Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Aplicación de una Arquitectura de Agentes de Servicios de IA Generativa

Autor: Javier Balza Díaz

Tutor: Antonio Jesús Díaz Honrubia

Madrid, junio de 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Aplicación de una Arquitectura de Agentes de Servicios de IA Generativa
junio de 2025

Autor: Javier Balza Díaz

Tutor:
Antonio Jesús Díaz Honrubia
Lenguajes y Sistemas Informáticos e Ingeniería de software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

La gestión eficiente del tiempo se ha convertido en un desafío crítico en el contexto actual, donde las personas enfrentan un ritmo de vida acelerado que exige equilibrar diversas responsabilidades, como el trabajo, los estudios, las actividades personales y las tareas del hogar. Muchas herramientas de planificación tradicionales, como agendas físicas o aplicaciones genéricas, requieren una entrada manual de datos que resulta tediosa y poco flexible, especialmente para usuarios no técnicos, lo que genera estrés, desorganización y una menor productividad. Este problema enfatiza la necesidad de soluciones innovadoras que automaticen la organización de tareas y se adapten de manera dinámica a los horarios y necesidades cambiantes de los usuarios, mejorando su bienestar y eficiencia diaria.

Paralelamente, el auge de la Inteligencia Artificial (IA) ha transformado la forma en que interactuamos con la tecnología, aunque su uso se ha centrado principalmente en aplicaciones conversacionales, como asistentes virtuales o chatbots, que responden preguntas o generan texto. Sin embargo, el potencial de la IA va más allá, pudiendo aplicarse a la creación de sistemas multiagentes que realicen tareas complejas de manera autónoma, como la planificación y gestión de actividades. Este enfoque, que combina modelos de lenguaje de gran escala (*LLM* por sus siglas en inglés) con agentes especializados, abre nuevas posibilidades para resolver problemas cotidianos de manera más eficiente e intuitiva, aprovechando la capacidad de la IA para entender y procesar instrucciones en lenguaje natural.

Este Trabajo de Fin de Grado (TFG) desarrolla el *Organizador de Tareas Inteligente*, una solución que utiliza IA generativa y una arquitectura multiagente para automatizar la planificación de tareas, permitiendo a los usuarios organizar sus actividades mediante instrucciones simples, como “sacar al perro esta tarde”. El sistema interpreta el contexto, asigna horarios evitando conflictos y ajusta las tareas dinámicamente, ofreciendo una herramienta accesible que reduce el esfuerzo manual y se adapta a horarios complejos, como los de un estudiante con clases, deportes y quehaceres domésticos. De esta forma, el TFG aborda la problemática de la gestión del tiempo, proporcionando una solución práctica que mejora la productividad y el bienestar del usuario.

Abstract

Efficient time management has become a critical challenge in today's context, where people face a fast-paced lifestyle that requires balancing various responsibilities such as work, studies, personal activities, and household tasks. Traditional planning tools, like physical agendas or generic applications, demand manual data entry that is tedious and inflexible, especially for non-technical users, leading to stress, disorganization, and reduced productivity. This issue underscores the need for innovative solutions that automate task organization and dynamically adapt to users' changing schedules and needs, enhancing their daily well-being and efficiency.

Parallel to this, the rise of Artificial Intelligence (AI) has transformed how we interact with technology, though its use has primarily focused on conversational applications, such as virtual assistants or chatbots, that answer questions or generate text. However, AI's potential extends further, enabling the creation of multi-agent systems that autonomously perform complex tasks, such as activity planning and management. This approach, which combines Large Language Models (LLM) with specialized agents, opens new possibilities for addressing everyday problems more efficiently and intuitively, leveraging AI's ability to understand and process natural language instructions.

This Final Degree Project (FDP) develops the *Intelligent Task Organizer*, a solution that uses generative AI and a multi-agent architecture to automate task planning, allowing users to organize their activities through simple instructions, such as "take the dog out this afternoon." The system interprets the context, assigns schedules avoiding conflicts, and dynamically adjusts tasks, providing an accessible tool that reduces manual effort and adapts to complex schedules, such as those of a student with classes, sports, and household chores. In this way, the FDP addresses the time management problem, offering a practical solution that improves user productivity and well-being.

Índice general

Índice de figuras

1 Introducción

En la actualidad, la gestión eficiente de las tareas diarias representa un desafío significativo para muchas personas, dado el ritmo acelerado de la vida moderna y la necesidad de equilibrar diversas responsabilidades, como el trabajo, los estudios y las actividades personales. Este contexto ha impulsado el desarrollo de herramientas tecnológicas que buscan simplificar la planificación y organización del tiempo. Los avances en inteligencia artificial, especialmente en el procesamiento de lenguaje natural y los sistemas multiagentes [1][2], han abierto nuevas posibilidades para crear soluciones que automaticen y optimicen la gestión de tareas, adaptándose de manera inteligente a las preferencias y necesidades de los usuarios. Estas tecnologías permiten a los sistemas entender instrucciones en lenguaje natural y organizarlas de forma autónoma, ofreciendo una experiencia más intuitiva y eficiente. En este Trabajo de Fin de Grado (TFG), se propone el desarrollo de un sistema basado en una arquitectura de agentes que aproveche estas capacidades para procesar entradas del usuario, analizar el contexto de las tareas, generar cronogramas y ajustarlos dinámicamente, todo ello implementado de manera local para garantizar autonomía y simplicidad en su uso.

1.1 Motivación y justificación

La motivación principal de este TFG surge de la necesidad de abordar el problema de la gestión del tiempo en un entorno donde las personas enfrentan dificultades para organizar sus actividades diarias de manera efectiva. Muchas veces, las herramientas de planificación tradicionales, como agendas físicas o aplicaciones genéricas, requieren que el usuario invierta un tiempo considerable en introducir y organizar manualmente sus tareas, lo que puede resultar tedioso y poco práctico, especialmente para quienes manejan múltiples responsabilidades. Además, estas soluciones suelen carecer de flexibilidad para adaptarse a imprevistos, como cambios de horario o nuevas prioridades, lo que genera estrés y desorganización. Este proyecto busca resolver dicho problema mediante el desarrollo de un sistema que automatice la planificación y ajuste de tareas, permitiendo a los usuarios dedicar menos tiempo a la organización y más a la ejecución de sus actividades, mejorando así su productividad y bienestar.

Otro aspecto que motivó este trabajo fue el deseo de explorar el potencial de los modelos de lenguaje avanzados, como *Gemini-2.0-Flash* [3], de una manera innovadora. En lugar de utilizarlos únicamente para tareas conversacionales, como responder preguntas o generar texto, este TFG propone aprovechar su capacidad para interpretar lenguaje natural dentro de un sistema multiagente que planifique y gestione tareas de forma autónoma. El sistema desarrollado está compuesto por cuatro componentes clave: el *EvaluatorAgent*, que analiza las instrucciones del usuario; el *RootAgent*, que organiza el flujo de trabajo y actualiza el calendario; el *PlanningAgent*, que programa las tareas; y el *AdjusterAgent*, que modifica el cronograma según las necesidades del usuario. Los datos se gestionan localmente mediante archivos JSON y un calendario interno, eliminando dependencias externas para garantizar autonomía. Además, se implementó una interfaz gráfica con *Streamlit* para facilitar la interacción y validar el sistema en un entorno real.

Este TFG pretende resolver el problema de la planificación manual ofreciendo una solución que combine la inteligencia de los *LLM* con una arquitectura modular de agentes, capaz de entender instrucciones en lenguaje natural y generar cronogramas adaptativos. Al hacerlo, no solo se reduce el esfuerzo del usuario, sino que también se proporciona una herramienta flexible que puede ajustarse a cambios imprevistos, como posponer una tarea por un imprevisto, y que puede integrarse en diferentes aplicaciones gracias a su diseño escalable. Este enfoque innovador busca contribuir al campo de la IA generativa, demostrando cómo los sistemas multiagentes pueden aplicarse a problemas cotidianos de manera práctica y eficiente.

1.2 Objetivos

Teniendo en cuenta la motivación expresada anteriormente, este TFG tiene como objetivo principal desarrollar un *framework* basado en una arquitectura de agentes de servicios de IA generativa, permitiendo la integración modular de herramientas en diferentes aplicaciones.

Para alcanzar este objetivo, se han definido los siguientes objetivos específicos:

1. Analizar y seleccionar el entorno tecnológico más adecuado para el desarrollo del *framework*.
2. Diseñar la arquitectura del sistema definiendo la estructura y las interacciones entre los agentes.
3. Implementar el *framework* con los agentes de IA generativa asegurando su funcionalidad y escalabilidad.
4. Desarrollar un caso de uso que valide la utilidad y aplicación del *framework* en un entorno real.
5. Realizar pruebas para evaluar el desempeño, eficiencia y facilidad de integración del sistema.

1.3 Planificación seguida

La realización de este Trabajo de Fin de Grado (TFG) ha seguido una planificación estructurada para garantizar el cumplimiento de los objetivos establecidos dentro del plazo establecido. A continuación, se detalla el listado de tareas realizadas y un diagrama de *Gantt* que refleja la cronología y duración aproximada de cada tarea.

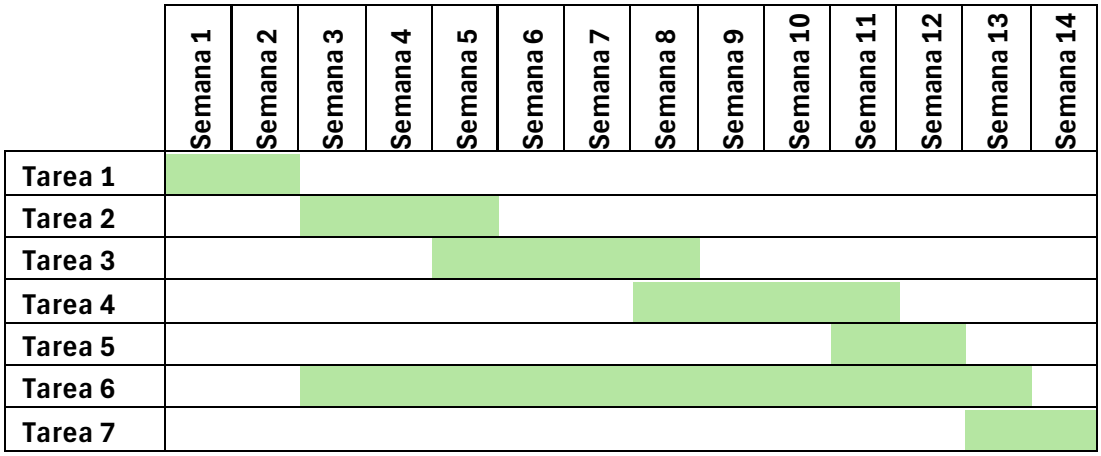
Lista de tareas

- Tarea 1: Investigación inicial y análisis del entorno tecnológico. Completada, se investigaron los fundamentos de las arquitecturas de agentes y la IA generativa, seleccionando *Python* y *Gemini-2.0-Flash* como tecnologías principales.

- Tarea 2: Diseño de la arquitectura del framework.
  Completada. Se diseñó una arquitectura multiagentes con cuatro agentes (*EvaluatorAgent*, el *RootAgent*, el *PlannerAgent* y el *AdjusterAgent*).
- Tarea 3: Implementación del framework.
- Completada. Se implementaron los agentes, permitiendo el procesamiento de entradas en lenguaje natural y la generación del cronograma, junto con una interfaz funcional en Streamlit para pruebas.
- Tarea 4: Desarrollo de un caso de uso.
  Completada. Se diseñó y ejecutó un caso de uso basado en un estudiante universitario, probando la gestión de tareas validando la funcionalidad del sistema.
- Tarea 5: Pruebas y evaluación del sistema.
  Completada. Se realizaron pruebas que demostraron un alto porcentaje de éxito en la interpretación de instrucciones válidas.
- Tarea 6: Redacción de la memoria del TFG.
  Completada. Este documento representa la memoria del TFG.
- Tarea 7: Preparación de la presentación del TFG.
  Completada. Se preparó un Power Point para realizar la defensa del TFG.

