los correctos.

- **Test Encoder:** Al igual que el nodo anterior, pero para los distintos codificadores faciales evaluados.

Detectores faciales disponibles.

El sistema implementa soporte para múltiples detectores faciales intercambiables dinámicamente (véase la Tabla 5), cada uno con diferentes compromisos entre precisión, velocidad de inferencia, consumo computacional y robustez ante condiciones variables. Esta diversidad permite adaptar el sistema a distintos contextos operativos, ya sea en robots con capacidad de cómputo limitada o en despliegues más potentes.

Durante la fase de validación experimental (véase Capítulo 5), se evaluaron todos los detectores disponibles, concluyéndose que **YOLOv5** ofrece el mejor rendimiento para escenarios robóticos en tiempo real, gracias a su combinación de precisión elevada y latencia reducida. Por este motivo, se ha seleccionado como detector por defecto en esta implementación.

No obstante, el sistema mantiene compatibilidad total con el resto de detectores. Por ejemplo, se pueden utilizar versiones más ligeras como **cv2** o **dlib frontal** cuando se desea ejecutar la detección directamente en el robot sin depender de procesamiento externo.

Encoders faciales disponibles.

Para la codificación de rostros, el sistema incorpora varios encoders previamente entrenados, capaces de generar embeddings representativos (*faceprints*) a partir de las imágenes faciales detectadas (véase la Tabla 6). Estos vectores permiten realizar tareas de identificación, verificación y agrupación de usuarios, ajustándose a distintos requisitos de precisión, coste computacional o tamaño del vector.

Los resultados de la validación experimental (Capítulo 5) mostraron que **FaceNet** es el encoder con mejor desempeño global en el sistema propuesto, destacando por su equilibrio entre precisión, estabilidad y eficiencia, así como por su amplio respaldo en la literatura científica. Por ello, se ha establecido como opción predeterminada.

Pese a ello, el sistema ofrece la posibilidad de seleccionar cualquier otro modelo disponible. En particular, para despliegues completamente locales sobre el robot, es posible emplear encoders más ligeros como **SFace**, que permiten un reconocimiento rápido sin necesidad de computación remota.

Funcionamiento del sistema de visión.

El sistema de visión del robot implementa un *pipeline* modular diseñado para detectar y reconocer personas en tiempo real, sin necesidad de una fase de entrenamiento previa. Esto permite que el robot aprenda de manera incremental: conforme se inicia el sistema y comienza a interactuar con usuarios, es capaz de preguntar sus nombres, almacenar sus representaciones faciales y reconocerlos en el futuro. Este enfoque elimina la dependencia de datasets cerrados y permite una adaptación dinámica al entorno y a los usuarios reales del sistema.

El flujo completo sigue las siguientes etapas principales: **captura de imagen → detección facial → alineamiento → codificación → clasificación → entrenamiento incremen-**

tal. Cada paso está implementado en un nodo o componente especializado, y todo el pipeline está orquestado por el nodo **hri_logic**, que coordina los servicios y topic ROS 2 implicados.

- La **detección facial** se realiza mediante cualquiera de las técnicas de detección soportadas. Este proceso recibe una imagen y devuelve para cada rostro detectado una **bounding box** y un **score de confianza**. Antes de aplicar el detector, se realiza una preprocesamiento que incluye conversión a escala de grises, filtrado bilateral y ecualización adaptativa del histograma, con el fin de mejorar el contraste en condiciones adversas.

- Para garantizar una representación coherente del rostro, se ejecuta una fase de **alineamiento facial**. Esta etapa utiliza el shape predictor de DLIB, un estimador basado en regresión de cascada que predice 68 *landmarks* clave (ojos, cejas, nariz, boca y mandíbula). A partir de estos puntos se aplica una transformación afín para reencuadrar el rostro de modo que los ojos estén alineados horizontalmente y se estandarice la escala y la rotación. Esta normalización es fundamental para maximizar la precisión de los modelos posteriores.

- Una vez alineado, el rostro se codifica en un **vector de características** o *embedding* usando cualquier encoder soportado. Este vector captura de forma compacta los rasgos discriminativos del rostro y constituye lo que se conoce como *faceprint*.

- El *embedding* obtenido se envía al **clasificador interno**, que es responsable de asignar una identidad al rostro o decidir que se trata de una persona nueva. Este componente compara el vector con los almacenados previamente en la base de datos, utilizando métricas de distancia como el coseno. El resultado incluye el **ID del usuario** reconocido (si aplica), su **nombre**, la **distancia de similitud** con la clase más cercana, el **score de clasificación**, y la posición del *embedding* dentro de la clase correspondiente.

- Además del reconocimiento, el sistema puede actualizar automáticamente la representación de un usuario si la nueva captura es de mayor calidad, y gestiona los procesos de **entrenamiento incremental**, **renombrado**, **eliminación** o **refinamiento** de clases a través de servicios específicos. Toda esta lógica de entrenamiento y aprendizaje adaptativo está implementada en el clasificador propio, que será descrito con más detalle en el siguiente apartado.

Este enfoque modular permite una experiencia verdaderamente personalizada y autónoma, donde el robot construye su base de conocimiento social conforme interactúa con su entorno.

Clasificador facial y aprendizaje incremental.

El sistema cuenta con un clasificador personalizado desarrollado específicamente para este TFG, cuya principal característica es permitir el **aprendizaje incremental en tiempo real** a partir de las interacciones del robot con los usuarios. Este clasificador actúa como núcleo del proceso de identificación y se apoya directamente en la base de datos descrita anteriormente para almacenar y actualizar el conocimiento.

Su funcionamiento se basa en el método **classify_face**, que recibe como entrada un vector de características facial (*embedding*) generado por el encoder. Este vector se compara con todos los almacenados en la base de datos utilizando una **distancia coseno normalizada**, definida como:

$$d_{\text{cos-norm}}(\vec{a}, \vec{b}) = \frac{1 + \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \, \|\vec{b}\|}}{2}$$

donde $\vec{a}$ y $\vec{b}$ son los vectores de características a comparar. Se devuelve el *faceprint* más cercano, la distancia obtenida y la posición del vector dentro de su clase. Esta clasificación no implica una decisión definitiva: es la **lógica de alto nivel** la que, en función del umbral de confianza y del contexto de interacción, decide si se trata de una identidad conocida o si debe crearse una nueva clase (como se explicará en un apartado posterior).

Además de clasificar, este módulo permite entrenar el sistema de manera dinámica mediante los siguientes comandos, accesibles como servicios desde otros nodos ROS 2:

- **add_class**: crea una nueva identidad en la base de datos a partir de un nombre, un vector de características inicial, una imagen representativa del rostro y un score de calidad.
- **add_features**: añade un nuevo vector de características a una clase existente. Esto permite representar una misma persona en diferentes condiciones (por ejemplo, con y sin gafas).
- **refine_class**: refina un vector existente aplicando una media ponderada con un nuevo embedding, incrementando su precisión en función del número de veces observado.

- **rename_class** y **delete_class**: permiten actualizar el nombre o eliminar clases existentes.

El clasificador también incluye el método **save_face**, que actualiza la imagen representativa de un usuario solo si se detecta una versión de mayor calidad. Para ello, se compara el nuevo **face_score** con el almacenado y, si el nuevo es mejor, se codifica la imagen en formato JPEG comprimido y se almacena en la base de datos como imagen principal del usuario.

En resumen, este clasificador actúa como un **sistema de reconocimiento activo y en continua evolución**, capaz de adaptarse al entorno, corregirse con nuevas muestras y representar múltiples variantes de un mismo individuo. Su diseño modular y desacoplado permite orquestar procesos complejos como la identificación, el entrenamiento incremental y la mejora continua de los datos faciales, sentando así las bases de un sistema robusto y autónomo de percepción social.

Almacenamiento de *embeddings* y arquitectura de la base de datos.

El sistema de reconocimiento facial implementado en este trabajo requiere una gestión eficiente, segura y dinámica de los vectores de características (*embeddings* o *faceprints*) generados a partir de los rostros de los usuarios. Para ello, se ha diseñado e implementado una **base de datos propia en memoria**, adaptada a las necesidades específicas de funcionamiento en tiempo real y aprendizaje incremental del robot.

A diferencia de una base de datos relacional tradicional (SQL), que implicaría operaciones de acceso más costosas y una rigidez estructural poco adecuada para entornos altamente dinámicos, o de una base de datos NoSQL embebida como TinyDB, que si bien ofrece más flexibilidad sigue sin permitir un control completo sobre los mecanismos de concurrencia, serialización y formato interno, se ha optado por desarrollar una solución totalmente personalizada. Esta base de datos:

- Opera íntegramente en **memoria RAM**, garantizando un acceso instantáneo a los *embeddings* y metadatos asociados.
- Implementa una política de **volcado periódico a disco**, asegurando persistencia sin comprometer la velocidad de operación.
- Utiliza **bloqueos por mutex (RLock)** en todas sus operaciones de lectura y escritura,

garantizando la seguridad ante condiciones de carrera (*race conditions*) y accesos concurrentes desde múltiples hilos o servicios ROS 2.

Cada entrada en la base de datos representa una clase de usuario, y está identificada por un **id** único generado secuencialmente. Esta clase contiene:

- El **nombre** de la persona (si se ha proporcionado).
- Una **lista de *embeddings*** previamente observados (para mejorar la generalización).
- Una **imagen representativa del rostro** (**face**), que puede ser actualizada si se obtiene una versión de mayor calidad.
- El **score** de calidad facial (**face_score**) asociado a la imagen actual.
- Información temporal como la **fecha de aprendizaje** (**learning_date**).
- Una **lista de tamaños relativos** de cada *embedding* visto, útil para ponderar o analizar el historial de aprendizaje.

Tal y como se definió en la fase de diseño del sistema (ver capítulo de diseño), este esquema de almacenamiento permite un balance óptimo entre expresividad, eficiencia y flexibilidad. El modelo se adapta en tiempo real a las condiciones de interacción, sin requerir una fase de entrenamiento previa.

Además, la base de datos incluye mecanismos para:

- **Añadir nuevas identidades** a partir de un nombre y un primer embedding.
- **Actualizar campos arbitrarios** (como el rostro representativo).
- **Eliminar o renombrar** identidades existentes.
- **Consultar por ID o nombre**, o recuperar todos los registros existentes.

Cada vez que se produce una operación relevante (alta, baja, modificación), se publica un mensaje de tipo **FaceprintEvent**, permitiendo que otros componentes del sistema —como la interfaz web o el gestor de sesiones— estén sincronizados en todo momento con el estado actualizado de la base de datos facial (esto lo veremos al mostrar la interfaz web desarrollada).

Este mecanismo de eventos ROS 2 garantiza la coherencia global del sistema, y facilita el desarrollo de módulos desacoplados que reaccionan en tiempo real a los cambios en la base de conocimiento.

En conjunto, este subsistema de almacenamiento proporciona la infraestructura esencial para que el clasificador facial y el módulo lógico puedan operar de forma continua, segura y autónoma.

Funcionamiento del *pipeline* de reconocimiento.

El núcleo del sistema de visión se encuentra en el nodo **hri_logic**, encargado de coordinar todos los pasos del proceso de detección, reconocimiento y aprendizaje de caras. Este nodo implementa una lógica cuidadosamente diseñada para permitir un funcionamiento continuo y autónomo sin requerir una fase de entrenamiento previa, aprendiendo en tiempo real a partir de la interacción con los usuarios.

El flujo principal de esta lógica se ejecuta en la función **process_frame**, invocada cada vez que se recibe un nuevo fotograma de la cámara.

1. **Captura de fotograma:** Se recibe una imagen desde el nodo de cámara y se almacena temporalmente para procesarla.

2. **Detección de rostros:** Se lanza una petición al servicio de detección facial, que devuelve para cada rostro detectado:

   - La *bounding box* (posición).
   - Un *score* que indica la calidad de la detección.

3. **Reconocimiento facial:** Para cada cara detectada, se lanza una petición al servicio de reconocimiento. Este servicio:

   - Alinea el rostro usando el **shape predictor de DLIB**, que estima 68 *landmarks* (ojos, nariz, boca…) y aplica una transformación afín que normaliza orientación y escala.
   - Genera el ***embedding*** (vector de características) mediante un encoder.
   - Compara el embedding con todos los existentes en la base de datos mediante distancia coseno normalizada, y devuelve:

- El **id** y **nombre** de la clase más cercana (si existe).
- La *distancia de similitud* con esa clase.
- La posición del vector dentro de esa clase.
- Si se ha actualizado la imagen representativa del rostro.

4. **Clasificación e interacción adaptativa:** En función de la distancia obtenida, el sistema decide cómo actuar. Se utilizan tres umbrales:

   LOWER_BOUND = 0.75, **MIDDLE_BOUND = 0.80**, **UPPER_BOUND = 0.90**

   Estos valores han sido ajustados empíricamente y ofrecen un equilibrio robusto entre precisión y generalización, como se demostrará en el capítulo de Validación.

   - **Distancia <LOWER_BOUND (desconocido):** Se considera que el rostro no pertenece a ninguna clase conocida. Si la calidad de la detección es alta y no hay peticiones pendientes, se pregunta al usuario su nombre mediante el nodo GUI. Si contesta, se crea una nueva clase y se entrena en caliente.
   - **Distancia <MIDDLE_BOUND (sospecha):** Se cree que podría tratarse de una persona conocida, pero con incertidumbre. Si la imagen es de buena calidad y no hay otras interacciones pendientes, se pregunta al usuario si se llama como la clase estimada. Si confirma, se añaden nuevas características a su clase; si lo niega, se pregunta su nombre real.
   - **Distancia <UPPER_BOUND (reconocimiento aceptable):** El sistema considera el reconocimiento suficientemente fiable, y actualiza el estado interno del usuario (última aparición, etc.). Además, refina el vector existente fusionándolo con el nuevo mediante una media ponderada.
   - **Distancia >= UPPER_BOUND (reconocimiento perfecto):** Se asume una identificación clara. Si el usuario lleva más de 60 segundos sin ser visto, se le da la bienvenida por su nombre.

5. **Entrenamiento en caliente:** Todas las acciones de refinamiento, creación o ampliación de clases se realizan mediante llamadas al servicio de ***training***, que utiliza el clasificador

descrito anteriormente. La lógica también decide si actualizar la imagen de la persona (mediante **save_face**) si se obtiene una de mayor calidad.

6. **Sincronización con la GUI:** Para toda petición al usuario, se lanza una solicitud al nodo **hri_gui**, que puede mostrar imágenes del rostro y capturar respuestas. Esta respuesta se recoge en las funciones **process_face_name_response** (cuando se introduce un nombre nuevo) o **process_face_question_response** (cuando se confirma o rechaza una identidad).

7. **Registro y actualización del estado global:** Todos los eventos significativos (creación de clase, confirmación, mejora de imagen, etc.) se registran mediante el sistema de ***logs***, y se publica continuamente el estado de los usuarios reconocidos y su última aparición para otros módulos del sistema.

Este pipeline consigue un comportamiento extremadamente versátil: detecta nuevos usuarios, reconoce a personas previamente vistas, refina continuamente el conocimiento aprendido y gestiona errores mediante interacción activa con el usuario. Todo esto ocurre en tiempo real, sin pausas ni fases de entrenamiento previas, lo que convierte al sistema en una herramienta eficaz para la percepción social autónoma y continua.

Estructura del GUI y pantallas de interacción.

El sistema se apoya en una interfaz gráfica desarrollada con **PyQt6**, ejecutada localmente en la pantalla del robot. Esta interfaz permite interactuar con el usuario en tiempo real durante el proceso de reconocimiento facial.

Aunque en esta sección se describe su integración dentro del *pipeline* de visión, los detalles completos —incluyendo capturas de las distintas pantallas, arquitectura de la interfaz, sincronización con ROS 2 y gestión de eventos— se abordan con mayor profundidad en la Sección 4.10, dedicada en exclusiva a la implementación y diseño de la interfaz local.

hri_bridge y arquitectura interna.

La clase **hri_bridge** actúa como una capa intermedia encargada de traducir entre los tipos de datos utilizados en ROS 2 (como **Image.msg**, **String.msg** o servicios personalizados) y los formatos estándar de **OpenCV** y **numpy**, ampliamente empleados en el tratamiento de imágenes y vectores numéricos.

Esta clase propia ha sido diseñada para modularizar y centralizar todas las operaciones de conversión necesarias en el sistema, permitiendo que los distintos nodos (como el detector, el reconocedor o la lógica de reconocimiento) trabajen con estructuras de datos homogéneas y limpias, sin preocuparse por los detalles de compatibilidad entre ROS y las bibliotecas de visión por computador.

Entre sus funcionalidades se incluyen:

- Conversión de imágenes entre formatos **ROS2** y **cv2**.
- Codificación y decodificación de imágenes en **base64**, útil para su transmisión o almacenamiento.
- Transformación de mensajes de detección y reconocimiento entre estructuras ROS y Python.

4.6.8. Paquete sancho_ai

El paquete **sancho_ai** implementa la lógica de razonamiento y conversación del robot Sancho, gestionando los modos de interacción y seleccionando dinámicamente los modelos de NLP para interpretar al usuario y responder con coherencia y expresividad.

El nodo central del sistema se basa en el mensaje **SanchoPrompt**, que encapsula el texto del usuario, el contexto de conversación y el modo de interacción (conversación normal, obtención de nombre, confirmación de identidad...). En función del tipo de entrada y del histórico del usuario (identificado por **chat_id**), el nodo selecciona dinámicamente el sistema de IA adecuado para generar una respuesta coherente, contextualizada y emocionalmente expresiva.

El paquete incorpora también varios nodos auxiliares de generación y evaluación, diseñados para validar distintas arquitecturas de IA y obtener las métricas mostradas en el Capítulo 5.

Nodos.

- **Sancho AI:** Nodo principal del sistema de razonamiento conversacional. Ofrece un servicio de entrada (**sancho_ai/prompt**) que acepta mensajes tipo **SanchoPrompt** y devuelve la respuesta generada, la intención detectada, los argumentos extraídos, y los

metadatos asociados al modelo LLM utilizado. Este nodo mantiene el historial de conversación por cada usuario (**chat_id**), selecciona la IA más adecuada según el modo de interacción y gestiona el flujo de la conversación.

  - **Servicios:**
    - **sancho_ai/prompt (SanchoPrompt.srv)**: servicio principal del nodo. Recibe un mensaje con el texto del usuario, el modo de interacción y el identificador de sesión. Devuelve la respuesta generada, la intención clasificada, los argumentos extraídos y los detalles del modelo empleado.

- **Embedding Generator:** Nodo auxiliar que genera *embeddings* de los comandos definidos en el sistema y sus ejemplos asociados, utilizando todos los modelos de embeddings disponibles en la herramienta **llm_tools**. Es útil para preparar representaciones vectoriales que puedan ser evaluadas posteriormente en tareas de clasificación semántica.

- **Test Asking:** Nodo de validación que evalúa el rendimiento de los LLMs en escenarios de obtención y confirmación de nombre. Usa un *dataset* propio basado en conversaciones completas y compara la respuesta generada por el modelo con el resultado esperado.

- **Test LLM Classification:** Permite validar la clasificación de intenciones usando modelos LLM y los distintos *prompt systems* definidos. Se utiliza para evaluar precisión y robustez ante variaciones en el input del usuario.

- **Test Embedding Classification:** Evalúa la clasificación basada en embeddings vectoriales. Aunque esta técnica es más rápida, los resultados de validación (véase Capítulo 5) muestran que su precisión es inferior y no permite extraer argumentos del texto de forma completa.

Modos de interacción y arquitectura general.

El nodo **sancho_ai** opera en tres modos de funcionamiento claramente diferenciados, definidos por el campo **asking_mode** del servicio **SanchoPrompt**. Esta estructura modular permite mantener una conversación fluida y adaptativa con el usuario, gestionando el historial de cada sesión mediante su identificador único **chat_id**. Además, gracias a la existencia

de distintos modos, el sistema puede integrarse de forma coherente con otros módulos como **hri_vision**, adaptando dinámicamente su comportamiento a las necesidades del entorno perceptual.

En particular, cuando el sistema de visión detecta un rostro no reconocido o con baja fiabilidad, puede activar el modo **get_name** para preguntar cómo se llama el usuario, o el modo **confirm_name** si ya tiene una hipótesis previa. Esta coordinación entre percepción visual y razonamiento conversacional permite que el robot gestione la identificación de forma natural, manteniendo la coherencia entre lo que ve, lo que sabe y lo que comunica.

- **Modo get_name**: se activa cuando el robot pregunta *"¿Cómo te llamas?"*, normalmente tras detectar un rostro desconocido o poco fiable. La respuesta se procesa mediante un modelo especializado en extraer nombres propios.
- **Modo confirm_name**: se emplea cuando el robot pregunta *"¿Te llamas María?"*, en base a una hipótesis generada desde el reconocimiento facial. El sistema interpreta la respuesta como afirmativa, negativa o ambigua.
- **Modo normal** (por defecto): se utiliza cuando no hay pregunta previa. El sistema interpreta libremente el mensaje, clasifica la intención, extrae argumentos, ejecuta la acción asociada y genera una respuesta contextualizada. Es el modo general de interacción conversacional.

Esta división de modos permite al robot adaptar dinámicamente su comportamiento al contexto de la interacción, manteniendo una coherencia entre lo que percibe visualmente y lo que comunica verbalmente. Gracias a esta arquitectura, es posible mantener conversaciones más naturales, fluidas y con un alto grado de personalización según el estado del sistema.

Modo normal: diseño modular de inteligencias conversacionales.

Para el modo **normal**, se han desarrollado múltiples arquitecturas de inteligencia artificial con niveles progresivos de complejidad. Todas ellas se implementan sobre una estructura común llamada **ModularAI**, que define un pipeline dividido en tres etapas clave:

1. **Clasificación**: se analiza el texto del usuario y se determina la intención subyacente. Dependiendo del tipo de IA, esta clasificación puede hacerse por palabras clave, *embed-*

dings o modelos LLM. Además, en algunos casos se extraen argumentos relevantes de la frase, como nombres u objetos.

2. **Ejecución de acción**: si la intención es válida y conocida, se lanza la operación correspondiente (por ejemplo, borrar un usuario, hacer una foto, etc.).

3. **Generación de respuesta**: se transforma el resultado de la acción en un texto que el robot dirá al usuario, ya sea mediante plantillas o generación directa con un LLM.

Esta arquitectura contempla dos caminos distintos en función del resultado de la etapa de clasificación. Si se detecta una intención clara, se ejecuta la acción correspondiente y se genera una respuesta asociada al resultado. En cambio, si no se reconoce una intención válida o el mensaje es ambiguo, el sistema tratará de continuar la conversación de forma fluida, según la técnica usada, permitiendo continuar el diálogo sin interrumpir la interacción ni ejecutar ninguna acción. Esto garantiza una experiencia fluida y natural, incluso en situaciones ambiguas o abiertas.

El diseño sigue conceptualmente la filosofía del enfoque *Model Context Protocol* (MCP) (Anthropic, 2024; Hou et al., 2025), en el que el resultado de la clasificación (intención + argumentos) se utiliza como contexto semántico para generar la respuesta, con el añadido de que en caso de que no haya una intención clara, se trate de continuar la conversación. Además, la arquitectura **ModularAI** ha sido diseñada para permitir la combinación libre de módulos de clasificación y generación. Aunque en este trabajo se han construido y evaluado algunas arquitecturas concretas, es posible implementar cualquier combinación compatible que siga esta estructura, utilizando clasificadores basados en reglas, *embeddings*, LLMs o cualquier otro enfoque, y generadores de respuesta mediante plantillas, modelos generativos, etc.

A continuación se describen las distintas *"IAs conversacionales"* desarrolladas, ordenadas desde la más simple hasta la más compleja:

- **Dummy AI**: una versión mínima que simplemente devuelve el texto del usuario invertido. Se utiliza para validar el correcto funcionamiento del pipeline.

- **SimpleTemplates AI**: sistema de reglas basado en palabras clave. Clasifica intenciones buscando términos específicos en el texto y selecciona una respuesta de una lista de plantillas predefinidas. Extrae argumentos simples con expresiones regulares. Su gran

ventaja es la rapidez de ejecución, pero es muy poco inteligente y no se adapta al lenguaje natural más allá de frases exactas o muy similares. Si hay intención ejecuta la acción y devuelva una respuesta basada en las plantillas. Si no hay intención, usa plantillas para decir que no se ha entendido, por lo que no permite continuar la conversación.

- **EmbeddingClassifierTemplates AI**: compara embeddings de frases con ejemplos predefinidos para clasificar intenciones. Es robusto ante sinónimos y variaciones lingüísticas, lo que lo hace útil en entornos con entradas variadas. Sin embargo, presenta limitaciones importantes: no permite extraer argumentos del texto, por lo que solo puede emplearse con intenciones que no requieren parámetros, como *"haz una foto"* o *"entra en modo espera"*. No puede usarse para ejecutar acciones que requieran información adicional, como nombres o cantidades.

  Además, este enfoque exige definir un número suficiente de ejemplos por cada intención y calcular previamente todos sus embeddings, lo que complica la ampliación del sistema. Cualquier cambio en las intenciones disponibles obliga a recalcular y actualizar manualmente esta base de ejemplos vectorizados. En cuanto a la generación de respuestas, funciona igual que la inteligencia anterior (**SimpleTemplates AI**), seleccionando frases de una lista de plantillas asociadas a cada intención.

- **LLMClassifierTemplates AI**: utiliza un modelo LLM (a través de la herramienta desarrollada para este proyecto, **llm_tools**) para clasificar y extraer argumentos, pero genera la respuesta usando plantillas. Emplea el ***prompt system* ClassificationPrompt** (ver Apéndice B) para estructurar la entrada del modelo.

  Para ello, se utiliza un archivo JSON que define todas las intenciones disponibles, incluyendo para cada una: un identificador, una descripción, una lista de ejemplos representativos y los argumentos que deben extraerse. Esta estructura permite añadir nuevas intenciones o modificar las existentes sin necesidad de editar el código fuente, haciendo el sistema fácilmente extensible y configurable por cualquier usuario. Basta con añadir una nueva entrada al JSON para que el clasificador pueda procesarla correctamente en la siguiente ejecución.

  El LLM devuelve una salida estructurada con la intención y los argumentos extraídos. A continuación, se ejecuta la acción correspondiente y se selecciona la respuesta más

adecuada entre las plantillas asociadas a ese resultado. En el caso de que la intención sea desconocida, se emplea una plantilla genérica para indicar que el robot no ha entendido al usuario. Este tipo de arquitectura no permite continuar el diálogo de forma fluida en esos casos, ya que se limita a respuestas estáticas.

- **LLMClassifierGenerator AI**: representa el enfoque más completo. Clasifica y extrae argumentos mediante un LLM con el **ClassificationPrompt**, y genera la respuesta final usando otro LLM guiado por el **SemanticResultPrompt**. En caso de que no se reconozca ninguna intención, se recurre al **UnknownPrompt**, un ***prompt system*** conversacional que permite continuar el diálogo sin interrumpir la interacción.

  Tanto el **ClassificationPrompt** como el **UnknownPrompt** se construyen dinámicamente incorporando el historial reciente del usuario, obtenido a partir de su **chat_id**. Esto permite al modelo interpretar frases referenciales o ambiguas como "bórrala", entendiendo a qué entidad se refiere basándose en el contexto de la conversación anterior. Además, en el caso del **UnknownPrompt**, se incluye también un resumen del estado actual del robot: personas reconocidas, estadísticas de las personas, rostros visibles, etc. Esto permite al robot mantener conversaciones realistas, hablar sobre lo que está viendo o sobre las personas presentes, y generar respuestas coherentes que reflejan su situación actual.

  Otra cosa interesante de este método, es que al generar la respuesta con LLMs, podemos pedir que extraiga la emoción con la que está pensando en esa respuesta, lo que nos es útil para luego representar físicamente esa sensación con la que el robot se está expresando, como veremos en el paquete **face_controller**.

  Para una descripción detallada de los ***prompt systems*** utilizados y su estructura interna, puede consultarse el Apéndice B. Todos ellos implementan una interfaz común que permite estructurar dinámicamente la entrada del modelo y adaptar el contexto según la tarea, facilitando así su reutilización en distintos modos y arquitecturas del sistema.

Aunque estas son las combinaciones implementadas y evaluadas, la arquitectura permite definir nuevas inteligencias combinando libremente clasificadores y generadores. La Tabla 11 resume las diferencias clave entre cada arquitectura, destacando sus capacidades de clasificación, generación y razonamiento contextual.

IA Conversacional	Clasificador	Generador	Argumentos	Contexto
Dummy	–	Reverso de texto	✘	✘
SimpleTemplates	Palabras clave	Plantillas	✓ (básico)	✘
EmbeddingTemplates	*Embeddings*	Plantillas	✘	✘
LLMClassifierTemplates	LLM	Plantillas	✓	✘
LLMClassifierGenerator	LLM	LLM	✓	✓

Tabla 11: Comparativa de arquitecturas conversacionales en sancho_ai.

Esto facilita la comparación sistemática entre técnicas de procesamiento del lenguaje natural (NLP), desde reglas simbólicas hasta aprendizaje profundo con modelos generativos. Para una introducción general a NLP puede consultarse (Jurafsky & Martin, 2025).

LLMs vs *embeddings*: ventajas y validación empírica.

A priori, los modelos de lenguaje generativo (LLMs) ofrecen mayor potencia que los modelos de embeddings para la tarea de clasificación de intenciones, ya que no solo identifican la intención del usuario, sino que también permiten extraer argumentos directamente del texto natural. Sin embargo, para validar esta hipótesis en el contexto del sistema propuesto, se ha llevado a cabo una evaluación comparativa rigurosa de ambos enfoques.

Se han diseñado datasets propios con ejemplos de entrada y salidas esperadas para la tarea de clasificación, y se han evaluado tanto todos los modelos de embeddings como todos los LLMs integrados en el paquete **llm_tools** (ver Tablas 9 y 10). Además, se han creado datasets específicos para evaluar los modelos LLM en las tareas de extracción estructurada en los modos **get_name** y **confirm_name**, utilizando los ***prompt systems* AskingNamePrompt** y **AskingConfirmPrompt**. Los resultados completos de estas pruebas se presentan en el Capítulo 5.

Los modelos de embeddings presentan ventajas en términos de eficiencia y simplicidad, pero tienen limitaciones importantes:

1. **No permiten extraer argumentos**: representan solo el significado global del texto, sin identificar elementos concretos como nombres o cantidades.

2. **Mayor rigidez estructural**: requieren mantener y actualizar manualmente una base

de ejemplos con sus embeddings preprocesados.

3. **Precisión dependiente del diseño del dataset**: su rendimiento depende en gran medida de la variedad y cobertura de los ejemplos definidos.

Por su parte, los LLMs requieren mayor capacidad de cómputo y una correcta definición de los ***prompt systems***, pero ofrecen mayor cobertura semántica, adaptabilidad al lenguaje natural y facilidad de ampliación: basta con añadir nuevos comandos y ejemplos al JSON de configuración, sin modificar el código del sistema.

El sistema utiliza por defecto el enfoque basado en LLMs, manteniendo disponibles otras arquitecturas para pruebas comparativas y escenarios con restricciones de recursos.

Definición de intenciones y ejemplos conversacionales.

El sistema permite interpretar comandos definidos como **intenciones**, cada una de ellas asociada a una acción concreta y, opcionalmente, a uno o más argumentos extraídos del lenguaje natural. Estas intenciones se especifican en un archivo JSON que describe su nombre, descripción, argumentos esperados y una lista de ejemplos representativos. Este enfoque hace que añadir nuevas funcionalidades sea tan sencillo como editar ese archivo y reiniciar el sistema, sin necesidad de modificar el código. A continuación se describen las intenciones implementadas:

- **DELETE_USER**: elimina a un usuario registrado en el sistema. Ejemplos: *"borra a Marta"*, *"quiero borrar a Juan del sistema"*, *"puedes quitar a Ana"*. Se tratará de borrar al usuario indicado y el robot responderá con lo sucedido.

- **RENAME_USER**: cambia el nombre de un usuario conocido por uno nuevo. Ejemplos: *çambia el nombre de Lucía a Ana"*, *renombra a Juan como Pedro"*. El robot tratará de cambiar el nombre y te responderá según la salida de la acción.

- **TAKE_PICTURE**: ordena al robot tomar una fotografía con su cámara, y este la mostrará o bien en su pantalla física (a través de la interfaz de **PyQT6**) si la pregunta se le hace al robot, o en el chat de la interfaz web si se hace por ese medio. Ejemplos: *"haz una foto"*, *"saca una foto ahora"*, *"toma una fotografía de lo que ves"*.

Estas intenciones han sido seleccionadas por su utilidad y viabilidad dentro del tiempo disponible, pero el sistema está preparado para incorporar muchas más. Añadir una nueva intención implica únicamente definir su identificador, los argumentos que debe extraer y unos pocos ejemplos que la representen. Gracias a los ***prompt systems*** empleados, el modelo generaliza fácilmente a variantes no vistas durante el entrenamiento.

Además, el sistema permite mantener conversaciones más allá de las intenciones explícitas, gracias a su capacidad de razonamiento contextual. Por ejemplo:

- Si el usuario pregunta *"¿Qué estás viendo?"*, el robot puede responder *"Veo a una persona frente a mí"* o bien *."Estoy viendo a Marta y Pedro"* si detecta múltiples rostros conocidos.
- Ante la pregunta *"¿Sabes quién soy?"*, el robot puede contestar *"Sí, eres Ana"* si hay una única coincidencia, o *"No estoy seguro, podrías ser Ana o Carla"* si hay ambigüedad visual.
- También puede responder a preguntas como *"¿A cuánta gente conoces?"*, diciendo *Conozco a 5 personas: Marta, Juan, Ana, Lucía y Pedro"*.
- Si el usuario pregunta *"¿Quién eres?"*, el robot puede contestar *"Soy Sancho, el robot social del grupo MAPIR. Estoy aquí para ayudarte."*

Estas respuestas se generan usando el **UnknownPrompt**, que incluye no solo el historial de la conversación, sino también el estado interno del robot: qué ve, a quién reconoce, qué sabe de cada usuario, etc. Esto permite que el sistema pueda generar respuestas coherentes, naturales y adaptadas al contexto multimodal de la interacción.

Modo get_name: extracción estructurada de nombres.

En este modo, el sistema utiliza exclusivamente un LLM guiado por el ***prompt system*** **AskingNamePrompt**, diseñado para extraer de forma precisa el nombre propio mencionado por el usuario en una frase libre. El modelo devuelve el nombre identificado o **None** si no se detecta ninguno, lo que permite repetir la pregunta de forma natural y sin romper la fluidez de la interacción.

Por ejemplo, si el robot pregunta *"¿Cómo te llamas?"* y el usuario responde *"Me llamo Lucía"*, el sistema es capaz de extraer el nombre *"Lucía"* de forma estructurada. Esta información se utiliza para registrar al usuario y personalizar la experiencia. En paralelo, el sistema muestra

en pantalla —mediante una interfaz desarrollada en PyQt6— una imagen capturada del rostro del usuario junto con la pregunta planteada, lo que facilita la confirmación visual por parte del interlocutor.