Diagrama *Gantt*

A continuación, se presenta el diagrama de Gantt que ilustra la planificación temporal de estas tareas:



1.4 Estructura de la memoria

Los siguientes capítulos de esta memoria son:

- Capítulo 2: Tecnologías empleadas. En este capítulo se define el estado actual de las diferentes tecnologías que se emplean en este TFG, además de definir y explicar cada una de ellas y diferentes alternativas.
- Capítulo 3: Análisis de requisitos y diseño del sistema. Justifica los requisitos que guían la implementación, explicando por qué se desarrollan las funcionalidades del sistema, además de explicar el diseño del sistema de agentes.

- Capítulo 4: Desarrollo del *Framework Organizador de Tareas Inteligente*. Explica el diseño e implementación de los diferentes agentes que forman parte del sistema y del desarrollo de la interfaz gráfica.
- Capítulo 5: Evaluación y discusión. Presenta el caso de uso y analiza diferentes pruebas que se han llevado a cabo para probar el sistema.
- Capítulo 6: Conclusiones y trabajo futuro. Se realiza una conclusión sobre el trabajo realizado, así como de los objetivos definidos y algunas funciones que podrían realizarse en un futuro para mejorar el sistema.
- Capítulo 7: Análisis de impacto. Se lleva a cabo un análisis del impacto potencial de los resultados obtenidos en este TFG en diferentes contextos.

2 Tecnologías Empleadas

La gestión eficiente de tareas en contextos laborales, académicos y personales ha sido transformada por avances en Inteligencia Artificial (IA), particularmente en sistemas multiagentes y modelos generativos de procesamiento de lenguaje natural [4]. Estas tecnologías forman la base del *Organizador de Tareas Inteligente*, un sistema diseñado para automatizar y optimizar la planificación de tareas generales, adaptándose a las necesidades de los usuarios. Este apartado describe las tecnologías existentes, como los sistemas multiagentes y los modelos de lenguaje de gran escala, las herramientas específicas empleadas en el desarrollo (*LangChain*, *Gemini-2.0-Flash* y *Streamlit*), y alternativas comunes a estas tecnologías, justificando las elecciones realizadas.

2.1 Sistemas multiagentes

Los sistemas multiagentes, conocidos como *MAS* por sus siglas en inglés (*Multi-Agent Systems*), son un tipo de tecnología que permite resolver problemas complejos mediante la colaboración de varias entidades, llamadas agentes, que trabajan juntas para alcanzar un objetivo común. Un agente, en este contexto, es como un pequeño programa que actúa de forma autónoma, es decir, que puede tomar decisiones por sí mismo, pero que también está diseñado para interactuar con otros agentes.

En un sistema multiagentes, cada agente tiene un rol especializado y puede realizar acciones concretas, como analizar información, planificar actividades o ajustar decisiones según sea necesario. Lo que hace especial a un *MAS* es que los agentes no trabajan de forma aislada, sino que se comunican entre sí para compartir información y tomar decisiones conjuntas, lo que permite dividir un problema grande en partes más pequeñas y manejables. Por ejemplo, en un sistema que organiza tareas, un agente podría encargarse de entender lo que el usuario quiere, mientras otro decide el mejor momento para esa tarea, y otro ajusta el plan si algo cambia. Esta colaboración hace que los MAS sean muy útiles para problemas que necesitan flexibilidad, coordinación y la capacidad de adaptarse a situaciones nuevas [5].

En el *Organizador de Tareas Inteligente*, los sistemas multiagentes son la base de su arquitectura, permitiendo que diferentes agentes trabajen juntos para gestionar las tareas del usuario. Aquí, el *EvaluatorAgent* es el encargado de entender las instrucciones del usuario, como “tengo que limpiar la cocina mañana”. El *RootAgent* actúa como el coordinador central, asegurándose de que la información fluya entre los agentes y actualizando el calendario con las tareas nuevas o modificadas. El *PlannerAgent* se ocupa de planificar las tareas, decidiendo el mejor horario para las nuevas tareas sin que se solape con otras actividades. Finalmente, el *AdjusterAgent* ajusta las tareas si el usuario pide un cambio, como “mover limpiar la cocina a la tarde”. Esta división de responsabilidades permite que el sistema procese las solicitudes de manera eficiente y sea más fácil de mejorar o ampliar en el futuro.

La aplicabilidad de los *MAS* se ha demostrado en diversos dominios. Por ejemplo, el trabajo de *Dorri et al.* (2018) ilustra su uso en la optimización de cadenas de suministro, donde agentes especializados gestionan inventarios, horarios de entrega y reabastecimiento, logrando una coordinación efectiva, aunque

dependen frecuentemente de infraestructuras centralizadas [6]. En el caso del *Organizador de Tareas Inteligente*, se integra con herramientas en la nube, como *Google Cloud*, para procesar datos a través de *Gemini*. Esta elección lo hace ideal para tareas generales como la programación de reuniones laborales, la gestión de proyectos académicos o la organización de actividades personales, ofreciendo una solución adaptable a usuarios con necesidades diversas.

Además de los agentes mencionados, los *MAS* modernos incorporan conceptos como la negociación entre agentes o el aprendizaje por refuerzo, que podrían integrarse en futuras versiones del sistema para mejorar la toma de decisiones. Sin embargo, para este TFG, se optó por una arquitectura básica pero robusta, priorizando la simplicidad y la viabilidad en un entorno académico con recursos limitados.

2.2 Modelos de lenguaje de gran escala (*LLM*)

Los modelos de lenguaje de gran escala, conocidos como *LLM* por sus siglas en inglés (*Large Language Models*), son un tipo de tecnología basada en inteligencia artificial que permite a las máquinas entender y generar texto de una manera muy similar a cómo lo hacen las personas. Estos modelos son como asistentes digitales avanzados que han sido entrenados con enormes cantidades de textos, como libros, artículos y páginas web, para aprender patrones del lenguaje humano. Gracias a este entrenamiento, un *LLM* puede comprender instrucciones escritas en lenguaje natural y responder de forma lógica y útil, por ejemplo, sugiriendo un horario o preguntando más detalles si algo no está claro. En esencia, un *LLM* actúa como un cerebro lingüístico que no solo entiende palabras, sino también el contexto y la intención detrás de ellas, permitiendo interacciones más naturales entre los usuarios y los sistemas tecnológicos.

Un *LLM* funciona utilizando redes neuronales, que son estructuras matemáticas inspiradas en cómo trabaja el cerebro humano. Estas redes han sido ajustadas con millones de ejemplos para predecir palabras o frases completas basándose en lo que se les ha dicho. Por ejemplo, si alguien escribe "tengo que estudiar", el *LLM* puede deducir que se trata de una tarea y podría preguntar "¿para qué día?" o sugerir un horario. Lo que hace especiales a los *LLM* es su capacidad para manejar instrucciones ambiguas o generales, como "por la tarde" o "próximamente", interpretándolas según el contexto. Esto los hace ideales para aplicaciones donde las personas quieren interactuar con un sistema sin necesidad de usar comandos técnicos o específicos. El trabajo de *Brown et al.* (2020) sobre *GPT-3*, un *LLM* pionero, destacó esta capacidad para interpretar instrucciones complejas y generar respuestas coherentes con pocos ejemplos, sentando las bases para modelos como *Gemini* y su uso en aplicaciones prácticas [7].

En el *Organizador de Tareas Inteligente*, se empleó un *LLM* específico llamado *Gemini-2.0-Flash*, al que se accede mediante una plataforma de *Google Cloud* llamada *Vertex AI*. Este modelo desempeña un papel central en el sistema, ya que es el encargado de analizar las instrucciones del usuario, como "organizar una reunión el viernes" o "sacar al perro esta tarde", y transformarlas en información que el sistema puede usar para planificar.

2.2.1 Versiones de *Gemini* y comparación

La familia *Gemini* ha evolucionado con versiones que ofrecen mejoras progresivas. *Gemini-1.0*, lanzado en diciembre de 2023, fue el debut, destacando por su capacidad multimodal (texto, imágenes) y un enfoque inicial en tareas generales, pero con limitaciones en velocidad y contexto (hasta 32,000 tokens). *Gemini-1.5*, introducido en febrero de 2024, marcó un avance con el modelo Pro, que amplió el contexto a 1 millón de tokens, permitiendo procesar hasta 2 horas de video o 2,000 páginas de texto. Esta versión mejoró el razonamiento y la eficiencia con una arquitectura *Mixture-of-Experts* (*MoE*), aunque seguía siendo más lenta que las variantes Flash.

Gemini-2.0, disponible desde diciembre de 2024, eleva el estándar con modelos como *Flash*, que es el doble de rápido que *Gemini-1.5 Flash*, manteniendo calidad comparable a 1.5 Pro, y añade salidas multimodales (texto, audio, imágenes) y uso nativo de herramientas. Su rendimiento en *benchmarks* supera a sus predecesores, ofreciendo una experiencia más versátil y eficiente. Por ejemplo, *Gemini-2.0-Flash* procesa tareas en menos tiempo y soporta integraciones como *GitHub* para codificación, lo que lo hace ideal para mi proyecto.

No utilizo *Gemini-2.5*, lanzado experimentalmente en marzo de 2025 con versiones Pro y Flash, porque su costo es significativamente mayor y su reciente disponibilidad lo hace menos estable para un TFG. *Gemini-2.5 Pro* destaca por su capacidad de razonamiento avanzado y un contexto de 1 millón de tokens, liderando en *benchmarks*. Sin embargo, su precio y el uso intensivo de recursos (hasta 100,000 tokens por hora en tareas complejas) exceden los créditos gratuitos y el alcance de mi proyecto, prefiriendo la relación costo-beneficio de *Gemini-2.0-Flash* [8].

2.2.2 Alternativas a *Gemini*

Entre las alternativas a *Gemini*, *GPT-3* de *OpenAI* ofrece un rendimiento excepcional en tareas de lenguaje, pero requiere una suscripción de pago, lo que lo hace menos viable para un TFG con presupuesto limitado. *LLaMA* de *Meta AI*, optimizado para investigación, proporciona eficiencia en entornos académicos, aunque su accesibilidad está restringida y requiere conocimientos avanzados para su implementación. Otro modelo emergente, *Grok* de *xAI*, destaca por su diseño para respuestas útiles y veraces, pero su disponibilidad es limitada y no ofrece créditos gratuitos como *Gemini*. Además, modelos más recientes como Mistral (desarrollado por Mistral AI) están ganando terreno por su eficiencia en tareas específicas y su potencial de uso local, aunque aún están en etapas tempranas de adopción.

La elección de *Gemini-2.0-Flash* se fundamentó en los créditos gratuitos ofrecidos por *Google* a través de *Vertex AI*, una ventaja crucial para un TFG con recursos restringidos. Además, su rendimiento competitivo en la interpretación de lenguaje natural, su integración con herramientas de *Google* (que facilitan su uso en entornos educativos), y su capacidad para operar localmente con ajustes mínimos lo convierten en una opción ideal. Esta decisión también asegura escalabilidad, permitiendo al sistema adaptarse a cargas de trabajo más complejas en el futuro.

2.3 Herramientas de gestión de tareas

Las herramientas tradicionales de gestión de tareas, como *Google Calendar* y *Todoist*, permiten registrar eventos y establecer recordatorios, pero dependen de entradas manuales y carecen de capacidades avanzadas de procesamiento de lenguaje natural. Por su parte, asistentes virtuales como *Siri* o *Google Assistant* han incorporado esta funcionalidad, interpretando comandos verbales para programar tareas, pero su enfoque en acciones aisladas (como establecer alarmas) los limitan en la gestión integral de agendas.

En el ámbito académico, investigaciones como la de *Zhang et al.* (2021) proponen sistemas multiagentes para optimizar flujos de trabajo en entornos empresariales, demostrando su eficacia en contextos específicos, aunque su complejidad técnica los hace poco accesibles para usuarios generales [9]. Por otro lado, *Russell y Norvig* (2020) exploran modelos de planificación con agentes, ofreciendo una base teórica sólida, pero requieren configuraciones avanzadas que superan el alcance de un proyecto académico típico [4]. El *Organizador de Tareas Inteligente* se diferencia por su diseño modular y local, combinando la flexibilidad de los *MAS* con una interfaz accesible, lo que lo hace adecuado para gestionar tareas variadas sin depender de infraestructuras externas.

2.4 Lenguaje de programación. Python

El desarrollo del *Organizador de Tareas Inteligente* se llevó a cabo utilizando *Python* como lenguaje de programación principal, una elección realizada por sus características que lo convierten en ideal para este proyecto, especialmente en el contexto de sistemas multiagentes. Python destaca por su simplicidad y legibilidad, lo que facilita la implementación y el mantenimiento del código, un aspecto crucial al diseñar una arquitectura con múltiples agentes que requieren una coordinación precisa. Además, su amplia comunidad y ecosistema de bibliotecas, como *LangChain* para la integración de *LLM* como *Gemini-2.0-Flash*, ofrecen soporte robusto para el desarrollo de aplicaciones de IA y procesamiento de datos, áreas esenciales para este framework.

La elección de Python se justifica también por su idoneidad para sistemas multiagentes, ya que permite una implementación modular y flexible gracias a su soporte para programación orientada a objetos y concurrencia, facilitando la definición de agentes como entidades independientes que interactúan entre sí. Además, Python ofrece bibliotecas específicas para IA, como *TensorFlow* o *PyTorch*, que podrían integrarse en el futuro, y su compatibilidad con herramientas de visualización como *Streamlit* refuerza su utilidad para crear interfaces gráficas accesibles.

2.5 *LangChain*

LangChain [10] es un *framework* de código abierto diseñado específicamente para facilitar la integración de los *LLM*, como *Gemini-2.0-Flash*, con aplicaciones que requieren procesamiento avanzado de lenguaje natural y lógica contextual. Desarrollado por la comunidad de código abierto y mantenido por *LangChain*

Inc., este *framework* se ha consolidado como una herramienta clave para construir sistemas inteligentes que combinan la potencia de los *LLM* con estructuras de datos y agentes autónomos. En esencia, *LangChain* actúa como un puente entre los modelos de IA y las aplicaciones prácticas, permitiendo a los desarrolladores aprovechar las capacidades de generación de texto y comprensión de los *LLM* mientras añaden funcionalidades como memoria a largo plazo, recuperación de información, y coordinación multiagente.

En el contexto del *Organizador de Tareas Inteligente*, *LangChain* sirve como el núcleo de la arquitectura, orquestando la interacción entre los agentes (*EvaluatorAgent*, *RootAgent*, *PlannerAgent*, y *AdjusterAgent*) y el modelo *Gemini-2.0-Flash*. Su función principal es procesar las entradas en lenguaje natural del usuario (por ejemplo, "reunión mañana a las 10") y transformarlas en comandos estructurados que los agentes pueden ejecutar. Esto se logra a través de sus componentes clave: las cadenas (chains), que encadenan múltiples pasos de procesamiento (como interpretación y planificación) y los agentes, que delegan tareas a herramientas específicas según el contexto. Además, *LangChain* incluye soporte para herramientas externas (como *APIs* o bases de datos), lo que permite al sistema acceder a información adicional o realizar acciones, como consultar un calendario interno.

2.5.1 Alternativas a *LangChain*

Haystack [11], desarrollado por *deepset*, se enfoca en recuperación de información y generación de respuestas basadas en documentos, pero es menos flexible para flujos multiagentes complejos. *Rasa* [12] está diseñado para *chatbots* con capacidades de diálogo, ofreciendo un enfoque sólido para interacciones conversacionales, aunque carece de la versatilidad de *LangChain* para gestionar agentes autónomos. *LlamaIndex [13]*, un *framework* emergente, optimiza la indexación y consulta de datos con *LLM*, permitiendo búsquedas eficientes sobre grandes corpus, pero requiere configuraciones más elaboradas y no soporta tan bien la coordinación multiagente. Recientemente, *AutoGen* [14] de *Microsoft* ha ganado atención por su capacidad de crear agentes conversacionales personalizados, aunque su implementación es más técnica y menos accesible para un TFG. Elegí *LangChain* por su compatibilidad con *Python* y su facilidad de uso, que agilizaron el desarrollo en un entorno académico.

2.6 Interfaz gráfica

La interfaz gráfica es un componente fundamental en el *Organizador de Tareas Inteligente*, ya que determina la experiencia del usuario al interactuar con el sistema para ingresar tareas, visualizar cronogramas y recibir retroalimentación. Para implementar esta interfaz, he seleccionado *Streamlit* [15], una librería de *Python* de código abierto diseñada específicamente para crear aplicaciones web interactivas de manera rápida y sencilla. Desarrollada por *Streamlit Inc.*, esta herramienta permite a los desarrolladores construir interfaces gráficas sin necesidad de conocimientos profundos en desarrollo web, utilizando únicamente *Python*, lo que la hace ideal para proyectos académicos como este TFG. *Streamlit* transforma scripts de *Python* en aplicaciones web dinámicas, eliminando la complejidad de *frameworks* tradicionales como *Flask* o *Django*, y

se ha convertido en una opción popular para prototipar y desplegar aplicaciones de datos e inteligencia artificial.

En el *Organizador de Tareas Inteligente*, *Streamlit* sirve como la capa de presentación, permitiendo a los usuarios introducir tareas en lenguaje natural (por ejemplo, "sacar al perro esta tarde") a través de campos de texto o botones. Su diseño se basa en diferentes componentes interactivos para capturar entradas, para enviar comandos y para visualizar cronogramas actualizados, como un calendario que muestra las tareas programadas tras cada acción. Por ejemplo, al ingresar una tarea, *Streamlit* actualiza en tiempo real una tabla con las columnas "Fecha", "Hora de inicio", "Hora de fin" y "Descripción", proporcionando una vista clara y accesible. Esta reactividad se logra gracias a su modelo de ejecución basado en eventos, donde cada cambio en la interfaz provoca una reejecución del script existente, lo que facilita la integración con la lógica de los agentes. Además, la experiencia previa con *Streamlit*, adquirida en proyectos anteriores, permitió acelerar el desarrollo, aprovechando su curva de aprendizaje suave y su documentación detallada, lo que resultó clave para validar rápidamente el prototipo en un entorno académico.

2.6.1 Alternativas a *Streamlit*

Dash [16], desarrollado por *Plotly*, ofrece mayor personalización con gráficos interactivos y *dashboards* avanzados, permitiendo a los usuarios crear aplicaciones complejas con controles detallados, pero su curva de aprendizaje es más pronunciada debido a su dependencia de *callbacks* y estructuras complejas, lo que requiere más tiempo de desarrollo y conocimientos de desarrollo web. *Flask* [17], un *framework* web ligero, permite desarrollar interfaces personalizadas con total control, aunque exige más código manual para lograr funcionalidades dinámicas como las de *Streamlit*, siendo menos eficiente para prototipos rápidos. *Gradio* [18], popular para crear interfaces de demostración de modelos de IA, destaca por su simplicidad y compatibilidad con modelos como *Gemini*, permitiendo subir archivos o interactuar con modelos en tiempo real, pero carece de las capacidades avanzadas de visualización de *Streamlit* para cronogramas o tablas extensas. *Shiny for Python* [19], una adaptación del *framework R Shiny*, está emergiendo como una opción para aplicaciones interactivas, ofreciendo reactividad similar a *Streamlit*, aunque aún está en desarrollo y no tiene la madurez ni la comunidad activa de *Streamlit*. Elegí *Streamlit* por la familiaridad previa, que agilizó el desarrollo, y por su simplicidad, compatibilidad con *Python*, y rapidez para prototipar y validar el sistema en un entorno académico.

3 Análisis de Requisitos y Diseño del Sistema

El *Organizador de Tareas Inteligente* se desarrolla para gestionar tareas generales (laborales, académicas, personales) de manera autónoma y adaptativa, priorizando la usabilidad y la adaptabilidad. Este apartado detalla los requisitos que guían su implementación, explicando las razones detrás de cada funcionalidad, y describe el diseño del sistema, con un enfoque especial en el flujo de trabajo entre agentes, incluyendo un diagrama de flujo detallado y la estructura de los JSONs utilizados.

3.1 Requisitos funcionales y no funcionales

Los requisitos funcionales especifican las capacidades esenciales que el sistema debe cumplir para satisfacer las necesidades de los usuarios. En primer lugar, el procesamiento de entradas en lenguaje natural es una funcionalidad crítica, permitiendo que los usuarios introduzcan tareas mediante frases naturales como "comprar el pan mañana por la mañana" o "posponer sacar al perro". Esta característica se justifica por su capacidad para reducir la barrera de entrada, eliminando la necesidad de aprender comandos específicos y haciéndolo accesible a personas sin conocimientos técnicos avanzados. La implementación de esta funcionalidad se basa en la integración con modelos como *Gemini-2.0-Flash*, que interpreta el lenguaje de forma precisa y contextual.

Otro requisito funcional clave es la generación de cronogramas optimizados, que implica asignar tiempos a las tareas evitando solapamientos con actividades existentes. Este aspecto es fundamental para maximizar la productividad, especialmente en agendas complejas donde los usuarios manejan múltiples responsabilidades. La capacidad de estimar la duración de las tareas (por ejemplo, 30 minutos para ir a comprar el pan) y seleccionar el mejor horario considerando fechas objetivo y referencias temporales garantiza que el sistema sea proactivo y eficiente. Además, la funcionalidad de ajustes dinámicos permite modificar tareas existentes según retroalimentación o cambios imprevistos, como posponer una tarea o reubicarla en un nuevo horario. Esta adaptabilidad es esencial para reflejar la naturaleza impredecible de las agendas diarias, asegurando que el sistema permanezca útil en situaciones reales.

La exportación de tareas a un calendario interno en formato local es otro requisito funcional importante. Esto permite a los usuarios visualizar y gestionar sus tareas sin depender de aplicaciones externas, reforzando la autonomía del sistema.

En cuanto a los requisitos no funcionales, la autonomía es un pilar fundamental. El sistema debe operar localmente, almacenando datos en archivos *JSON*, lo que garantiza la privacidad del usuario al evitar la transmisión de datos sensibles a servidores externos. Este enfoque se justifica por las crecientes preocupaciones sobre la seguridad de la información en plataformas en la nube y la necesidad de cumplir con normativas de protección de datos.

La escalabilidad del sistema es otro requisito no funcional clave. Su diseño modular permite incorporar nuevos agentes o funcionalidades (por ejemplo, un agente de notificaciones o integración con herramientas externas) sin alterar la arquitectura existente. Esta flexibilidad es crucial para adaptarse a futuros usos,

como la gestión de proyectos en entornos empresariales o educativos, asegurando que el sistema tenga un impacto a largo plazo.

La usabilidad también se prioriza como requisito no funcional. La interfaz, desarrollada con *Streamlit*, debe ser intuitiva, con un diseño claro para ingresar tareas y visualizar cronogramas, justificándose por la necesidad de que usuarios de diversos niveles técnicos puedan operarlo sin dificultades. Además, el rendimiento del sistema debe ser eficiente, procesando tareas y generando cronogramas en un tiempo razonable (menos de 5 segundos por operación), lo que garantiza una experiencia fluida y práctica.

3.2 Diseño del sistema

El diseño del *Organizador de Tareas Inteligente* se basa en una arquitectura multiagente que coordina cuatro agentes: *RootAgent*, *EvaluatorAgent*, *PlannerAgent*, y *AdjusterAgent*. El flujo de trabajo sigue un proceso estructurado que comienza con la entrada del usuario y culmina con la actualización del calendario interno. A continuación, se detalla el flujo con sus componentes:

- **Input del usuario**: El usuario proporciona una tarea en lenguaje natural, como “sacar al perro esta tarde” o “posponer comprar el pan”. Esta entrada desencadena el proceso.
- ***RootAgent***: Actúa como el coordinador central, recibiendo el input del usuario y enviándolo al *EvaluatorAgent* para su análisis. Una vez procesado, el *RootAgent* decide el siguiente paso según la intención del usuario: lo envía al *PlannerAgent* para crear tareas o al *AdjusterAgent* para modificarlas, finalmente actualizando el calendario interno con los resultados.
- ***EvaluatorAgent***: Recibe el input y procesa las intenciones del usuario mediante el *LLM*, *Gemini-2.0-Flash* y genera un *JSON* estructurado a partir de las conclusiones procesadas de esta forma:

  {
  "*intent*": " ",
  "*task_detail*": " ",
  "*target_date*": " ",
  "*time_reference*": " ",
  "*explanation*": " "
  }

  Los elementos de este JSON son:
  - "***intent***": Indica la acción que quiere realizar el usuario, siendo "crear_tarea" si es una nueva tarea o "modificar_tarea" si se ajusta una existente, determinando el flujo posterior.
  - "***task_detail***": Contiene una descripción detallada de la tarea (por ejemplo, “sacar al perro”), esencial para que el sistema entienda el contenido.
  - "***target_date***": Registra la fecha objetivo si se menciona (por ejemplo, “2025-05-15” o “el miércoles”), útil para priorizar plazos.
  - "***time_reference***": Captura cualquier referencia temporal explícita (por ejemplo, “por la tarde”), guiando la planificación.
  - "***explanation***": Incluye la interpretación que hace el *LLM* de la intención del usuario (por ejemplo, “El usuario quiere crear una nueva tarea para sacar a pasear al perro. Indica que quiere realizarla por la tarde.”), facilitando la trazabilidad y depuración.