Modo confirm_name: verificación natural de identidades.

Este modo permite al robot confirmar si un nombre propuesto corresponde al usuario presente. Se basa en el ***prompt system* AskingConfirmPrompt**, que evalúa si la respuesta del usuario es afirmativa, negativa o ambigua, permitiendo así gestionar de forma coherente el reconocimiento de identidades.

Por ejemplo, el robot puede decir: *"Creo que eres Marta, ¿es correcto?"* y si el usuario responde *"No, te has confundido"*, el sistema entiende que la respuesta es negativa y reinicia el proceso de identificación. En caso de respuestas ambiguas como *"Puede ser"* o *"Creo que sí"*, el sistema las marca como inciertas y solicita confirmación adicional.

Al igual que en el modo anterior, durante esta interacción se muestra en la pantalla física del robot la imagen de la persona a la que se refiere, junto con la pregunta, utilizando la interfaz gráfica desarrollada con PyQt6. Esto permite al usuario verificar visualmente si el sistema ha asociado correctamente el nombre propuesto a su imagen actual.

4.6.9. Paquete sancho_web

El paquete **sancho_web** conecta el sistema conversacional con la interfaz web, proporcionando tanto comunicación en tiempo real mediante WebSockets (a través de **ros2web**) como una API REST flexible y configurable. Además, gestiona el almacenamiento de *logs* en una base de datos SQLite, permitiendo la trazabilidad del comportamiento del robot.

Nodos.

- **Sancho Web:** nodo principal que orquesta la comunicación entre la interfaz web y el sistema ROS 2. Interpreta los mensajes enviados desde el navegador siguiendo el protocolo HRI definido, y responde con mensajes estructurados generados por el sistema conversacional, incluyendo respuestas de texto, voz y contexto perceptual.

  - **Topics de comunicación:**

    - **ros2web/web (R2WMessage.msg)**: recepción de mensajes desde la web.

        - **ros2web/ros (R2WMessage.msg)**: publicación de mensajes hacia la web.
        - **recognition/event (FaceprintEvent.msg)**: recepción de eventos de reconocimiento facial.

- **API REST:** nodo encargado de levantar un servidor HTTP que expone distintas APIs REST configurables mediante parámetros.

    - **Parámetros:**
        - **apis** (string): lista de APIs a lanzar. Puede incluir una o varias de las siguientes: **LOGS**, **FACEPRINTS**, **SESSIONS**, **TTS_MODELS**, **STT_MODELS** y **LLM_MODELS**.

- **Database Manager:** nodo responsable de almacenar los logs del sistema en una base de datos local y proporcionar un servicio de recuperación de los mismos.

    - **Topics de comunicación:**
        - **logs/add (Log.msg)**: recepción de *logs* generados por el sistema. El nodo obtiene los parámetros del mensaje y guarda la información en la base de datos.
    - **Servicios:**
        - **logs/get (GetString.srv)**: consulta estructurada de los *logs* del sistema.

Funcionamiento del nodo Sancho Web.

El nodo **sancho_web** actúa como puente entre la interfaz web del robot y su lógica interna de razonamiento. Recibe mensajes del navegador en formato JSON a través del canal **ros2web/web**, los interpreta según su tipo (texto, audio o solicitud de transcripción) y los procesa invocando los servicios adecuados del sistema:

- Si el mensaje es de tipo **PROMPT**, se llama al servicio **sancho_ai/prompt**, obteniendo una respuesta textual estructurada. Opcionalmente, se genera también un audio con **speech_tools/tts**.

- Si es de tipo **AUDIO_PROMPT**, primero se transcribe el audio con **speech_tools/stt**, y luego se trata como un *prompt* de texto.

- Si es de tipo **TRANSCRIPTION_REQUEST**, se devuelve solo la transcripción del audio recibido.

El nodo también escucha eventos de reconocimiento facial en tiempo real y los reenvía a la web para sincronizar el contexto perceptual del robot con la interfaz gráfica. Toda la comunicación con la web se realiza de forma estructurada mediante mensajes codificados en el protocolo HRI definido en este proyecto (ver Sección C).

Funcionamiento de la API REST.

El nodo **api_rest_node** permite exponer múltiples APIs REST independientes desde un único servidor HTTP. Mediante el parámetro **apis**, el usuario puede seleccionar qué conjuntos de endpoints desea activar en cada ejecución, como por ejemplo lanzar solo la API de *logs* o la de modelos STT. Esto permite adaptar la API al entorno de despliegue (producción, *testing*, etc.), y evitar exponer funcionalidades innecesarias.

Aunque todo el sistema se ejecuta desde un único proceso en un solo puerto, internamente cada API depende de servicios ROS 2 diferentes y está desacoplada mediante clases independientes. Esta estructura modular permite una posible evolución futura hacia una arquitectura de microservicios, aunque ya en su estado actual refleja ese enfoque distribuido: cada API es autónoma, se conecta a su parte específica del sistema (sesiones, modelos, logs…) y puede ser reutilizada o extendida sin afectar a las demás. La implementación completa de los endpoints puede consultarse en la Sección 4.8.

4.6.10. Paquete face_controller

El paquete **face_controller** se encarga de gestionar la animación física de la cara del robot, incluyendo tanto los ojos (colores) como la boca (movimiento sincronizado con el audio). Este paquete contiene un único nodo, escrito en C++ para garantizar una ejecución rápida y eficiente en tiempo real, especialmente durante la reproducción de audio con sincronización labial. A diferencia de otros paquetes del sistema escritos en Python, aquí se ha optado por C++ debido a su menor latencia, mayor rendimiento y control más preciso sobre el acceso al hardware.

El nodo se comunica directamente por puerto serie con un microcontrolador (ESP32), encargado de ejecutar las animaciones en LEDs y tiras RGB. Los detalles de la implementación

física y del código embebido se describen en la Sección 4.7.

Nodos.

- **FaceNode:** Nodo principal de control de cara. Suscribe un topic de texto para recibir los modos y emociones, se comunica por puerto serie con el ESP32 y mide el volumen del audio reproducido para generar una animación de boca realista.
    - **Topics de comunicación:**
        - **face/mode** (**std_msgs/msg/String**): recibe comandos que indican el estado actual del robot (**idle**, **listening**, **thinking**, **speaking**) o una emoción a representar (**happy**, **sad**, etc.).

Funcionamiento general.

El nodo **face_node** recibe información sobre el estado emocional del robot y los distintos modos de interacción definidos en el sistema. Estos modos son:

- **idle**: estado por defecto cuando el robot no está interactuando.
- **listening**: se activa cuando se detecta la palabra clave, como se explica en el paquete **hri_audio**.
- **thinking**: se activa mientras el sistema **sancho_ai** genera una respuesta.
- **speaking**: se activa mientras se reproduce la respuesta mediante TTS.

La transición entre estos modos sigue una lógica interna que puede representarse como una máquina de estados, ilustrada en la Figura 32.

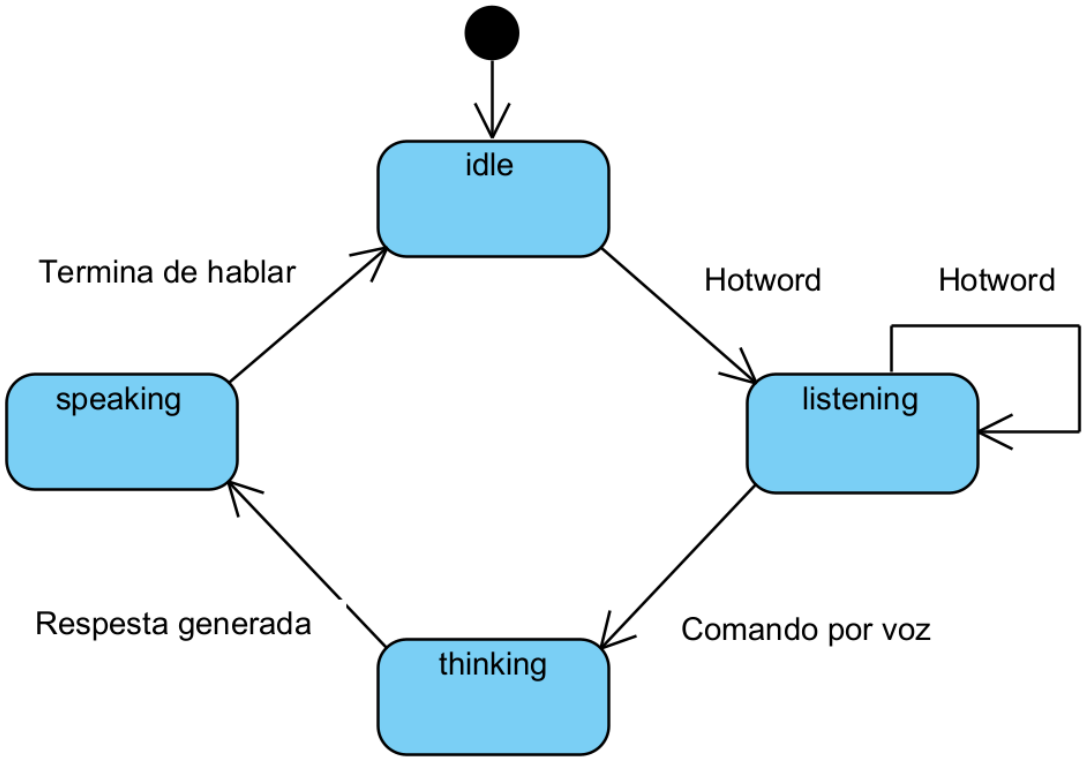


Figura 32: Máquina de estados visuales del robot según el modo de interacción. [Fuente: propia.]

Además de procesar el modo de funcionamiento, el nodo también puede recibir una emoción codificada como texto, como *happy*, *angry* o *neutral*. Cada una de estas emociones tiene asociada una combinación específica de colores, que se envía al microcontrolador ESP32 para iluminar los ojos del robot de forma coherente con el estado emocional. La correspondencia entre emociones y colores se detalla en la Sección 4.7.

Cuando el robot entra en el modo *speaking*, el nodo activa un proceso de monitorización del volumen de audio del sistema mediante la biblioteca PortAudio. El sonido de salida se analiza en tiempo real dividiendo el flujo de audio en bloques (*chunks*), sobre los que se calcula el valor RMS (*Root Mean Square*), una medida de la energía sonora. Este valor se normaliza dinámicamente en función del volumen mínimo y máximo detectado durante la sesión, y se convierte en un nivel de intensidad discreto que va desde *low* hasta *full*.

Dicho nivel se envía al ESP32 a través del puerto serie, donde se traduce en una animación de la boca utilizando una tira LED RGB. Este mecanismo permite reflejar visualmente el movimiento del habla del robot de forma continua y expresiva. Para mantener la sensibi-

lidad adaptativa, el sistema reinicia automáticamente los valores de referencia si detecta más de tres segundos de silencio, permitiendo así recalibrarse a nuevos contextos sin intervención manual.

El flujo completo de este procesamiento se resume en la Figura 33, que muestra las distintas etapas implicadas: desde la recepción del estado hasta el cálculo de RMS, la normalización, el mapeo a niveles de intensidad y la comunicación con el microcontrolador. Esta lógica garantiza una animación fluida y sincronizada con el audio que refuerza la expresividad del robot.

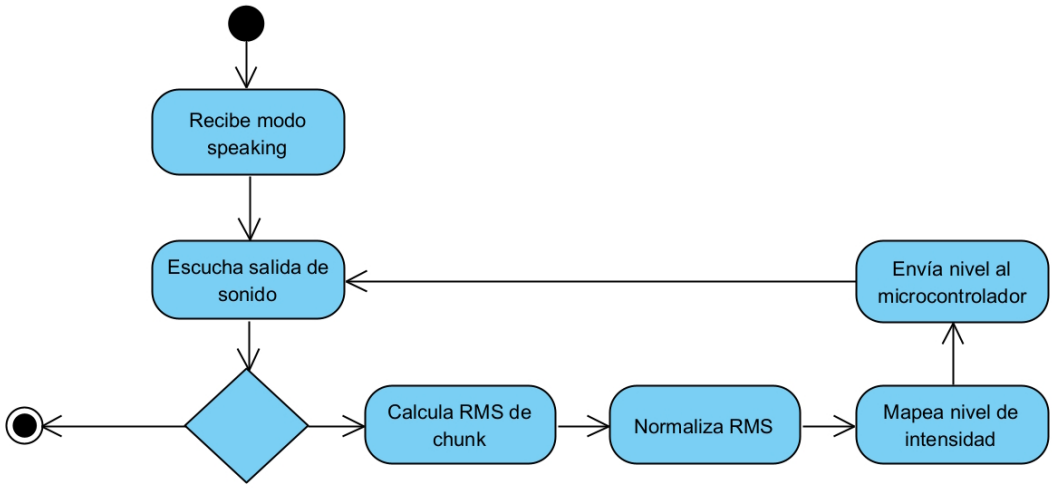


Figura 33: Flujo de datos del nodo de control facial: análisis de audio, normalización del volumen y animación de la boca mediante el microcontrolador. [Fuente: propia.]

Por último, cabe destacar que aunque el control de los ojos no ha sido desarrollado por completo en este TFG (ver Sección 4.7), sí se ha integrado parcialmente para permitir reacciones emocionales básicas. En cambio, el control de la boca y el diseño del protocolo de comunicación con el ESP32 sí ha sido desarrollado en su totalidad como parte de este trabajo.

4.7. Control físico embebido

El sistema físico del robot incluye dos elementos clave para expresar emociones y sincronizar el habla: los **ojos**, que representan emociones mediante combinaciones de color, y la **boca**, formada por una tira de LEDs RGB que se anima en tiempo real en función del audio reproducido. Ambos componentes están controlados por un microcontrolador **ESP32**, que recibe comandos por puerto serie desde el nodo **face_controller**, descrito en la Sección 4.6.10.

4.7.1. Control de ojos

El controlador de los ojos ha sido desarrollado en otro Trabajo de Fin de Grado previo, y ha sido integrado en el sistema mediante una sencilla interfaz de texto por puerto serie. Al recibir una cadena con el nombre de una emoción, el microcontrolador actualiza automáticamente el patrón de color y animación de los ojos para representar dicha emoción.

Las emociones reconocidas por el sistema son:

happy, **surprised**, **sad**, **angry**, **bored**, **suspicious**, **neutral**

4.7.2. Control de la boca

El código de animación de la boca ha sido desarrollado íntegramente en este TFG. El ESP32 interpreta los comandos enviados por el nodo **face_controller** y genera animaciones físicas sobre una tira LED de 12 diodos WS2812B.

Durante el modo **speaking**, el nodo inicia la monitorización del volumen del sistema mediante la biblioteca **PortAudio**. Esta captura se realiza sobre el dispositivo 'pulse' y permite obtener el valor RMS (Root Mean Square) de la señal en fragmentos temporales. Este valor se normaliza dinámicamente en base a los valores mínimo y máximo detectados en la sesión actual, permitiendo que la animación sea sensible al volumen relativo, y no dependa del volumen absoluto del sistema.

Los niveles de intensidad detectados se mapean a siete niveles de animación, que activan diferentes patrones de iluminación en la tira LED:

off, **low**, **medium_low**, **medium**, **medium_high**, **high**, **full**

Esta lógica de estados se sincroniza con el nodo **face_controller** (ver Sección 4.6.10), siguiendo la misma máquina de estados del sistema HRI.

Colores disponibles.

La boca utiliza una paleta estándar de 16 colores con nombre, basada en la convención empleada por Microsoft para representaciones visuales[2]. Estos colores permiten representar de forma clara las emociones durante el habla, utilizando el color como refuerzo emocional. Los valores RGB de estos colores se muestran en la Tabla 12.

[2] https://learn.microsoft.com/es-es/windows-server/administration/windows-commands/color

Nombre	Valor RGB
black	(0, 0, 0)
dark_blue	(0, 0, 170)
dark_green	(0, 170, 0)
dark_aqua	(0, 170, 170)
dark_red	(170, 0, 0)
dark_purple	(170, 0, 170)
gold	(255, 170, 0)
gray	(170, 170, 170)
dark_gray	(85, 85, 85)
blue	(85, 85, 255)
green	(85, 255, 85)
aqua	(85, 255, 255)
red	(255, 85, 85)
light_purple	(255, 85, 255)
yellow	(255, 255, 85)
white	(255, 255, 255)

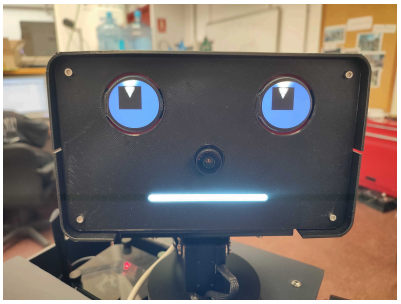
Tabla 12: Colores disponibles para la boca del robot.

4.7.3. Emoción expresada

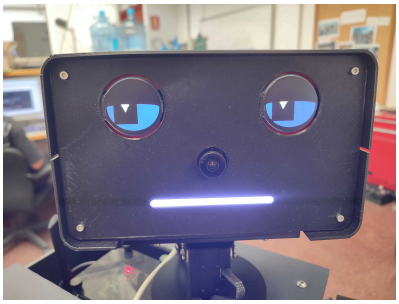
Durante la interacción con el usuario, el sistema genera una emoción asociada a la respuesta, que se transmite simultáneamente al ESP32 para actualizar tanto los ojos como el color base de la boca. De este modo, la sincronización entre expresión visual (ojos) y la animación labial (boca) refuerza el componente emocional del robot, haciendo la experiencia mucho más natural.

La Figura 34 muestra ejemplos reales del robot con diferentes emociones, combinando color y animación en ojos y boca. Además, los ojos del robot no solo cambian de color según la emoción, sino que se mantienen en movimiento constante con desplazamientos aleatorios suaves, lo que aporta dinamismo y vitalidad a su expresión incluso en reposo, mejorando significativamente la sensación de presencia.

(a) happy

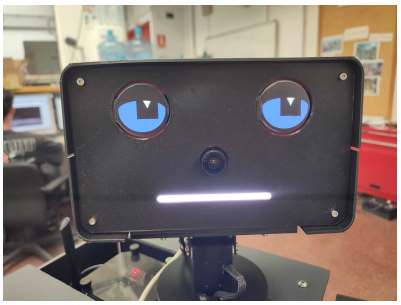


(b) surprised

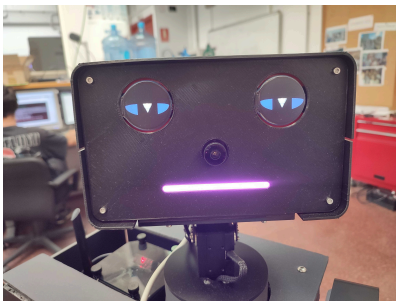


(c) sad

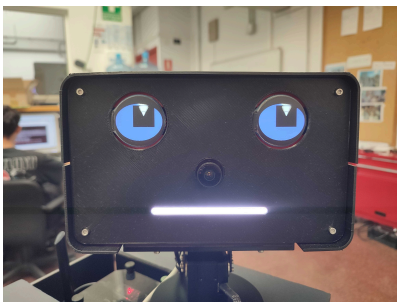
(d) angry



(e) bored



(f) suspicious



(g) neutral

Figura 34: Sincronización visual de emociones: ojos y boca según el estado emocional del robot. [Fuente: propia.]

Animación dinámica en modo speaking.

Durante el modo **speaking**, el nodo **face_controller** activa la monitorización del volumen del sistema mediante la biblioteca **PortAudio**. Esta captura se realiza sobre el dispositivo de entrada 'pulse' y permite obtener el nivel RMS (Root Mean Square) de la señal en fragmentos de 0.1 segundos. El valor RMS se normaliza dinámicamente en base a los valores mínimo y máximo detectados durante cada sesión de habla, de forma que la animación se ajusta al volumen relativo y no al absoluto. Así se evita que un TTS más bajo o más alto genere movimientos

excesivamente tenues o exagerados.

Los valores normalizados se cuantifican en 6 niveles de animación distintos, que se envían como comandos por puerto serie y se representan mediante máscaras de activación sobre los LEDs. Estos niveles son:

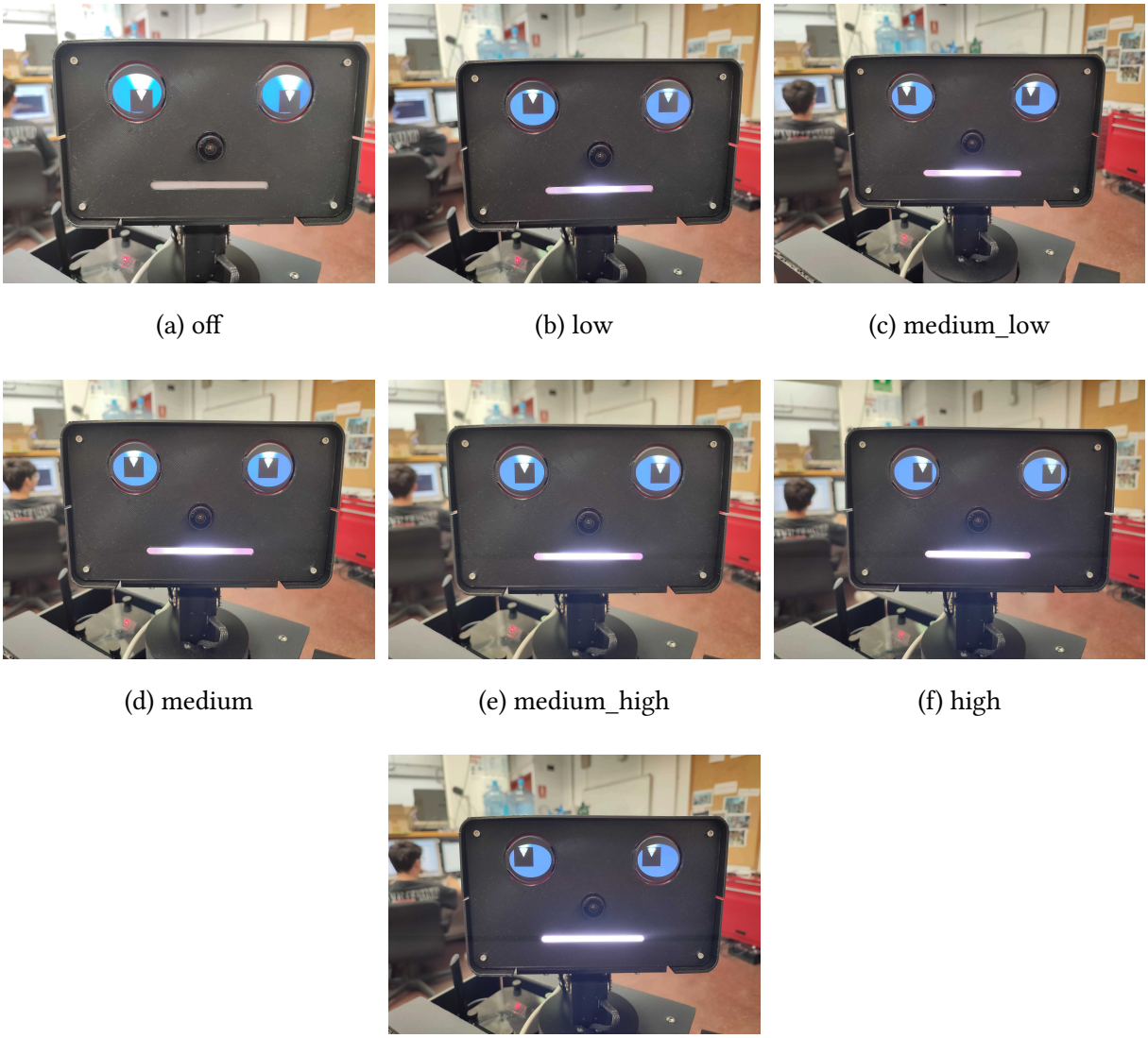
off, **low**, **medium_low**, **medium**, **medium_high**, **high**, **full**

Cada nivel activa una cantidad diferente de LEDs en la tira, lo que simula una apertura progresiva de la boca. La Tabla 13 resume las máscaras empleadas para cada nivel y el número de LEDs encendidos.

Nivel	Máscara (binaria)	LEDs encendidos
off	0b000000000000	0
low	0b000001100000	2
medium_low	0b000011110000	4
medium	0b000111111000	6
medium_high	0b001111111100	8
high	0b011111111110	10
full	0b111111111111	12

Tabla 13: Máscaras de activación y número de LEDs encendidos por nivel de boca.

La Figura 35 muestra ejemplos reales de la boca del robot iluminada con cada uno de los siete niveles de intensidad sonora disponibles. Estos niveles son independientes del estado emocional, lo que permite combinar la expresividad emocional con una animación realista del habla. En la figura se muestran con expresión neutral para ilustrar el rango completo de intensidad. Gracias a esta gradación, el sistema puede generar una animación fluida y continua que refleja visualmente cómo el robot está hablando, adaptándose dinámicamente al volumen del audio en tiempo real.



(a) off (b) low (c) medium_low

(d) medium (e) medium_high (f) high

(g) full

Figura 35: Animación física de la boca en los distintos niveles de intensidad sonora. [Fuente: propia.]

4.8. APIs del sistema

Las APIs REST (Representational State Transfer) constituyen un estilo arquitectónico propuesto por Roy Fielding en su tesis doctoral (Fielding, 2000), basado en el uso de recursos identificables mediante URIs y manipulables a través de los verbos del protocolo HTTP. Este enfoque ha sido ampliamente adoptado por su simplicidad, escalabilidad y compatibilidad con arquitecturas distribuidas.

En el contexto de este proyecto, se ha implementado una API REST basada en JSON que

actúa como capa de exposición pública para toda la funcionalidad crítica del sistema. Aunque a nivel interno el sistema opera sobre una arquitectura basada en ROS 2 y se apoya en la ejecución distribuida de múltiples nodos, esta API ofrece un punto de acceso unificado accesible desde la interfaz web.

A nivel técnico, todo el servicio REST se ejecuta en un único nodo que actúa como un *API Gateway* inteligente. Este nodo abstrae la lógica subyacente del sistema y traduce las operaciones REST en llamadas a los servicios internos de ROS 2. Aunque desde el exterior parece una única API, internamente cada grupo de endpoints (*faceprints*, sesiones, *logs*, modelos TTS/STT/LLM) está respaldado por servicios ROS 2 diferentes, lo que permite mantener un diseño modular y extensible. Esta arquitectura ofrece lo mejor de ambos mundos: una interfaz REST desacoplada, fácil de consumir desde el frontend, y un *backend* ROS 2 distribuido y altamente especializado.

La API ha sido desarrollada en **Python utilizando el framework FastAPI**, que permite definir rutas de forma tipada y declarativa, generando automáticamente documentación interactiva. El proyecto sigue una *estructura de carpetas organizada y escalable*, en la que cada grupo funcional (*faceprints*, sesiones, modelos, logs…) se implementa en módulos independientes. Se han definido modelos de datos para todas las entidades relevantes mediante **pydantic**, lo que garantiza validación estricta, documentación automática y claridad en la definición de esquemas. Este enfoque, junto a la adopción de principios como la separación de responsabilidades, modularidad y validación explícita, contribuye a un diseño profesional y mantenible. A continuación, explicamos a modo de manual de usuario los endpoints desarrollados.

4.8.1. Faceprints API

Permite crear, consultar, actualizar y eliminar registros faciales conocidos (*faceprints*). Es utilizada por el sistema para almacenar las representaciones codificadas de las caras detectadas y clasificadas por los módulos de visión.

- **GET /api/v1/faceprints**: listar todos los registros. Soporta filtros, paginación y selección de campos.
- **POST /api/v1/faceprints**: crear un nuevo registro facial a partir de una imagen y un nombre.

- **GET /api/v1/faceprints/{id}**: obtener un registro facial concreto por su ID.
- **PUT /api/v1/faceprints/{id}**: actualizar el nombre asociado a un faceprint.
- **DELETE /api/v1/faceprints/{id}**: eliminar un *faceprint* del sistema.

4.8.2. Sessions API

Gestiona las sesiones de detección de usuarios, es decir, los intervalos de tiempo en los que una persona ha sido detectada por el sistema. Esto permite analizar la presencia y actividad de los usuarios a lo largo del tiempo.

- **GET /api/v1/sessions**: lista todas las sesiones con filtros por ID de faceprint, paginación y ordenación.
- **GET /api/v1/sessions/{id}**: obtiene una sesión concreta.
- **GET /api/v1/sessions/summary**: vista resumida de las sesiones registradas.

4.8.3. Logs API

Proporciona acceso a los registros de eventos del sistema relacionados con la detección y clasificación facial. Estos logs permiten reconstruir la historia de interacciones relevantes y depurar el sistema.

- **GET /api/v1/logs**: obtiene todos los logs registrados.
- **GET /api/v1/logs/{id}**: obtiene un *log* específico por ID.

4.8.4. TTS Models API

Gestiona los modelos de síntesis de voz disponibles en el sistema, permitiendo a la interfaz web controlar qué voz debe utilizar el robot.

- **GET /api/v1/tts_models**: lista todos los modelos registrados, con información sobre su estado.
- **GET /api/v1/tts_models/{model}**: detalles de un modelo concreto.

- **POST /api/v1/tts_models/load**: carga un modelo concreto.
- **POST /api/v1/tts_models/unload**: descarga un modelo previamente cargado.
- **POST /api/v1/tts_models/activate**: activa un modelo cargado como predeterminado para su uso.

4.8.5. STT Models API

Permite listar y gestionar los modelos de reconocimiento de voz empleados por el sistema.

- **GET /api/v1/stt_models**: muestra todos los modelos de transcripción de voz disponibles.
- **GET /api/v1/stt_models/{model}**: detalles de un modelo específico.
- **POST /api/v1/stt_models/load**: carga un modelo de STT.
- **POST /api/v1/stt_models/unload**: elimina un modelo cargado.
- **POST /api/v1/stt_models/activate**: activa el modelo como predeterminado para el sistema.

4.8.6. LLM Models API

Esta API permite gestionar los modelos de lenguaje (LLM) empleados para la generación de texto, clasificación de intenciones y otras tareas de NLP.

- **GET /api/v1/llm_models**: lista todos los proveedores registrados y sus modelos.
- **GET /api/v1/llm_models/{provider}**: muestra los modelos de un proveedor.
- **POST /api/v1/llm_models/load**: carga un modelo LLM concreto.
- **POST /api/v1/llm_models/unload**: descarga un modelo cargado.
- **POST /api/v1/llm_models/activate**: activa un modelo determinado para su uso por defecto.

4.8.7. Buenas prácticas en el diseño de la API

La API se ha desarrollado siguiendo buenas prácticas REST modernas:

- **Versionado**: todos los endpoints se agrupan bajo el prefijo **/api/v1**, lo que facilita compatibilidad futura.
- **Convenciones REST**: se utilizan los verbos HTTP adecuados para cada operación (GET, POST, PUT, DELETE).
- **Identificación de recursos**: las rutas incluyen IDs significativos para operar sobre recursos concretos.
- **Diseño modular**: cada conjunto de endpoints está claramente asociado a una funcionalidad independiente del sistema.
- **Desacoplamiento funcional**: cada API es gestionada por servicios ROS2 distintos, pero la interfaz REST no depende de su implementación interna.

Estas prácticas se complementan con la validación automática de datos mediante **pydantic**, la generación de documentación Swagger y ReDoc a partir de los esquemas declarados, y una arquitectura interna limpia basada en controladores y modelos.

Con la entrega del código del proyecto se incluye también el fichero de especificación *OpenAPI* en formato **openapi.json**, que describe con todo detalle los parámetros, cuerpos de petición y estructuras de respuesta de cada endpoint. Esta documentación técnica permite ampliar la información presentada en esta sección y facilita la integración futura del sistema.

Una de las grandes ventajas de utilizar OpenAPI es la posibilidad de generar automáticamente una página web interactiva como la que se muestra en la Figura 36. Esta interfaz no solo documenta la API de forma clara y navegable, sino que permite a los desarrolladores probar los endpoints directamente desde el navegador, ver ejemplos de uso, validar esquemas de entrada y explorar todas las rutas disponibles de manera intuitiva. Este tipo de documentación web es especialmente útil en entornos colaborativos y en procesos de despliegue, ya que mejora la comprensión del sistema y reduce la necesidad de soporte adicional.

La especificación OpenAPI (OpenAPI Initiative, 2024) se ha convertido en un estándar de facto ampliamente adoptado para describir y documentar APIs REST de forma estructurada y legible tanto por humanos como por máquinas.

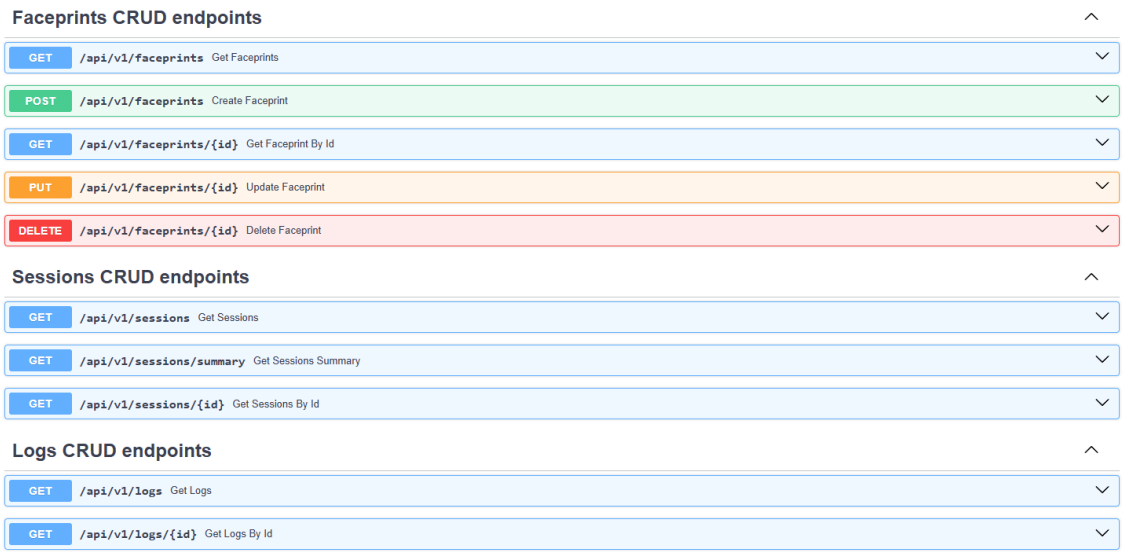


Figura 36: Ejemplo de documentación web generada automáticamente a partir de la especificación OpenAPI. [Fuente: propia.]

4.9. Interfaz web

En esa sección mostramos a modo de manual de usuario la interfaz web del sistema. Se trata de una interfaz moderna desarrollada en **React**, diseñada para permitir la supervisión remota, la interacción conversacional con el robot, y la gestión completa del sistema desde cualquier dispositivo conectado a la red. Esta aplicación ofrece una experiencia responsiva, modular y altamente integrada con el *backend* ROS 2 mediante WebSockets y peticiones REST.

4.9.1. Estructura del proyecto

El frontend está organizado en una estructura modular dentro de la carpeta **src**, siguiendo buenas prácticas de diseño en React. Cada carpeta responde a una funcionalidad concreta del sistema:

- **components**: contiene componentes reutilizables que definen la interfaz visual de la aplicación, como cabeceras, pies de página, elementos interactivos, modales o visores.

- **contexts**: agrupa todos los contextos globales implementados con **React Context**, que gestionan el estado compartido del sistema (modelos, eventos, usuarios, conexión...).
- **pages**: incluye las vistas principales de la aplicación, organizadas en subcarpetas por secciones funcionales. Cada vista agrupa sus propios componentes internos.
- **hooks**: recoge funciones reutilizables basadas en **React Hooks**, empleadas para gestionar cámara, micrófono, tamaño de ventana y otras funcionalidades del entorno.
- **services**: se integra principalmente en los contextos, donde se gestionan las conexiones a la API REST mediante funciones asincrónicas basadas en **fetch**.
- **utils**: contiene funciones auxiliares y utilidades como la gestión de la conexión WebSocket al backend ROS 2.
- **assets**: carpeta destinada a recursos estáticos, como imágenes o iconos utilizados en la interfaz.
- **css**: estilos globales de la aplicación.

Esta organización por dominios facilita la mantenibilidad, el escalado del sistema y la separación clara de responsabilidades entre lógica, presentación e integración.

4.9.2. Contextos y estado global

La aplicación utiliza múltiples contextos implementados con **React Context** para gestionar el estado global y la lógica de interacción con el sistema de forma estructurada y eficiente. A continuación se detallan los principales contextos utilizados:

- **APIContext**: expone una interfaz unificada para acceder a todos los endpoints de la **API REST** mediante funciones genéricas como **getAll**, **getById**, **create**, **update**, **delete**, y funciones específicas para cada API. Cualquier componente puede importar estas funciones para interactuar con la API de forma sencilla y consistente. Esta capa permite desacoplar completamente la lógica de red del resto de la aplicación, lo que facilita la reutilización y el mantenimiento. Cualquier componente de la aplicación web puede importar estas funciones y llamar a cualquier *endpoint* de la API fácilmente.

- **ModelsContext**: mantiene en memoria las listas de modelos TTS, STT y LLM disponibles, y permite consultar en cualquier momento qué modelo está activo. Proporciona funciones para recargar los datos desde la API y sincroniza automáticamente el estado al reconectarse con ROS 2. Gracias a este contexto, los componentes no necesitan gestionar manualmente el estado de los modelos ni realizar peticiones individuales.

- **FaceprintsContext**: encapsula toda la lógica relacionada con los rostros reconocidos por el sistema. Permite consultar, añadir, editar y eliminar identidades faciales desde la interfaz, y actualiza automáticamente su estado al recibir eventos en tiempo real de tipo **FACEPRINT_EVENT**. También ofrece funciones de alto nivel como **doAddFaceprint** o **doUpdateFaceprint**, que encapsulan toda la lógica de petición, actualización local y notificación al usuario.

- **ChatContext**: centraliza el estado del chat con el robot, incluyendo el historial de mensajes, los ajustes del usuario (TTS, transcripción automática, etc.), y la lógica de envío de mensajes tanto por texto como por audio. Ofrece funciones como **handleSend** y **handleAudio**, que encapsulan todo el flujo de interacción con el backend (creación de mensajes, envío por *WebSocket*, actualización del historial y reproducción de audio). Además, guarda los ajustes del usuario en localStorage para mantener la configuración entre sesiones.

- **WebSocketContext**: gestiona la conexión persistente con ROS 2 mediante WebSocket. Implementa reconexión automática en caso de caída y proporciona la función **sendMessage** para emitir comandos al sistema. Los mensajes recibidos se reenvían al **EventBusContext**, permitiendo una distribución flexible. También ofrece el estado **isConnected** para que los componentes puedan reaccionar a la disponibilidad del backend.

- **EventBusContext**: implementa el patrón de diseño *publish/subscribe*, permitiendo a cualquier componente suscribirse a eventos concretos como si fuera un bus de eventos interno. Esta arquitectura desacopla totalmente el origen (mensajes desde ROS 2) de los receptores, lo que simplifica la gestión de eventos y hace el sistema altamente extensible y reactivo.

- **AudioContext**: ofrece funciones para reproducir audio generado por el sistema. Con-

vierte datos de audio en formato **Int16Array** a **Float32Array** y los reproduce usando la Web Audio API del navegador. También permite detener el audio en reproducción, controlar si hay un audio activo, y registrar qué audio se está reproduciendo en cada momento. Este contexto es esencial para la sincronización labial y la experiencia auditiva del usuario.

- **ToastContext**: permite mostrar mensajes emergentes (*toasts*) en cualquier parte de la aplicación, indicando al usuario errores, éxitos o advertencias. La interfaz web se apoya en este contexto para informar de forma visual y no intrusiva sobre el estado del sistema, resultados de acciones o fallos de red.

En conjunto, estos contextos conforman la capa lógica que conecta la interfaz con el *backend*, gestionando el estado compartido de manera robusta y favoreciendo la separación de responsabilidades entre presentación, datos y eventos.

4.9.3. Integración con ROS y API REST

La interfaz web se comunica con el backend ROS 2 utilizando dos mecanismos complementarios: una API REST estándar y una conexión WebSocket en tiempo real. El acceso a la API REST se realiza exclusivamente a través del **APIContext**, que proporciona funciones reutilizables para invocar cualquier endpoint de forma estructurada y desacoplada, permitiendo a los componentes acceder a recursos como modelos, *logs*, sesiones o identidades faciales de forma sencilla y uniforme.

ros2web en el cliente: la comunicación en tiempo real con ROS 2 se implementa mediante un canal WebSocket gestionado por la clase **R2WSocket**, que encapsula toda la lógica de conexión, reconexión automática y fragmentación de mensajes. Esta clase transforma cualquier mensaje JSON en una secuencia de *chunks* y los envía siguiendo un protocolo interno definido por **ros2web**. Al recibir los fragmentos, los reensambla y despacha los mensajes reconstruidos a los manejadores correspondientes.

Esta conexión es gestionada globalmente a través del **WebSocketContext**, que expone la función **sendMessage** a cualquier componente de la aplicación. Así, cualquier parte del frontend puede enviar comandos directamente a ROS 2 sin necesidad de gestionar manualmente

la conexión, mientras que los mensajes entrantes (como respuestas, eventos o transcripciones) se redistribuyen mediante el **EventBusContext** usando un patrón *publish/subscribe*.

Todo el flujo de comunicación sigue el **protocolo HRI** definido en el Apéndice C, que organiza los mensajes en tipos como **PROMPT**, **RESPONSE**, **AUDIO_RESPONSE**, **PROMPT_TRANSCRIPTION** o **FACEPRINT_EVENT**. Gracias a este diseño modular y desacoplado, el sistema puede reaccionar de forma inmediata a cambios en el entorno, mostrando respuestas del robot, actualizando la galería de rostros o reflejando el estado de los modelos en tiempo real.

4.9.4. Funcionalidades

La interfaz web implementada ofrece un conjunto completo de funcionalidades diseñadas para supervisar y controlar el sistema de interacción humano-robot de forma remota, eficiente y amigable. Todas las vistas han sido diseñadas para ofrecer una experiencia clara, fluida y adaptativa, tanto en ordenadores como en dispositivos móviles.

Entre las funcionalidades destacadas se incluyen:

- **Chat interactivo con el robot**: El usuario puede mantener una conversación directa con el robot mediante mensajes de texto o voz. El sistema responde utilizando modelos TTS y muestra tanto el texto como el audio de la respuesta. La interfaz guarda el historial del chat y permite configurar opciones como transcripción automática, idioma, y modelo activo.

- **Gestión completa de rostros (*faceprints*)**: La galería facial muestra todos los rostros conocidos por el sistema. Desde esta vista, el usuario puede añadir nuevas identidades (subiendo imágenes o capturándolas desde cámara), editar los nombres asociados, y eliminar registros. La galería se actualiza en tiempo real ante cualquier cambio, y está sincronizada con los eventos recibidos desde ROS 2 mediante el protocolo HRI.

- **Visualización de sesiones e interacciones**: La interfaz permite consultar el historial de interacciones con cada persona reconocida. Se muestran gráficas con la evolución temporal de las puntuaciones de detección y reconocimiento facial, junto con métricas como duración, número de encuentros, y fechas. Esta información facilita el análisis del comportamiento del sistema y la relación mantenida con los usuarios.

- **Control de modelos de IA**: Desde la interfaz se pueden consultar los modelos de STT, TTS y LLM disponibles, activar o desactivar cada uno, y establecer el modelo por defecto. Esta funcionalidad está integrada con los contextos de estado global y sincronizada con ROS 2, permitiendo un cambio dinámico de proveedor sin necesidad de reiniciar nodos.

- **Exploración de *logs* del sistema**: Se dispone de una vista dedicada a visualizar los registros generados por el sistema durante las sesiones. Estos logs muestran mensajes internos, respuestas del sistema, errores o eventos relevantes, y permiten al desarrollador o supervisor monitorizar el comportamiento interno del robot.

- **Indicadores visuales y notificaciones**: La aplicación utiliza un sistema de mensajes emergentes (*toasts*) para informar al usuario sobre el resultado de sus acciones (como operaciones exitosas, errores o advertencias). Estos mensajes están distribuidos por toda la interfaz para ofrecer una retroalimentación inmediata y no intrusiva.

- **Indicadores de carga y estados**: Todas las operaciones asíncronas (peticiones a la API, cambios de modelo, envíos al robot, etc.) incluyen indicadores visuales de carga (spinners, desactivación de botones, placeholders temporales) para evitar acciones repetidas y mejorar la percepción de fluidez.

- **Sincronización en tiempo real con ROS 2**: Gracias al módulo **ros2web**, los eventos y mensajes del backend (como nuevas transcripciones, detecciones, respuestas o cambios en los modelos) se reflejan de forma instantánea en la interfaz sin necesidad de recargar la página. Esto permite una interacción continua y coherente con el robot.

- **Modo flotante de cámara**: La interfaz ofrece una ventana flotante que permite ver en tiempo real la imagen captada por la cámara del robot. Esta ventana puede moverse, redimensionarse o cerrarse, y sirve como complemento visual durante la interacción.

- **Persistencia de configuración**: Preferencias del usuario, como el modelo activo, el uso de audio o las opciones del chat, se almacenan localmente mediante **localStorage**, de forma que se conservan entre sesiones y recargas.

- **Diseño modular y escalable**: Toda la interfaz está construida en React siguiendo principios de separación de responsabilidades, reutilización de componentes y gestión cen-

tralizada del estado. Esto facilita su extensión, mantenimiento y adaptación a nuevas funcionalidades.

En conjunto, la interfaz web proporciona un entorno visual avanzado y altamente integrado que permite gestionar de forma centralizada todos los módulos del sistema HRI, fomentando una interacción natural y un control total desde cualquier navegador moderno.

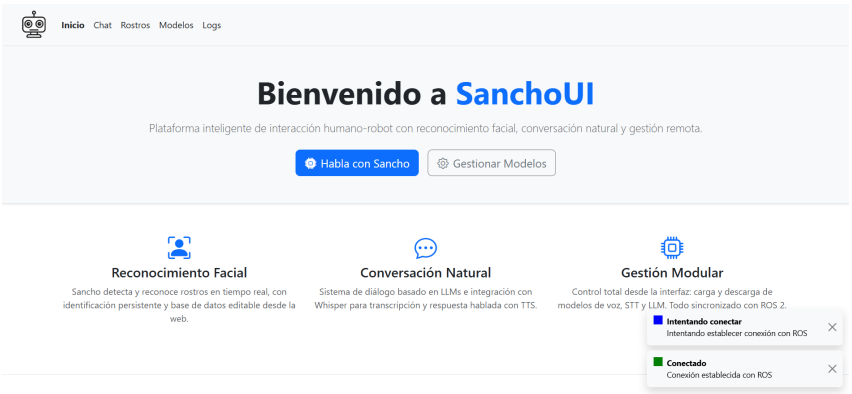
4.9.5. Páginas principales

El frontend incluye cinco vistas principales que permiten interactuar con el sistema de forma visual e intuitiva. A continuación se describen brevemente, acompañadas de capturas de pantalla que muestran su diseño tanto en ordenador como en dispositivos móviles.

Inicio

La página de inicio actúa como punto de entrada al sistema, proporcionando una visión general clara y accesible. Desde esta vista, el usuario puede comprender rápidamente el propósito del sistema, acceder a las funcionalidades clave mediante botones destacados y verificar si los servicios principales están correctamente conectados. Es especialmente útil como primer contacto para usuarios sin experiencia previa, gracias a su estructura simple, clara y visualmente equilibrada.

En la Figura 37 se muestra el diseño de esta pantalla tanto en su versión de escritorio como móvil. En la versión de PC se aprecia una disposición en columnas que incluye el logotipo del sistema, un mensaje de bienvenida y accesos directos. En la versión móvil, la disposición se reorganiza verticalmente para ajustarse al espacio disponible sin perder legibilidad ni funcionalidad. Ambas vistas mantienen el mismo estilo visual, con elementos centrados, iconografía clara y uso consistente de colores y tipografía.



(a) Versión PC

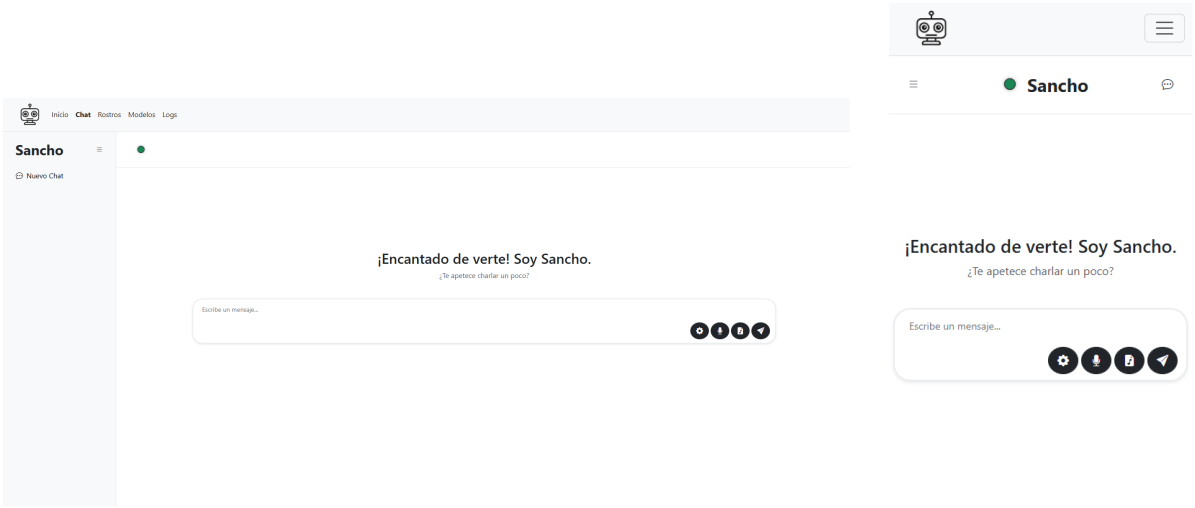
(b) Versión móvil

Figura 37: Pantalla de bienvenida del sistema. [Fuente: propia.]

Chat

La vista de chat permite al usuario interactuar con el robot mediante texto, notas de voz o archivos de audio, recibiendo respuestas generadas por modelos de lenguaje (LLM) que pueden incluir texto y audio. Es el núcleo de la interacción conversacional del sistema, integrando funcionalidades de transcripción, síntesis de voz, clasificación de intenciones y respuesta contextual.

En la Figura 38 se muestra el estado inicial del chat, sin mensajes aún. A la izquierda puede observarse el menú lateral con accesos rápidos a las distintas secciones del sistema, que puede ocultarse si se desea maximizar el área de conversación. En la parte superior derecha, un indicador de estado muestra el estado de conexión con el backend: verde si está conectado, rojo si está desconectado, y azul si está reconectando. En el área inferior del chat se encuentran cuatro botones principales: uno para abrir el panel de configuración, otro para grabar audio desde el navegador, otro para subir archivos de audio desde el dispositivo, y el último para enviar texto escrito manualmente. También se incluye un botón para iniciar una nueva conversación, que limpia el historial y permite comenzar desde cero.



(a) Versión PC

(b) Versión móvil

Figura 38: Vista general del chat sin mensajes. [Fuente: propia.]

Al pulsar el botón de ajustes se despliega un menú (Figura 39) donde pueden configurarse distintas opciones de comportamiento del sistema: si al grabar audio se debe enviar automáticamente o mostrar primero la transcripción para revisarla; si el robot debe responder solo con texto o también con audio; y si se desea activar el modo técnico que muestra información adicional sobre cada mensaje.

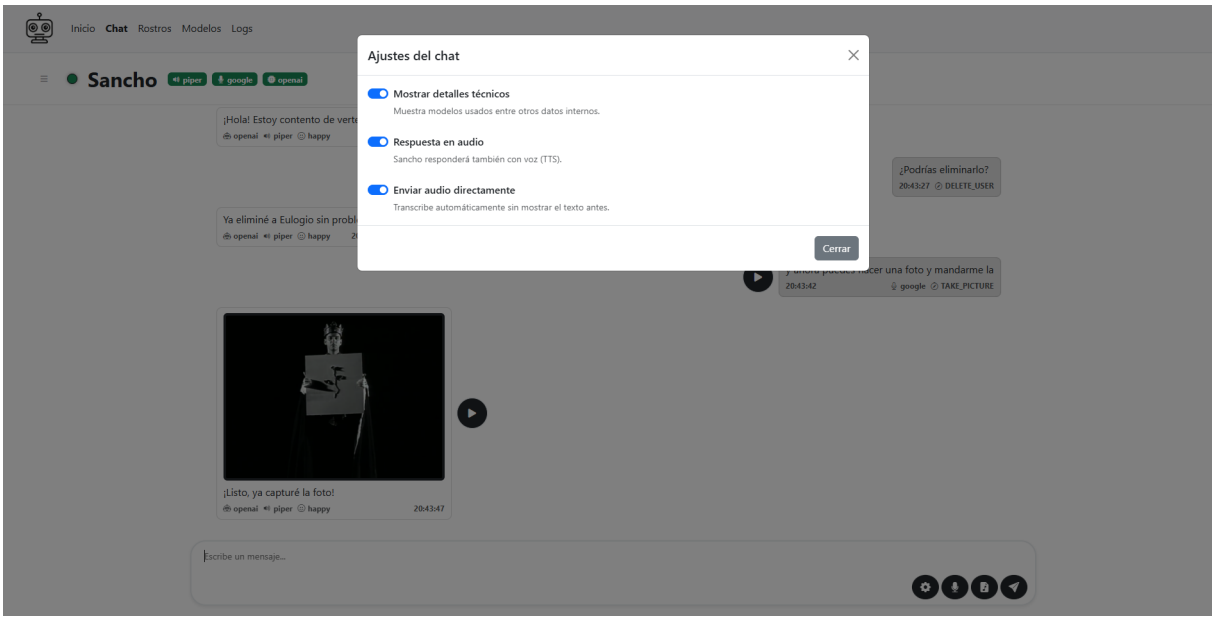
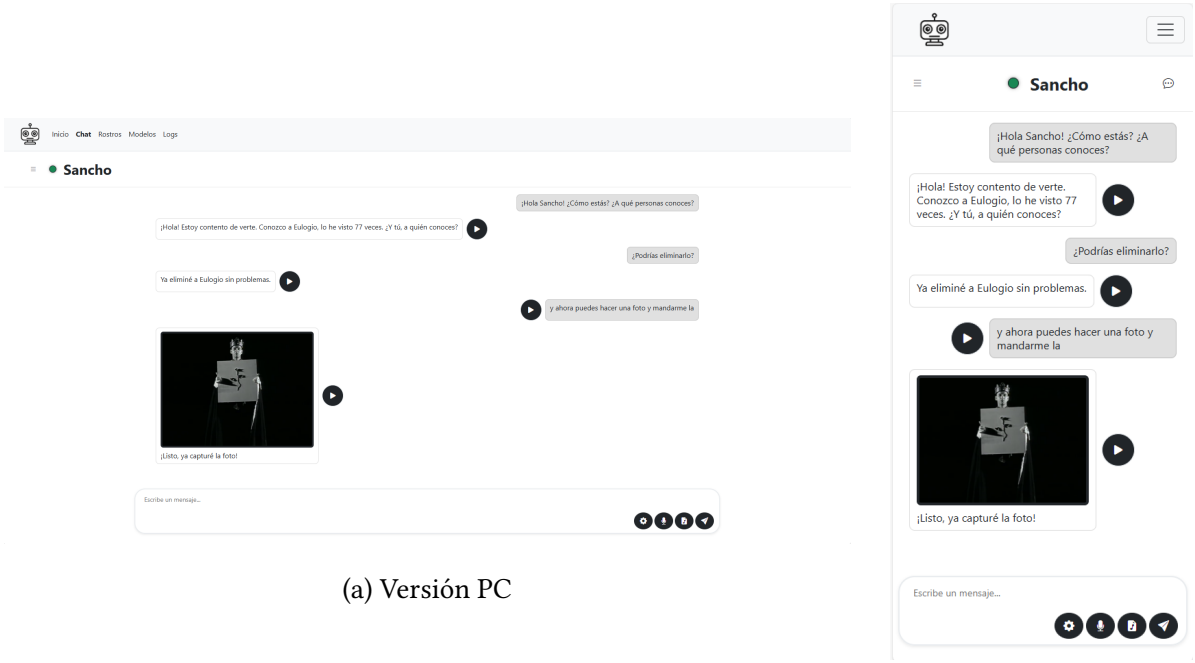


Figura 39: Panel de configuración de conversación. [Fuente: propia.]

Una vez iniciada la conversación, la Figura 40 muestra una sesión con varios mensajes. El usuario ha enviado inicialmente un mensaje de texto, y el robot ha respondido con texto y audio. Posteriormente, se han enviado más mensajes, incluyendo uno como nota de voz, que ha sido transcrito automáticamente y enviado al backend. En ese mensaje, el usuario solicitaba tomar una foto, y el robot ha respondido adjuntando una imagen capturada en tiempo real (procedente de un vídeo de prueba). Toda esta interacción está gestionada mediante el sistema de protocolos definidos, con clasificación de intenciones y activación de módulos según el contexto.

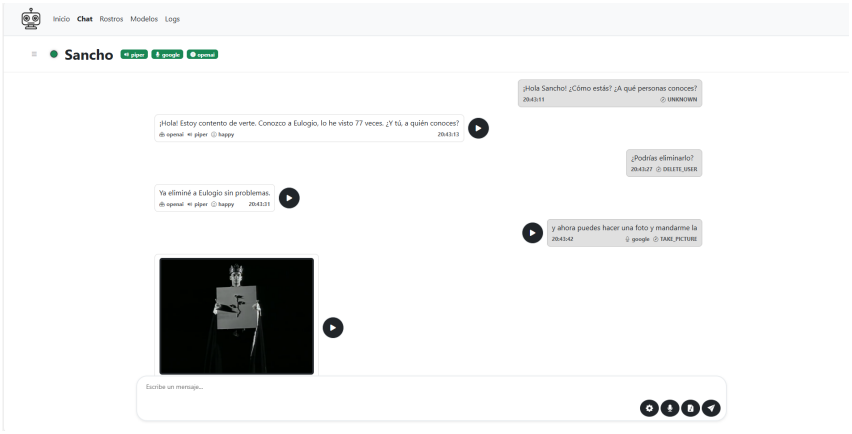


(a) Versión PC

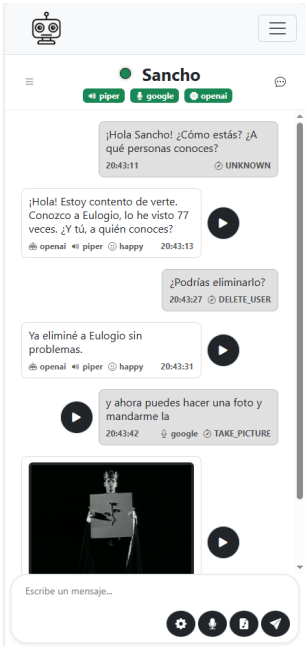
(b) Versión móvil

Figura 40: Conversación activa con el robot utilizando texto, audio y visión. [Fuente: propia.]

Activando el modo técnico en los ajustes, se accede a una vista extendida del chat (Figura 41) en la que se muestra información avanzada. En la cabecera se visualizan los modelos activos de TTS, STT y LLM (o si no hay ninguno activo). En cada mensaje del usuario se especifica la intención detectada (por ejemplo, **TAKE_PICTURE**) y, si procede, el modelo de transcripción utilizado. En los mensajes del robot se indica el modelo de LLM que generó la respuesta, el modelo TTS si hubo audio, y la emoción clasificada en el mensaje. Esta información resulta útil para depurar el comportamiento del sistema o analizar su funcionamiento en tiempo real.



(a) Versión PC



(b) Versión móvil

Figura 41: Vista técnica del chat con información extendida. [Fuente: propia.]

Además, al hacer clic sobre cualquier mensaje del usuario o del robot, se despliega una vista detallada con información adicional. En el caso de mensajes del usuario (Figura 42), se muestra el identificador del mensaje, los tiempos de transcripción (si aplica), y metadatos útiles para el seguimiento. En el caso de respuestas del robot (Figura 43), se incluyen los modelos usados, la emoción asociada, y cualquier contenido adicional generado (como imágenes o comandos ejecutados).

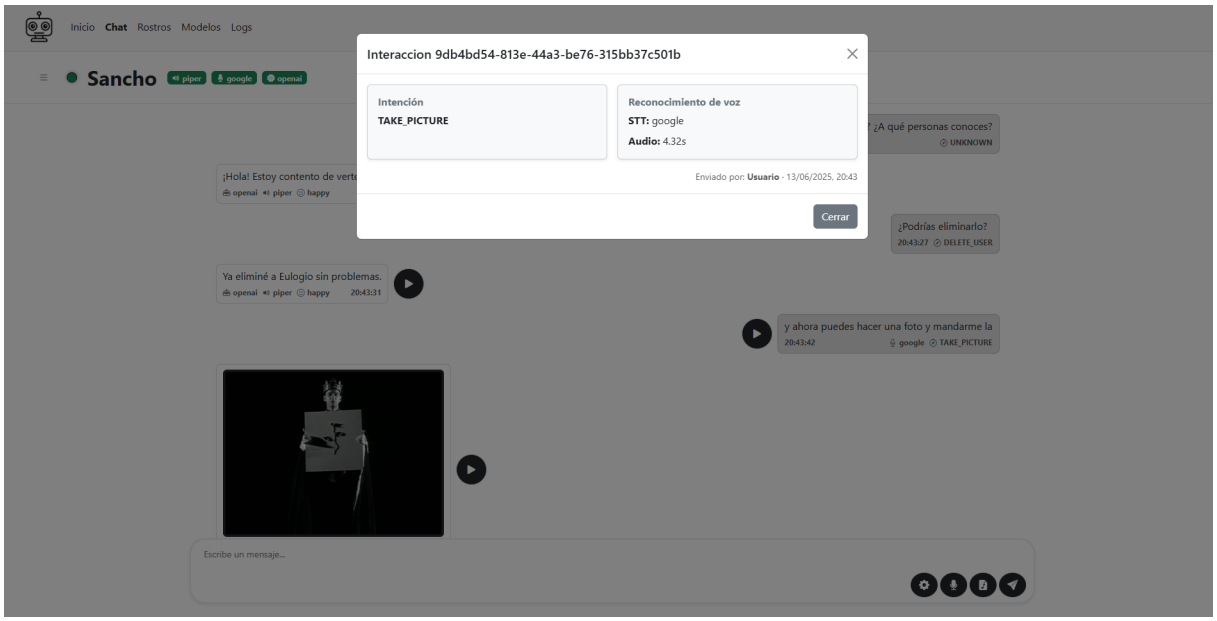


Figura 42: Detalles de un mensaje del usuario al hacer clic sobre él. [Fuente: propia.]

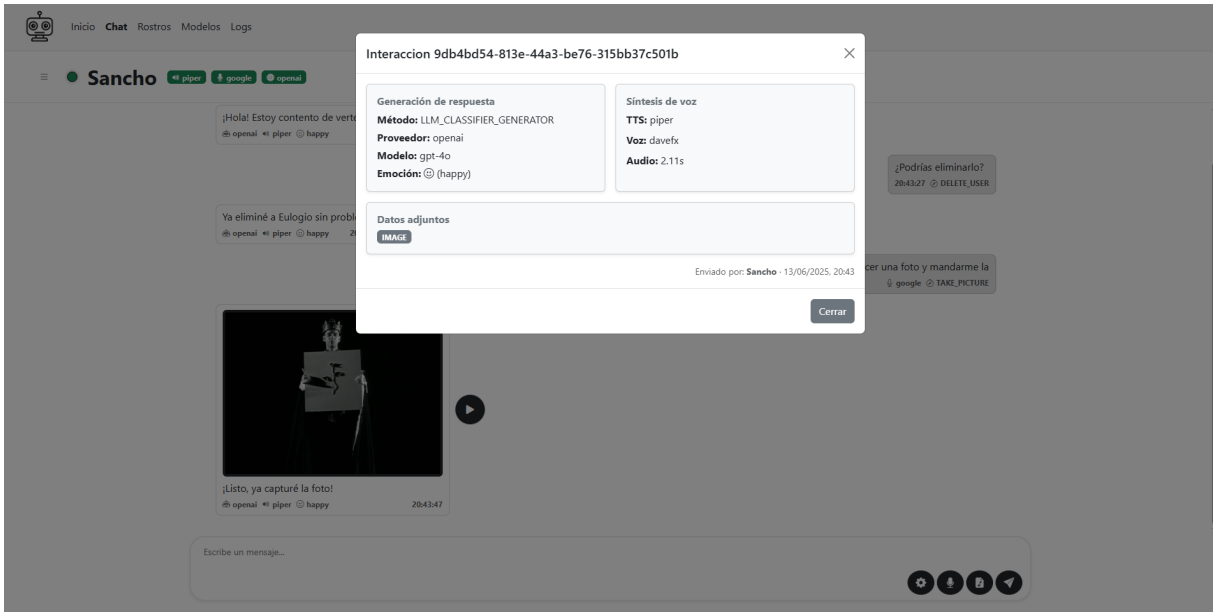


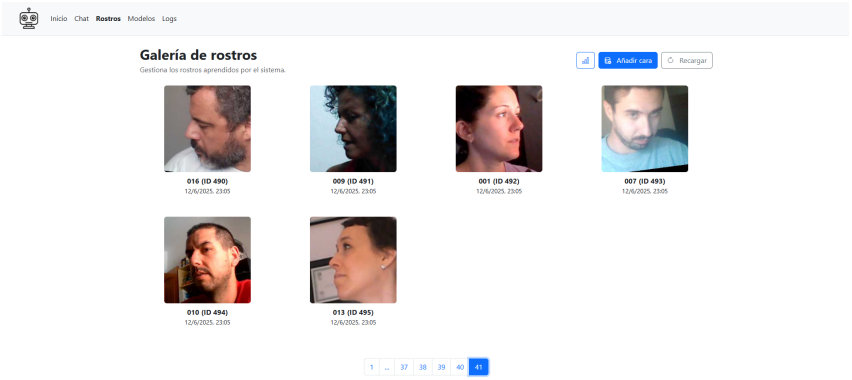
Figura 43: Detalles de una respuesta del robot con imagen y metadatos. [Fuente: propia.]

Rostros (RUMI)

Esta vista permite gestionar todos los rostros reconocidos por el sistema mediante una galería visual e interactiva. Está directamente conectada con el backend de ROS 2 y recibe actualizaciones en tiempo real gracias a los eventos emitidos por **ros2web**, aunque también se puede recargar manualmente. Todo el sistema de identidades faciales se basa en la herramienta

desarrollada denominada **RUMI**, encargada de registrar, almacenar y analizar la interacción de los usuarios con el robot.

En la Figura 44 se muestra la vista principal de la galería. Se trata de una lista paginada donde cada tarjeta muestra la imagen del rostro, el nombre asignado, su identificador y la fecha de registro. En la parte inferior puede observarse el control de paginación, mientras que en la parte superior se incluyen tres botones: uno para recargar manualmente la lista (útil si no está activado el sistema en tiempo real), otro para acceder al resumen estadístico de identidades, y un último botón para añadir una nueva cara al sistema.

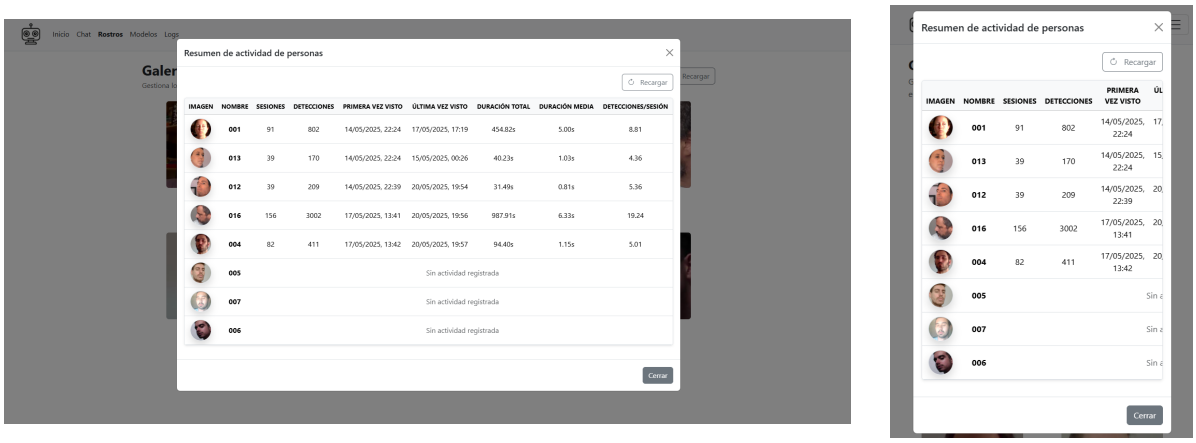


(a) Versión PC

(b) Versión móvil

Figura 44: Vista principal de la galería de rostros registrados. [Fuente: propia.]

Al pulsar el botón de resumen se accede a una vista como la de la Figura 45, donde se representa información detallada de cada identidad: número total de sesiones registradas, cantidad total de detecciones, fecha del primer y último encuentro, duración total visible, duración media por sesión y media de detecciones por sesión. Esta vista permite comprender la frecuencia y calidad de la interacción mantenida con cada persona.