Este *JSON* se devuelve al *RootAgent* para orientar el siguiente paso.

- ***PlannerAgent*** (para crear tareas): Recibe el *JSON* generado por el *EvaluatorAgent*, si el *intent* es "crear_tarea". Utiliza el *LLM* para estimar la duración de la tarea basada en el *task_detail* (por ejemplo, 30 minutos para sacar al perro, o 15 minutos para bajar la basura), determina el mejor horario considerando el *target_date* y el *time_reference*, y evita solapamientos con tareas existentes en el calendario. Devuelve un *JSON* de la siguiente forma:

```
{
        "scheduled_time": {
            "date": "",
            "start_time": "",
            "end_time": ""
        },
        "duration_minutes": "",
        "task_type": "",
        "notes": ""
}
```

  Los elementos de este *JSON* son:
  - **"*scheduled_time*":** Un objeto con "*date*", siendo la fecha que ha elegido para la tarea (por ejemplo, "2025-05-15"), "*start_time*", será la hora a la que empieza la tarea (por ejemplo, "10:00"), y "*end_time*", que es la hora a la que acaba la tarea (por ejemplo, "11:00"), definiendo el hueco asignado.
  - **"*duration_minutes*":** El tiempo estimado que durará la tarea (por ejemplo, 60), calculado por el *LLM*.
  - **"*task_type*":** Clasifica la tarea (por ejemplo, "tareas del hogar"), útil para el *LLM* para poder decidir mejor la duración de la tarea y para un mejor filtrado.
  - **"*notes*":** Similar a "*explanation*" del *EvaluatorAgent*, mejorando la usabilidad.

  A partir de este *JSON*, genera un objeto "*slot*" que es el que usara el *RootAgent* para añadir la tarea al calendario, es el hueco que determina el agente en el que se añadirá la tarea en el calendario, este objeto *slot* contiene: "*start_time*", "*end_time*", "*task_type*" y "*task_detail*", ya explicado anteriormente.

- ***AdjusterAgent*** (para modificar tareas): Recibe el *JSON* generado por el *EvaluatorAgent*, si el *intent* es "modificar_tarea". Ajusta la tarea existente evitando solapamientos:
  - Si *time_reference* está vacío, es decir, no se especifica una nueva hora para la tarea, pospone la tarea 1 hora, si la nueva hora solapa con una tarea ya existente, pospondrá la tarea 1 hora más, así hasta que encuentre un hueco disponible. En caso de no encontrar un hueco disponible para ese día, pasará al día siguiente y procederá de la misma manera.
  - Si se especifica una hora (por ejemplo, "15:00"), es decir, se indica la nueva hora en el *time_reference*, intenta reubicarla a la hora indicada; si hay solapamiento, la mueve después de la tarea conflictiva. Devuelve un *JSON* con:

- **"*schedule_time*":** Un objeto igual que el del *PlannerAgent*, con la fecha y las nuevas horas de comienzo y finalización de la tarea.
- **"*reason*":** Una breve explicación de por qué se eligió ese nuevo horario.

Al igual que el *PlannerAgent*, este agente usará el *JSON* para generar el objeto "*slot*" y el *RootAgent* eliminará la tarea con el horario anterior y añadirá la tarea con el nuevo horario.

La Figura 3-1 muestra el diagrama de flujo explicado anteriormente.

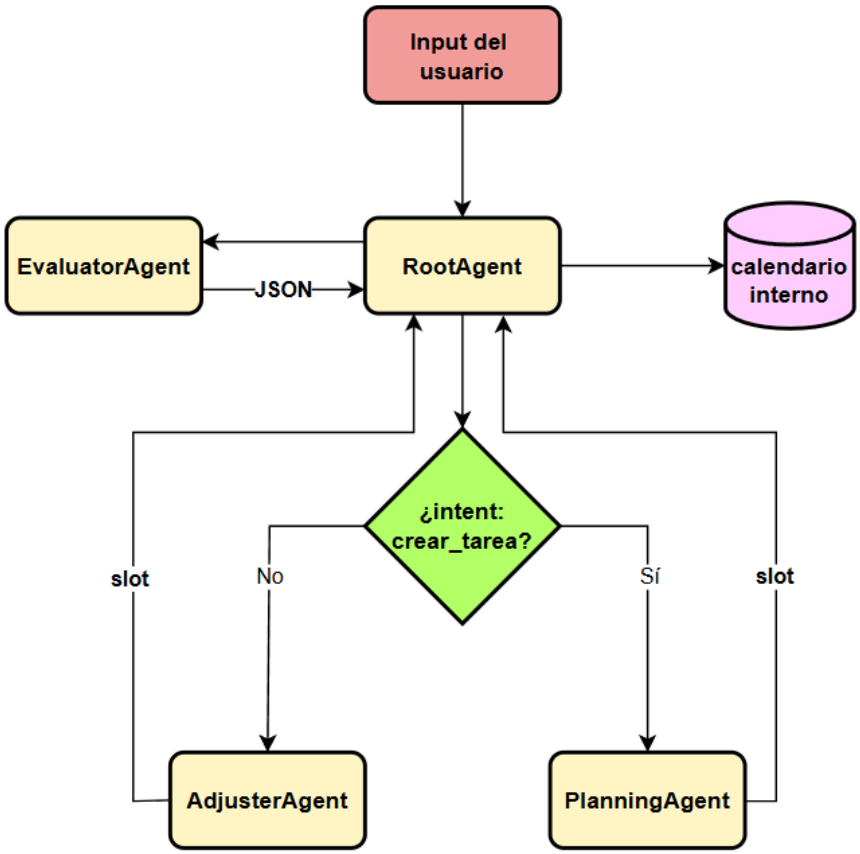


Figura 3-1. Diagrama de flujo

4 Desarrollo del *Framework Organizador de Tareas Inteligente*

Este capítulo presenta el proceso de desarrollo del *Organizador de Tareas Inteligente*, describiendo de manera detallada las etapas técnicas llevadas a cabo para la creación del framework. Se abordan aspectos clave como la implementación de la arquitectura multiagente con los agentes especializados (*EvaluatorAgent*, *RootAgent*, *PlannerAgent* y *AdjusterAgent*), y la integración del modelo de lenguaje *Gemini-2.0-Flash* para el procesamiento de instrucciones en lenguaje natural. Además, se explica el desarrollo de la interfaz gráfica mediante *Streamlit*, todo ello acompañado de ejemplos prácticos y decisiones técnicas tomadas para garantizar la operatividad y escalabilidad del framework.

4.1 Introducción al desarrollo

El desarrollo del *Organizador de Tareas Inteligente* ha constituido un proceso metódico y estructurado orientado a diseñar una herramienta que facilite la organización de actividades cotidianas, tales como el estudio, el ejercicio o la realización de compras, sin recurrir a métodos de planificación complejos. El objetivo principal de este proyecto ha sido crear un sistema capaz de interpretar instrucciones expresadas en lenguaje natural, similar al utilizado en una conversación habitual, y de optimizar la planificación del tiempo de manera autónoma. Asimismo, se buscó implementar una interfaz gráfica intuitiva que permitiera visualizar y gestionar las tareas de forma eficiente, además de incorporar la capacidad de adaptarse a modificaciones imprevistas, como el aplazamiento de actividades.

El procedimiento de desarrollo se llevó a cabo mediante un enfoque iterativo, empleando el lenguaje de programación *Python*, reconocido por su versatilidad y amplia adopción en el ámbito académico y profesional. A esta base se integró una solución de inteligencia artificial proporcionada por *Google Cloud* (*Gemini-2.0-Flash*), que permitió incorporar capacidades avanzadas de comprensión y procesamiento del lenguaje, equiparables a las de un asistente virtual sofisticado. Para estructurar el sistema, se diseñó una arquitectura basada en agentes, dividida en cuatro componentes principales: un agente central de coordinación, denominado (*RootAgent)*, un agente encargado de interpretar las instrucciones del usuario (*EvaluatorAgent*), otro responsable de planificar las tareas (*PlannerAgent*), y un cuarto dedicado a realizar ajustes (*AdjusterAgent*). Complementariamente, se utilizó la herramienta *Streamlit* para desarrollar una interfaz gráfica atractiva y funcional.

En las siguientes secciones, se detallará el proceso seguido para la creación de cada componente del sistema, desde la organización de los agentes hasta el diseño de la interfaz gráfica, utilizando ejemplos prácticos como la programación de “comprar pan mañana” o el ajuste de “sacar al perro” a un horario diferente.

4.2 Arquitectura general del sistema

La arquitectura del *Organizador de Tareas Inteligente* se diseñó para garantizar un funcionamiento eficiente y coordinado entre sus diferentes partes, permitiendo que cada componente cumpla un rol específico dentro del proceso de organización de tareas. Como ya se explicó en un apartado anterior, el sistema se basa en una estructura de agentes que trabajan juntos, con un agente principal que coordina las acciones y otros agentes especializados que se encargan de entender, planificar y ajustar las tareas. En esta sección, se ofrece un resumen breve para recordar cómo está organizado el sistema, antes de entrar en los detalles del desarrollo de cada componente.

El núcleo del sistema lo forma un agente central, denominado *RootAgent*, que actúa como el coordinador general. Este agente es el encargado de recibir las instrucciones del usuario, como "estudiar mañana a las 10:00", y distribuir las responsabilidades a los otros agentes según sea necesario. Además, se ocupa de guardar y actualizar un archivo donde se almacenan todas las tareas, para poder generar el calendario. El resto de los agentes incluyen el que interpreta lo que el usuario quiere hacer (*EvaluatorAgent*), el que se encarga de decidir el mejor momento para establecer la tarea (*PlannerAgent*), y uno más que ajusta las tareas si hace falta cambiarlas (*AdjusterAgent*). Para que el usuario pueda interactuar con el sistema de forma sencilla, se creó una pantalla principal con la herramienta *Streamlit*, que muestra las tareas y permite añadir más, modificar las existentes o eliminarlas. La Figura 4-1 muestra un diagrama de la comunicación existente entre los agentes.

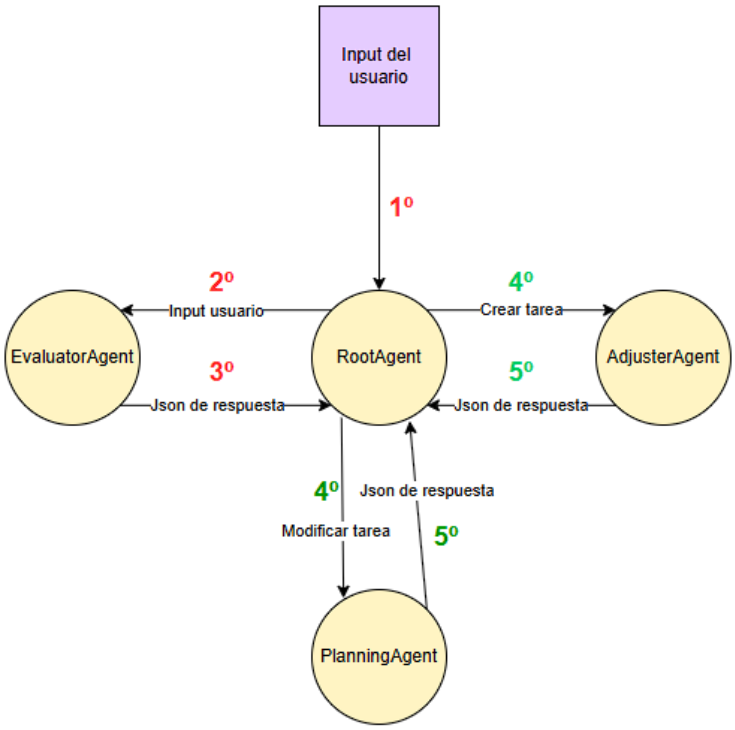


Figura 4-1. Diagrama de comunicación de los agentes

4.3 Desarrollo del *RootAgent*

El desarrollo del *RootAgent* representó un pilar esencial en la construcción del *Organizador de Tareas Inteligente*, al establecerse como el componente central que coordina todas las operaciones del sistema. Este agente fue diseñado para recibir las instrucciones del usuario, gestionar el almacenamiento de las tareas y supervisar la interacción con los demás agentes especializados, asegurando que el sistema funcione de manera fluida y eficiente. A continuación, se describe el procedimiento seguido para su implementación, con un enfoque especial en su función clave de orquestación, acompañado de ejemplos prácticos para ilustrar su rol.