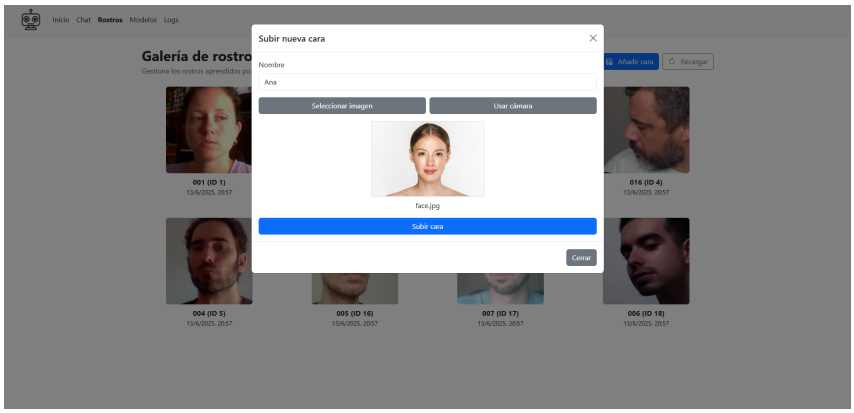


(a) Versión PC

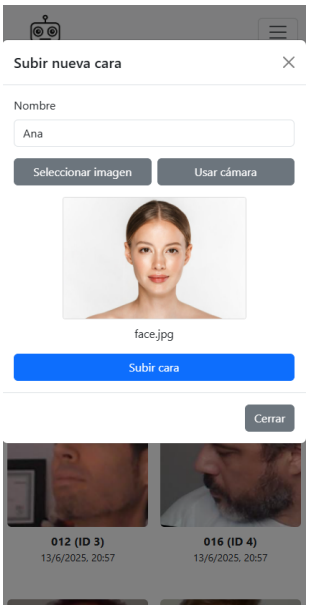
(b) Versión móvil

Figura 45: Resumen estadístico de interacción por rostro. [Fuente: propia.]

La Figura 46 muestra la vista que aparece al pulsar el botón de añadir una nueva cara. El proceso es muy sencillo: basta con introducir el nombre de la persona y seleccionar una imagen del dispositivo o capturarla directamente desde la cámara. Tras pulsar el botón de subir, el sistema ejecuta su *pipeline* de detección y reconocimiento. Dependiendo del resultado, puede que la imagen se acepte (rostro válido), se rechace (no se detecta rostro) o se advierta que la persona ya existe en la base de datos. Todos estos resultados se notifican al usuario mediante *toasts*.



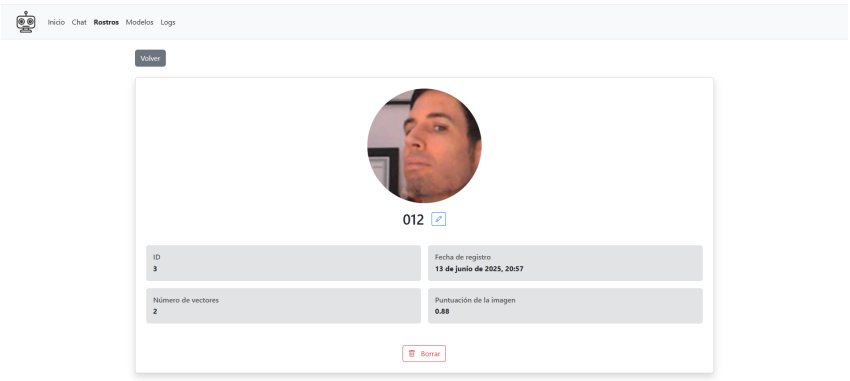
(a) Versión PC



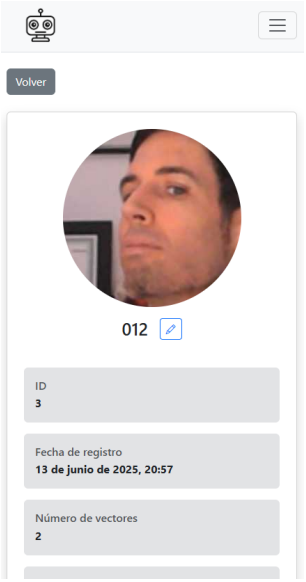
(b) Versión móvil

Figura 46: Formulario para añadir un nuevo rostro al sistema. [Fuente: propia.]

Una vez añadido un rostro, al hacer clic sobre él se accede a una vista detallada como la mostrada en la Figura 47. En ella se puede editar el nombre asignado, eliminar el rostro completamente, y consultar datos adicionales como el identificador interno, fecha de alta y estado actual. Esta información resulta útil para gestionar manualmente los registros o revisar los casos más relevantes.



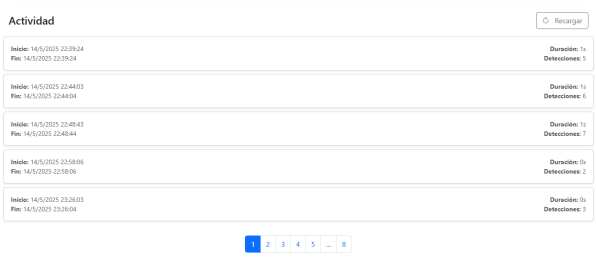
(a) Versión PC



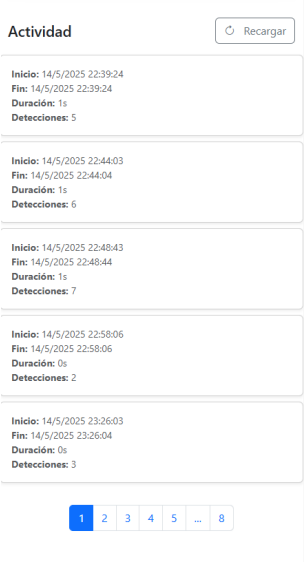
(b) Versión móvil

Figura 47: Vista detallada de un rostro con opciones de edición y eliminación. [Fuente: propia.]

Al descender en la misma página, como se muestra en la Figura 48, se accede a la lista de sesiones de interacción asociadas a esa persona. Esta lista está paginada e incluye un botón de recarga, junto con accesos directos para navegar entre páginas. Cada sesión incluye información resumida de duración, fecha y eventos capturados.



(a) Versión PC



(b) Versión móvil

Figura 48: Listado de sesiones de interacción asociadas al rostro. [Fuente: propia.]

Al hacer clic en cualquiera de las sesiones, se despliega un modal (Figura 49) que muestra información más detallada. Este modal incluye estadísticas generales, gráficas con la evolución temporal de la puntuación de detección y clasificación, así como métricas agregadas del rendimiento del sistema durante esa sesión concreta.

(a) Versión PC

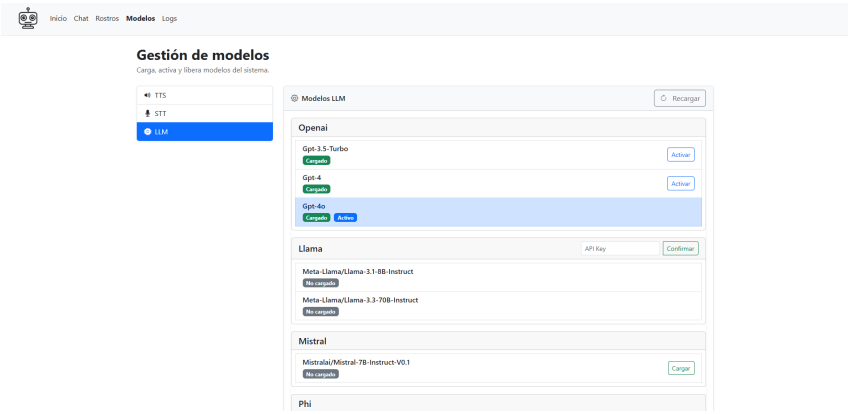
(b) Versión móvil

Figura 49: Modal con análisis detallado de una sesión de interacción. [Fuente: propia.]

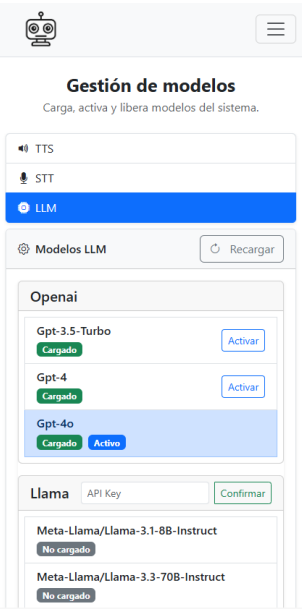
Modelos

Esta vista permite gestionar todos los modelos de inteligencia artificial utilizados por el sistema, incluyendo modelos de lenguaje (LLM), de transcripción (STT) y de síntesis de voz (TTS). Desde aquí es posible cargar, descargar, activar o cambiar configuraciones de cada uno de ellos, tanto si están disponibles localmente como si requieren conexión a servicios en la nube.

La Figura 50 muestra el panel dedicado a los modelos de lenguaje (LLM). En esta vista se listan los distintos proveedores disponibles (OpenAI, Gemini, Mistral, etc.), y dentro de cada proveedor se despliegan sus modelos correspondientes. En el caso de proveedores que funcionan por API (como OpenAI), basta con introducir la clave API una única vez para habilitar el acceso a todos sus modelos. Para los proveedores locales, se pueden cargar modelos de forma individual si no requieren clave, o introducir una API key si es necesaria. Esta estructura modular permite gestionar múltiples orígenes de modelos de forma unificada, clara y extensible.



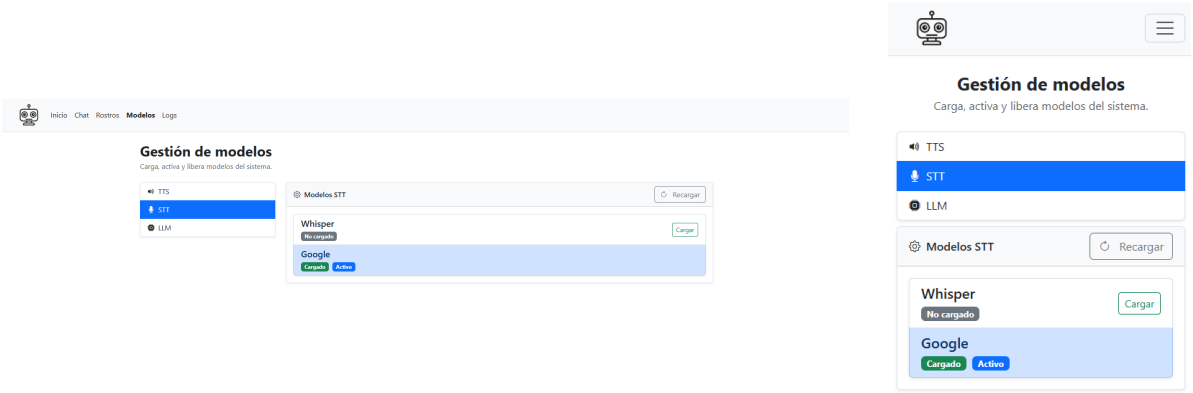
(a) Versión PC



(b) Versión móvil

Figura 50: Gestión de modelos de lenguaje. [Fuente: propia.]

En la Figura 51 se representa la interfaz de modelos de transcripción (STT). A diferencia de los LLM, en este caso los modelos se cargan directamente sin necesidad de autenticación adicional. La interfaz permite activar y desactivar los modelos disponibles de forma rápida y visual, indicando el estado actual de cada uno mediante iconos intuitivos.

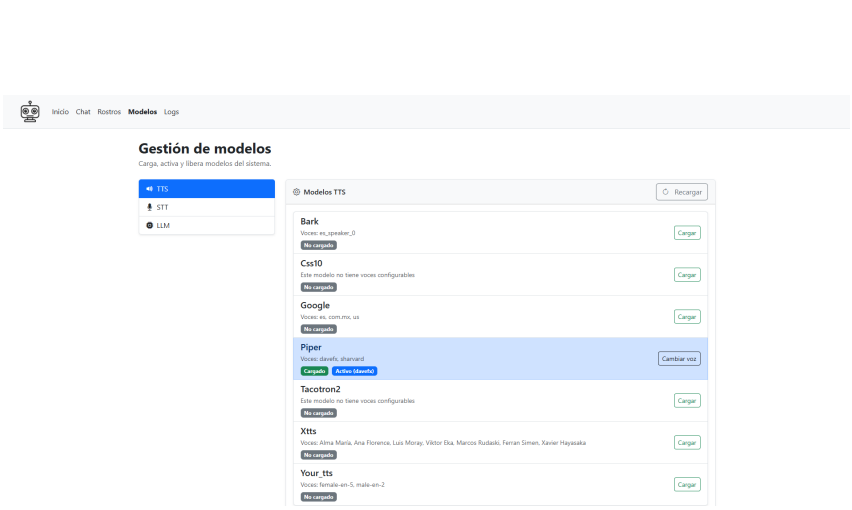


(a) Versión PC

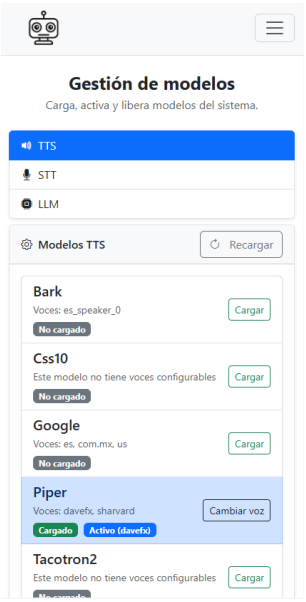
(b) Versión móvil

Figura 51: Gestión de modelos de transcripción. [Fuente: propia.]

Por último, la Figura 52 muestra el panel de modelos de síntesis de voz (TTS). Al igual que en los modelos de STT, la carga y activación de los modelos es directa. Sin embargo, algunos modelos TTS ofrecen múltiples voces disponibles. En ese caso, una vez cargado el modelo, es posible pulsar sobre el botón de selección de voz, lo que abre un modal específico como se muestra en la Figura 53, donde se puede elegir fácilmente entre distintas voces compatibles con el modelo activo.



(a) Versión PC



(b) Versión móvil

Figura 52: Gestión de modelos de síntesis de voz. [Fuente: propia.]

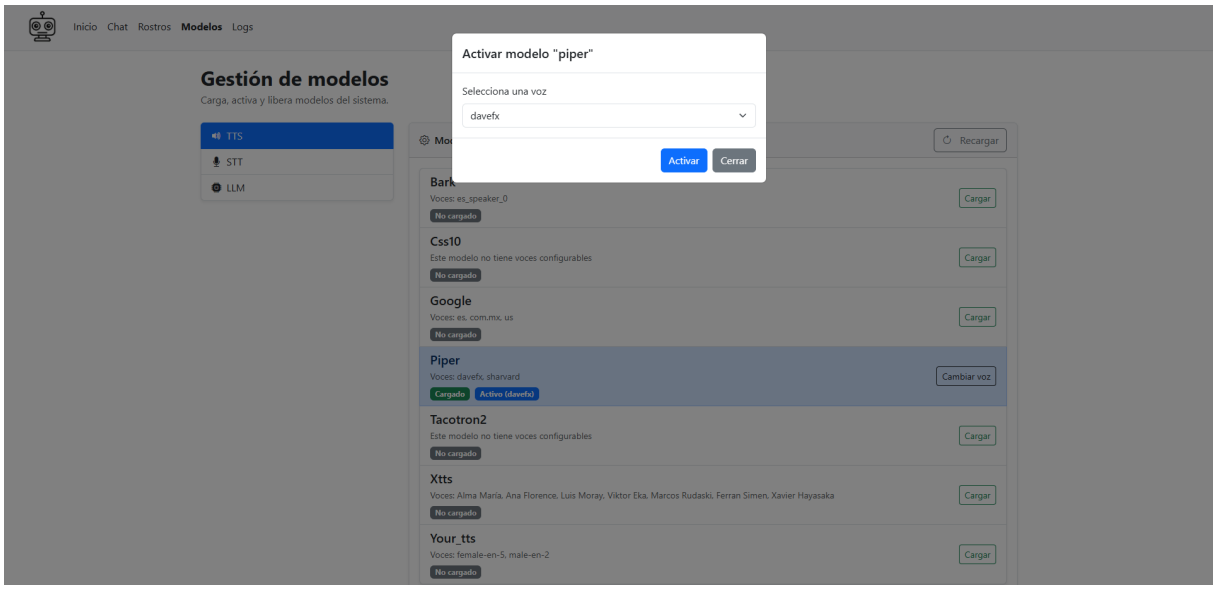


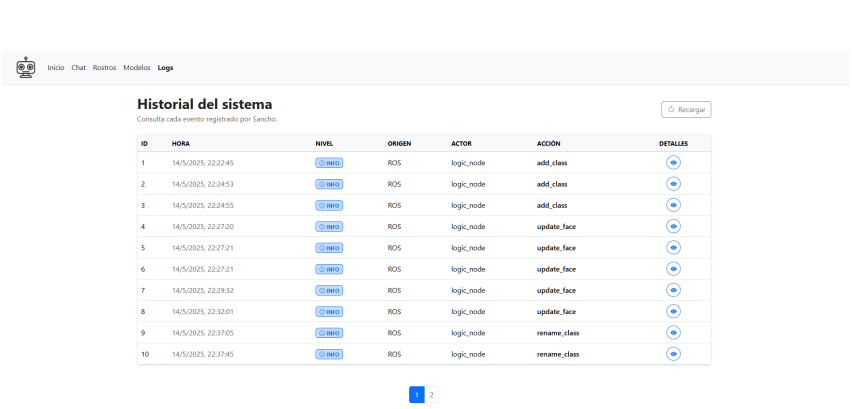
Figura 53: Modal para seleccionar entre múltiples voces disponibles en modelos TTS. [Fuente: propia.]

Logs

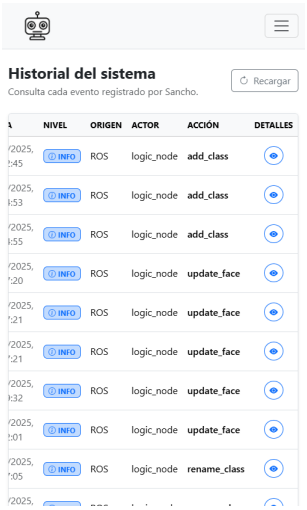
La interfaz web incluye una sección dedicada a la visualización de registros del sistema. Esta funcionalidad es especialmente útil durante el desarrollo, validación o diagnóstico del

comportamiento del robot, ya que permite inspeccionar en tiempo real los eventos más relevantes que se producen internamente.

En la Figura 54 se muestra el visor principal de logs. Este componente muestra mensajes generados por los distintos nodos del sistema, incluyendo respuestas del robot, cambios de modelo, errores, advertencias, eventos relacionados con los usuarios, y transcripciones de voz. La vista incluye filtros de búsqueda y segmentación temporal, lo que facilita la depuración y el análisis.



(a) Versión PC



(b) Versión móvil

Figura 54: Visor en tiempo real de registros del sistema. [Fuente: propia.]

Adicionalmente, como se observa en la Figura 55, es posible acceder a detalles técnicos de cada *log*, visualizando información estructurada como el tipo de evento, su contenido, el nodo ROS que lo ha emitido y cualquier dato complementario útil. Esta vista estructurada facilita el análisis del comportamiento del sistema y permite identificar de forma rápida posibles fallos, comportamientos inesperados o métricas clave durante las interacciones.

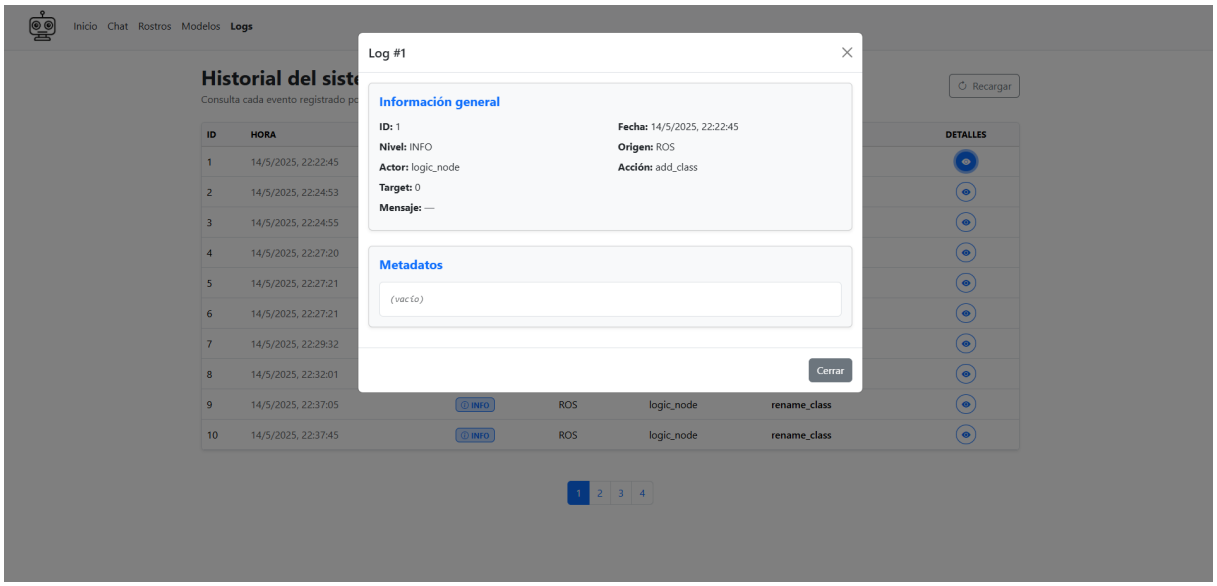


Figura 55: Vista detallada de un *log* individual con metadatos técnicos. [Fuente: propia.]

4.9.6. Elementos complementarios de interacción visual

Además de las páginas principales descritas anteriormente, la interfaz incluye varios elementos visuales complementarios que enriquecen la experiencia del usuario y mejoran la interacción en tiempo real con el sistema.

Ventana flotante de cámara Uno de estos elementos es la **ventana flotante de cámara** (Figura 56), que permite visualizar en todo momento lo que está captando la cámara del robot. Esta vista es especialmente útil en contextos de supervisión remota, ya que ofrece una referencia visual inmediata del entorno físico en el que se encuentra el robot. La ventana puede moverse libremente por la interfaz, minimizarse o cerrarse según las necesidades del usuario, manteniendo siempre su funcionalidad sin interferir con el resto de componentes de la aplicación.

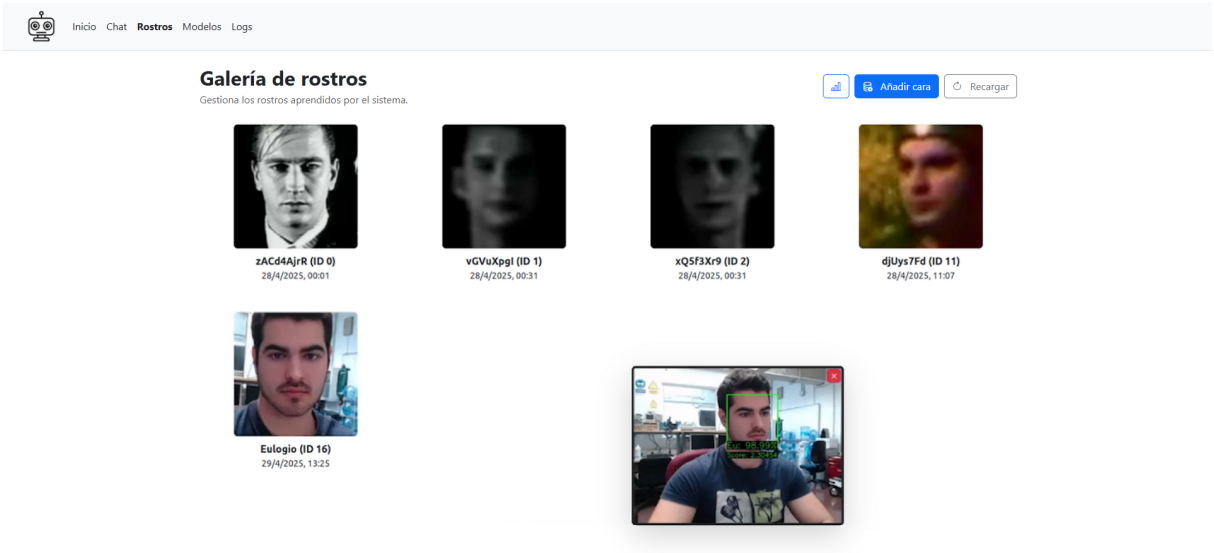
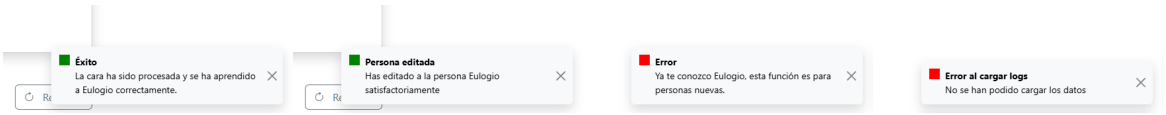


Figura 56: Ventana flotante con vista de cámara en tiempo real. [Fuente: propia.]

Toasts informativos

Otro elemento destacado son los ***toasts*** o mensajes emergentes (Figura 57), diseñados para proporcionar retroalimentación visual rápida y no intrusiva al usuario. Estos mensajes aparecen en la parte inferior derecha de la pantalla cada vez que ocurre una acción relevante: adición o edición de rostros, errores de red, o notificaciones del sistema.

Los *toasts* son acumulativos (pueden mostrarse varios a la vez), desaparecen automáticamente a los cinco segundos y pueden ser cerrados manualmente por el usuario si lo desea. Esta solución, además de elegante, permite mantener informado al usuario en todo momento sin interrumpir su flujo de trabajo.



(a) Rostro añadido (b) Identidad editada (c) Fallo en la detección (d) Error de carga

Figura 57: Ejemplos de *toasts* informativos mostrados en la interfaz web. [Fuente: propia.]

4.9.7. Responsividad, usabilidad y accesibilidad

Durante el desarrollo de la interfaz web se ha prestado especial atención a los principios de **responsividad**, **usabilidad** y **accesibilidad**, con el objetivo de proporcionar una experiencia

fluida, intuitiva y universalmente accesible en cualquier dispositivo y contexto de uso.

En cuanto a la **responsividad**, toda la interfaz ha sido diseñada para adaptarse automáticamente a distintos tamaños de pantalla, resoluciones y dispositivos (PC, tablets, móviles). Para ello, se ha utilizado el sistema de *breakpoints* de **Bootstrap**, que permite reorganizar dinámicamente los elementos en función del ancho disponible. Este enfoque ha facilitado la creación de una interfaz flexible y adaptativa, manteniendo la legibilidad y funcionalidad en todos los contextos. Además, se han verificado visualmente las vistas móviles y de escritorio, como se muestra en las figuras correspondientes, para asegurar una correcta presentación y experiencia de uso en ambos formatos.

Respecto a la **usabilidad**, se han seguido los principios de diseño centrado en el usuario, guiándose por las *heurísticas de usabilidad* propuestas por Jakob Nielsen (Nielsen, 1994). Se ha buscado que la interfaz sea coherente, predecible y clara, ofreciendo retroalimentación inmediata al usuario (por ejemplo, con mensajes emergentes o indicadores de carga), minimizando la carga cognitiva y facilitando el aprendizaje progresivo del sistema. Aunque no se ha realizado un estudio formal de usabilidad con usuarios externos, las decisiones de diseño se han basado en buenas prácticas ampliamente validadas en la literatura.

En cuanto a la **accesibilidad**, se ha intentado maximizar la compatibilidad con herramientas de asistencia y garantizar el acceso a personas con distintos perfiles. Para ello, se han utilizado colores con buen contraste para facilitar la lectura, se ha procurado que los elementos interactivos sean navegables mediante teclado (tabulables), y se han definido etiquetas semánticas adecuadas cuando ha sido posible. Si bien no se han llevado a cabo auditorías específicas de accesibilidad, el desarrollo ha tratado de alinearse con las recomendaciones generales establecidas por iniciativas como *WCAG 2.1*.

En conjunto, la interfaz web busca ser no solo funcional y estética, sino también inclusiva y accesible, promoviendo una experiencia coherente y satisfactoria para cualquier tipo de usuario.

4.10. Interfaz de escritorio

La interfaz gráfica local del sistema, desarrollada en **PyQt6**, cumple un papel fundamental en la interacción social directa del robot con el usuario. Como se expuso en la fase de diseño (véase Capítulo 3), esta interfaz está pensada para ejecutarse en la pantalla táctil del robot

Sancho, y está organizada siguiendo una arquitectura **MVC** que facilita la separación de lógica, presentación y control de eventos.

Está completamente sincronizada con los nodos ROS 2 del sistema, permitiendo mostrar información contextual, recibir entradas del usuario y participar activamente en procesos como el reconocimiento facial o la gestión de identidades. Uno de sus mayores valores es que las pantallas se actualizan en tiempo real según el estado de la conversación, ofreciendo al usuario un *feedback visual directo* mientras interactúa verbalmente con el robot. Esta coherencia multimodal mejora la comprensión, genera confianza y refuerza el vínculo comunicativo.

El sistema define cuatro pantallas principales. A continuación, explicamos como manual de usuario cada una de estas pantallas, asociada a un estado concreto del flujo de interacción:

1. **Pantalla de bienvenida:** Se muestra por defecto cuando no hay ningún usuario presente o identificado. Presenta un mensaje simple como *"¡Hola! Soy Sancho"*, indicando que el robot está activo y listo para comenzar una nueva interacción. Esta pantalla genera presencia y favorece la espontaneidad del encuentro humano-robot (véase Figura 58).

Figura 58: Pantalla de bienvenida mostrada cuando el sistema está en reposo. [Fuente: propia.]

2. **Pantalla de introducción de nombre:** Aparece cuando el sistema detecta un rostro desconocido y lanza una petición para registrar una nueva identidad. Se muestra la ima-

gen capturada del usuario, una caja de texto y un botón de envío, lo que facilita el registro inmediato desde la pantalla táctil del robot. Esta pantalla aporta un componente didáctico y participativo a la interacción (véase Figura 59).

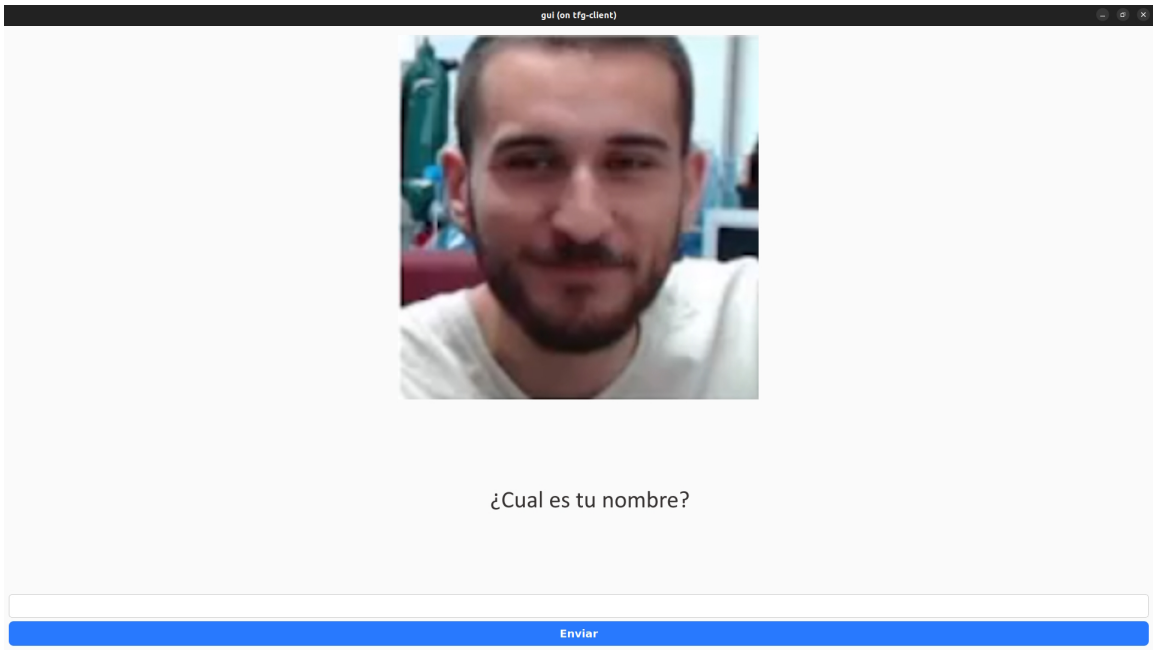


Figura 59: Pantalla de registro de nombre para un nuevo usuario detectado. [Fuente: propia.]

3. **Pantalla de confirmación de identidad:** Se utiliza cuando el sistema reconoce un rostro con una confianza baja o intermedia. En ella se presenta la imagen del usuario junto con el nombre estimado, y se le pregunta si la identificación es correcta mediante dos botones (*"Sí"* y *"No"*). Esta verificación visual aporta robustez al reconocimiento y genera una experiencia participativa (véase Figura 60).

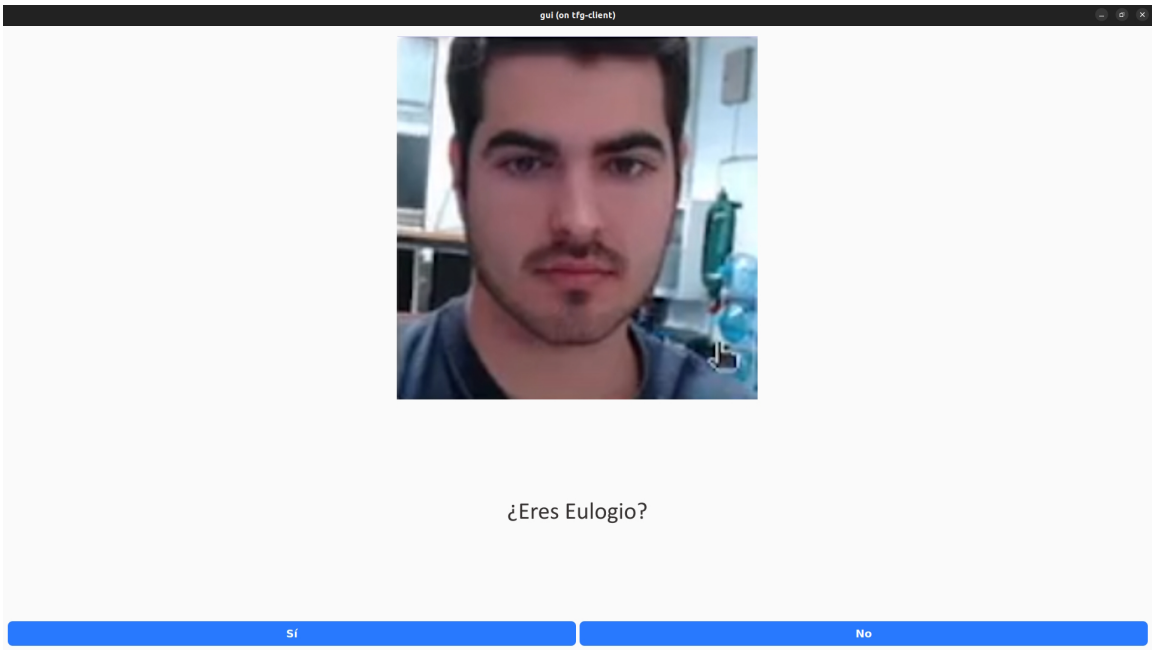


Figura 60: Pantalla para confirmar si la identidad estimada es correcta. [Fuente: propia.]

4. **Pantalla de visualización de imagen:** Permite mostrar temporalmente una fotografía capturada por el robot, por ejemplo tras una detección o como parte de una interacción didáctica. Aunque actualmente se usa principalmente como confirmación visual, en futuras versiones puede utilizarse para enseñar objetos, mostrar resultados o representar emociones mediante imágenes (véase Figura 61).

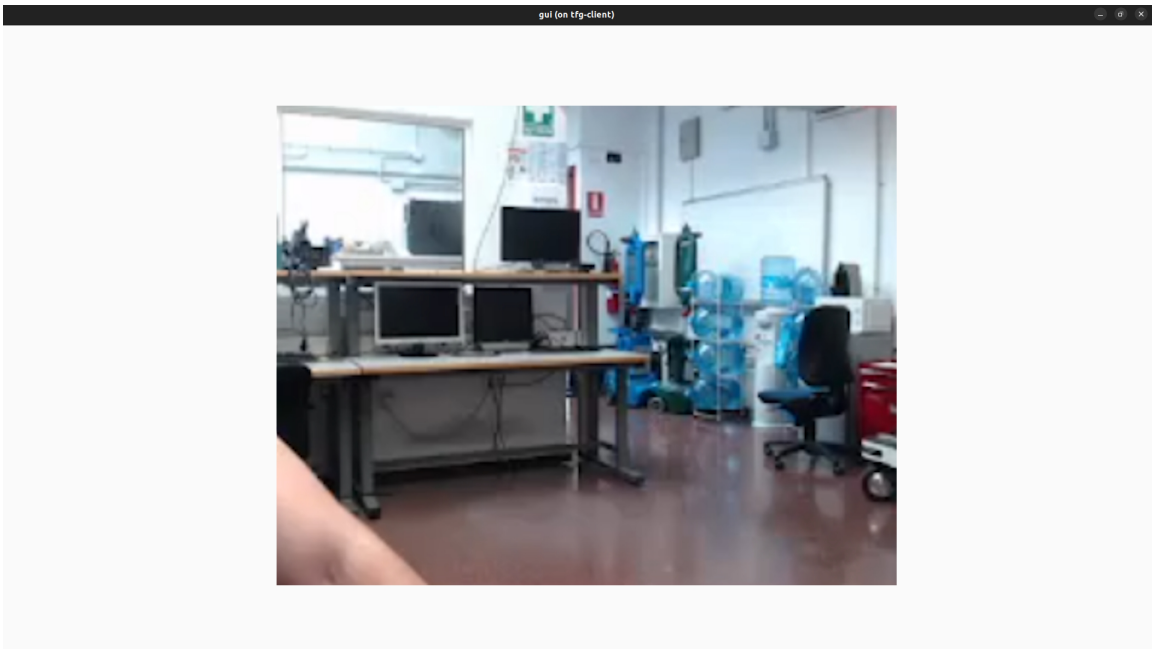


Figura 61: Pantalla de visualización temporal de imagen capturada. [Fuente: propia.]

4.11. Modos de ejecución del sistema

El sistema ha sido diseñado con el objetivo de adaptarse a distintos entornos operativos y capacidades de hardware, permitiendo una ejecución flexible y escalable. En función del contexto, se contemplan dos modos principales de despliegue:

- **Ejecución local con apoyo en la nube:** es el escenario más habitual cuando se implementa el sistema en un robot autónomo con recursos limitados, como una Raspberry Pi. En este modo, el procesamiento local se limita a tareas ligeras como la detección facial (mediante detectores como **cv2** o **dlib frontal**), el reconocimiento facial con codificadores como **FaceNet**, y la síntesis de voz con modelos como **Piper**. Para otras tareas computacionalmente más exigentes —como la transcripción de voz (STT) o la generación de lenguaje natural (LLM)— se emplean servicios en la nube, tales como la API de **Google Speech** o los modelos de **OpenAI** y **Gemini**. Este modo ofrece una buena relación entre autonomía y rendimiento sin requerir infraestructura adicional.

- **Ejecución distribuida con servidor de computación propio:** en entornos con acceso a un servidor potente, ya sea en red local o en la nube privada del usuario, el sistema puede aprovechar dicho servidor para realizar las tareas más costosas. En este caso se

pueden utilizar modelos avanzados como **YOLOv5** para detección de rostros, **FaceNet** o **ArcFace** como codificadores de alta precisión, **Whisper** para STT, **XTTS** para síntesis de voz y cualquier modelo LLM compatible con los módulos del sistema, ejecutado localmente en el servidor. Esta arquitectura distribuida mantiene al robot ligero mientras delega las tareas pesadas de forma transparente y eficiente.

Ambos modos comparten una arquitectura modular común, basada en ROS 2 y servicios desacoplados, que permite cambiar los modelos o el punto de ejecución de cada componente (TTS, STT, LLM, reconocimiento facial, etc.) sin modificar el resto del sistema. Esta flexibilidad facilita tanto el desarrollo como el mantenimiento y la adaptación a nuevas condiciones operativas.

5

Validación y pruebas

Este capítulo presenta el proceso de validación del sistema desarrollado, con el objetivo de demostrar su correcto funcionamiento, robustez y adecuación a los requisitos definidos. La validación se ha llevado a cabo mediante una combinación de pruebas sistemáticas, evaluaciones funcionales y verificación empírica de los resultados obtenidos.

Se describen en primer lugar las pruebas unitarias aplicadas a los componentes críticos del sistema, seguidas por una batería de tests de integración que validan la interoperabilidad entre módulos a través de la API REST. A continuación, se presentan los tests de carga realizados con usuarios virtuales, que permiten analizar el rendimiento del sistema bajo distintas condiciones de estrés.

Además de las pruebas estructuradas, se han llevado a cabo evaluaciones específicas sobre los modelos de inteligencia artificial integrados en el sistema, incluyendo modelos de visión, de lenguaje y de *embeddings* semánticos, para distintas tareas que son cruciales en este trabajo. Estas evaluaciones permiten cuantificar el rendimiento, comparar alternativas y justificar las elecciones adoptadas.

Por último, se documenta visualmente el resultado final del sistema, incluyendo capturas de pantalla e imágenes del robot en funcionamiento, mostrando su capacidad de interacción multimodal en un entorno realista.

5.1. Pruebas del *software*

La validación del sistema ha comenzado por la base: asegurar la fiabilidad y corrección del código desarrollado. Para ello, se ha implementado una estrategia de pruebas estructurada en distintos niveles, comenzando con pruebas unitarias sobre funciones y módulos individuales, y continuando con tests de integración centrados en la interacción entre componentes y servicios clave del sistema.

Estas pruebas permiten verificar el cumplimiento de los requisitos funcionales, prevenir regresiones durante el desarrollo iterativo y garantizar la interoperabilidad entre las distintas capas de la arquitectura. Además, se han ejecutado tests de carga para evaluar el comportamiento del sistema bajo distintas condiciones de concurrencia y estrés, especialmente en la interfaz web y los servicios expuestos a través de la API REST.

Este apartado recoge de forma detallada el diseño, alcance y resultados de cada uno de estos niveles de prueba, proporcionando evidencia concreta del correcto funcionamiento del sistema desde el punto de vista del *software*.

5.1.1. Tests unitarios

Las pruebas unitarias constituyen un pilar fundamental en la validación del software, ya que permiten comprobar el comportamiento correcto de componentes individuales del sistema en condiciones controladas. Su objetivo es verificar que cada función, clase o módulo actúe conforme a lo esperado, reduciendo así la probabilidad de errores en fases posteriores de integración.

En este proyecto se ha implementado un conjunto representativo de tests unitarios sobre los paquetes más críticos y susceptibles de fallo. Aunque alcanzar una cobertura total sería el objetivo ideal, se ha optado por una estrategia pragmática: focalizar los esfuerzos en aquellas funciones que presentan lógica relevante, ramificaciones complejas o alta reutilización. Esta decisión responde tanto a criterios de impacto como a limitaciones razonables de tiempo.

Se han aplicado diferentes estrategias de prueba, incluyendo tests de **caja blanca** (basados en el conocimiento del código interno), **caja negra** (centrados en el comportamiento externo esperado) y pruebas con **mocking**, que permiten simular componentes externos o dependencias complejas. En todos los casos se ha seguido el patrón *Arrange–Act–Assert (AAA)* para estructurar las pruebas, una buena práctica ampliamente reconocida en la ingeniería del software (Osherove, 2014).

En total, se han definido y ejecutado **52 *tests* unitarios**, distribuidos entre los distintos paquetes del sistema, tal como se resume en la Tabla 14. Todos ellos han sido superados correctamente, lo que demuestra la robustez del diseño modular y la fiabilidad de los componentes más relevantes.

Paquete	Nº tests	Tipo de pruebas	Técnicas empleadas
hri_audio	10	Caja blanca	Mocking, acceso interno
hri_vision	7	Caja negra	Validación externa
ros2web	10	Mixto	Mocking, asincronía
rumi_web	12	Caja blanca/negra	Mocking, simulación temporal
sancho_ai	2	Caja negra	Mocking del pipeline
sancho_web	11	Caja negra	Validación de errores
Total	**52**		

Tabla 14: Resumen de tests unitarios por paquete

hri_audio: Se ha testeado exhaustivamente la clase **STTHotword**, encargada de detectar la palabra de activación a partir del audio. Los tests cubren tanto la normalización del texto como el funcionamiento interno del buffer, el control temporal y la detección del nombre objetivo. Se han utilizado técnicas de *mocking* para simular la función de transcripción STT, permitiendo comprobar el comportamiento del sistema sin depender de modelos externos. La mayoría de pruebas son de caja blanca, accediendo a métodos privados y estados internos.

hri_vision: Se han desarrollado tests para el módulo **FaceprintsDatabase**, que gestiona el almacenamiento en memoria de vectores faciales. Se han comprobado operaciones de inserción, actualización, eliminación y consulta, incluyendo filtros y consistencia estructural de los datos. Estos tests son de caja negra y validan la lógica esperada de la base de datos sin acceder a detalles internos de implementación.

ros2web: Este paquete incluye pruebas para dos componentes clave. Por un lado, **ChunkManager**, cuya lógica de fragmentación y reconstrucción de mensajes ha sido verificada con escenarios básicos, orden aleatorio y múltiples mensajes simultáneos. Por otro lado, el servidor **WebSocketServer** ha sido evaluado con pruebas asíncronas que simulan conexiones reales usando clases dummy, permitiendo validar flujos completos de conexión, envío y desconexión. Se combinan aquí técnicas de caja blanca y mocking, especialmente relevantes para manejar concurrencia y asincronía.

rumi_web: Se han testado tanto la lógica del gestor de sesiones **SessionManager**, con fo-

co en control temporal, detecciones filtradas y cierres automáticos, como el módulo **SessionsDatabase**, que interactúa con una base de datos SQLite. En ambos casos se han empleado mocks del sistema de tiempo para validar condiciones bajo distintas simulaciones temporales. Los tests combinan mocking, caja blanca (control del buffer y estructuras internas) y caja negra (consultas a base de datos).

sancho_ai: Se ha testeado la clase **ModularAI**, que representa el pipeline completo de razonamiento del sistema. Se han validado dos flujos diferenciados: uno con intención reconocida y ejecución completa, y otro con intención desconocida, donde se delega en la generación de respuesta. Los componentes del pipeline han sido simulados mediante mocks, permitiendo verificar llamadas esperadas y salidas correctas sin necesidad de modelos reales. Se trata de pruebas de caja negra con fuerte apoyo en mocking.

sancho_web: Finalmente, se ha verificado la lógica del módulo **SystemDatabase**, encargado de registrar logs del sistema. Los tests comprueban la creación, consulta y validación de logs, incluyendo el tratamiento de errores cuando se introducen niveles, orígenes o acciones no válidas. Estas pruebas siguen una estrategia de caja negra y validan el correcto comportamiento de la lógica de negocio y restricciones impuestas por el sistema.

En conjunto, esta batería de tests aporta una base sólida para garantizar la estabilidad del sistema y facilitar su mantenimiento a futuro. Si bien el número total de tests podría ampliarse, el enfoque adoptado ha permitido cubrir con garantías los módulos más críticos del sistema, demostrando su correcto funcionamiento en múltiples escenarios controlados.

Todos los tests unitarios se han ubicado siguiendo la convención habitual de ROS 2, dentro de una carpeta **test** específica en cada paquete correspondiente. Su ejecución es directa mediante la herramienta **pytest**, bastando con ejecutar el siguiente comando **pytest nombre_archivo_test.py** desde la raíz del paquete para comprobar el resultado de cada conjunto de pruebas.

5.1.2. Tests de integración

Las pruebas de integración tienen como objetivo verificar que los distintos componentes del sistema funcionan correctamente en conjunto. A diferencia de los tests unitarios, que se centran en módulos aislados, estas pruebas validan el comportamiento del sistema desde una

perspectiva más global, asegurando que las interacciones entre los elementos del backend, la lógica de negocio y la base de datos producen los resultados esperados.

En este caso, se ha optado por centrar las pruebas de integración en la API REST expuesta por el endpoint **/api/v1/faceprints**, ya que constituye uno de los puntos más críticos del sistema. Esta API es responsable de operaciones como la creación, consulta, actualización y eliminación de registros biométricos, además de la validación de imágenes y la prevención de duplicados. Dado que el resto de APIs siguen una estructura análoga pero gestionan entidades más simples (como logs, sesiones o modelos), su cobertura implícita se considera suficientemente garantizada al validar en profundidad esta interfaz.

Las pruebas se han implementado en un único fichero utilizando el framework **pytest** y se ejecutan directamente contra un servidor local en ejecución. Cada test comienza con una limpieza de los datos existentes mediante una *fixture* que borra todos los rostros registrados antes de cada ejecución, garantizando así un entorno limpio y determinista. En total se han desarrollado **13 tests de integración**, todos ellos superados con éxito. La Tabla 15 resume los principales casos validados.

Test	Descripción
CRUD completo	Crear, obtener, actualizar y eliminar un faceprint
Obtener por ID	Recuperación correcta por identificador
Listado completo	Listar todos los registros disponibles
Errores 404	Consulta de ID inexistente
Error 422	Actualización sin campos requeridos
Eliminación inválida	Eliminar un ID inexistente
Imagen sin rostro	Rechazo de imágenes sin caras detectadas
Imagen con varias caras	Rechazo de imágenes con más de un rostro
Imagen duplicada	Detección de faceprint previamente registrado
Actualización exitosa	Modificación del nombre de un registro
Comprobación de unicidad	Validación de que no se repiten registros
Validación de mensaje de error	Verificación del contenido de los errores
Limpieza previa (fixture)	Reseteo de base de datos antes de cada prueba

Tabla 15: Resumen de tests de integración sobre la API de faceprints

Entre los aspectos más relevantes verificados en estas pruebas, destacan:

- **Flujo completo CRUD**: se valida el ciclo completo de operaciones sobre un faceprint (crear, leer, actualizar y eliminar), asegurando que los datos persistidos se reflejan correctamente y que las operaciones intermedias modifican el estado de forma consistente.
- **Comprobación de errores y casos límite**: se prueban múltiples situaciones excepcionales, como intentar acceder a un ID inexistente, actualizar sin proporcionar un nombre, o eliminar entradas no registradas. Estos casos devuelven los códigos HTTP esperados (**400**, **404**, **422**) y mensajes de error significativos en el cuerpo de la respuesta.
- **Validación del reconocimiento facial**: la API ha sido diseñada para aceptar únicamente imágenes con un único rostro válido. Por ello, se incluyen pruebas con imágenes que contienen múltiples caras, ninguna, o bien una ya registrada previamente. En cada caso, el sistema responde con mensajes claros, impidiendo registros duplicados o inválidos.
- **Gestión de duplicados**: se verifica que el sistema detecta correctamente si se intenta registrar una imagen ya conocida, devolviendo un error coherente del tipo *"ya te conozco"*, lo que garantiza la unicidad de los faceprints almacenados.

Estas pruebas refuerzan la fiabilidad del sistema y confirman que la API principal se comporta de forma robusta ante escenarios realistas y variados. Además, sientan las bases para una futura ampliación del sistema de tests hacia otras rutas de la API REST y módulos externos, asegurando una integración sin fisuras.

Al igual que en las pruebas unitarias, la ejecución de estos tests puede realizarse mediante el comando **pytest nombre_archivo_test.py**, y se recomienda integrarlos en el flujo de desarrollo continuo. El archivo correspondiente se ha ubicado en la ruta adecuada dentro de la carpeta **test** del paquete **sancho_web**.

5.1.3. Tests de carga

Las pruebas de carga son una técnica esencial en la validación de sistemas web, ya que permiten medir el comportamiento del sistema bajo distintos niveles de demanda concurrente.

Su finalidad es detectar cuellos de botella, degradaciones de rendimiento o errores cuando múltiples usuarios acceden simultáneamente al servicio.

Para realizar estas pruebas se ha utilizado la herramienta **K6**, una plataforma moderna de pruebas de carga open-source orientada a desarrolladores. K6 permite definir los escenarios de prueba mediante scripts escritos en JavaScript, lo que facilita su integración en flujos de trabajo automatizados y entornos CI/CD. Además, sus resultados pueden visualizarse en tiempo real a través del panel de **Grafana**, gracias a su integración nativa mediante el módulo **k6/xk6-output-grafana-cloud**. Esto proporciona una monitorización rica en métricas como latencias, tasas de error y uso de recursos.

El sistema evaluado corresponde a la API REST implementada en el paquete **sancho_web**, focalizando los tests en el endpoint **/api/v1/faceprints**, por ser el más demandante en términos de consultas, acceso a base de datos y volumen de datos transferidos. Los demás endpoints comparten la misma infraestructura de red y servidor, por lo que sus tiempos de respuesta se consideran equivalentes o inferiores.

Los tests se han ejecutado sobre el equipo **uedge**, cuyas características técnicas se detallan en la Tabla 7.

Smoke

El test *smoke* sirve como verificación preliminar del correcto funcionamiento del endpoint bajo condiciones controladas. Se ejecutó con 5 usuarios virtuales durante 1 minuto, simulando un uso normal y sin estrés para el sistema.

Durante la prueba se completaron con éxito 7 398 peticiones, todas con código de estado 200 y sin errores registrados. La latencia media fue de tan solo 40.47 ms, cumpliendo ampliamente los umbrales establecidos (menos de 200 ms de media y 0 % de fallos). Este resultado confirma que el servicio responde de forma fiable y rápida bajo carga ligera, y valida la base para ejecutar el resto de pruebas más exigentes.

Breakpoint

El objetivo del test de breakpoint es identificar el número máximo de usuarios virtuales (VUs) que el sistema puede manejar simultáneamente antes de que su rendimiento se degrade de forma significativa. Para ello, se empleó un escenario con incremento progresivo de la tasa de llegada de usuarios durante un total de 10 minutos, utilizando el ejecutor **ramping-arrival-**

rate. En este tipo de prueba no se fija un número de usuarios de antemano, sino que la carga aumenta gradualmente hasta que se alcanza el límite operativo del sistema.

Durante la prueba, el sistema fue capaz de escalar hasta un total de **28 993 usuarios virtuales** activos, momento en el cual comenzó a registrar una tasa de fallos muy elevada (80.66 %). De las 2 441 peticiones realizadas, 1 969 fallaron, y más de 77 000 iteraciones fueron directamente descartadas antes de poder ser atendidas. Además, las latencias se dispararon: aunque el promedio global fue de 6.1 segundos, las respuestas exitosas (las pocas que lo fueron) alcanzaron tiempos medios de **31.58 segundos**, con picos de hasta 32.71 segundos.

La Figura 62 muestra un resumen visual del comportamiento del sistema durante el test de breakpoint. En ella se puede observar la evolución de métricas clave como la tasa de peticiones HTTP, la duración de las respuestas, el número de usuarios virtuales activos, el volumen de datos transferidos y la tasa de errores. Esta visualización confirma que el sistema colapsa alrededor del umbral de los 29 000 usuarios concurrentes, lo que justifica su uso como referencia para definir los niveles de carga relativos en los tests posteriores (spike, average y stress).

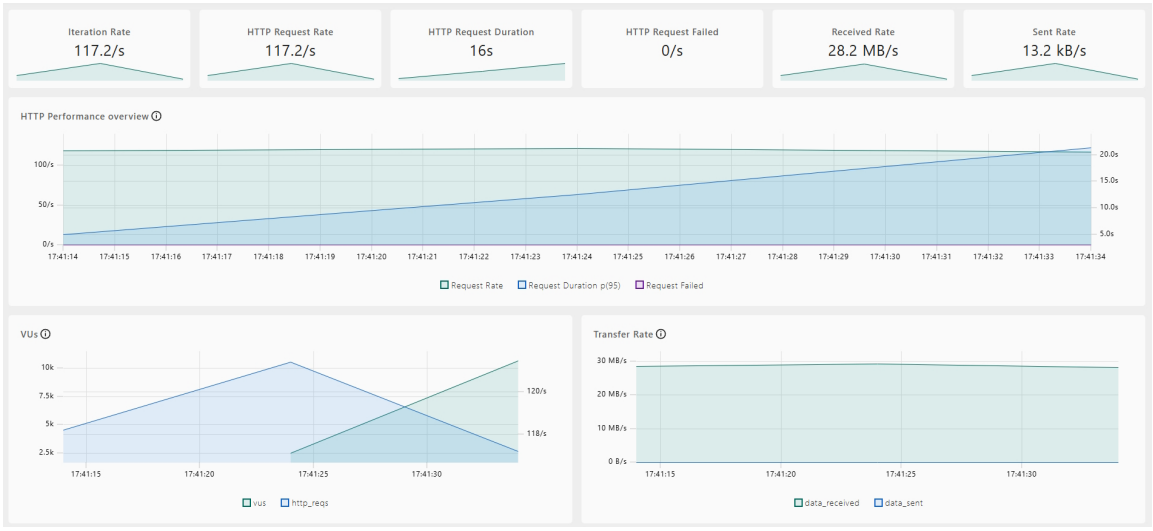


Figura 62: Resumen de métricas de rendimiento durante el test de breakpoint. [Fuente: propia.]

Spike

El test de tipo *spike* se diseñó para simular un aumento súbito de carga, alcanzando en apenas un minuto el 40 % del umbral de usuarios determinado previamente (11 597 VUs). El objetivo es comprobar cómo responde el sistema ante una avalancha rápida de peticiones.

Los resultados muestran que el sistema no es capaz de absorber este tipo de cargas abruptas

sin una degradación significativa: más del 28 % de las peticiones fallaron (6 011 de 21 155), y las latencias promedio se situaron por encima de los 34 segundos, alcanzando incluso el minuto en los percentiles superiores. Aunque se mantuvo operativo, la experiencia de usuario quedaría claramente comprometida en un escenario real de pico instantáneo de tráfico.

Average

El test *average* evalúa el comportamiento del sistema ante una carga moderada pero sostenida en el tiempo, equivalente al 50 % del número máximo de usuarios virtuales soportados (14 496 VUs). El objetivo es analizar si el sistema puede operar de forma estable y eficiente durante varios minutos sin llegar a saturarse.

Los resultados evidencian que, incluso con esta carga intermedia, el sistema comienza a degradarse progresivamente. La latencia media de las peticiones fue de **24.57 segundos**, y la tasa de error alcanzó el **3.26 %**, superando los umbrales definidos como aceptables para un funcionamiento fluido.

En la Figura 63 se aprecia una tendencia creciente tanto en la duración de las peticiones como en el número de usuarios activos. La tasa de transferencia de datos se mantiene relativamente estable, lo que sugiere que el cuello de botella no está en la red, sino en la capacidad de procesamiento interno del sistema.

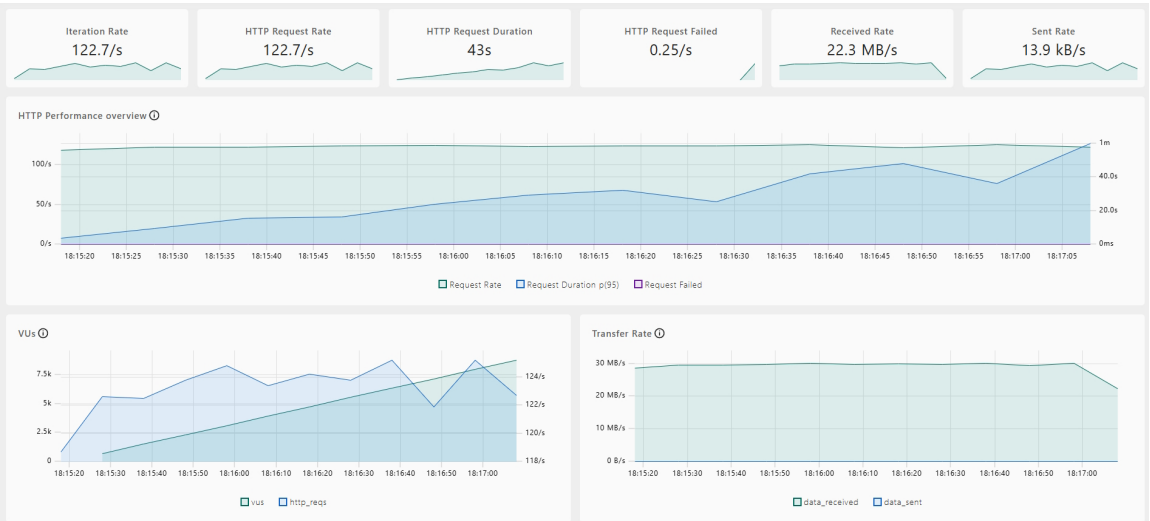


Figura 63: Evolución de carga, latencia y transferencia durante el test *average*. [Fuente: propia.]

La Figura 64 muestra con más detalle las métricas de HTTP. Se observa cómo los percentiles p95 y p99 de la latencia se disparan hasta rozar los **60 segundos**, indicando una acumulación

crítica de peticiones pendientes. Además, la tasa de fallos crece al final de la prueba, coincidiendo con un pico en los tiempos de espera, lo que confirma que, aunque el sistema es capaz de aguantar una carga media durante un tiempo, eventualmente acaba saturándose.

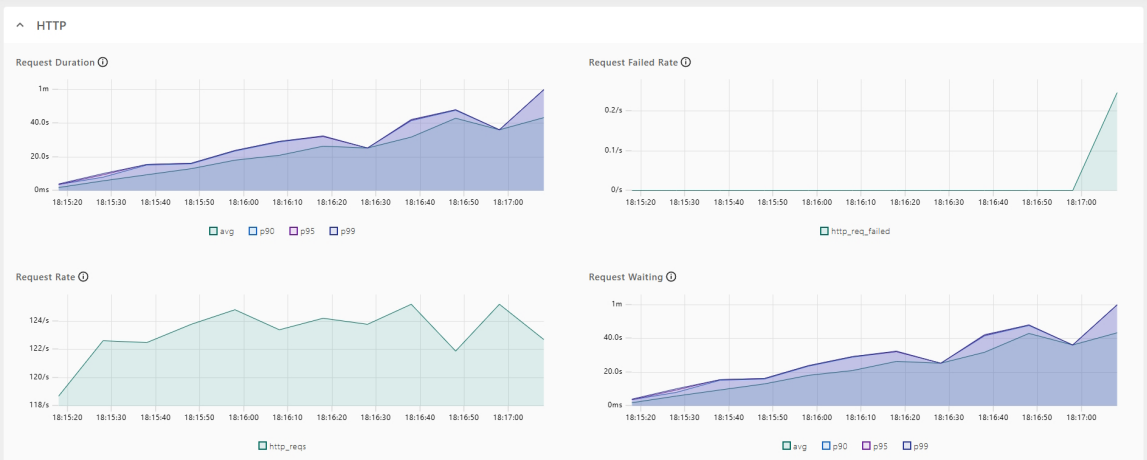


Figura 64: Análisis detallado de métricas HTTP en el test *average*. Se aprecian los picos en latencia y errores. [Fuente: propia.]

La Tabla 16 presenta un desglose detallado de las métricas recopiladas durante el test *average*, incluyendo valores medios, medianas, máximos y percentiles relevantes (p90, p95, p99) para las distintas fases de cada petición HTTP. Además, se recogen los volúmenes totales de datos transferidos y las tasas de error. El tiempo medio de espera de 22 segundos, junto con latencias que alcanzan el minuto en los percentiles superiores, confirma que el cuello de botella se encuentra en el procesamiento interno del backend y no en la red ni en la conexión TLS.

Métrica	Media	Mediana	Máx	p90	p95	p99
http_req_duration	22 s	23 s	1 min	41 s	46 s	1 min
http_req_waiting	22 s	23 s	1 min	41 s	46 s	1 min
iteration_duration	22 s	23 s	1 min	41 s	46 s	1 min
http_req_connecting	32 μs	0 ms	21 ms	75 μs	89 μs	128 μs
http_req_blocked	45 μs	2 μs	21 ms	110 μs	133 μs	188 μs
http_req_receiving	136 μs	104 μs	3 ms	251 μs	301 μs	418 μs
http_req_sending	15 μs	8 μs	2 ms	31 μs	39 μs	63 μs
http_req_tls_handshaking	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms
Resumen de tráfico y tasa de error						
data_received	3.49 GB (29.1 MB/s)					
data_sent	1.67 MB (13.9 kB/s)					
http_reqs	14,800 (123.2 req/s)					
iterations	14,800 (123.2 it/s)					
http_req_failed	0.02 errores/s (3.26 %)					

Tabla 16: Resumen técnico del test *average* a partir de métricas obtenidas con Grafana (K6).

Stress

El test *stress* se diseñó para evaluar el comportamiento del sistema ante una carga muy elevada sostenida, fijada en el 80 % del punto de ruptura previamente identificado (23 194 usuarios virtuales). El propósito era comprobar si el sistema podía mantener un nivel de servicio aceptable en condiciones de saturación cercana a su límite teórico.

Durante la prueba, aunque no se produjeron errores (0 % de peticiones fallidas), el sistema experimentó una degradación drástica en los tiempos de respuesta: la latencia media fue de 36.97 segundos, con valores prácticamente constantes y máximos de hasta 36.99 segundos. Este comportamiento indica que el sistema mantiene la integridad funcional bajo estrés, pero a costa de ofrecer tiempos de espera totalmente inaceptables desde el punto de vista de la experiencia de usuario.

Además, el test fue abortado automáticamente antes de completarse debido al incumplimiento del umbral de latencia establecido (5 segundos de media), lo cual confirma que, si bien

no se colapsa estructuralmente, el sistema no está preparado para sostener cargas tan elevadas durante largos periodos sin una pérdida severa de rendimiento.

En conjunto, los tests revelan que la API puede operar con fiabilidad hasta aproximadamente 10 000 usuarios sin comprometer el rendimiento. Más allá de este punto, la latencia aumenta rápidamente, incluso sin errores de conexión. Este análisis permitirá priorizar futuras mejoras en el escalado y rendimiento del backend web.

Los tests de carga han sido implementados utilizando la herramienta **k6**, que permite definir escenarios de carga en scripts escritos en JavaScript. Cada prueba puede ejecutarse de forma independiente mediante el comando **k6 run nombre_del_script.js**, lo que facilita su reproducción y análisis. Además, k6 proporciona una interfaz web opcional con métricas en tiempo real (**K6_WEB_DASHBOARD=true**), que ha sido utilizada para visualizar latencias, tasas de error y número de usuarios virtuales durante la ejecución de las pruebas.

5.2. Verificación formal del sistema de control de audio

Con el objetivo de garantizar la corrección del sistema en partes especialmente sensibles, se ha aplicado una técnica de verificación formal sobre el componente encargado del flujo de estados de audio. Este módulo determina si un fragmento de audio recibido debe añadirse o no a una ventana de grabación, basándose en el nivel de energía (RMS). Un fallo en esta lógica podría ocasionar errores graves como interrupciones prematuras, pérdidas de voz o respuestas incorrectas del asistente.

Se ha utilizado el verificador SPIN junto con el lenguaje Promela para modelar este comportamiento de forma abstracta y verificar propiedades temporales clave mediante lógica temporal lineal (LTL), siguiendo el enfoque introducido por Pnueli (1977) y concretado en la herramienta SPIN (Holzmann, 1997).

5.2.1. Modelo

A continuación, se muestra el modelo especificado en lenguaje Promela. El siguiente apartado explica en detalle este modelo.

```
#define THRESHOLD 1
#define TIMEOUT 3
#define NO_AUDIO 0
```

```
#define SOME_AUDIO 1
#define END_AUDIO 2

int rms = 0;
int timer = 0;
int state = NO_AUDIO;
bool attached = false;

chan rms_channel = [0] of { int };

proctype Microphone() {
    do
    :: rms_channel!0;
    :: rms_channel!1;
    :: rms_channel!2;
    od
}

proctype AssistantHelper() {
    int current_rms;
    int received_rms;

    do
    :: rms_channel?current_rms ->
        received_rms = current_rms;

        if
        :: received_rms >= THRESHOLD ->
            if
            :: state == NO_AUDIO ->
  atomic {
      rms = received_rms;
                    state = SOME_AUDIO;
                    timer = 0;
                    attached = true;
  }
            :: else ->
                timer = 0;
                attached = true;
            fi
        :: else ->
            if
            :: state != NO_AUDIO ->
                timer++;
                if
                :: timer >= TIMEOUT ->
      atomic {
                        state = END_AUDIO;
                        attached = false;
      }
                :: else -> skip;
                fi
            :: else -> skip;
            fi
        fi
    :: state == END_AUDIO -> break;
    od
}

init {
    run Microphone();
    run AssistantHelper();
}
```

```
ltl p1 { [] (state == SOME_AUDIO -> rms >= THRESHOLD) }

ltl p2 { [] (state == SOME_AUDIO -> attached) }

ltl p3 { []<> (state == NO_AUDIO) && []<> (state == SOME_AUDIO) && []<> (state == END_AUDIO) ->
         [] (rms >= THRESHOLD -> <> (state == SOME_AUDIO)) }

ltl p4 { []<> (state == NO_AUDIO) && []<> (state == SOME_AUDIO) && []<> (state == END_AUDIO) ->
         [] ((state == SOME_AUDIO && rms < THRESHOLD) -> <> (state == END_AUDIO)) }

ltl p5 { []<> (state == NO_AUDIO) && []<> (state == SOME_AUDIO) && []<> (state == END_AUDIO) -> ((state == NO_AUDIO) U (state
     == SOME_AUDIO)) }
```

El sistema modela un componente de asistencia auditiva que cambia de estado en función del nivel de señal RMS recibido:

- **Microphone**: genera continuamente valores RMS entre 0 y 2, que representan el nivel de entrada de audio.
- **AssistantHelper**: procesa los valores RMS recibidos y cambia de estado entre **NO_AUDIO**, **SOME_AUDIO** y **END_AUDIO**.