En primer lugar, se realizó un enfoque en habilitar la carga de un archivo de almacenamiento, "*calendar.json*", que actúa como una agenda o calendario donde se registran todas las tareas organizadas por fecha. Para ello, se desarrolló un método para leer este archivo y convertir la información del formato *JSON* a un formato que el sistema pueda manejar, organizando los datos por días con detalles como la hora de inicio, la hora de fin y el tipo de tarea. Por ejemplo, al cargar el archivo, el *RootAgent* transforma una entrada como "2025-05-21" con la tarea "clase uni de 10:00 a 14:00" en un formato interno útil para los demás agentes. Este método incluyó medidas para manejar posibles errores, como la ausencia del archivo o datos incompletos, garantizando que el sistema pudiera continuar funcionando sin interrupciones.

A continuación, se implementó otro método que hace lo contrario al explicado anteriormente, que convierte la información a formato *JSON* y asegura que los cambios realizados en el calendario se guarden en "*calendar.json*". Este paso fue crucial para mantener las tareas actualizadas de forma permanente. El procedimiento consistió en transformar los datos del sistema en un formato compatible con el archivo, incluyendo detalles como la hora, el tipo de tarea y su estado.

La funcionalidad más importante del *RootAgent*, y el núcleo de su rol como orquestador del sistema, se encuentra en un método denominado "*process_user_input*". Este método actúa como el director principal, encargándose de recibir las instrucciones del usuario y coordinar las acciones de los agentes especializados para cumplir con ellas. El proceso comienza cuando el usuario introduce una solicitud, como "estudiar mañana a las 10:00". El *RootAgent* toma esta entrada y la pasa al *EvaluatorAgent* para que analice la intención del usuario, determinando si se trata de crear una nueva tarea o modificar una existente. Según la respuesta del *EvaluatorAgent*, el *RootAgent* decide a quién recurrir: si es una tarea nueva, llama al *PlannerAgent* para que encuentre el mejor momento para programarla; si es un ajuste, como "mover la reunión a las 15:00", solicita la intervención del *AdjusterAgent* para reubicar la tarea. Una vez que el *PlannerAgent* o el *AdjusterAgent* terminan su trabajo, el *RootAgent* toma la información resultante, actualiza el calendario interno con los nuevos datos y guarda los cambios en el calendario interno. Por ejemplo, tras planificar "estudiar mañana a las 10:00", el *RootAgent* añade esta tarea al calendario para el día siguiente y la guarda en el archivo. Este método también fue diseñado para reconocer comandos como "salir" o "quit", permitiendo cerrar el sistema de manera ordenada.

Otro aspecto clave fue el desarrollo de una función que permite al usuario ver las tareas programadas para una fecha específica. Por ejemplo, al consultar las tareas para el 21 de mayo de 2025, el *RootAgent* muestra la tarea "clase uni"

junto con su horario, como se puede observar en la Figura 4-2. Este método fue creado para filtrar las tareas por día y presentarla de forma clara. Asimismo, se implementó una función para eliminar tareas concretas, como "sacar al perro" de una fecha dada, y otra que borra todas las tareas del calendario cuando se solicita, con un procedimiento que actualiza el archivo tras cada acción.

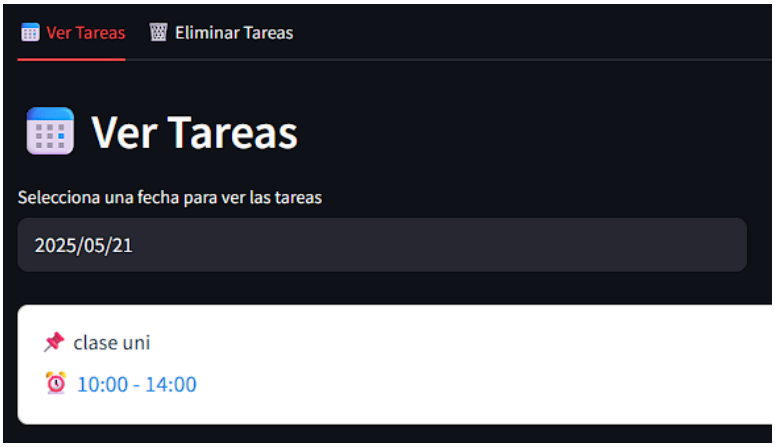


Figura 4-2. Consulta de tareas del día 21/05/2025

El desarrollo del *RootAgent* implicó varias iteraciones para garantizar que la comunicación con los agentes especializados fuera eficiente y que el calendario se mantuviera actualizado en todo momento. Este enfoque consolidó al *RootAgent* como el orquestador central del sistema, asegurando que las instrucciones del usuario se traduzcan en acciones concretas y que las tareas se gestionen de manera confiable y organizada.

4.4 Desarrollo del *EvaluatorAgent*

El *EvaluatorAgent* desempeña un papel esencial en el *Organizador de Tareas Inteligente*, al ser el encargado de interpretar las instrucciones del usuario y determinar qué acción desea realizar, ya sea programar una nueva tarea o modificar una existente. Este agente fue diseñado para procesar el lenguaje natural de manera efectiva, delegando la toma de decisiones al *LLM* proporcionado por *Google Cloud*, *Gemini-2.0-Flash*. A continuación, se detalla el procedimiento seguido para su desarrollo y se explica su funcionamiento.

El desarrollo del *EvaluatorAgent* comenzó con la creación de una clase que integrara las capacidades de inteligencia artificial necesarias para analizar las instrucciones del usuario. El primer paso fue establecer una conexión con *Google Cloud* para emplear el modelo *Gemini-2.0-Flash*, una herramienta capaz de comprender frases como si fueran expresadas por una persona. Para guiar al *LLM* en su tarea, se elaboró una plantilla de instrucciones detallada, también denominado *prompt*, donde se le indicó que analizara el texto del usuario y determinara su intención, ya fuera crear una tarea nueva, como "estudiar mañana a las 10:00", o modificar una existente, como "mover la reunión a las 15:00" y generara un *JSON*, con los contenidos que ya se especificaron en anteriores capítulos de este TFG. Además, se le proporcionó indicaciones

específicas para identificar intenciones de cambio, basándose en palabras comunes como “cambiar”, “posponer”, “adelantar” o “no me va bien esa hora”.

El funcionamiento del *EvaluatorAgent* se fundamenta en el *LLM* como el componente que toma las decisiones clave. Cuando el usuario introduce una instrucción, el *EvaluatorAgent* la envía al *LLM* junto con la plantilla de instrucciones. El *LLM* analiza la frase y decide la intención del usuario, confiando exclusivamente en su capacidad para interpretar el lenguaje natural. Por ejemplo, si el usuario escribe “estudiar mañana a las 10:00”, el *LLM* determina que la intención es crear una nueva tarea, identifica “estudiar” como el detalle de la tarea, “mañana” como la fecha, y “10:00” como la hora específica. De manera similar, si el usuario dice “mover sacar al perro a las 11:00”, el *LLM* reconoce la palabra “mover” y decide que la intención es ajustar una tarea existente, identificando “sacar al perro” como la tarea a modificar y “11:00” como el nuevo horario.

Un aspecto importante del funcionamiento del *LLM* es su manejo de referencias temporales generales, como “por la mañana”, “por la tarde” o “el sábado por la tarde”, en lugar de convertirlas directamente en horas concretas. Por ejemplo, si el usuario indica “sacar al perro mañana por la mañana”, el *LLM* no asigna una hora específica como las 09:00, sino que guarda “mañana por la mañana” como referencia temporal, preservando la intención original del usuario. Esta flexibilidad permite que el sistema mantenga la ambigüedad natural del lenguaje y la pase a los agentes posteriores, como el *PlannerAgent*, para que decidan el horario exacto según el contexto. Otro ejemplo sería “reunión el viernes por la tarde”: el *LLM* interpreta la intención como crear una tarea, identifica “reunión” como la tarea a crear, y guarda “viernes por la tarde” como referencia temporal, sin especificar una hora concreta.

Una vez que el *LLM* toma estas decisiones, el *EvaluatorAgent* organiza la información en un formato estructurado que incluye la intención (crear o modificar), el detalle de la tarea, la fecha objetivo (si se menciona) y la referencia temporal (como “mañana por la mañana” o “11:00”). Este formato es crucial para que el *RootAgent* sepa cómo proceder, ya sea enviando la tarea al *PlannerAgent* para programarla o al *AdjusterAgent* para ajustarla. Durante el desarrollo, se realizaron varias iteraciones sobre la plantilla de instrucciones para afinar la capacidad del *LLM* de reconocer intenciones y referencias temporales, ajustándola para incluir más expresiones comunes y reducir malentendidos. Por ejemplo, se añadieron términos como “retrasar” o “por la noche” para mejorar la detección de intenciones y temporalidades.

El proceso también contempló medidas para manejar instrucciones poco claras. En caso de ambigüedad, el *EvaluatorAgent* confía en la mejor interpretación del *LLM* y devuelve un resultado básico, permitiendo que el sistema continúe funcionando. La razón principal de delegar las decisiones al *LLM* radica en su habilidad para procesar el lenguaje natural de forma flexible y precisa, lo que hace que el sistema sea accesible y útil para cualquier usuario. De esta manera, el *EvaluatorAgent* se convierte en un puente esencial entre las instrucciones del usuario y el resto del sistema, garantizando que las intenciones se comprendan correctamente y se canalicen hacia la acción adecuada.

4.5 Desarrollo del *PlannerAgent*

El *PlannerAgent* es el encargado de determinar el mejor momento para programar nuevas tareas, asegurando que se ajusten al calendario del usuario de manera eficiente y sin conflictos. Este agente fue diseñado para trabajar en colaboración con *Gemini-2.0-Flash*, el *LLM* de *Google Cloud*, como también hacía el *EvaluatorAgent*. A continuación, se detalla el procedimiento seguido para su desarrollo y se explica su funcionamiento, con ejemplos prácticos que ilustran cómo organiza las tareas de forma efectiva.

El desarrollo del *PlannerAgent* comenzó con la creación de una clase que integrara las capacidades del *LLM*, específicamente el modelo *Gemini-2.0-Flash* de *Google Cloud*, para procesar las solicitudes de planificación. El primer paso fue diseñar una plantilla de instrucciones claras que guiaran al *LLM* en su tarea. En esta plantilla, se le indicó que analizara la información de la tarea, como "estudiar mañana", y decidiera el mejor horario, respetando ciertas reglas: las tareas solo se pueden programar entre las 9:00 y las 14:00 o entre las 15:00 y las 21:00, evitando que se superpongan con otras actividades ya programadas. También se le añadió que considerara las preferencias temporales del usuario, como "por la mañana" o "por la tarde", y que sugiriera el siguiente día disponible si no había espacio en la fecha indicada.

El funcionamiento del *PlannerAgent* se basa en la capacidad del *LLM* para evaluar el contexto y proponer una solución. Cuando el *RootAgent* envía una solicitud, como "estudiar mañana a las 10:00" el *PlannerAgent* consulta el calendario existente y pasa esta información al *LLM* junto con la plantilla. El *LLM* analiza las tareas ya programadas para esa fecha, por ejemplo, "clase uni de 10:00 a 14:00", y decide si el horario solicitado es viable. Si no hay conflictos, el *PlannerAgent* programa la tarea en el horario indicado. Sin embargo, si hay un solapamiento, el *LLM* sugiere el siguiente hueco libre en el mismo día o, si no lo hay, en el día siguiente. Este proceso asegura que las tareas se coloquen de forma ordenada y práctica. Además, el *LLM* se encarga de determinar el tiempo necesario que necesitará para la tarea, por ejemplo, si la tarea es "sacar al perro", decidirá que el tiempo necesario será entre 30 minutos o 1 hora, pero si la tarea es "jugar un partido de fútbol", decidirá que la tarea durará al menos 2 horas, ya que los partidos de fútbol tienen una duración de unos 90 minutos.

Una vez que el *LLM* toma la decisión, el *PlannerAgent* organiza la información en un formato que incluye la fecha, la hora de inicio, la hora de fin, el tipo de tarea y una duración estimada. Por ejemplo, para "comprar pan por la mañana", el *LLM* podría sugerir un horario de 9:00 a 9:30, basándose en las referencias temporales y las tareas existentes. El agente luego convierte esta propuesta en un formato interno que el *RootAgent* puede usar para actualizar el calendario. Durante el desarrollo, se iteró sobre la plantilla para asegurarse de que el *LLM* respetara las restricciones horarias y considerara adecuadamente las preferencias del usuario, ajustándola para incluir los detalles que fueran necesarios.

El procedimiento también incluyó pasos para manejar casos en los que la referencia temporal sea general, como "por la tarde" o "mañana por la mañana". En estos casos, el *LLM* no asigna una hora exacta de inmediato, sino que propone un rango basado en las reglas establecidas (por ejemplo, "por la mañana" se traduce a un inicio entre 9:00 y 12:00), dejando que el sistema ajuste según el calendario. Un ejemplo práctico sería "reunión el viernes por la tarde": el *PlannerAgent*, con ayuda del *LLM*, podría programarla en una hora

entre las 15:00 y 21:00 del viernes más próximo, evitando solapamientos con otras tareas y definiendo una duración que considere adecuada, es decir, que primero identifica el rango de horas, luego el tiempo necesario para la tarea y finalmente elige un hueco disponible en ese rango.

La razón de diseñar el *PlannerAgent* de esta manera radica en su capacidad para adaptarse a las necesidades del usuario mientras mantiene un calendario organizado. Al confiar en el *LLM* para evaluar el contexto y proponer horarios, este agente asegura que las nuevas tareas se integren de forma lógica, apoyando la funcionalidad central del sistema y facilitando una experiencia de planificación eficiente.

4.6 Desarrollo del *AdjusterAgent*

El *AdjusterAgent* se encarga de ajustar tareas ya programadas cuando el usuario solicita un cambio, como mover una actividad a otro horario o fecha. Este agente fue diseñado para tomar decisiones inteligentes sobre la reubicación de tareas, utilizando exclusivamente la inteligencia artificial. A continuación, se detalla el procedimiento seguido para su desarrollo y se explica su funcionamiento, ilustrando cómo el *LLM* decide los nuevos horarios y asegura que las tareas se reorganicen de manera eficiente.

El desarrollo del *AdjusterAgent* comenzó con la creación de una clase que aprovechara las capacidades del *LLM*, específicamente el modelo *Gemini-2.0-Flash* de *Google Cloud*, para procesar solicitudes de ajuste de tareas. El primer paso fue diseñar una plantilla de instrucciones detallada que guiara al *LLM* en su tarea de reubicación. En esta plantilla, se le indicó que analizara la tarea a modificar, como "sacar al perro de 10:00 a 10:30 el 21 de mayo de 2025", junto con la solicitud del usuario, por ejemplo, "mover sacar al perro a las 11:00". También se le especifcó que considerara las tareas ya programadas en el calendario para evitar solapamientos, respetando las mismas reglas horarias que el *PlannerAgent*: las tareas solo pueden programarse entre las 9:00 y las 14:00 o entre las 15:00 y las 21:00. Además, se le solicitó que mantuviera las preferencias del usuario, como "por la mañana" o "por la tarde", y que, si no había espacio en la fecha indicada, sugiriera el siguiente día disponible.

El funcionamiento del *AdjusterAgent* se basa completamente en el *LLM* para tomar decisiones sobre los nuevos horarios. Cuando el *RootAgent* envía una solicitud de ajuste, como "mover sacar al perro a las 11:00", el *AdjusterAgent* recopila la información relevante: la tarea actual, el nuevo horario solicitado y las tareas ya programadas en el calendario para esa fecha. Esta información se pasa al *LLM* junto con la plantilla de instrucciones. El *LLM* analiza el contexto y decide el mejor horario para reubicar la tarea, asegurándose de evitar solapamientos. En este caso, el *LLM* confirma que las 11:00 están libres y reubica "sacar al perro" de 11:00 a 11:30, respetando la duración original de la tarea. Si no se especifica la nueva hora de la tarea o un momento concreto para posponer la tarea, por ejemplo, "posponer sacar al perro", el *LLM* pospondrá la tarea 1 hora, esta decisión permite que haya consistencia a la hora de posponer tareas sin que se especifique el nuevo horario.

Si el nuevo horario solicitado genera un conflicto, el *LLM* busca el siguiente hueco disponible en el mismo día o en el día siguiente. Por ejemplo, si el usuario solicita "mover sacar al perro a las 11:00" pero ese día ya hay una tarea asignada a las 11:00, el *LLM* buscará el siguiente hueco disponible en el mismo día. Si no

hubiera espacio en ese día, el *LLM* podría sugerir el día siguiente a las 11:00, asegurándose de que no haya conflictos con otras tareas programada para ese día.

El *AdjusterAgent* también maneja referencias temporales generales, como "por la tarde" o "mañana por la mañana". Por ejemplo, si el usuario dice "mover reunión a mañana por la tarde", el *LLM* analiza las tareas existentes para ese día y podría decidir programar la reunión entre las 15:00 y las 21:00, proponiendo un horario como de 15:00 a 16:00 si está libre. Una vez que el *LLM* toma la decisión, el *AdjusterAgent* organiza la información en un formato que incluye la nueva fecha, hora de inicio y hora de fin, y la pasa al *RootAgent* para actualizar el calendario.

Durante el desarrollo, se realizaron varias iteraciones sobre la plantilla de instrucciones para garantizar que el *LLM* tomara decisiones precisas. Se ajustaron las indicaciones para que priorizara mantener las preferencias del usuario, como el momento del día, y para que manejara casos complejos, como cuando no hay espacio disponible en la fecha solicitada.

La decisión de confiar exclusivamente en el *LLM* para determinar los nuevos horarios se fundamenta en su capacidad para evaluar el contexto de manera inteligente, considerando las restricciones horarias y las preferencias del usuario. Esto hace que el *AdjusterAgent* sea una herramienta flexible, capaz de adaptar el calendario a los cambios solicitados, asegurando que las tareas se reorganicen de manera lógica y sin complicaciones para el usuario.

4.7 Desarrollo de la interfaz gráfica

La interfaz gráfica del *Organizador de Tareas Inteligente* fue diseñada utilizando *Streamlit*, una herramienta que permite crear aplicaciones web interactivas de manera sencilla y eficiente. Esta interfaz se convirtió en el punto de contacto principal entre el usuario y el sistema, ofreciendo una experiencia visual e intuitiva para gestionar las tareas. Su desarrollo tuvo como objetivo principal facilitar la creación, visualización y eliminación de tareas, mientras se presentaba la información de forma clara y atractiva. A continuación, se detalla el procedimiento seguido para su implementación, se describen las acciones que el usuario puede realizar, y se incorporan las capturas de pantalla proporcionadas para ilustrar los elementos clave de la interfaz, como los botones y las secciones disponibles.

El desarrollo de la interfaz comenzó con la configuración inicial de *Streamlit*, utilizando su funcionalidad básica para estructurar la página. Se estableció un diseño general con un título principal, "Sistema de Gestión de Tareas", que se muestra en la parte superior de la pantalla con un ícono de una agenda para darle un toque visual. Para que la interfaz fuera atractiva y fácil de leer, se añadieron estilos personalizados mediante un bloque de código que define colores y formatos, como un fondo blanco para las tarjetas de tareas y sombras suaves que se activan al pasar el cursor por encima. Este diseño hace que la experiencia sea más agradable, destacando elementos clave como los horarios de las tareas en azul y los detalles en un texto más grande y oscuro. La pantalla principal se muestra en la Figura 4-3.

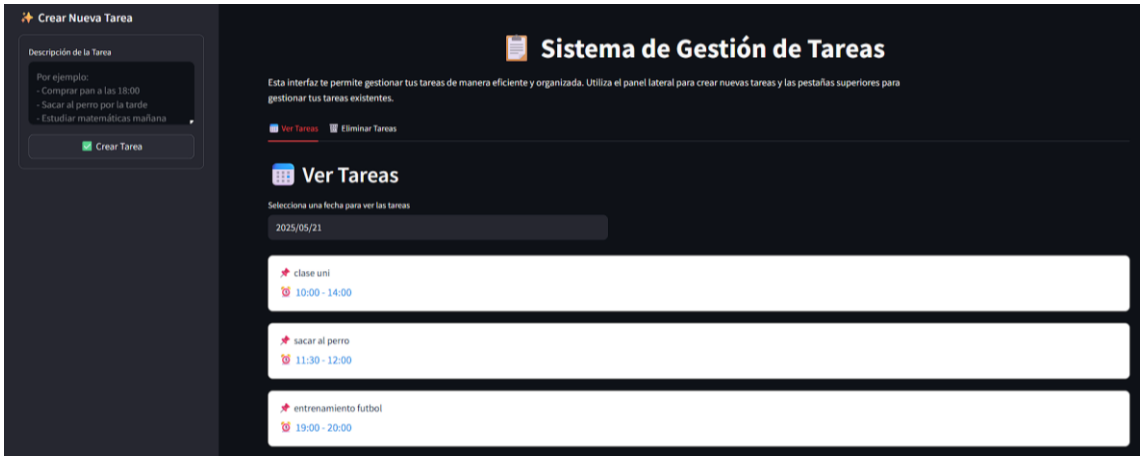


Figura 4-3. Pantalla principal de la interfaz

La interfaz se divide en varias secciones funcionales, comenzando con una barra lateral a la izquierda, dedicada a la creación de nuevas tareas (véase la Figura 4-4). En esta sección, titulada "Crear Nueva Tarea", se encuentra un formulario donde el usuario puede escribir instrucciones, como "comprar pan por la tarde". El formulario incluye un área de texto con ejemplos sugeridos ("- Comprar pan a las 18:00", "- Sacar al perro por la tarde", "- Estudiar matemáticas mañana") y un botón "Crear Tarea" que, al ser pulsado, envía la instrucción al *RootAgent* para procesarla. Una vez que la tarea se crea, por ejemplo, "sacar al perro por la tarde", aparece un mensaje de confirmación verde que dice algo como "Tarea creada: sacar al perro | Día: miércoles | Hora: 20:30 – 21:00", como se aprecia en la Figura 4-5, desapareciendo tras unos segundos para mantener la pantalla limpia.

Figura 4-4. Formulario de creación de tareas

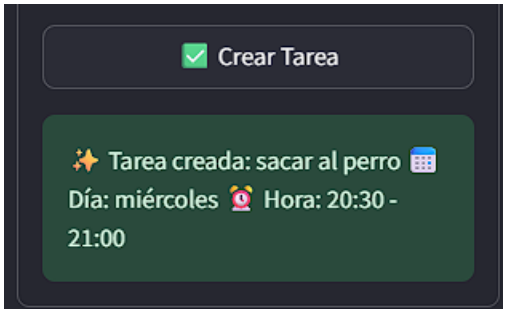


Figura 4-5. Pop-up de tarea creada

La sección principal de la interfaz está organizada en dos pestañas: "Ver Tareas" y "Eliminar Tareas" (véase la Figura 4-6). En la pestaña "Ver Tareas", el usuario puede seleccionar una fecha mediante un calendario interactivo. Por ejemplo, al elegir el 20 de mayo de 2025, la interfaz muestra una lista de tareas programadas para ese día, como "clase uni de 10:00 a 14:00". Cada tarea se presenta en una tarjeta con un diseño claro, mostrando el detalle (clase uni) y el horario (10:00 - 14:00) en letras grandes y colores distintivos, Figura 4-7. Si no hay tareas para la fecha seleccionada, se muestra un mensaje azul que dice "No hay tareas programadas para esta fecha", asegurando que el usuario siempre reciba retroalimentación (véase la Figura 4-8). Además, la interfaz permite al usuario navegar por el calendario para seleccionar otras fechas, como se observa en la Figura 4-9.

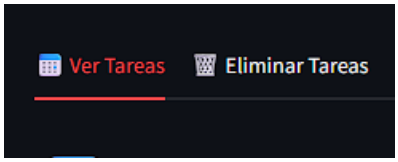


Figura 4-6. Pestañas disponibles

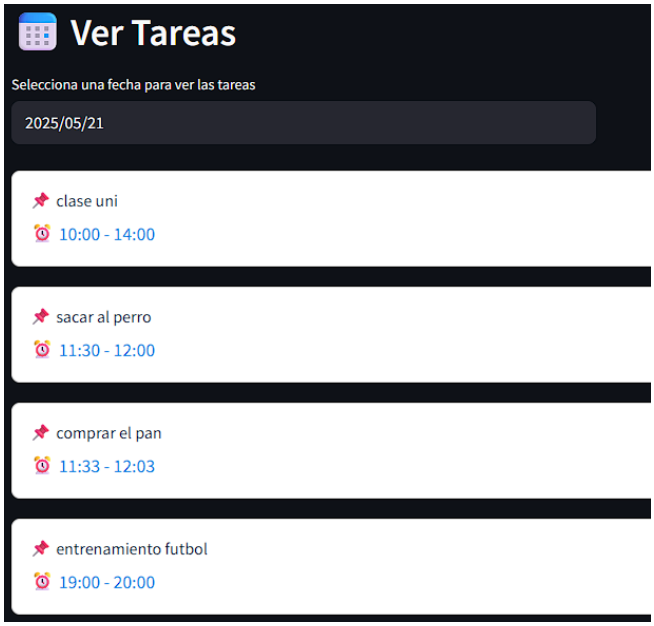


Figura 4-7. Tareas disponibles

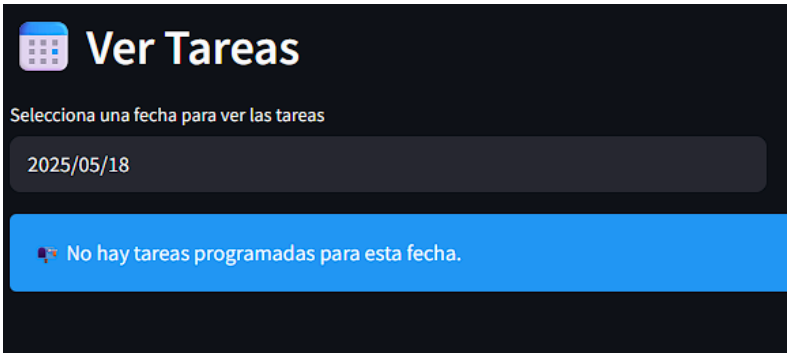


Figura 4-8. Día sin tareas programadas

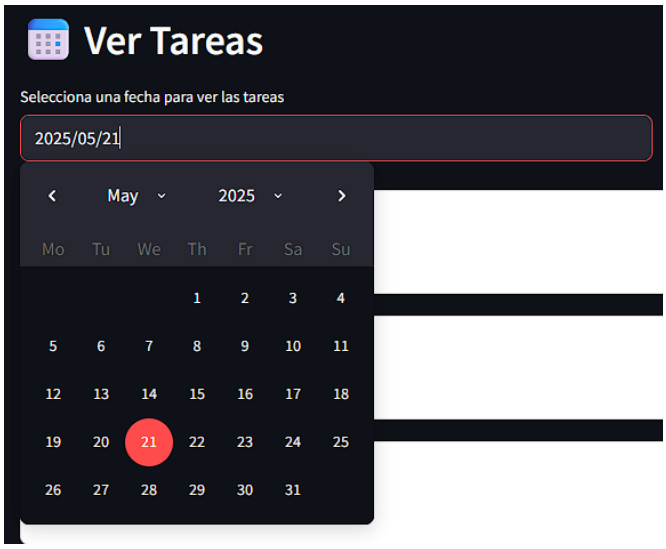


Figura 4-9. Calendario

La pestaña “Eliminar Tareas” se divide en dos subpestañas: “Eliminar Tarea Específica” y “Eliminar Todas” (Figura 4-10). En la primera, el usuario selecciona una fecha y ve las tareas disponibles, presentadas de manera similar a la pestaña anterior. A la derecha de cada tarjeta de tarea, hay un botón rojo “Eliminar”. Por ejemplo, al seleccionar el 22 de mayo de 2025 y pulsar “Eliminar” junto a “clase uni de 10:00 a 14:00”, la tarea se borra y aparece un mensaje verde que confirma “¡Tarea eliminada exitosamente!”. Si no hay tareas, se muestra el mensaje “No hay tareas en esta fecha” (véase la Figura 4-11). En la subpestaña “Eliminar Todas”, hay un botón rojo destacado “Eliminar Todas las Tareas”, acompañado de un mensaje de advertencia que dice “Esta acción eliminará todas las tareas y no se puede deshacer”. Al pulsarlo, todas las tareas se borran, y se muestra un mensaje de confirmación verde “¡Todas las tareas han sido eliminadas!”, como se puede ver en la Figura 4-12.

Figura 4-10. Sección de eliminar tareas

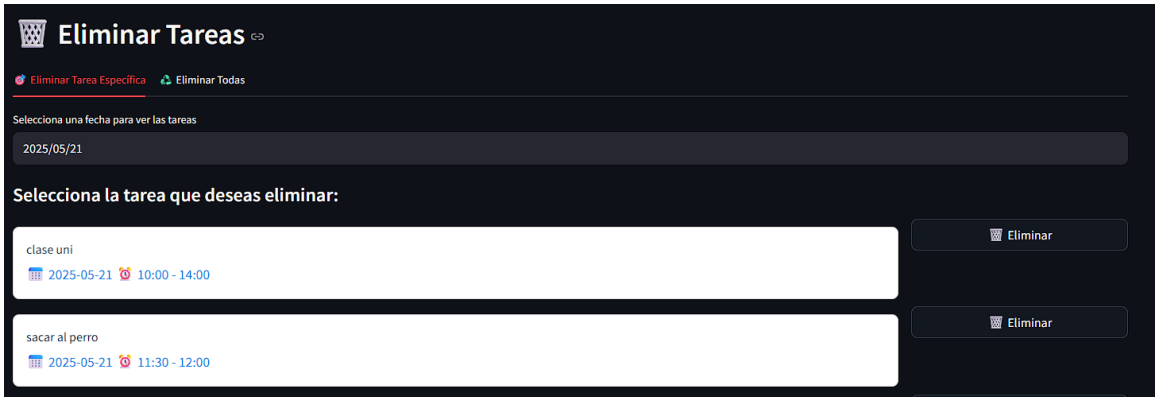


Figura 4-11. Eliminar tarea específica

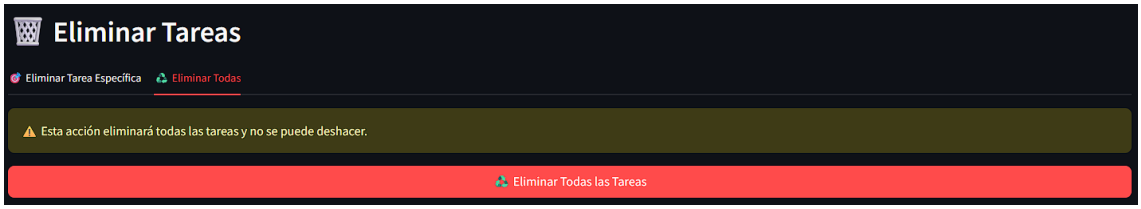


Figura 4-12. Eliminar todas las tareas

El procedimiento de desarrollo incluyó varias iteraciones para conectar la interfaz con el *RootAgent*, asegurándose de que cada acción (crear, ver, eliminar) interactuara correctamente con el calendario. También se ajustaron los estilos para que los botones y mensajes fueran claros y consistentes, usando colores como verde para confirmaciones y rojo para advertencias. *Streamlit* fue elegido por su simplicidad y capacidad para crear interfaces interactivas rápidamente, lo que permitió sentrarse en la funcionalidad sin complicaciones técnicas. Esta interfaz proporciona al usuario una manera sencilla y visual de gestionar sus tareas, haciendo que el sistema sea accesible y práctico para cualquier persona.

5 Evaluación y Discusión

La evaluación del *Organizador de Tareas Inteligente* se basa en un análisis práctico que permite validar su diseño y funcionalidad a través de un caso de uso realista, reflejando las necesidades de un usuario típico. Este apartado tiene como objetivo principal demostrar cómo el sistema procesa instrucciones en lenguaje natural, gestiona tareas de manera eficiente y se adapta a cambios dinámicos, utilizando la arquitectura multiagente y la interfaz gráfica desarrollada. A través de este proceso, se podrán identificar fortalezas, limitaciones y áreas de mejora, proporcionando una base sólida para discutir su desempeño en comparación con herramientas existentes y para proponer futuras optimizaciones.

5.1 Evaluación del caso de uso

El caso de uso es una parte fundamental de la evaluación del *Organizador de Tareas Inteligente*, ya que permite demostrar su aplicabilidad práctica en un escenario real y analizar cómo responde a diferentes tipos de entradas del usuario, además de ser uno de los objetivos impuestos en este TFG. A continuación, se presenta la descripción del caso de uso, incluyendo los detalles de los inputs utilizados (tanto válidos como no válidos), que reflejan las actividades y necesidades del usuario creado para este caso de uso.

Descripción del escenario

El caso de uso se centra en un estudiante universitario ficticio que necesita gestionar sus tareas diarias durante una semana típica, del 26 de mayo de 2025 al 1 de junio de 2025. Este estudiante asiste a clases de lunes a viernes de 10:00 a 14:00, es un deportista que juega al fútbol y entrena los lunes y miércoles de 19:00 a 20:00, y comparte la responsabilidad de pasear a su perro con su hermano sin un horario fijo. Además, es el encargado de comprar pan en un supermercado cercano con cierta regularidad, disfruta de ir de compras los fines de semana, y debe planificar la entrega de trabajos universitarios y tareas del hogar como limpiar su cuarto, sacar el lavaplatos y bajar la basura. El objetivo de este caso de uso fue validar la capacidad del sistema para interpretar instrucciones en lenguaje natural, programar tareas considerando horarios fijos y variables, y ajustarlas dinámicamente según las necesidades del estudiante, utilizando la interfaz gráfica de *Streamlit* para facilitar la interacción. Para probar este caso de uso, primero se estableció en el calendario los horarios fijos del estudiante, es decir, tanto la tarea de ir a la universidad de lunes a viernes de 10:00 a 14:00, como la tarea de entrenar al fútbol los lunes y miércoles de 19:00 a 20:00.

Inputs utilizados

Para comenzar a probar este caso de uso, se empezó realizando pruebas básicas, primero se escribió el input “sacar al perro esta tarde”, para añadir la tarea en el mismo día 26 de mayo, lo que resultó en que el sistema creo la tarea “sacar al perro” para el día 26 de mayo de 17:15 a 17:45, como se puede ver en la Figura 5-1.

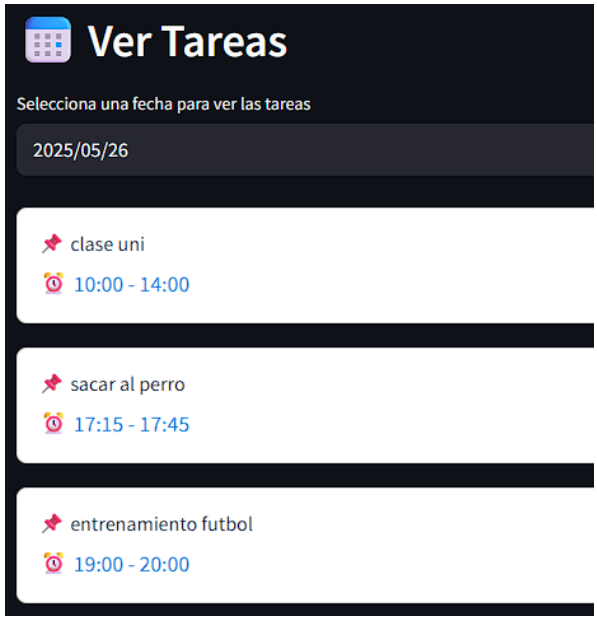


Figura 5-1. Primera prueba

Tras esto, se decidió probar a posponer la tarea de sacar al perro, sin especificar una hora concreta, para probar la funcionalidad, por lo que el sistema lo pospuso una hora, al no haber solapamiento con otras tareas. Además, se añadió una nueva tarea para sacar el lavaplatos, para ello se escribió el input "sacar el lavaplatos hoy", y el sistema decidió añadir la tarea antes de sacar al perro, como se puede ver en la Figura 5-2. Como última tarea del día 26 de mayo, se decidió usar un input más complicado, ya que se quería ver si el sistema era capaz de añadir una tarea antes o después de otra tarea especificándolo en el propio input, por lo que se escribió "bajar la basura después del entrenamiento de futbol", el *LLM* fue capaz de identificar correctamente los requisitos del usuario, ya que se pudo ver en la "*explanation*" devuelta por el *JSON* del *EvaluatorAgent*, que efectivamente interpretó de manera correcta el input: `"explanation": "El usuario quiere crear una nueva tarea para bajar la basura. El momento en que desea realizar la tarea es después del entrenamiento de fútbol. No hay información sobre la fecha específica."` Y por tanto añadió la tarea correctamente, como puede apreciarse en la Figura 5-3.