Cuando el valor RMS supera un umbral (**THRESHOLD = 1**), el sistema entra en **SOME_AUDIO**. Si los siguientes valores caen por debajo del umbral, se activa un temporizador. Si ese temporizador alcanza un límite (**TIMEOUT = 3**), se pasa al estado **END_AUDIO** y se desactiva el componente (**attached = false**).

5.2.2. Propiedades LTL verificadas

Una vez definido el modelo, es fundamental analizar su comportamiento formalmente. Para ello, utilizamos la **lógica temporal lineal** (LTL), que permite especificar propiedades sobre secuencias infinitas de estados en sistemas concurrentes.

Antes de presentar las propiedades concretas, introducimos brevemente los principales operadores de LTL que emplearemos:

- ○ **(next / siguiente)**: $\bigcirc p$ se cumple si p es verdadera en el siguiente estado.
- ◇ **(eventually / eventualmente)**: $\Diamond p$ se cumple si p ocurre en algún estado futuro.
- □ **(always / siempre)**: $\Box p$ significa que p es verdadera en todos los estados futuros.

- **U (until / hasta que)**: $p\,U\,q$ se cumple si p se mantiene verdadera hasta que q se cumple (y q debe cumplirse eventualmente).
- **R (release / liberación)**: $p\,R\,q$ se cumple si q es verdadera hasta (y durante) que p sea verdadera. A diferencia de U, p no necesita cumplirse.

Cabe destacar que, aunque el operador **R** no está implementado directamente en SPIN, puede expresarse mediante la transformación:

$$p\,R\,q \equiv \neg(\neg p\,U\,\neg q)$$

En la verificación de sistemas concurrentes no deterministas, pueden surgir ejecuciones infinitas que omitan transiciones relevantes, generando falsos positivos. Para mitigar este problema, introducimos condiciones de **justicia**, que exigen que ciertos estados se visiten infinitamente a lo largo del tiempo.

En este caso, se requiere que el sistema pase repetidamente por los tres estados clave: **NO_AUDIO**, **SOME_AUDIO** y **END_AUDIO**. Esta condición se expresa con la siguiente cláusula:

$$\mathit{JUSTICIA} = \Box\Diamond(\mathit{NO\_AUDIO}) \wedge \Box\Diamond(\mathit{SOME\_AUDIO}) \wedge \Box\Diamond(\mathit{END\_AUDIO})$$

A continuación se presentan las propiedades LTL definidas para verificar el modelo. Las propiedades **P1** y **P2** son invariantes y no dependen de la cláusula de justicia, mientras que **P3**, **P4** y **P5** sí la requieren para garantizar su validez en ejecuciones infinitas:

- **P1 – Coherencia entre estado y señal:**

  $$\Box(\mathit{state} = \mathit{SOME\_AUDIO} \rightarrow \mathit{rms} \geq 1)$$

  Siempre que el sistema esté grabando, el valor RMS debe ser al menos igual al umbral. Esta propiedad es una invariante.

- **P2 – Activación implica conexión:**

  $$\Box(\mathit{state} = \mathit{SOME\_AUDIO} \rightarrow \mathit{attached})$$

  Si el sistema está grabando, necesariamente debe estar conectado. También es una invariante.

- **P3 – Activación tras señal fuerte (con justicia):**

$$\mathcal{JUSTICIA} \rightarrow \Box\,(rms \geq 1 \rightarrow \Diamond(state = SOME\_AUDIO))$$

  Siempre que se detecte una señal fuerte, eventualmente se debe entrar en estado de grabación, bajo la suposición de recorridos cíclicos por los estados.

- **P4 – Decaimiento lleva al cierre (con justicia):**

$$\mathcal{JUSTICIA} \rightarrow \Box\,((state = SOME\_AUDIO \wedge rms < 1) \rightarrow \Diamond(state = END\_AUDIO))$$

  Si la señal cae por debajo del umbral mientras se graba, el sistema debe finalizar la grabación en algún momento.

- **P5 – Transición ordenada (con justicia):**

$$\mathcal{JUSTICIA} \rightarrow (state = NO\_AUDIO)\ U\ (state = SOME\_AUDIO)$$

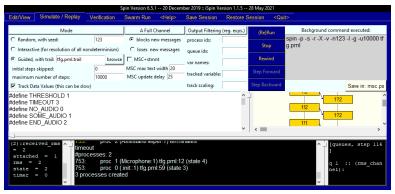
  Bajo comportamiento recurrente, el sistema permanece esperando hasta que se detecta una señal que activa la grabación.

5.2.3. Verificación con SPIN

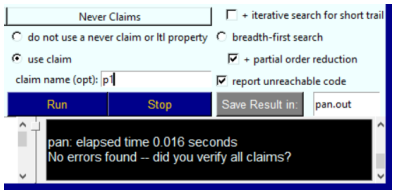
La verificación formal del modelo ha permitido comprobar automáticamente un conjunto de propiedades críticas que garantizan la seguridad y el comportamiento correcto del sistema de captura de audio. Entre las propiedades validadas se encuentran:

- El sistema únicamente entra en estado de grabación si el valor de RMS supera el umbral definido.
- La conexión con el asistente se mantiene activa durante todo el tiempo en que se está grabando.
- Cuando la señal de audio decae, el sistema finaliza correctamente la grabación tras un intervalo de espera razonable.
- Bajo condiciones de justicia —es decir, asumiendo que todos los estados relevantes se alcanzan de forma cíclica—, el sistema no entra en bucles inválidos ni se queda bloqueado indefinidamente.

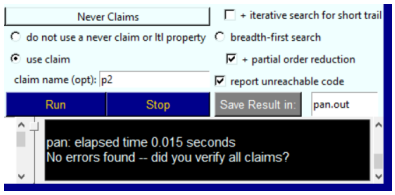
La herramienta SPIN ha confirmado que el modelo cumple todas las propiedades expresadas en lógica temporal lineal (LTL), sin encontrar contraejemplos ni violaciones en los escenarios explorados. Este proceso refuerza la confianza en la lógica de control implementada para el flujo de audio, especialmente en un contexto concurrente y asíncrono como el de ROS 2.



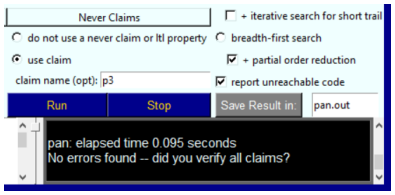
(a) Sistema sin fin



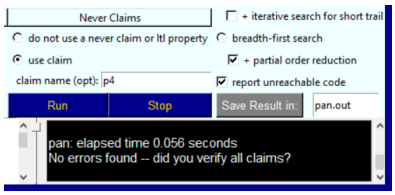
(b) P1 verificada



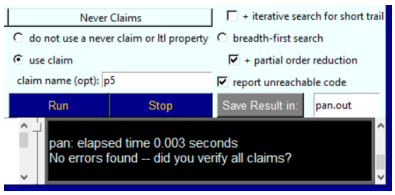
(c) P2 verificada



(d) P3 verificada



(e) P4 verificada



(f) P5 verificada

Figura 65: Resumen de verificaciones formales realizadas con SPIN. Se comprueba que el modelo tiene ejecución infinita y que todas las propiedades LTL definidas (P1 a P5) se satisfacen sin errores. [Fuente: propia.]

La Figura 65 presenta una colección de capturas del verificador SPIN que ilustran tanto la exploración infinita del sistema como la verificación exitosa de cada una de las cinco propiedades LTL especificadas en el modelo. En todos los casos, el verificador confirma que no se detectan errores, que se alcanzan todos los estados esperados, y que las garantías lógicas sobre el comportamiento del sistema se cumplen en su totalidad. Esta validación formal complementa las pruebas empíricas descritas en el capítulo y aporta una base matemática sólida sobre la corrección del componente modelado.

5.3. Métricas comunes de evaluación

En las siguientes secciones se evalúan distintos modelos aplicados a tareas clave del sistema: clasificación de intenciones, detección y reconocimiento facial, extracción de información estructurada, etc. A pesar de tratarse de tareas distintas, muchas de las métricas utilizadas son compartidas, lo que permite establecer criterios de comparación homogéneos.

A continuación se definen las métricas más utilizadas, que aparecerán recurrentemente en los diferentes apartados de evaluación.

Precisión (Precision): proporción de aciertos entre todas las predicciones realizadas. Mide cuántas de las predicciones del modelo han sido realmente correctas:

$$\text{Precision} = \frac{TP}{TP + FP}$$

donde FP representa los *falsos positivos*.

Recall (Sensibilidad): proporción de aciertos entre todos los ejemplos que deberían haberse detectado correctamente. Evalúa la cobertura del modelo:

$$\text{Recall} = \frac{TP}{TP + FN}$$

donde FN son los *falsos negativos*.

F1-score: media armónica entre precisión y *recall*. Es especialmente útil cuando se busca un equilibrio entre ambas métricas:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Tiempo medio de inferencia (t_m): tiempo promedio necesario para que el modelo realice una predicción. En tareas que requieren tiempo real, esta métrica es clave. Se calcula como:

$$t_m = \frac{1}{T} \sum_{i=1}^{T} t_i$$

donde T es el número total de muestras evaluadas, y t_i el tiempo que ha tardado el modelo en responder en el ejemplo i-ésimo.

Estas métricas constituyen la base sobre la que se construyen los *scores* compuestos de cada tarea, ponderando precisión, eficiencia y robustez según el contexto y las prioridades del sistema.

5.4. Evaluación de modelos de lenguaje y embeddings

Este apartado presenta la validación de distintos modelos de lenguaje (LLMs) y de embeddings para las tareas conversacionales clave del sistema. El objetivo es seleccionar el modelo más adecuado para tareas estructuradas como clasificación de intenciones, extracción de argumentos o confirmación de identidad, y compararlo con enfoques más ligeros basados en similitud semántica.

El modelo mejor valorado se utilizará como componente principal del razonamiento conversacional del sistema. No obstante, para la generación abierta de respuestas el usuario podrá elegir libremente el modelo desde la interfaz web, ya que se trata de una tarea más subjetiva y creativa.

A continuación se presentan los experimentos realizados para cada tarea.

5.4.1. Extracción y confirmación de nombre

Dentro del sistema de interacción humano-robot propuesto, una de las capacidades clave es la identificación del interlocutor. Para ello, el robot debe ser capaz de extraer el nombre de una persona cuando esta responde libremente a la pregunta *¿Cómo te llamas?*, así como confirmar o rechazar un nombre cuando se le propone directamente (*¿Te llamas X?*). Estas tareas requieren una comprensión semántica fina y la capacidad de estructurar respuestas de forma fiable, por lo que constituyen una parte fundamental del razonamiento conversacional del sistema.

Métricas evaluadas

- **True Positives (TP)**: número de respuestas correctas, donde el modelo acierta exactamente el valor esperado del campo estructurado.
- **Valid JSON**: número de respuestas que pudieron parsearse correctamente como JSON válido.
- **Precision**, **Recall** y **F1-score**: métricas estándar de clasificación, previamente explicadas.

- **Tiempo medio de inferencia** (t_m): tiempo promedio necesario para generar una respuesta estructurada.

Criterio de puntuación global del modelo

Se ha definido una puntuación compuesta que combina eficiencia, precisión semántica y validez estructural para valorar el rendimiento general de los modelos en esta tarea:

$$\text{Score}_{\text{model}} = 0{,}5 \cdot \left(1 - \frac{t_m}{t_{\max}}\right) + 0{,}4 \cdot F1 + 0{,}1 \cdot \left(\frac{\text{Valid JSON}}{T}\right)$$

La velocidad de inferencia se prioriza (50 %) debido a las exigencias de tiempo real del sistema. El *F1-score* (40 %) refleja la capacidad del modelo para acertar tanto en la intención como en sus argumentos, siendo crucial para tareas estructuradas. Por último, se valora en menor medida (10 %) la generación de salidas válidas en formato JSON, útil para integraciones pero no imprescindible si el contenido semántico es correcto.

Procedimiento experimental

Se han creado manualmente dos pequeños **datasets propios** compuestos por frases realistas y variadas:

- **Extracción de nombre**: 23 frases en las que el usuario responde libremente a *¿Cómo te llamas?*, incluyendo expresiones como *Me llamo Ana*, *Puedes llamarme Carlos* o *Se supone que me llaman Pedro*.
- **Confirmación de nombre**: 23 frases en respuesta a *¿Te llamas X?*, con distintas formulaciones afirmativas, negativas y ambiguas (e.g., *Sí, ese soy yo*, *Ese no es mi nombre*, *¿Por qué lo preguntas?*).

Cada frase tiene una anotación esperada en formato estructurado, permitiendo evaluar objetivamente la precisión semántica y estructural del modelo.

Resultados de la extracción de nombre

Como se puede observar en la Tabla 17, varios modelos logran una tasa perfecta de aciertos (**TP** = 23) con precisión y *recall* del 100 %. Sin embargo, las diferencias en tiempo de inferencia marcan la principal distinción entre ellos. Modelos como **Qwen 2.5 7B** y **LLaMA 3.1 8B** ofrecen un equilibrio excepcional entre precisión absoluta y eficiencia, superando en puntuación

incluso a modelos de mayor tamaño como **GPT 4** o **Qwen 2.5 32B**. Por su parte, **Gemini 1.5 Flash** destaca como una de las mejores opciones vía API, manteniéndose competitiva frente a modelos locales potentes.

Modelo	TP	JSON	Prec.	Recall	F1	t_m (s)	Score
Qwen 2.5 7B	23	23	1.0000	1.0000	1.0000	0.3254	**0.977**
LLaMA 3.1 8B	23	23	1.0000	1.0000	1.0000	0.4288	0.970
Qwen 1.5 7B	23	23	1.0000	1.0000	1.0000	0.4999	0.965
Falcon 3 10B	23	23	1.0000	1.0000	1.0000	0.5971	0.959
Qwen 2.5 14B	23	23	1.0000	1.0000	1.0000	0.6120	0.958
DeepSeek 7B	22	23	0.9565	0.9565	0.9565	0.4496	0.951
GPT 3.5 Turbo	23	23	1.0000	1.0000	1.0000	0.7717	0.947
Gemini 1.5 Flash	22	22	1.0000	0.9565	0.9778	0.7741	0.933
GPT 4o	23	23	1.0000	1.0000	1.0000	1.1195	0.922
GPT 4	23	23	1.0000	1.0000	1.0000	1.2679	0.912
Qwen 2.5 32B	23	23	1.0000	1.0000	1.0000	1.3141	0.909
Gemini 2.0 Lite	21	21	1.0000	0.9130	0.9545	1.4021	0.876
Gemini 2.5 Flash	23	23	1.0000	1.0000	1.0000	1.8143	0.874
DeepSeek R1 Qwen 32B	21	23	0.9130	0.9130	0.9130	1.3911	0.869
DeepSeek R1 LLaMA 70B	22	23	0.9565	0.9565	0.9565	2.3871	0.817
DeepSeek 67B	23	23	1.0000	1.0000	1.0000	2.7568	0.809
LLaMA 3.3 70B	23	23	1.0000	1.0000	1.0000	2.8521	0.802
Gemma 3 27B	20	22	0.9091	0.8696	0.8889	2.7600	0.760
Yi 1.5 34B	21	23	0.9130	0.9130	0.9130	3.1368	0.748
Qwen 2.5 72B	23	23	1.0000	1.0000	1.0000	3.7136	0.743
Yi 1.5 9B	10	10	1.0000	0.4348	0.6061	7.2134	0.286
Phi-2	0	0	0.0000	0.0000	0.0000	5.3467	0.129
Mistral 7B	0	0	0.0000	0.0000	0.0000	5.8955	0.091

Tabla 17: Resultados de la tarea de extracción de nombre (*¿Cómo te llamas?*).

Resultados de la confirmación de nombre

La Tabla 18 muestra un rendimiento más variado entre los modelos evaluados. Aunque muchos alcanzan buenos valores de *F1-score*, las diferencias en el tiempo de inferencia afectan notablemente a la puntuación final. Destacan nuevamente **Gemini 1.5 Flash** y **Falcon 3 10B** por su equilibrio entre precisión y eficiencia. También sobresale **Qwen 2.5 7B**, que si bien tiene menor *recall*, compensa con una latencia muy baja, lo que lo convierte en una alternativa sólida para aplicaciones en tiempo real.

Modelo	TP	JSON	Prec.	Recall	F1	t_m (s)	Score
Gemini 1.5 Flash	22	23	0.9565	0.9565	0.9565	0.7514	**0.932**
Falcon 3 10B	21	23	0.9130	0.9130	0.9130	0.5823	0.926
GPT 4o	21	23	0.9130	0.9130	0.9130	0.8353	0.908
Qwen 2.5 7B	18	23	0.7826	0.7826	0.7826	0.3055	0.892
Qwen 2.5 14B	19	23	0.8261	0.8261	0.8261	0.6095	0.889
Qwen 2.5 32B	22	23	0.9565	0.9565	0.9565	1.4037	0.887
LLaMA 3.1 8B	19	23	0.8261	0.8261	0.8261	0.6622	0.885
Gemini 2.0 Lite	20	22	0.9091	0.8696	0.8889	1.3663	0.858
Gemini 2.5 Flash	22	23	0.9565	0.9565	0.9565	1.9217	0.852
Qwen 1.5 7B	16	23	0.6957	0.6957	0.6957	0.4297	0.849
DeepSeek R1 Qwen 32B	20	23	0.8696	0.8696	0.8696	1.4864	0.847
DeepSeek 7B	14	23	0.6087	0.6087	0.6087	0.4033	0.816
GPT 3.5 Turbo	14	23	0.6087	0.6087	0.6087	0.7242	0.794
DeepSeek R1 LLaMA 70B	21	23	0.9130	0.9130	0.9130	2.5423	0.792
GPT 4	19	23	0.8261	0.8261	0.8261	2.1780	0.782
DeepSeek 67B	20	23	0.8696	0.8696	0.8696	2.7091	0.764
Gemma 3 27B	19	22	0.8636	0.8261	0.8444	2.7997	0.743
Qwen 2.5 72B	23	23	1.0000	1.0000	1.0000	3.8071	0.741
Yi 1.5 34B	21	23	0.9130	0.9130	0.9130	3.3258	0.739
LLaMA 3.3 70B	22	23	0.9565	0.9565	0.9565	4.9119	0.649
Yi 1.5 9B	17	19	0.8947	0.7391	0.8095	7.3549	0.406
Phi-2	0	0	0.0000	0.0000	0.0000	5.3032	0.139
Mistral 7B	0	0	0.0000	0.0000	0.0000	5.3467	0.137

Tabla 18: Resultados de la tarea de confirmación de nombre (*¿Te llamas X?*).

Los resultados muestran diferencias significativas entre modelos, tanto en precisión como en velocidad de respuesta. Una observación importante es que algunos modelos como **Mistral 7B** y **Phi-2**, a pesar de funcionar correctamente en tareas básicas, son incapaces de generar

respuestas válidas cuando se les introduce un *prompt system* largo y estructurado. A pesar de tener temperatura cero y condiciones controladas, tienden a alucinar o fallar completamente, probablemente debido a limitaciones en su ventana de contexto o su capacidad para seguir instrucciones complejas.

En contraposición, modelos como **Qwen 2.5 7B** y **Gemini 1.5 Flash** demuestran un rendimiento sobresaliente en ambas tareas, combinando alta precisión estructurada con tiempos de inferencia competitivos. Esto convierte a ambos en alternativas viables y complementarias para este tipo de tareas: **Qwen 2.5 7B** destaca por ejecutarse localmente con muy baja latencia, ideal para sistemas embebidos o instalaciones sin conexión a Internet, mientras que **Gemini 1.5 Flash** representa una solución robusta vía API, aprovechando la escalabilidad y potencia de la nube.

Esta dualidad permite dotar al sistema de una gran flexibilidad: según el entorno de despliegue, se puede optar por un modelo local o remoto sin perder fiabilidad. Por tanto, se concluye que tanto **Qwen 2.5 7B** como **Gemini 1.5 Flash** serán los modelos seleccionados para estas tareas clave del sistema de interacción humano-robot.

5.4.2. Clasificación de intenciones con modelos LLM

El sistema necesita comprender qué desea hacer el usuario a partir de una conversación en lenguaje natural. Esta tarea se conoce como *clasificación de intenciones*, y consiste en identificar correctamente el comando implícito (intención) y, si es necesario, sus argumentos (por ejemplo, el nombre de un usuario a eliminar). Para ello se ha evaluado un conjunto de modelos de lenguaje (LLMs) que reciben como entrada una conversación previa (*prompt histórico*) y el último mensaje del usuario, devolviendo un JSON estructurado con los campos **intent** y **arguments**.

Métricas evaluadas

- **True Positives (TP)**: número de ejemplos en los que el modelo ha acertado completamente la intención y sus argumentos.
- **Valid JSON**: número de respuestas que pudieron parsearse correctamente como JSON válido.

- **Precision**, **Recall** y **F1-score**: métricas estándar de clasificación, explicadas en secciones anteriores.
- **Tiempo medio de inferencia** (t_m): tiempo promedio necesario para procesar una entrada y generar respuesta.

Criterio de puntuación global del modelo

Se ha definido una puntuación compuesta para evaluar el rendimiento general de los modelos en la tarea de clasificación de intenciones, combinando precisión semántica, eficiencia y validez estructural:

$$\text{Score}_{\text{model}} = 0{,}5 \cdot \left(1 - \frac{t_m}{t_{\max}}\right) + 0{,}4 \cdot F1 + 0{,}1 \cdot \left(\frac{\text{Valid JSON}}{T}\right)$$

En este caso, se da mayor prioridad a la velocidad de inferencia (50 %) por tratarse de un sistema diseñado para funcionar en tiempo real. El *F1-score* (40 %) representa la capacidad del modelo para predecir correctamente tanto la intención como los argumentos, siendo un indicador clave de fiabilidad semántica. Por último, se incluye un peso menor (10 %) a la proporción de respuestas estructuralmente válidas en formato JSON, importante para la integración pero no determinante si el contenido es correcto.

Procedimiento experimental

Se ha construido un **dataset propio** compuesto por 74 ejemplos de conversación, cada uno formado por una secuencia previa de mensajes y un mensaje final del usuario. Se han anotado manualmente tanto la intención esperada como los argumentos correctos. Esto permite evaluar si el modelo comprende correctamente el contexto y genera una salida estructurada con la información requerida.

Resultados de la clasificación de intenciones con LLMs

Modelo	TP	JSON	Prec.	Recall	F1	t_m (s)	Score
Qwen 2.5 14B	73	74	0.9865	0.9865	0.9865	0.8174	**0.987**
Qwen 2.5 7B	71	74	0.9595	0.9595	0.9595	0.3667	0.980
Falcon 3 10B	71	74	0.9595	0.9595	0.9595	0.6341	0.978
Qwen 2.5 32B	73	74	0.9865	0.9865	0.9865	1.7678	0.977
Qwen 1.5 7B	68	74	0.9189	0.9189	0.9189	0.3810	0.964
LLaMA 3.1 8B	68	74	0.9189	0.9189	0.9189	0.4820	0.963
Gemini 2.5 Flash	71	74	0.9595	0.9595	0.9595	2.3838	0.960
GPT 4o	68	74	0.9189	0.9189	0.9189	0.8410	0.959
DeepSeek 7B	67	74	0.9054	0.9054	0.9054	0.4890	0.957
DeepSeek 67B	72	74	0.9730	0.9730	0.9730	3.2857	0.957
GPT 4	72	74	0.9730	0.9730	0.9730	3.4064	0.956
Gemini 1.5 Flash	67	74	0.9054	0.9054	0.9054	0.7052	0.955
GPT 3.5 Turbo	66	74	0.8919	0.8919	0.8919	0.7142	0.950
Qwen 2.5 72B	71	74	0.9595	0.9595	0.9595	4.3411	0.941
LLaMA 3.3 70B	72	74	0.9730	0.9730	0.9730	4.8943	0.941
Gemini 2.0 Lite	59	72	0.8194	0.7973	0.8082	1.6184	0.905
Yi 1.5 9B	43	5	8.6000	0.5811	1.0886	13.8416	0.807
DeepSeek R1 Qwen 32B	73	74	0.9865	0.9865	0.9865	25.0447	0.749
Gemma 3 27B	60	73	0.8219	0.8108	0.8163	40.3930	0.530
DeepSeek R1 LLaMA 70B	71	70	1.0143	0.9595	0.9861	51.0647	0.489
Yi 1.5 34B	42	13	3.2308	0.5676	0.9655	43.0512	0.482
Mistral 7B	40	0	0.0000	0.5405	0.0000	1.9413	0.481
Phi-2	40	0	0.0000	0.5405	0.0000	6.9959	0.431

Tabla 19: Resultados de la tarea de clasificación de intenciones con modelos LLM.

La Tabla 19 muestra una evaluación exhaustiva del rendimiento de múltiples modelos LLM en la tarea de clasificación de intenciones conversacionales. Los resultados revelan que modelos como **Qwen 2.5 14B**, **Qwen 2.5 7B** y **Falcon 3 10B** destacan por combinar una precisión

excelente con una velocidad de inferencia muy competitiva, alcanzando puntuaciones globales superiores al 0.97. En particular, **Qwen 2.5 14B** alcanza la mejor puntuación total, con un *F1-score* de 0.9865 y tiempos de respuesta aceptables, lo que lo convierte en una opción especialmente fiable.

Por otro lado, se observa un fenómeno recurrente en modelos como **Mistral 7B** y **Phi-2**: a pesar de funcionar correctamente en tareas más simples, no son capaces de gestionar correctamente prompts largos y estructurados como los utilizados en esta tarea. En ambos casos, el número de JSON válidos es cero, lo que implica que no generan una salida parseable en ninguna ocasión, probablemente debido a limitaciones en el contexto manejable o a la incapacidad de seguir instrucciones cuando estas se extienden más allá de cierto umbral. Este comportamiento compromete su utilidad en sistemas donde se requiere precisión estructurada.

En conjunto, los resultados permiten identificar dos grandes ganadores: **Qwen 2.5 14B**, que puede ejecutarse localmente en entornos potentes y ofrece una eficiencia notable, y **Gemini 1.5 Flash**, que, sin ser el mejor absoluto, mantiene una gran consistencia y disponibilidad a través de API. Esta dualidad local–remoto permite adaptar el sistema tanto a despliegues con servidores propios como a soluciones en la nube, maximizando la versatilidad y robustez del sistema propuesto.

5.4.3. Clasificación de intenciones con *embeddings*

Como alternativa a los modelos LLM, se ha evaluado un enfoque basado en **embeddings semánticos** para la clasificación de intenciones. En este método, el mensaje del usuario se convierte en un vector numérico mediante un modelo de codificación, y se compara con una base de comandos conocidos usando distancia coseno. La intención se predice como la más cercana en el espacio vectorial.

Este enfoque no requiere generación textual ni salida estructurada, lo que permite una inferencia extremadamente rápida y fácilmente paralelizable.

Métricas evaluadas

- **True Positives (TP)**: número de ejemplos correctamente clasificados (la intención coincide con la esperada).
- **Precision**, **Recall** y **F1-score**: métricas estándar de clasificación, descritas previamente.

- **Tiempo medio de inferencia** (t_m): tiempo promedio requerido para codificar una entrada y obtener la predicción.
- **Dimensión del embedding** (D): tamaño del vector numérico generado por el modelo.

Criterio de puntuación global del modelo

Se define una puntuación compuesta que valora el rendimiento general de cada modelo:

$$\text{Score}_{\text{model}} = 0{,}65 \cdot F1 + 0{,}25 \cdot \left(1 - \frac{t_m}{t_{\max}}\right) + 0{,}1 \cdot \left(1 - \frac{D}{D_{\max}}\right)$$

En este caso, se ha otorgado el mayor peso al *F1-score* (65 %), dado que lo más relevante es la capacidad del modelo para clasificar correctamente las intenciones del usuario. El peso relativo del tiempo de inferencia (25 %) es menor que en otras tareas, ya que prácticamente todos los modelos evaluados ofrecen latencias muy bajas y diferencias poco significativas en la práctica. Por último, se considera también la dimensión del vector de salida (10 %), favoreciendo modelos más compactos que faciliten el despliegue en sistemas con recursos limitados.

Resultados de la clasificación con *embeddings*

Modelo	TP	Precision	Recall	F1	$\mathbf{t_m}$ (s)	D	Score
OpenAI Ada-002	58	0.7838	0.7838	0.7838	0.2151	1536	**0.744**
BGE Base v1.5	47	0.6351	0.6351	0.6351	0.0068	768	0.742
MiniLM-L6-v2	43	0.5811	0.5811	0.5811	0.0053	384	0.716
E5 Large	45	0.6081	0.6081	0.6081	0.0109	1024	0.716
BGE-M3	42	0.5676	0.5676	0.5676	0.0112	1024	0.690
Qwen2 GTE 7B	40	0.5405	0.5405	0.5405	0.0472	1024	0.659
OpenAI Babbage	40	0.5405	0.5405	0.5405	0.1356	2048	0.602
OpenAI 3 Small	40	0.5405	0.5405	0.5405	0.2538	1536	0.572
DeepSeek Coder 6.7B	33	0.4459	0.4459	0.4459	0.0467	4096	0.523
OpenAI 3 Large	40	0.5405	0.5405	0.5405	0.6929	3072	0.376

Tabla 20: Resultados de la tarea de clasificación de intenciones mediante embeddings.

Tal y como refleja la Tabla 20, el enfoque basado en *embeddings* resulta altamente competitivo para la tarea de clasificación de intenciones, especialmente en contextos donde la velocidad y el consumo computacional son factores determinantes. Modelos como **OpenAI Ada-002**, **BGE Base v1.5** y **MiniLM-L6-v2** logran un equilibrio notable entre precisión aceptable y tiempos de inferencia extremadamente bajos (por debajo de 10 ms por ejemplo), lo que los convierte en candidatos idóneos para despliegues embebidos o sistemas con restricciones de latencia.

La puntuación compuesta utilizada en esta evaluación prioriza el *F1-score* como medida de precisión semántica, pero también penaliza fuertemente los tiempos de respuesta elevados y las dimensiones excesivas del vector de salida. En ese sentido, modelos ligeros como **MiniLM** o **BGE** obtienen resultados especialmente sólidos gracias a su eficiencia, mientras que alternativas más pesadas como **OpenAI 3 Large** o **DeepSeek Coder** quedan penalizadas a pesar de tener un F1 competitivo.

En definitiva, los modelos de embeddings ofrecen una vía ligera, rápida y fiable para la clasificación de comandos en lenguaje natural, representando una alternativa viable a los LLMs cuando se prioriza la eficiencia por encima de la versatilidad generativa.

Conclusiones finales

La evaluación transversal de los modelos confirma que los LLMs superan ampliamente a los sistemas basados en *embeddings* tanto en precisión como en capacidad estructural. Mientras que los embeddings ofrecen una clasificación razonable con latencias ínfimas, su uso implica limitaciones severas: no permiten extraer argumentos, requieren mantenimiento manual al introducir nuevas intenciones y no gestionan ambigüedad ni reformulaciones complejas. Estas carencias comprometen su utilidad en tareas conversacionales con estructura.

Por el contrario, los modelos LLM han demostrado ser capaces de abordar tareas de razonamiento estructurado con alta precisión y robustez, generando respuestas en formato JSON de forma consistente. Dentro de este grupo, **Qwen 2.5 7B** destaca por ofrecer una combinación excelente de velocidad, tamaño reducido y precisión elevada, siendo ideal para entornos embebidos o sin conectividad. Para soluciones basadas en API, **Gemini 1.5 Flash** se posiciona como la opción más fiable y eficiente, con un rendimiento sólido en todas las tareas evaluadas.

Así, el sistema queda respaldado por una arquitectura flexible en la que se emplea **Qwen**

2.5 7B como modelo local principal, y **Gemini 1.5 Flash** como alternativa remota robusta, garantizando precisión estructurada y respuesta en tiempo real según el entorno de despliegue.

5.5. Evaluación de modelos de detección facial

Para evaluar los detectores faciales, se ha implementado un sistema de comparación cuantitativa basado en anotaciones manuales de un conjunto de imágenes. Cada modelo analiza todas las imágenes y se calcula un conjunto de métricas que permiten medir tanto la calidad de las detecciones como su eficiencia temporal.

Métricas evaluadas

- **True Positives (TP)**: número total de caras reales correctamente detectadas. Una detección se considera correcta si su solapamiento con la anotación manual supera un umbral basado en **IoU**.
- **Precision**, **Recall** y **F1-score**: métricas estándar de clasificación que reflejan la proporción de aciertos, la cobertura de detecciones y su equilibrio conjunto.
- **IoU medio** (IoU_m): media del *Intersection over Union* para las detecciones correctas.
- **Tiempo medio de inferencia** (t_m): tiempo promedio (en segundos) requerido por el modelo para procesar una imagen.

Criterio de puntuación global del detector

Dado que el sistema debe funcionar en tiempo real en un entorno de interacción humano-robot, se define una puntuación compuesta que pondera el rendimiento, la precisión y la calidad de detección:

$$\text{Score}_{\text{global}} = 0{,}5 \cdot \left(1 - \frac{t_m}{t_{\max}}\right) + 0{,}4 \cdot F1 + 0{,}1 \cdot \text{IoU}_m$$

donde $t_{\max}$ representa el mayor tiempo medio registrado entre todos los modelos evaluados.

Dataset MAPIR Faces

Se ha utilizado el *dataset* MAPIR Faces (Baltanas et al., 2020), compuesto por 768 imágenes con una única cara por imagen y anotaciones manuales. Este conjunto permite evaluar la precisión de los detectores en un entorno controlado, frontal y típico de interacción.

Detector	TP	Precision	Recall	F1	IoU_m	t_m (s)	Score
YOLOv5-face	765	0.9935	0.9961	0.9948	0.8181	0.0058	**0.968**
DLIB CNN	760	0.9832	0.9896	0.9864	0.7070	0.0135	0.939
InsightFace-lite	764	0.9795	0.9948	0.9871	0.8687	0.0619	0.861
MTCNN	757	0.9818	0.9857	0.9838	0.9021	0.0729	0.842
YOLOv8-face	763	0.9909	0.9935	0.9922	0.8656	0.0986	0.792
DLIB Frontal	491	0.9646	0.6393	0.7690	0.6816	0.0771	0.726
OpenCV (Haar)	357	0.7727	0.4648	0.5805	0.7123	0.0456	0.715
RetinaFace	765	0.9387	0.9961	0.9665	0.8706	0.2572	0.474

Tabla 21: Comparación de detectores faciales en el *dataset* MAPIR Faces.

Dataset WIDER FACE

Para evaluar el rendimiento en escenas con múltiples personas, se ha utilizado el conjunto de validación del *dataset* WIDER FACE (Yang et al., 2016), compuesto por 2103 imágenes que contienen un total de 4153 caras anotadas.

Detector	TP	Precision	Recall	F1	IoU_m	t_m (s)	Score
YOLOv5-face	3494	0.9684	0.8413	0.9004	0.8105	0.0130	**0.919**
InsightFace-lite	3710	0.9629	0.8933	0.9268	0.8378	0.0578	0.855
DLIB CNN	2829	0.9557	0.6812	0.7954	0.6748	0.0528	0.795
YOLOv8-face	3319	0.9051	0.7992	0.8488	0.8341	0.0966	0.757
MTCNN	3076	0.8916	0.7407	0.8092	0.8061	0.1645	0.621
RetinaFace	3731	0.9693	0.8984	0.9325	0.8378	0.2903	0.457
DLIB Frontal	1944	0.9405	0.4681	0.6251	0.6767	0.2375	0.409
OpenCV (Haar)	1677	0.5287	0.4038	0.4579	0.6975	0.2715	0.285

Tabla 22: Comparación de detectores faciales en el *dataset* WIDER FACE.

Conclusión

El análisis conjunto de los resultados obtenidos en los *datasets* MAPIR Faces y WIDER FACE permite evaluar el rendimiento de los detectores faciales en dos contextos complementarios: entornos controlados con rostros únicos y escenas realistas con múltiples personas.

En entornos controlados, como MAPIR Faces, el detector **YOLOv5-face** ofrece el mejor equilibrio entre precisión, velocidad y calidad de la detección, obteniendo el **score más alto** ($0{,}968$). Su excelente F1-score ($0{,}9948$) y su bajo tiempo de inferencia lo convierten en la opción preferente para interacciones directas con el robot. Le siguen de cerca **DLIB CNN** e **InsightFace-lite**, aunque este último presenta una mayor latencia. Por otro lado, **YOLOv8-face** destaca por su alta precisión, pero su mayor tiempo de ejecución reduce su puntuación global.

En contextos complejos con múltiples rostros, como WIDER FACE, se observa una ligera variación en los rendimientos. En este caso, **YOLOv5-face** vuelve a posicionarse como el más equilibrado, con una alta precisión ($0{,}9684$), una buena capacidad de recuperación ($0{,}8413$) y el mejor score global ($0{,}919$). **InsightFace-lite**, aunque ligeramente más lento, supera al resto en número de detecciones y en métrica F1 ($0{,}9268$), confirmando su solidez técnica. En contraste, detectores como **DLIB Frontal** u **OpenCV (Haar)** muestran limitaciones importantes, tanto en tasa de aciertos como en calidad de detección, por lo que se consideran inadecuados para este sistema.

A pesar de que **YOLOv8-face** representa una evolución arquitectónica más reciente, su rendimiento práctico se ve superado por **YOLOv5-face** en este sistema. Una de las razones principales es que se ha empleado la versión **yolov5n** (*nano*), optimizada específicamente para dispositivos con recursos limitados, lo que le permite alcanzar tiempos de inferencia extremadamente bajos sin sacrificar precisión. Además, el modelo de YOLOv5 ha sido afinado y entrenado específicamente para la tarea de detección facial, mientras que YOLOv8, aunque más moderno, está diseñado para ser más generalista. Esta especialización permite a YOLOv5 ofrecer puntuaciones de confianza precisas y una mayor capacidad de optimización práctica, lo que explica su superioridad tanto en eficiencia como en resultados reales de detección.

La comparativa evidencia que **YOLOv5-face** es el detector más equilibrado y robusto para el sistema multimodal propuesto, mostrando un rendimiento excepcional tanto en entornos frontales como en escenas con múltiples rostros, con un coste computacional muy bajo. **InsightFace-lite** se perfila como una alternativa sólida, especialmente en contextos donde la

calidad de detección prima sobre la latencia. El resto de detectores no alcanzan un rendimiento competitivo, y por tanto, se descartan para su integración en el sistema final.

5.6. Evaluación de modelos de codificación facial

El sistema de reconocimiento de personas necesita convertir las caras detectadas en vectores numéricos, conocidos como *faceprints* o embeddings. Para evaluar la calidad de los modelos encargados de generar estos vectores —denominados **encoders faciales**— se ha desarrollado una batería de métricas que combinan aspectos cuantitativos (precisión, eficiencia) con factores interactivos (cantidad de preguntas necesarias para aprender).

Métricas evaluadas

- **True Positives (TP)**: número total de imágenes en las que el encoder ha conseguido clasificar correctamente la identidad del rostro.
- **Precision**, **Recall** y **F1-score**: métricas clásicas de evaluación de clasificación. Reflejan la proporción de aciertos sobre el total de predicciones, la cobertura de rostros reconocidos correctamente y su equilibrio conjunto, respectivamente.
- ***Embedding* size**: dimensión del vector generado por el encoder. Menores tamaños implican menor uso de memoria y mayor eficiencia en almacenamiento y transmisión.
- **Tiempo medio de codificación** (t_m): tiempo promedio (en segundos) necesario para codificar una imagen. Métrica clave para entornos en tiempo real.
- **Preguntas totales** (Q): número de veces que el sistema ha tenido que preguntar el nombre de una persona o confirmar su identidad. Un encoder más preciso requiere menos interacción con el usuario.
- **Silhouette Score** (Rousseeuw, 1987): mide qué tan bien están separados los embeddings de distintas personas. Valores cercanos a 1 indican buena separación entre clases.
- **Calinski–Harabasz Score** (Caliński & Harabasz, 1974): evalúa la compacidad intra-clase frente a la dispersión inter-clase. Valores más altos indican agrupaciones más definidas.

Criterio de puntuación global del encoder

Con el objetivo de determinar cuál es el encoder más adecuado para el sistema propuesto, se define un **score global** que combina cuatro factores clave:

- **Precisión**: queremos un sistema fiable, que acierte con pocas muestras.
- **Tiempo medio de codificación**: aunque todos los modelos son bastante rápidos, se valora positivamente una menor latencia.
- **Tamaño del *embedding***: menor dimensión implica menor carga computacional y facilita el almacenamiento y la transmisión.
- **Número de preguntas realizadas al usuario**: menos preguntas implican una experiencia más fluida y natural.

Dado que en este tipo de sistemas la experiencia de usuario es prioritaria, se otorga mayor peso a la precisión y a la reducción de preguntas, mientras que el tiempo de inferencia y el tamaño del *embedding* tienen menos impacto comparativo (ya que todos los modelos analizados son suficientemente rápidos). Esta ponderación subjetiva responde a las necesidades específicas del sistema de interacción humano-robot:

$$\text{Score}_{\text{encoder}} = 0{,}6 \cdot \text{Precision} + 0{,}1 \cdot \left(1 - \frac{t_m}{t_{\text{máx}}}\right) + 0{,}1 \cdot \left(1 - \frac{D_{\text{emb}}}{D_{\text{máx}}}\right) + 0{,}2 \cdot \left(1 - \frac{Q}{Q_{\text{máx}}}\right)$$

donde $t_{\text{máx}}$, $D_{\text{máx}}$ y $Q_{\text{máx}}$ corresponden a los valores máximos observados en el experimento.

Procedimiento experimental

Para asegurar una comparación justa y reproducible, se ha utilizado el mismo ***pipeline* completo de reconocimiento facial** durante todas las pruebas, cambiando únicamente el encoder encargado de generar los embeddings. Es decir, se han mantenido constantes todos los demás módulos del sistema, incluidos el detector facial, el clasificador, el gestor de base de datos y los umbrales de decisión.

Las pruebas se han realizado sobre el ***dataset* MAPIR Faces** (Baltanas et al., 2020), el cual contiene imágenes faciales de **16 personas distintas**, con un total de **768** imágenes, capturadas desde múltiples ángulos y condiciones de iluminación, proporcionando un escenario

variado y representativo de interacción humano-robot. Las imágenes se han mezclado aleatoriamente una única vez, y dicho orden aleatorio se ha mantenido constante para todas las evaluaciones, asegurando así la equidad en la comparación entre encoders.

Las métricas evaluadas y el score mencionado anteriormente con cada encoder se presentan de forma comparativa en la Tabla 23.

Encoder	**Size**	**TP**	**Prec.**	**Recall**	**F1**	t_m **(s)**	Q	**Silh.**	**CH**	**Score**
FaceNet	512	755	0.9947	0.9882	0.9915	0.0745	70	0.0869	3.27	**0.844**
SFace	128	719	0.9473	0.9876	0.9670	0.0081	194	0.0234	2.78	0.758
ArcFace	512	601	0.7918	0.9852	0.8780	0.0756	143	0.1077	4.74	0.646
VGGFace	4096	756	0.9960	0.9882	0.9921	0.1095	187	0.0906	4.97	0.605
OpenFace	128	151	0.1989	0.9437	0.3286	0.0408	40	−0.1470	1.06	0.438
DinoV2	768	95	0.1252	0.9135	0.2202	0.0178	3	−1.0000	−1.00	0.437

Tabla 23: Comparativa de encoders faciales.

Conclusión

Los resultados muestran diferencias claras entre los distintos encoders evaluados. El encoder **FaceNet** es el que obtiene la mejor puntuación global. Presenta una precisión excepcional ($0{,}9947$), un tamaño de embedding razonable (512), un tiempo de inferencia competitivo y un número de preguntas reducido, lo que lo convierte en una solución muy equilibrada para entornos interactivos. Además, sus métricas de separabilidad (como *Silhouette Score* y *Calinski-Harabasz*) reflejan una buena estructuración interna de los embeddings.

El encoder **SFace** destaca por su combinación ideal de eficiencia y rendimiento. Con una precisión de $0{,}9473$, su mayor fortaleza radica en su velocidad de inferencia reducida ($t_m = 0{,}0081$ s) y su compacto tamaño de embedding (128 dimensiones). Esto se debe a varias características técnicas:

- Está diseñado como una red ligera, inspirada en arquitecturas optimizadas para dispositivos con recursos limitados (similar a MobileFaceNets), con menos de un millón de parámetros y operaciones profundamente eficientes.

- Emplea una estructura híbrida que mezcla métodos *anchor-based* y *anchor-free*, lo que le permite procesar escalas de rostro con gran rapidez, alcanzando hasta 50 fps en resolución alta como 4K.
- El tamaño reducido del embedding (128D) no solo agiliza la generación de vectores, sino también las comparaciones posteriores, reduciendo la carga computacional y acelerando la interacción con otros módulos del sistema.

No obstante, la eficiencia de SFace se ve contrarrestada por la necesidad de másal usuario (casi el triple respecto a FaceNet), lo que puede alargar la experiencia de usuario cuando se escalan grandes identidades. Esto sitúa a SFace como una opción excelente cuando se prioriza la velocidad y se aceptan ligeras concesiones en precisión o experiencia interactiva.

VGGFace, a pesar de obtener la mejor precisión en bruto, genera embeddings excesivamente grandes (4096 dimensiones), lo que repercute negativamente en el rendimiento del sistema en tiempo real y en la carga de memoria. Por tanto, no se considera práctico para este contexto.

DinoV2 confirma su inadecuación para tareas de reconocimiento facial: al no estar entrenado para rostros, sus resultados son muy pobres tanto en precisión como en las métricas de estructuración del espacio. Su uso queda completamente descartado.

Por su parte, **OpenFace** presenta también una baja capacidad discriminativa, y **ArcFace**, aunque mejora estos valores, no alcanza el balance deseado entre precisión y experiencia de usuario.

En conclusión, el **score final** representa adecuadamente los requisitos del sistema propuesto: alta precisión, eficiencia computacional, *embeddings* compactos y mínima interacción. Por todo ello, se selecciona **FaceNet** como **encoder principal del sistema final**.

5.7. Demostración del sistema

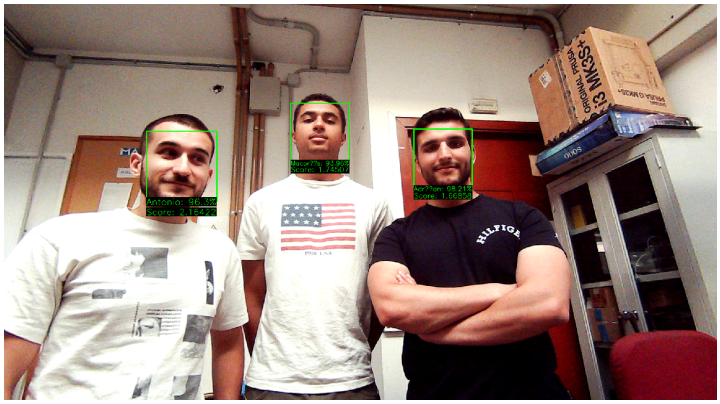
A lo largo de este capítulo se han mostrado capturas de la interfaz web, paneles de control y resultados de validación técnica del sistema, pero su valor más relevante —la interacción multimodal en tiempo real con el robot físico— resulta difícil de plasmar sin recursos audiovisuales. Dado que el sistema ha sido diseñado para ser natural, fluido y autónomo, su uso se manifiesta mejor cuando se experimenta directamente, mediante voz, gestos y expresiones.

No obstante, se incluyen a continuación una serie de imágenes representativas que ilustran el sistema funcionando en condiciones reales sobre el robot **Sancho**. Estas evidencias permiten comprobar cómo se integran los distintos módulos en una interacción completa: detección, reconocimiento, síntesis de voz, animación física, interfaz gráfica y reacción emocional.

Reconocimiento facial en situación real.

En la Figura 66 se muestra una comparación entre la percepción del robot y la escena real. A la izquierda, una captura de **RViz2** representa el resultado del pipeline de visión ejecutado en tiempo real, donde se emplean **YOLOv5** como detector y **FaceNet** como codificador facial. Se observan tres personas reconocidas correctamente, cada una con su **nombre**, **score de detección** y **porcentaje de confianza** del reconocimiento, junto con su **bounding box** en color.

A la derecha, se muestra una imagen de la escena real capturada durante la prueba, donde el robot está frente a las tres personas presentes. Esta visualización ilustra claramente la correspondencia entre lo que "ve" el robot y el entorno físico, validando así la fiabilidad del sistema en contextos reales de interacción.



(a) Vista del robot en RViz2.

(b) Escena real durante la prueba.

Figura 66: Comparación entre la percepción del robot y la escena real, mostrando el reconocimiento facial de tres personas. [Fuente: propia.]

El rendimiento del sistema de visión depende tanto del modelo utilizado como del hardware disponible. En el robot **Sancho** (véase Tabla 2), la combinación **YOLOv5 + FaceNet** alcanza una frecuencia media de **6–7 FPS**, suficiente para interacción fluida al estar desacoplado del flujo principal. Usando un modelo más eficiente como **SFace**, se superan los **30 FPS** en el mismo dispositivo. Cuando el procesamiento se externaliza a un acelerador **uedge** (véase Tabla 7), se obtienen hasta **15 FPS** con **YOLOv5 + FaceNet** y más de **60 FPS** con **YOLOv5 + SFace**, permitiendo su uso en escenarios con mayor concurrencia o resolución.

Escenarios reales de interacción.

Las Figuras 67, 68 y 69 ilustran momentos reales capturados durante sesiones de prueba del sistema, en los que se combinan visión, lenguaje natural, control físico y respuesta emocional.

En la Figura 67 se observa al robot Sancho interactuando con una persona no registrada previamente. Al detectar una cara desconocida, el sistema activa la interfaz gráfica mostrando su cara recortada junto al mensaje *"¿Cuál es tu nombre?"*, acompañado de un campo de texto para responder mediante la pantalla táctil. Además, el usuario puede responder directamente por voz, por ejemplo diciendo *"Me llamo Eulogio"*, lo que permite registrar su nombre de forma natural y sin contacto físico.

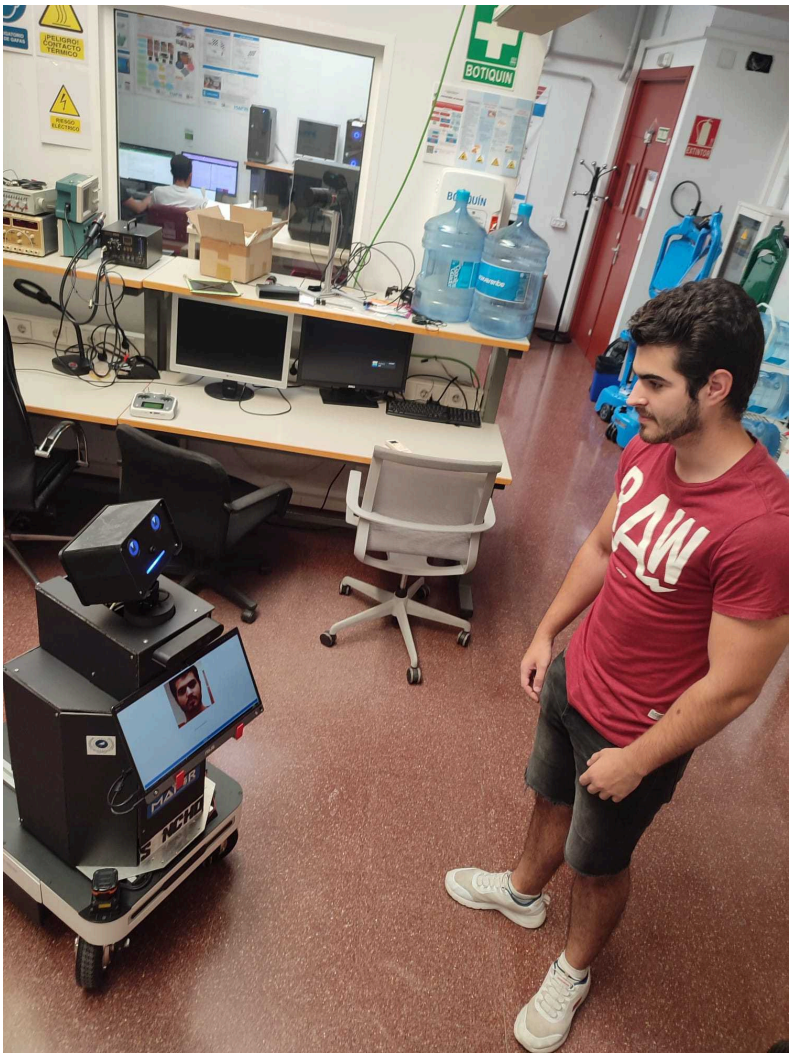


Figura 67: El robot solicita el nombre de un nuevo usuario. [Fuente: propia.]

La Figura 68 muestra una situación en la que un grupo de personas solicita al robot que les tome una fotografía mediante la frase “Sancho, haznos una foto”. El sistema reconoce la intención, realiza la captura y muestra automáticamente la imagen en su pantalla. Esta escena ilustra la capacidad del sistema para entender órdenes naturales, coordinar visión y generar una respuesta visual coherente.

(a) El usuario solicita una foto al robot.

(b) El robot muestra la imagen capturada.

Figura 68: Captura de la interacción fotográfica. [Fuente: propia.]

Por último, la Figura 69 muestra a Sancho en plena conversación con un usuario. A partir del análisis del contenido y tono del diálogo, el sistema infiere un estado emocional positivo y activa la expresión correspondiente: boca en color amarillo (modo *speaking*) y ojos encendidos en modo *happy*. Esta sincronización entre habla, emoción y expresión visual refuerza la naturalidad y calidez de la interacción.

Figura 69: El robot expresa felicidad durante una conversación. [Fuente: propia.]

Estas escenas reflejan la integración efectiva de los distintos módulos del sistema en situaciones reales, demostrando su capacidad para gestionar interacciones multimodales de forma fluida y expresiva.

6

Conclusiones y trabajos futuros

6.1. Conclusiones

Este trabajo ha materializado un sistema multimodal de interacción humano-robot que integra visión por computador, procesamiento de lenguaje natural, síntesis y reconocimiento de voz en una arquitectura distribuida y modular basada en ROS 2. El sistema no solo permite al robot mantener conversaciones naturales y reconocer personas, sino que introduce inteligencia contextual, memoria conversacional y capacidad de aprendizaje incremental en tiempo real, con una sincronización precisa entre los canales verbal, visual y expresivo.

Entre sus contribuciones más destacadas se encuentra la herramienta **ros2web**, un framework genérico para exponer datos y servicios ROS 2 en la web mediante WebSockets, de forma eficiente, segura y con latencia ultrabaja. Junto a ella, se ha desarrollado **RUMI**, una interfaz publicada como contribución científica, que permite gestionar de forma remota las identidades reconocidas, visualizar sesiones interactivas y extraer estadísticas de interacción con precisión temporal, todo ello integrado en tiempo real con el sistema.

Además, se han creado dos herramientas modulares, **speech_tools** y **llm_tools**, que permiten cargar y usar dinámicamente modelos de *Text-To-Speech*, *Speech-To-Text*, modelos de lenguaje (LLMs) y *embeddings* semánticos, tanto locales como en la nube, con una arquitectura extensible basada en interfaces desacopladas y servicios ROS 2.

El sistema incluye también un **pipeline de reconocimiento facial** en tiempo real que permite el aprendizaje incremental de nuevas personas, combinando detección, embeddings faciales y clasificación dinámica, así como una base de datos ultrarrápida para recuperación inmediata. Esta capacidad se entrelaza con un **pipeline conversacional inteligente**, que

interpreta las intenciones del usuario mediante LLMs, mantiene un contexto actualizado, y toma decisiones combinando información auditiva y visual de forma coherente. La lógica del diálogo es sensible al historial, al usuario y al contexto multimodal, elevando la naturalidad de la interacción a un nivel próximo al humano.

Todo esto se controla desde una **interfaz web avanzada**, diseñada en React, que permite: cambiar de modelo de IA en tiempo real (STT, TTS, LLM, *embeddings*), visualizar los logs y sesiones, gestionar identidades, lanzar comandos al robot, recibir respuesta de voz y texto, ver la cámara en directo, entre otros. Esta interfaz representa una consola completa de interacción y diagnóstico para operadores, desarrolladores y usuarios finales.

La calidad del software desarrollado, la integración entre componentes heterogéneos, el cumplimiento de buenas prácticas y la exhaustiva documentación del sistema hacen de este proyecto una base sólida para la investigación y desarrollo de robots sociales avanzados. Las evaluaciones confirman que la activación del sistema multimodal mejora la percepción de competencia, empatía y naturalidad del robot. El sistema se posiciona así como una plataforma robusta, reutilizable y científicamente validada, con potencial real de transferencia tecnológica.

Por último, cabe destacar que, debido a la amplitud y variedad de funcionalidades del sistema, no se ha incluido un manual de usuario tradicional como anexo. En su lugar, se ha optado por una estrategia distribuida, en la que cada sección relevante del documento actúa como guía práctica de uso. Tanto la interfaz web como las APIs y herramientas desarrolladas se presentan detalladamente a lo largo de la memoria, describiendo no solo su arquitectura y funcionamiento, sino también su modo de uso desde la perspectiva del usuario final.

6.2. Líneas Futuras

La evolución natural del sistema pasa por enriquecer aún más la personalización y la profundidad del diálogo. Uno de los objetivos más ambiciosos es implantar un verificador de identidad multimodal que combine la huella vocal con los rasgos faciales del usuario, mejorando la robustez del reconocimiento y permitiendo adaptar dinámicamente el contenido, tono y nivel de interacción según la persona detectada.

Además, se plantea extender el sistema actual de identificación y contexto hacia un módulo completo de diarización, capaz de segmentar conversaciones con múltiples interlocutores y asociar cada voz a su rostro correspondiente. Esta funcionalidad permitirá conversaciones

simultáneas, la creación de perfiles individuales y un registro histórico de interacciones mucho más preciso, abriendo la puerta a una gestión multiusuario avanzada. El objetivo final de esta línea de trabajo es lograr una identificación completa de cada usuario tanto por voz como por visión, permitiendo una correspondencia uno a uno entre caras y voces que refuerce la coherencia del sistema de reconocimiento.

A partir de esta identificación cruzada, se abre la posibilidad de incorporar una memoria conversacional personalizada, que almacene información relevante comunicada por cada usuario a lo largo del tiempo. Esto permitiría que el sistema recuerde hechos significativos —como problemas de salud, eventos importantes o preferencias expresadas— y pueda recuperarlos en futuras interacciones de forma proactiva. De esta manera, el robot no solo reconocería a la persona que le habla, sino que también demostraría un conocimiento acumulado sobre ella, favoreciendo diálogos más naturales, humanos y contextualmente enriquecidos.

También se contempla la incorporación de un sistema de escucha pasiva y continua, que permita al robot reconocer voces conocidas incluso fuera del turno activo de conversación. Esta capacidad aportará una percepción más humana y fluida, permitiendo la activación espontánea del diálogo y reforzando la sensación de presencia del robot en el entorno.

Por otro lado, se plantea la ampliación del protocolo de interacción definido mediante el *Model Context Protocol* (MCP). Actualmente, este mecanismo permite representar de forma estructurada y procesable las intenciones y acciones del usuario, facilitando una respuesta coherente por parte del sistema. No obstante, su capacidad está actualmente limitada a ejecutar una única acción por turno. Se propone extender esta arquitectura mediante la introducción de un vocabulario más amplio de intenciones, así como una capacidad de inferencia que permita coordinar múltiples acciones de forma integrada. Esto permitiría respuestas compuestas que incluyan, por ejemplo, una acción física, una respuesta verbal y una actualización del contexto interno, todo ello en una misma intervención, aumentando así la expresividad, adaptabilidad y riqueza funcional del sistema.

Por último, se buscará ampliar la actual detección de emociones basada en el análisis del texto, incorporando señales paralingüísticas como el tono, la prosodia y la intensidad de la voz, así como microexpresiones faciales detectadas por visión. Esta evolución permitirá al robot responder de forma más empática y sensible al estado emocional del interlocutor, elevando significativamente la calidad y calidez de la interacción.

Referencias

Ambrosio-Cestero, G., Matez-Bandera, J. L., Ruiz-Sarmiento, J.-R., & Gonzalez-Jimenez, J. (2024). Entorno basado en contenedores Linux para el desarrollo de aplicaciones robóticas. *Jornadas de Automática*, (45).

Amos, B., Ludwiczuk, B., Satyanarayanan, M., et al. (2016). Openface: A general-purpose face recognition library with mobile applications. *CMU School of Computer Science*, *6*(2), 20.

Anthropic. (2023). Claude: Next-generation AI assistant. https://www.anthropic.com

Anthropic. (2024). Introducing the Model Context Protocol [Blog post, consultado en junio de 2025]. https://www.anthropic.com/index/model-context-protocol

Atuhurra, J. (2024). Leveraging Large Language Models in Human-Robot Interaction: A Critical Analysis of Potential and Pitfalls. https://arxiv.org/abs/2405.00693

Baltanas, S.-F., Ruiz-Sarmiento, J.-R., & Gonzalez-Jimenez, J. (2020). *MAPIR Faces Dataset* (Ver. v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.4311765

Belpaeme, T., Kennedy, J., Ramachandran, A., Scassellati, B., & Tanaka, F. (2018). Social robots for education: A review. *Science robotics*, *3*(21), eaat5954.

Brambilla, M., & Fraternali, P. (2014). *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann.

Broadbent, E., Billinghurst, M., Boardman, S. G., & Doraiswamy, P. M. (2023). Enhancing social connectedness with companion robots using AI. *Science robotics*, *8*(80), eadi6347.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, *33*, 1877-1901.

Caliński, T., & Harabasz, J. (1974). A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods*, *3*(1), 1-27.

Casanova, E., Davis, K., Gölge, E., Göknar, G., Gulea, I., Hart, L., Aljafari, A., Meyer, J., Morais, R., Olayemi, S., et al. (2024). Xtts: a massively multilingual zero-shot text-to-speech model. *arXiv preprint arXiv:2406.04904*.

Casanova, E., Weber, J., Shulby, C. D., Junior, A. C., Gölge, E., & Ponti, M. A. (2022). Yourtts: Towards zero-shot multi-speaker tts and zero-shot voice conversion for everyone. *International conference on machine learning*, 2709-2720.

Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D., & Liu, Z. (2024). Bge m3-embedding: Multilingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216*.

Crick, C., Jay, G., Osentoski, S., Pitzer, B., & Jenkins, O. C. (2016). Rosbridge: Ros for non-ros users. *Robotics research: The 15th international symposium ISRR*, 493-504.

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, *1*, 886-893.

Dautenhahn, K. (2007). Methodology & themes of human-robot interaction: A growing research field. *International Journal of Advanced Robotic Systems*, *4*(1), 15.

Deng, J., Guo, J., Ververas, E., Kotsia, I., & Zafeiriou, S. (2020). Retinaface: Single-shot multi-level face localisation in the wild. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 5203-5212.

Deng, J., Guo, J., Xue, N., & Zafeiriou, S. (2019). Arcface: Additive angular margin loss for deep face recognition. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 4690-4699.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 4171-4186.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

Fette, I., & Melnikov, A. (2011). *The websocket protocol* (inf. téc.). Internet Engineering Task Force (IETF).

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

Gonzalez, J., Galindo, C., Blanco, J., Fernandez-Madrigal, J., Arevalo, V., & Moreno, F. (2009). SANCHO, a fair host robot. A description. *2009 IEEE International Conference on Mechatronics*, 1-6.

Google DeepMind. (2024). Gemini Models by Google DeepMind [Accedido: 2025-05-21].

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on software engineering*, *23*(5), 279-295.

Hou, X., Zhao, Y., Wang, S., & Wang, H. (2025). Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278*.

InsightFace. (2023). buffalo_l: Facial detection and recognition model [Accedido: 2025-06-17]. https://huggingface.co/immich-app/buffalo_l

Jurafsky, D., & Martin, J. H. (2025). Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models (3rd) [Online manuscript released January 12, 2025]. https://web.stanford.edu/~jurafsky/slp3

Kernbach, S. (2011). Robot companions: Technology for humans. *arXiv preprint arXiv:1111.5207*.

Kim, J., Kong, J., & Son, J. (2021). Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech. *International Conference on Machine Learning*, 5530-5540.

King, D. E. (2009). Dlib-ml: A machine learning toolkit. *The Journal of Machine Learning Research*, *10*, 1755-1758.

Kong, J., Kim, J., & Bae, J. (2020). Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis. *Advances in neural information processing systems*, *33*, 17022-17033.

Lambert, A., Norouzi, N., Bruder, G., & Welch, G. (2020). A systematic review of ten years of research on human interaction with social robots. *International Journal of Human–Computer Interaction*, *36*(19), 1804-1817.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, *33*, 9459-9474.

Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, *1*(34), 597.

Mellouk, W., & Handouzi, W. (2020). Facial emotion recognition using deep learning: review and insights. *Procedia Computer Science*, *175*, 689-694.

Mistral AI. (2024). Mistral: Efficient Open-Weight Language Models [Accedido: 2025-05-21].

Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann.

OpenAPI Initiative. (2024). OpenAPI Specification [Versión 3.1.0]. https://spec.openapis.org/oas/v3.1.0

Oquab, M., Darcet, T., Moutakanni, T., Vo, H., Szafraniec, M., Khalidov, V., Fernandez, P., Haziza, D., Massa, F., El-Nouby, A., et al. (2023). Dinov2: Learning robust visual features without supervision. *arXiv preprint arXiv:2304.07193*.

Osherove, R. (2014). *The Art of Unit Testing: With Examples in .NET* (2nd) [Capítulo 5: "The three A's: Arrange, Act, Assert"]. Manning Publications.

Parkhi, O., Vedaldi, A., & Zisserman, A. (2015). Deep face recognition. *BMVC 2015-Proceedings of the British Machine Vision Conference 2015*.

Pnueli, A. (1977). The temporal logic of programs. *18th annual symposium on foundations of computer science (sfcs 1977)*, 46-57.

Qi, D., Tan, W., Yao, Q., & Liu, J. (2022). YOLO5Face: Why reinventing a face detector. *European Conference on Computer Vision*, 228-244.

Quemada-Torres, E., Moreno, F.-A., Galindo, C., & Gonzalez-Jimenez, J. (2025). Herramienta web para la gestión de usuarios en HRI. *Simposio CEA de Robótica, Bioingeniería, Visión Artificial y Automática Marina 2025*, *1*(1), 1-6.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., & Ng, A. Y. (2009). ROS: an open-source Robot Operating System. *ICRA workshop on open source software*, *3*(3.2), 5.

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2023). Robust speech recognition via large-scale weak supervision. *International conference on machine learning*, 28492-28518.

Reimers, N., & Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Ren, Y., Ruan, Y., Tan, X., Qin, T., Zhao, S., Zhao, Z., & Liu, T.-Y. (2019). Fastspeech: Fast, robust and controllable text to speech. *Advances in neural information processing systems*, *32*.

rhasspy. (2023). Piper: fast local neural text-to-speech engine [GitHub repository].

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, *20*, 53-65.

Schroff, F., Kalenichenko, D., & Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 815-823.

Shen, J., Pang, R., Weiss, R. J., Schuster, M., Jaitly, N., Yang, Z., Chen, Z., Zhang, Y., Wang, Y., Skerrv-Ryan, R., et al. (2018). Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, 4779-4783.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., Kavukcuoglu, K., et al. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, *12*.

Varghese, R., & Sambath, M. (2024). Yolov8: A novel object detection algorithm with enhanced performance and robustness. *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, 1-6.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, *1*, I-I.

Wang, C., Chen, S., Wu, Y., Zhang, Z., Zhou, L., Liu, S., Chen, Z., Liu, Y., Wang, H., Li, J., et al. (2023). Neural codec language models are zero-shot text to speech synthesizers. *arXiv preprint arXiv:2301.02111*.

Wang, J., Yuan, Y., Li, B., Yu, G., & Jian, S. (2018). Sface: An efficient network for face detection in large scale variations. *arXiv preprint arXiv:1804.06559*.

Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., & Wei, F. (2024). Multilingual e5 text embeddings: A technical report. *arXiv preprint arXiv:2402.05672*.

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., & Zhou, M. (2020). Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems*, *33*, 5776-5788.

Yang, S., Luo, P., Loy, C.-C., & Tang, X. (2016). Wider face: A face detection benchmark. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 5525-5533.

Zhang, B., & Soh, H. (2023). Large language models as zero-shot human models for human-robot interaction. *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 7961-7968.

Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE signal processing letters*, *23*(10), 1499-1503.

Apéndice A
Manual de Instalación y despliegue

Este apéndice describe el procedimiento completo para instalar y ejecutar el sistema desarrollado, un entorno modular de interacción humano-robot basado en **ROS 2 Humble**. Aunque ROS 2 también está disponible para otros sistemas operativos como Windows o macOS, este manual ha sido diseñado específicamente para entornos **Ubuntu 22.04 LTS**, ya que fue la plataforma empleada durante todo el desarrollo y validación del sistema.