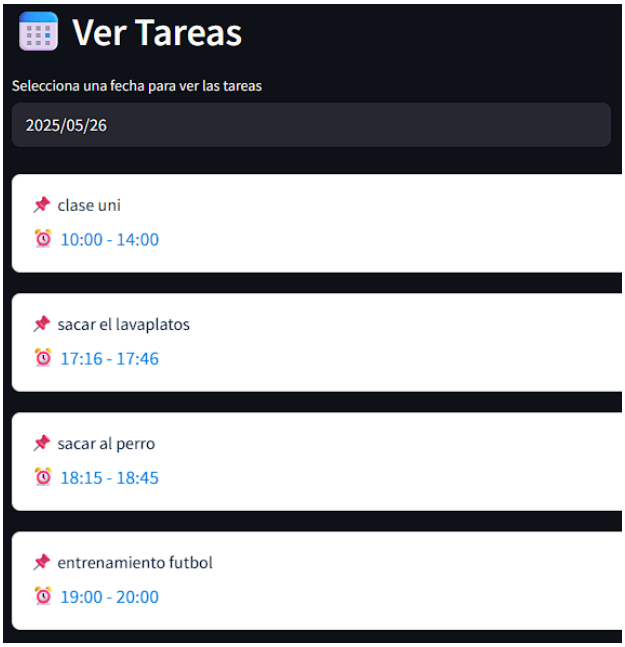


Figura 5-2. Segunda prueba

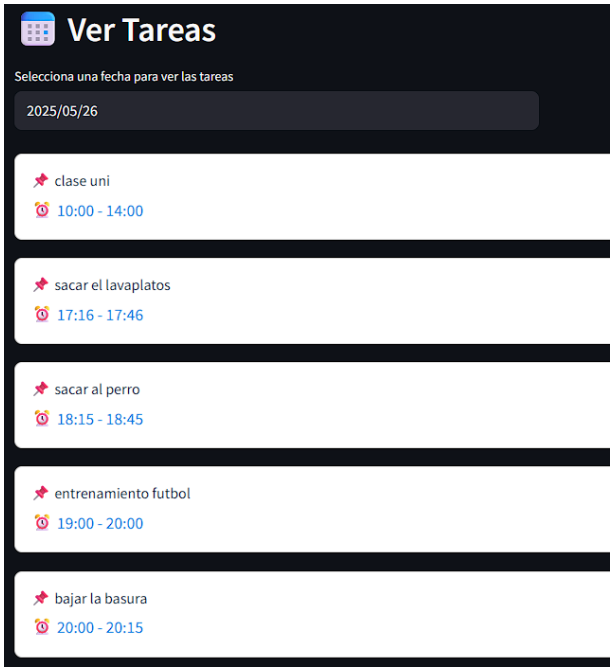


Figura 5-3. Tercera prueba

Una vez acabadas las pruebas para el primer día, ahora se empezó a usar inputs con referencias temporales de distintos días, “mañana”, “el martes”, “dentro de dos días” y otras más que se definirán a continuación. Por lo tanto, primeramente, se probó el siguiente input: “comprar el pan mañana por la mañana”, para ver si el sistema era capaz de añadir la tarea antes de la clase de universidad, pero después de las 9:00 que es la primera hora a la que se le especifica al *LLM* que puede añadir una tarea y efectivamente lo realizó de forma correcta, añadió la tarea comprar el pan de 9:00 a 9:15, como muestra la Figura 5-4.

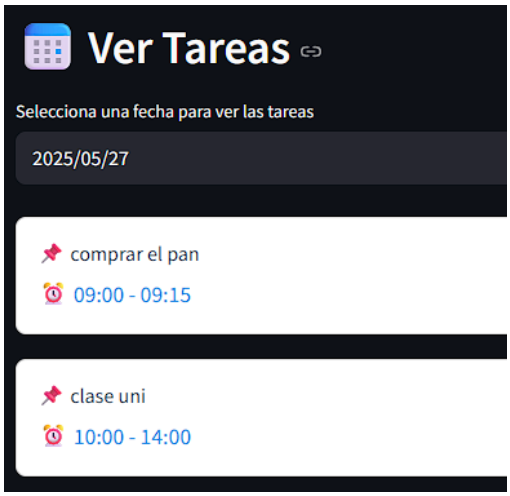


Figura 5-4. Cuarta prueba

A continuación, se probó con un input para crear una tarea en un horario en el que ya hay una tarea creada, por lo tanto, al haber solapamiento el sistema debería modificar la hora para añadirla en un hueco disponible. El input sería: “Sacar al perro mañana a las 10:00”, que solapa con la tarea de “clase uni” sin embargo, el sistema fue capaz de identificar el solapamiento y por tanto el *LLM* decidió añadir la tarea después, concretamente a las 15:00, por lo que además queda asegurado que no se añaden tareas entre las 14:00 y las 15:00, como se estableció en el *prompt* de los diferentes agentes, esta prueba puede verse en la Figura 5-5.

Figura 5-5. Quinta prueba

Para las siguientes pruebas, se usaron inputs para crear tareas sencillas, pero con diferentes referencias temporales, para corroborar cuales era capaz de identificar correctamente el *LLM* y si había alguna que no fuera capaz de comprender. Por ello, se probaron los siguientes inputs: "limpiar el cuarto el miércoles", "estudiar pasado mañana", "ir de compras el sábado por la mañana", "comprar leche en 3 días" o "sacar al perro el viernes entre las 17:00 y las 20:00" (esta última prueba se realizó para comprobar si el *LLM* interpretaba correctamente que el usuario quería añadir la tarea en esa franja horaria, sin que la tarea durara las 2 horas). El *LLM* fue capaz de reconocer todas las referencias temporales y añadió las tareas correctamente, por lo que quedaba demostrado que es capaz de interpretar correctamente el lenguaje natural usado en los inputs, los resultados de estas pruebas pueden verse en las figuras: Figura 5-6, Figura 5-7, Figura 5-8, Figura 5-9.

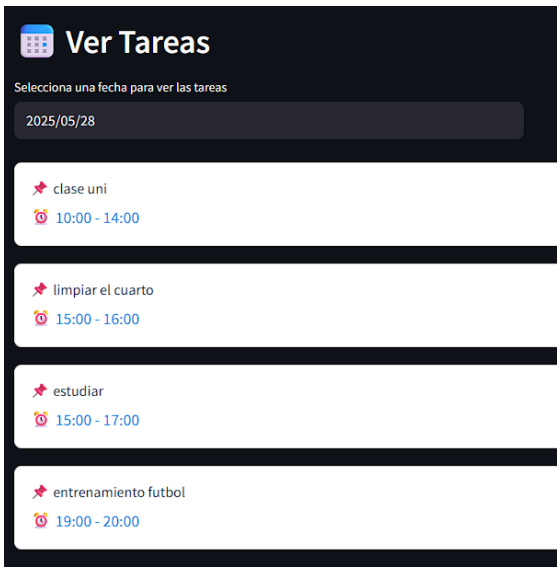


Figura 5-6. Sexta prueba

Figura 5-7. Séptima prueba

Figura 5-8. Octava prueba

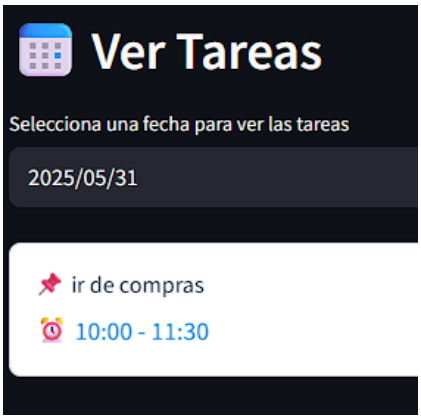


Figura 5-9. Novena prueba

A continuación, se realizaron pruebas para verificar el funcionamiento del *AdjusterAgent*, y ya que la creación de tareas funciona, comprobar si a la hora de modificar el sistema funciona de la misma manera. Para ello se probaron los diferentes inputs: "posponer sacar al perro", "cambiar sacar la basura a las 20:30", "posponer sacar el lavaplatos a mañana", "mover comprar el pan al viernes". En estos casos, al igual que con la creación de tareas, el *LLM* era capaz de interpretar correctamente las entradas del usuario, modificaba las tareas correctamente, identificando que no hubiera solapamientos y si los había, modificando la hora conforme a ello. Sin embargo, en la tarea de sacar el lavaplatos, el *LLM* cambió la duración inicial que tenía la tarea, esto puede deberse a que decidió que la duración que primeramente estableció el *PlannerAgent* no era correcta para la tarea.

Los resultados de estas pruebas se reflejan en las figuras: Figura 5-10, Figura 5-11, Figura 5-12.

Figura 5-10. Décima prueba

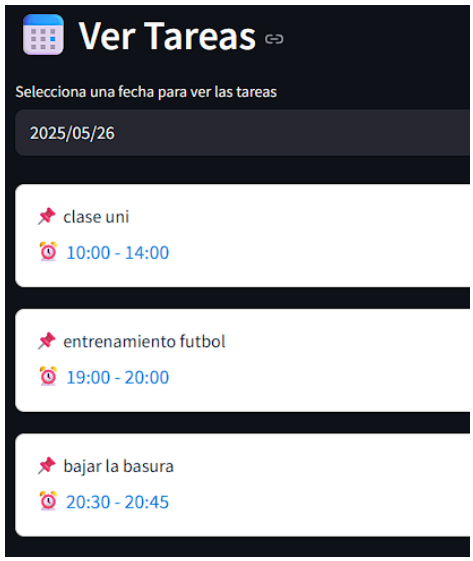


Figura 5-11. Undécima prueba

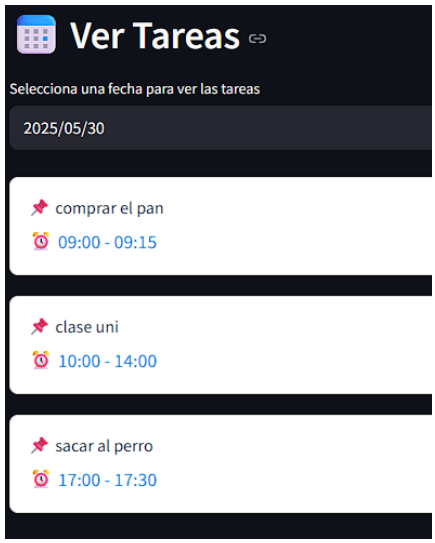


Figura 5-12. Duodécima prueba

Por último, se realizaron otras pruebas para comprobar que el sistema no modificaba pruebas que no existían previamente o que si se le pasaba un input que no tenía que ver con tareas lo interpretara de manera correcta. Para ello se probaron los siguientes inputs: "posponer ir al médico" (no existía la tarea ir al médico previamente), "hola que tal", "adsgsadg" o "qué temperatura hace?". Para el caso de posponer una tarea que no existe previamente, el *EvaluatorAgent* interpreta de manera correcta que la intención del usuario es modificar una tarea, sin embargo, al pasarle la información al *AdjusterAgent*, este agente devuelve que no existe la tarea a modificar, por lo tanto, se para la ejecución y no modifica ninguna tarea. Para el input de "hola que tal", el *EvaluatorAgent* interpreta que el usuario simplemente está saludando y que no hay información de ninguna tarea, lo mismo sucede con el input "¿qué temperatura hace?", para el que el *LLM* interpreta que el usuario pregunta por la temperatura y que no hay información sobre ninguna tarea. Sin embargo, para el input "adsgsadg", el *LLM* interpreta que el usuario quiere crear una tarea, pero que, al no contener información útil, lo analiza como si el usuario proporcionara una entrada vacía, y decide suponer que el usuario está intentando crear una nueva tarea, pero sin proporcionar detalles, por lo que crea una tarea llamada "tarea no especificada" y le otorga una duración de 3 horas.