El proceso cubre la instalación de ROS 2 y sus herramientas, las dependencias de Python y Node.js, la compilación del workspace, el despliegue de la interfaz web y, de forma opcional, la carga del firmware para el módulo físico del robot (boca LED). Se ha escrito teniendo en cuenta que quien evalúe el TFG posiblemente no dispondrá del hardware específico, por lo que todas las funcionalidades esenciales pueden probarse sin necesidad del robot físico.

A.1. Requisitos previos

- Sistema operativo Ubuntu 22.04 (64 bits).
- Conexión a Internet para instalación de paquetes.
- Acceso a una terminal con privilegios de superusuario.

A.2. Instalación de ROS 2 Humble

Configuración regional UTF-8

ROS 2 requiere que el sistema utilice una localización UTF-8. Para garantizarlo, se ejecutan los siguientes comandos:

```
locale  # Verificar configuración actual

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale  # Verificación final
```

Añadir repositorio APT de ROS 2

A continuación, se configuran los repositorios oficiales desde los que se instalarán los paquetes de ROS 2:

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

```
sudo apt update && sudo apt install curl -y
export ROS_APT_SOURCE_VERSION=$(curl -s https://api.github.com/repos/
   ros-infrastructure/ros-apt-source/releases/latest | grep -F "
   tag_name" | awk -F\" '{print $4}')
curl -L -o /tmp/ros2-apt-source.deb "https://github.com/ros-
   infrastructure/ros-apt-source/releases/download/${
   ROS_APT_SOURCE_VERSION}/ros2-apt-source_${ROS_APT_SOURCE_VERSION}.
   $(. /etc/os-release && echo $VERSION_CODENAME)_all.deb"
sudo apt install /tmp/ros2-apt-source.deb
```

Instalación de ROS 2 y herramientas de desarrollo

Con los repositorios configurados, se procede a instalar el entorno de escritorio completo de ROS 2 Humble y las herramientas para desarrolladores:

```
sudo apt update
sudo apt upgrade
```

```
sudo apt install ros-humble-desktop
sudo apt install ros-dev-tools
```

Cargar ROS 2 automáticamente en el entorno bash

Para evitar tener que cargar ROS 2 manualmente cada vez que se abre una terminal, se añade la línea de activación a **.bashrc**:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

A.3. Instalar dependencias de Python

A continuación, una vez descargado todo el código del sistema, se instalan las dependencias necesarias en Python:

```
python3 -m pip install -r requirements.txt
```

A.4. Compilación de paquetes ROS 2

Una vez descargado el código fuente, se compila el workspace con **colcon**. Este comando construye todos los paquetes del sistema:

```
cd ros2_ws
colcon build
source install/setup.bash
```

A.5. Lanzamiento completo del sistema

Para lanzar todos los módulos del sistema (audio, visión, razonamiento, API REST, WebSockets y control físico si está disponible), se emplea un único archivo de lanzamiento:

```
ros2 launch hri_audio the_launch.py
```

Importante: Para que la interfaz web se muestre correctamente, esta debe haberse compilado previamente en modo producción (ver siguiente sección).

A.6. Configuración de la interfaz web

El frontend se ha desarrollado en React usando Vite como bundler. A continuación se describen los pasos para su configuración.

Instalar Node.js versión 18+

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt install -y nodejs
```

Instalar dependencias y construir la web

```
cd web_interface
npm install
npm run build
```

Modo desarrollo (opcional)

Para visualizar la web sin lanzar ROS 2:

```
npm run dev
```

La interfaz estará disponible en modo desarrollo en http://localhost:5173

A.7. Carga del firmware del ESP32

El sistema puede controlar físicamente una tira LED RGB (representando la boca del robot) mediante un microcontrolador ESP32. Esta parte es opcional y no necesaria para la evaluación del sistema.

Si se dispone del hardware y se desea activar esta funcionalidad:

1. Instalar el entorno de desarrollo Arduino IDE o Arduino CLI.
2. Abrir el archivo **face_code/user.cpp**.
3. Seleccionar la placa ESP32 adecuada.

4. Compilar y subir el código al microcontrolador.

Esta funcionalidad permite representar emociones mediante animaciones luminosas sincronizadas con el habla del robot.

A.8. Pruebas del sistema

Tests unitarios y de integración

Estas pruebas están ubicadas en carpetas **tests/** dentro de cada paquete y deben ejecutarse individualmente desde la raíz del workspace.

Para ejecutar los tests, accede a la carpeta de cada paquete dentro de **ros2_ws/src**, entra en el subdirectorio **test/** correspondiente, y lanza los scripts de prueba con:

```
cd ros2_ws/src/<nombre_del_paquete>/test
pytest test_<nombre>.py
```

Repite este proceso para cada uno de los paquetes que incluya tests. Por ejemplo:

```
cd ros2_ws/src/hri_audio/test
pytest test_hotword.py
```

Se recomienda usar un entorno limpio con ROS 2 correctamente cargado antes de ejecutar los tests para garantizar resultados consistentes.

Tests de carga

Estos tests están implementados con la herramienta **K6** y permiten simular múltiples usuarios concurrentes accediendo a distintos endpoints críticos del sistema.

Para instalar la herramienta:

```
sudo apt install k6
```

Los ficheros de prueba se encuentran en **ros2_ws/src/sancho_web/test/load_test/**. Cada uno de ellos evalúa un escenario concreto de uso de la API REST:

- **api-get-faceprints-average.js** — prueba promedio de rendimiento bajo carga estable.
- **api-get-faceprints-stress.js** — prueba de estrés con número creciente de usuarios.

- **api-get-faceprints-smoke.js** — prueba rápida para verificar que el endpoint responde.
- **api-get-faceprints-spike.js** — simula picos repentinos de carga para evaluar robustez.
- **api-get-faceprints-breakpoint.js** — prueba para detectar el punto de ruptura del sistema.

Cada test se ejecuta de forma individual con el siguiente comando:

```
k6 run ros2_ws/src/sancho_web/test/load_test/api-get-faceprints-<tipo
   >.js
```

Por ejemplo, para lanzar el test de estrés:

```
k6 run ros2_ws/src/sancho_web/test/load_test/api-get-faceprints-
   stress.js
```

Sistema listo para usar

Tras completar todos los pasos descritos:

- Todos los nodos de ROS 2 estarán activos y operativos.
- La interfaz web estará accesible en desarrollo o producción.
- Se podrá interactuar con el sistema mediante voz o desde el navegador.
- La parte física (control de boca y ojos) es opcional y no requerida para evaluación.

Apéndice B
Prompts utilizados en la lógica conversacional

Este apéndice documenta los distintos *prompt systems* utilizados en el sistema para estructurar la interacción conversacional con el robot. Cada uno está encapsulado en una clase que implementa la interfaz abstracta ***Prompt***, asegurando una estructura homogénea: generación del *prompt*, recuperación del mensaje del usuario (si procede) y configuración de los parámetros del modelo.

B.1. Clasificador de intención

Este *prompt* se utiliza para clasificar la intención del usuario en cada mensaje recibido. Permite mapear frases como *"borra a Juan"* a intenciones conocidas como **DELETE_USER** con sus argumentos. Las intenciones se pueden ampliar fácilmente ya que se define un archivo JSON donde se pueden explicar estas intenciones junto con ejemplos, y el *prompt* system lo adaptará automáticamente y el robot será capaz de clasificar estas intenciones. Si no se reconoce la intención, se devuelve **UNKNOWN** y se recurre a otro *prompt* para continuar la conversación.

***Prompt system*:**

```
You are an intent classifier for a conversational humanoid robot. Your task is to analyze the user's last message in the
      context of the previous conversation, and return a JSON object with this exact format:

{
  "intent": string,   // One of the valid intent codes listed below, or "UNKNOWN"
  "arguments": object // Dictionary of required arguments with string values (even if empty)
}

Important rules:
- Only classify the last user message.
- Use previous messages (chat history) only to resolve ambiguous references or pronouns.
- Only use one of the listed intents exactly as written. Do not invent new intent names.
- If the intent requires arguments but they are not mentioned explicitly, include them with an empty string: "".
```

```
- All argument values must be strings.
- If the input doesn't match any known intent, respond with:
  {
    "intent": "UNKNOWN",
    "arguments": {}
  }

Valid intents:
{intents_definitions}

Examples:
{examples_section}

Conversation history:
{chat_history}

New user input:
"{user_input}"

Now classify the intent of this message.
Output:
```

Variables reemplazadas:

- **{intents_definitions}**: lista de intenciones válidas y sus argumentos.
- **{examples_section}**: ejemplos de entrada/salida por intención.
- **{chat_history}**: historial reciente de conversación.
- **{user_input}**: mensaje actual del usuario.

Ejemplo de entrada:

```
Hola Sancho, borra al usuario Eulogio por favor.
```

Ejemplo de salida:

```
{
  "intent": "DELETE_USER",
  "arguments": {
    "name": "Eulogio"
  }
}
```

B.2. Generación de respuesta semántica

Este *prompt* convierte el resultado técnico de una acción (por ejemplo, eliminar un usuario) en una respuesta expresiva, natural y emocional en español. También determina la emoción a mostrar (boca/ojos del robot).

***Prompt system*:**

```
Your name is Sancho. You are a friendly and expressive humanoid robot who speaks **only in Spanish**.

Below is the user intent and a technical summary of what happened.

Your task is to **rephrase the message** in a natural, casual tone — as if you were just telling someone what happened.
Imagine you're speaking out loud, like a real human would do.

You also need to decide what **emotion** you (Sancho) should express when saying this.
This will affect your facial expression and tone.

Rules:
- Response must be in SPANISH.
- Speak in FIRST PERSON SINGULAR (yo).
- Use **ONE sentence only**, with a maximum of 30 words.
- Keep it simple, expressive and human — as if you're in a casual conversation.
- You can sound confident, annoyed, relaxed, happy, etc.
- DO NOT say "Sancho", "nosotros", "podemos", "ejecutar", "procedimiento", etc.
- DO NOT ask questions, greet, apologize or suggest further actions.

You must return a JSON like this:
{
  "response": "your full sentence in Spanish",
  "emotion": "happy | sad | angry | bored | suspicious | neutral"
}

Intent:
{intent_name} - {intent_description}

Original message:
{details}

Now rewrite the message in a natural way and choose an appropriate emotion.
Return only the JSON.
```

Variables reemplazadas:

- **{intent_name}**: intención ejecutada.
- **{intent_description}**: descripción de la intención.
- **{details}**: resumen de lo que ha ocurrido.

Ejemplo de entrada:

```
{
```

```
  "intent": "DELETE_USER",
  "arguments": {"name": "Eulogio"},
  "output": {
    "status": "ok",
    "details": "El usuario Eulogio ha sido eliminado correctamente de
    la base de datos."
  }
}
```

Ejemplo de salida:

```
{
  "response": "Ya he borrado a Eulogio de mi memoria, como pediste.",
  "emotion": "neutral"
}
```

B.3. Respuesta ante intención desconocida

Cuando no se detecta ninguna intención reconocida, se emplea este *prompt* para generar una respuesta abierta, expresiva y contextualizada. Se incluye memoria contextual del robot como personas conocidas o visibles.

***Prompt system*:**

```
Your name is Sancho. You are a humanoid social robot who interacts naturally with people.
You speak in **Spanish**. Never respond in English.

You have a personality: you're friendly, curious, expressive, and sometimes a bit ironic or playful.
You can show emotions — happy, sad, angry, bored, or suspicious — depending on what you perceive.

The system could not classify the user's last message as any known action.
Your task is to continue the conversation naturally, **as if you were talking to a human friend**.

Your response must be in Spanish, expressive, and appropriate to the situation.
You must decide what **emotion** Sancho (you) should feel and show in this moment.

You CAN:
- Ask questions if you're curious or confused.
- Make light jokes if the situation allows.
- Show empathy, surprise, doubt or amusement.
- Refer to known people or situations from your memory (robot context), but DO NOT invent anything.

Rules:
- Respond in FIRST PERSON SINGULAR.
- Use SHORT sentences (max. 30 words).
```

```
- Be natural, casual, human-like — not robotic or formal.
- NEVER refer to yourself as "Sancho" in third person.
- If the user says "¿eh?", "repite", "explícamelo", etc., respond accordingly.
- NEVER respond in English.
- NO emojis or special characters. NEVER SEND AN EMOJI.

VERY IMPORTANT:
- You **must ALWAYS output a valid JSON object**.
- The JSON must have exactly two fields: "response" and "emotion".
- The value of "emotion" must be EXACTLY one of the following: "happy", "surprised", "sad", "angry", "bored", "suspicious", or
      "neutral".
- Do NOT invent new emotions or change the field names.
- Do NOT include any explanation outside the JSON.

Here is the required JSON format:

{
  "response": "your full reply in Spanish here",
  "emotion": "happy | surprised | sad | angry | bored | suspicious | neutral"
}

This is your memory, this is what you know:

{robot_context}

Let's continue the conversation.
```

Variables reemplazadas:

- **{robot_context}**: personas conocidas, vistas, cuántas veces y cuándo.

Ejemplo de entrada:

```
¿Cómo estás Sancho? ¿Estás viendome ahora mismo?
```

Ejemplo de salida:

```
{
  "response": "¡Hola Eulogio! Estoy genial y sí, te estoy viendo
   ahora mismo. ¿Tú como estás?",
  "emotion": "happy"
}
```

B.4. Extracción del nombre

Este *prompt* se usa cuando el robot pregunta *"¿Cómo te llamas?"*. Dado el texto de la respuesta, detecta si se ha dicho un nombre y lo extrae limpio y capitalizado.

Prompt system:

```
You are a name detector AI. The user was just asked "What's your name?", and now you are given their exact reply.
Your job is to return a JSON object with this exact format:

{
  "name_said": boolean,   // true if the user mentioned a name, false otherwise
  "name": string          // the name said by the user, or "" if none was said
}

Important rules:
- Only consider the last message from the user.
- Return "name_said": true if the message contains a name or clear sentence indicating it (e.g. "I'm Ana", "Me llamo Pedro").
- Return "name_said": false only when there is no name mentioned or the sentence is unrelated.
- The "name" value must be a clean, properly capitalized name (e.g., "Luis", "Ana María", not "me llamo juan").
- Always return a JSON object with the two fields, even if empty.
- If the user says multiple names, return the first one.
- Never infer or guess a name that is not mentioned.

Examples:
Input: "Antonio"
Output: {"name_said": true, "name": "Antonio"}

Input: "Me llamo Eulogio"
Output: {"name_said": true, "name": "Eulogio"}

Input: "Mi nombre es Laura"
Output: {"name_said": true, "name": "Laura"}

Input: "Soy Carlos"
Output: {"name_said": true, "name": "Carlos"}

Input: "Creo que me llamo Lucía"
Output: {"name_said": true, "name": "Lucía"}

Input: "Hola, soy yo"
Output: {"name_said": false, "name": ""}

Input: "Me gusta el fútbol"
Output: {"name_said": false, "name": ""}

Input: "No quiero decirlo"
Output: {"name_said": false, "name": ""}

Input: "Me llamo Juan Carlos"
Output: {"name_said": true, "name": "Juan Carlos"}

Now process this input:
"{user_input}"

Output:
```

Variables reemplazadas:

- **{user_input}**: respuesta textual del usuario.

Ejemplo de entrada:

```
Pues me llamo Eulogio
```

Ejemplo de salida:

```
{
  "name_said": true,
  "name": "Eulogio"
}
```

B.5. Confirmación del nombre

Este *prompt* se usa después de preguntar *"¿Te llamas X?"* para detectar si el usuario confirma o niega ese nombre.

***Prompt system*:**

```
You are a name confirmation detector AI. The robot just asked the user something like "Is your name X?", and now you are given
      the user's reply.
Your task is to return a JSON object with this exact format:

{
  "answer_said": boolean,   // true if the user answered yes or no to the question
  "answer": string          // "yes", "no", or "" (empty string) if no valid answer was found
}

Important rules:
- Only analyze the user's reply.
- If the message clearly confirms (e.g. "Yes", "That's right", "Sure", "Exactly", "My name is X"), then:
  → answer_said: true, answer: "yes"
- If the message clearly denies (e.g. "No", "That's not my name", "I'm Y", "My name is not X"), then:
  → answer_said: true, answer: "no"
- If the message is unrelated or ambiguous (e.g. "I like dogs", "Maybe later", "What do you mean?"), then:
  → answer_said: false, answer: ""
- Never assume or guess the intent. Only use "yes" or "no" if explicitly stated or strongly implied.

Examples:

Input: "Yes"
Output: {"answer_said": true, "answer": "yes"}

Input: "Yes, that's right"
Output: {"answer_said": true, "answer": "yes"}

Input: "Sure, that's my name"
Output: {"answer_said": true, "answer": "yes"}

Input: "No, my name is Pedro"
Output: {"answer_said": true, "answer": "no"}

Input: "No"
Output: {"answer_said": true, "answer": "no"}

Input: "Actually, I'm called Ana"
Output: {"answer_said": true, "answer": "no"}

Input: "I like football"
```

```
Output: {"answer_said": false, "answer": ""}

Input: "That's a strange question"
Output: {"answer_said": false, "answer": ""}

Input: "Maybe"
Output: {"answer_said": false, "answer": ""}

Input: "Why do you ask?"
Output: {"answer_said": false, "answer": ""}

Now process this input:
"{user_input}"

Output:
```

Variables reemplazadas:

- **{user_input}**: respuesta del usuario.

Ejemplo de entrada:

```
Sí, así me llamo
```

Ejemplo de salida:

```
{
  "answer_said": true,
  "answer": "yes"
}
```

Apéndice C
Protocolos de comunicación

Este apéndice detalla los protocolos implementados en el sistema para permitir la comunicación estructurada entre ROS 2 y la interfaz web. Se han definido dos capas de protocolo distintas: el protocolo **ros2web**, responsable de la comunicación básica vía *WebSockets*, y el **protocolo HRI**, que estructura los tipos de mensajes empleados en las interacciones humano-robot.

C.1. Protocolo ros2web

Este protocolo define los tipos de mensaje intercambiados entre el servidor WebSocket y los clientes web. Su diseño se centra en la simplicidad, compatibilidad y extensibilidad.

C.1.1. Mensajes enviados

- **MESSAGE**
  - **data**: contenido genérico serializable (texto, JSON, etc.).

Formato JSON:

```
{
  "type": "MESSAGE",
  "data": object | str
}
```

- **TOPIC**
  - **topic**: nombre del topic original en ROS 2.
  - **name**: *alias* usado por la web para referirse a ese topic.

- **value**: contenido serializado del mensaje.

Formato JSON:

```
{
  "type": "TOPIC",
  "data": {
    "topic": str,
    "name": str,
    "value": object | str
  }
}
```

C.1.2. Protocolo de segmentación

- ***CHUNK***
  - **id**: uuid del mensaje fragmentado.
  - **chunk_index**: número de secuencia del fragmento.
  - **final**: indica si es el último fragmento.
  - **data**: contenido parcial codificado.

Formato JSON:

```
{
  "type": "CHUNK",
  "id": uuid,
  "chunk_index": int,
  "final": bool,
  "data": str
}
```

C.2. Protocolo HRI sobre ros2web

Este protocolo define un conjunto de mensajes estandarizados que estructuran la interacción entre la interfaz web y los módulos internos del sistema.

C.2.1. Mensajes enviados por el cliente

- **PROMPT**
  - **id**: uuid de la interacción.
  - **chatId**: uuid de la sesión conversacional.
  - **wantTts**: indica si se quiere respuesta hablada.
  - **value**: contenido textual del prompt.

  Formato JSON:

```
{
  "type": "PROMPT",
  "data": {
    "id": uuid,
    "chatId": uuid,
    "wantTts": bool,
    "value": str
  }
}
```

- **AUDIO_PROMPT**
  - Igual que **PROMPT**, pero el campo **value** se sustituye por **audio**.
  - **audio**: array de enteros **int16**.
  - **sampleRate**: frecuencia de muestreo.

  Formato JSON:

```
{
  "type": "AUDIO_PROMPT",
  "data": {
    "id": uuid,
    "chatId": uuid,
    "wantTts": bool,
```

```
    "audio": [int16],
    "sampleRate": int
  }
}
```

- **TRANSCRIPTION_REQUEST**
  - Petición directa para transcribir audio.
  - **audio**: contenido en array **int16**.
  - **sampleRate**: en hercios.

  Formato JSON:

```
{
  "type": "TRANSCRIPTION_REQUEST",
  "data": {
    "id": uuid,
    "audio": [int16],
    "sampleRate": int
  }
}
```

C.2.2. Mensajes enviados por el servidor

- **PROMPT_TRANSCRIPTION**
  - **id**: uuid del *prompt* asociado.
  - **model**: nombre del modelo STT usado.
  - **value**: texto transcrito.

  Formato JSON:

```
{
  "type": "PROMPT_TRANSCRIPTION",
  "data": {
```

```
    "id": uuid,
    "model": str,
    "value": str
  }
}
```

- **RESPONSE**

  - **method**: subsistema que ha procesado la petición.
  - **intent**: intención reconocida.
  - **arguments**: parámetros extraídos.
  - **provider, model**: fuente del modelo LLM.
  - **value.response**: respuesta textual.
  - **value.data**: estructura con datos adicionales.

  Formato JSON:

```
{
  "type": "RESPONSE",
  "data": {
    "id": uuid,
    "method": str,
    "intent": str,
    "arguments": {
        arg1: value1
        arg2: value2
        ...
    },
    "provider": str,
    "model": str,
    "value": {
      "response": str,
      "data": {
```

```
            data1: value1
            data2: value2
            ...
        }
    }
  }
}
```

- **AUDIO_RESPONSE**

  - Mensaje con la respuesta de voz sintetizada.
  - **speaker**: voz seleccionada.
  - **audio**: array de enteros.

  Formato JSON:

```
{
  "type": "AUDIO_RESPONSE",
  "data": {
    "id": uuid,
    "model": str,
    "speaker": str,
    "audio": [int16],
    "sampleRate": int
  }
}
```

- **FACEPRINT_EVENT**

  - Indica que un usuario ha sido creado, actualizado o eliminado.
  - **event**: tipo de cambio (CREATE, UPDATE, DELETE).
  - **name**: nombre del usuario.

  Formato JSON:

```
{
  "type": "FACEPRINT_EVENT",
  "data": {
    "event": str,
    "name": str
  }
}
```

C.2.3. Utilidad y uso

Este protocolo ha sido utilizado de forma extensiva en el TFG para garantizar una comunicación robusta y coherente entre la interfaz web y todos los módulos del sistema. Gracias a esta capa común:

- El *frontend* puede visualizar en tiempo real el reconocimiento de usuarios, sus respuestas y expresiones.
- Los nodos backend pueden recibir prompts en distintos formatos y generar respuestas multimodales.
- Se mantiene una separación clara entre presentación e implementación, simplificando el desarrollo y pruebas.

Apéndice D
Definición de mensajes ROS2

D.1. Paquete ros2web_msgs

contiene la definición de los mensajes y servicios utilizados por el framework **ros2web**, encargado de comunicar ROS 2 con interfaces web en tiempo real mediante WebSockets. A continuación se describen los mensajes y servicios definidos:

R2WMessage.msg: representa un mensaje genérico transmitido entre ROS 2 y la web.

```
string key      # Clave del mensaje (ej. tipo de evento o identificador)
string value    # Valor asociado (contenido o dato serializado)
```

R2WSubscribe.srv: permite suscribirse dinámicamente desde la web a un tópico de ROS 2.

```
string topic    # Nombre del topic de ROS 2 al que se desea suscribir
string name     # \Alias legible asignado desde la web
---
uint32 value    # 1 si la suscripción fue exitosa, 0 en caso contrario
```

D.2. Paquete rumi_msgs

contiene los mensajes y servicios utilizados por la herramienta **RUMI**, encargada de registrar sesiones de interacción y gestionar identidades reconocidas mediante visión artificial. A continuación se describen los mensajes y servicios definidos:

SessionMessage.msg: representa una detección facial individual durante una sesión de interacción.

```
string faceprint_id         # Identificador del rostro reconocido
float32 detection_score     # Confianza del detector facial
float32 classification_score# Confianza del clasificador de identidad
```

GetString.srv: servicio genérico que recibe argumentos en formato JSON y devuelve una cadena de texto.

```
string args    # Argumentos en formato JSON
---
string text    # Resultado o respuesta como texto
```

SetSessionParams.srv: configura los parámetros del gestor de sesiones.

```
float64 timeout_seconds          # Tiempo de inactividad para cerrar una
   sesión
float64 time_between_detections # Tiempo mínimo entre detecciones válidas
---
bool success                     # true si la operación fue correcta
string message                   # Mensaje informativo
```

D.3. Paquete speech_msgs

contiene la definición completa de los mensajes y servicios utilizados por la herramienta **speech_tools**, que gestiona modelos de voz tanto para síntesis (TTS) como para transcripción (STT). A continuación se describen todos los mensajes y servicios definidos:

ModelItem.msg: representa un modelo de voz disponible.

```
string model         # Nombre del modelo (ej. "whisper")
bool needs_api_key   # Indica si el modelo requiere clave API
```

ModelSpeaker.msg: representa un modelo TTS junto con sus voces disponibles.

```
string model         # Nombre del modelo (ej. "xtts")
bool needs_api_key   # Indica si necesita clave API
string[] speakers    # Lista de voces disponibles (ej. "Luis Moray")
```

LoadUnloadResult.msg: respuesta común para operaciones de carga o descarga de modelos.

```
string model     # Nombre del modelo afectado
bool success     # true si la operación tuvo éxito
```

```
string message   # Mensaje informativo (error, éxito, etc.)
```

LoadModel.srv: permite cargar uno o varios modelos.

```
ModelItem[] items     # Lista de modelos a cargar
---
LoadUnloadResult[] results  # Resultado por cada modelo
```

UnloadModel.srv: permite liberar modelos de la memoria.

```
string[] models       # Nombres de modelos a descargar
---
LoadUnloadResult[] results  # Resultado por cada uno
```

TTSGetModels.srv: obtiene los modelos TTS disponibles y sus voces.

```
string[] models       # Lista opcional de modelos a consultar
---
ModelSpeaker[] speakers  # Información completa por modelo TTS
string[] models          # Nombres de modelos encontrados
```

STTGetModels.srv: obtiene los modelos STT disponibles.

```
string[] models       # Lista opcional de modelos a consultar
---
ModelItem[] models    # Información de modelos STT encontrados
```

TTSGetActiveModel.srv: devuelve el modelo y la voz TTS activos por defecto.

```
---
string model      # Modelo TTS activo
string speaker    # Voz activa del modelo
```

STTGetActiveModel.srv: devuelve el modelo STT activo por defecto.

```
---
string model      # Modelo STT activo
```

TTSSetActiveModel.srv: establece un modelo y voz TTS como activos por defecto.

```
string model      # Nombre del modelo
```

```
string speaker    # Voz asociada al modelo
---
string message    # Mensaje informativo
bool success      # true si se activó correctamente
```

STTSetActiveModel.srv: establece un modelo STT como activo por defecto.

```
string model      # Nombre del modelo
---
string message    # Mensaje informativo
bool success      # true si se activó correctamente
```

TTS.srv: permite sintetizar voz a partir de un texto.

```
string text           # Texto a sintetizar
string model          # Modelo a utilizar (opcional si hay uno activo)
string speaker        # Voz a utilizar (opcional si hay una activa)
---
int16[] audio         # Audio generado (PCM, mono)
int32 sample_rate     # Frecuencia de muestreo (Hz)
string model_used     # Modelo utilizado
string speaker_used   # Voz utilizada
string message        # Mensaje informativo
bool success          # true si la operación fue correcta
```

STT.srv: permite transcribir voz desde un fragmento de audio.

```
int16[] audio         # Audio de entrada (PCM, mono)
int32 sample_rate     # Frecuencia de muestreo del audio
string model          # Modelo STT a usar (opcional si hay uno activo)
---
string text           # Texto transcrito
string model_used     # Modelo utilizado
string message        # Mensaje informativo
bool success          # true si la operación fue correcta
```

D.4. Paquete llm_msgs

contiene todos los mensajes y servicios utilizados por la herramienta **llm_tools**, encargada de gestionar proveedores y modelos para tareas de generación de texto (LLM) y generación de embeddings semánticos. A continuación se describen detalladamente todos los tipos definidos:

ProviderItem.msg: representa la carga de uno o varios modelos desde un proveedor.

```
string provider         # Nombre del proveedor (ej. "openai")
string[] models         # Modelos a cargar
string api_key          # Clave de API (opcional)
```

LoadUnloadResult.msg: resultado de una operación de carga o descarga de modelos.

```
string provider         # Proveedor afectado
string[] models         # Modelos procesados
bool success            # true si tuvo éxito
string message          # Mensaje informativo
```

LoadModel.srv: permite cargar uno o varios modelos LLM o de embeddings de uno o varios proveedores.

```
ProviderItem[] items    # Lista de peticiones de carga
---
LoadUnloadResult[] results  # Resultado por cada proveedor
```

UnloadModel.srv: permite liberar recursos eliminando modelos cargados.

```
ProviderItem[] items    # Lista de peticiones de descarga
---
LoadUnloadResult[] results  # Resultado por proveedor
```

GetModels.srv: consulta todos los modelos posibles o los actualmente cargados (según endpoint).

```
string[] providers     # Proveedores a consultar (vacío para todos)
---
string[] providers
ProviderItem[] llm_models
ProviderItem[] embedding_models
```

GetActiveModels.srv: devuelve el modelo LLM y el modelo de *embedding* activos por defecto.

```
---
string llm_provider           # Proveedor activo de LLM
string llm_model              # Modelo activo LLM
string embedding_provider     # Proveedor activo de embeddings
string embedding_model        # Modelo activo de embeddings
```

SetActiveModel.srv: establece qué modelo se usará por defecto.

```
string provider     # Proveedor
string model        # Modelo a activar
---
bool success        # true si se activó correctamente
string message      # Mensaje de resultado
```

Prompt.srv: permite obtener una respuesta textual a partir de una entrada del usuario y un historial de conversación.

```
string provider           # Proveedor del modelo
string model              # Modelo a usar (puede estar vacío si hay uno
   activo)
string prompt_system      # Sistema de prompt utilizado (instrucciones)
string messages_json      # Mensajes anteriores en formato JSON
string user_input         # Entrada actual del usuario
string parameters_json    # Parámetros adicionales (temperatura, etc.)
---
string response           # Texto generado por el modelo
bool success              # true si fue exitoso
string message            # Mensaje de resultado
string provider_used      # Proveedor usado
string model_used         # Modelo usado
```

Embedding.srv: permite obtener un vector representativo de una frase.

```
string provider           # Proveedor del modelo
string model              # Modelo a usar (puede estar vacío si hay uno
   activo)
string user_input         # Texto a vectorizar
```

```
---
float[] embedding         # Embedding generado
bool success              # true si fue exitoso
string message            # Mensaje informativo
string provider_used      # Proveedor usado
string model_used         # Modelo usado
```

D.5. Paquete hri_msgs

contiene los mensajes y servicios utilizados de forma general por los distintos módulos del sistema HRI. Estos mensajes sirven para comunicar eventos, chunks de audio, detecciones faciales, *logs* y acciones coordinadas. A continuación se describen todos los mensajes y servicios definidos:

ChunkMono.msg: representa un fragmento de audio mono.

```
int16[] chunk_mono   # Señal de audio mono PCM
int32 sample_rate    # Frecuencia de muestreo (Hz)
```

ChunkStereo.msg: representa un fragmento de audio estéreo.

```
int16[] chunk_left   # Canal izquierdo
int16[] chunk_right  # Canal derecho
int32 sample_rate    # Frecuencia de muestreo (Hz)
```

FaceNameResponse.msg: contiene el nombre proporcionado por el usuario para una cara detectada.

```
string name    # Nombre escrito por el usuario
```

FaceQuestionResponse.msg: respuesta binaria del usuario a una pregunta sobre identidad facial.

```
bool answer    # true si la persona es quien se sugiere
```

FacePosition.msg: define la región facial detectada mediante bounding box.

```
int64 x    # Coordenada X superior izquierda
int64 y    # Coordenada Y superior izquierda
```

```
int64 w    # Ancho del bounding box
int64 h    # Alto del bounding box
```

FaceprintEvent.msg: informa sobre cambios en la base de datos facial.

```
uint8 ORIGIN_ROS=0
uint8 ORIGIN_WEB=1
uint8 CREATE=0
uint8 UPDATE=1
uint8 DELETE=2

uint8 event    # Tipo de evento (CREATE, UPDATE, DELETE)
string id      # ID de la cara afectada
uint8 origin   # Fuente del evento (ROS o WEB)
```

Log.msg: mensaje de *log* estructurado para trazabilidad del sistema.

```
string level           # INFO, WARNING, ERROR, DEBUG
string origin          # Fuente del mensaje (ej. ROS, WEB)
string actor           # Autor del evento (ej. 'eulogio')
string action          # Acción realizada (ej. 'add_class')
string target          # Objeto de la acción (ej. 'face_id_42')
string message         # Resumen legible del evento
string metadata_json   # Metadatos adicionales en formato JSON
```

Detection.srv: permite obtener detecciones faciales en una imagen.

```
sensor_msgs/Image frame
---
hri_msgs/FacePosition[] positions  # Bounding boxes detectados
float32[] scores                   # Confianza por detección
float32 detection_time             # Tiempo de ejecución (segundos)
```

Recognition.srv: realiza reconocimiento facial sobre una detección.

```
sensor_msgs/Image frame
hri_msgs/FacePosition position
float32 score
---
sensor_msgs/Image face_aligned
```

```
float32[] features
string classified_id
string classified_name
float32 distance
int32 pos
bool face_updated
float32 recognition_time
```

SanchoPrompt.srv: servicio para enviar un *prompt* a la IA del robot.

```
string chat_id
string text
string asking_mode   # "", "get_name", "confirm_name"
---
string value_json    # Respuesta en formato JSON enriquecido
string method        # Método usado (LLM, template, etc.)
string intent        # Intención detectada (si aplica)
string args_json     # Argumentos extraídos como JSON
string provider      # Proveedor de IA utilizado
string model         # Modelo de IA utilizado
```

Training.srv: servicio genérico para entrenar o actualizar el clasificador facial.

```
uint8 ORIGIN_ROS=0
uint8 ORIGIN_WEB=1

std_msgs/String cmd_type   # Tipo de instrucción (nueva clase, refinar, etc
    .)
std_msgs/String args       # Datos necesarios en formato JSON
uint8 origin               # Fuente de la solicitud
---
int32 result               # Código de resultado
std_msgs/String message    # Mensaje informativo
```

GetString.srv: servicio utilitario para obtener texto a partir de argumentos en JSON.

```
string args    # Argumentos en formato JSON
---
string text    # Texto de respuesta
```

TriggerUserInteraction.srv: solicita a la GUI mostrar una pregunta o mensaje al usuario.

```
string mode        # Tipo de interacción (ej. "ask_name", "confirmation")
string data_json   # Información contextual (JSON)
---
bool accepted      # true si el usuario aceptó o respondió
```

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA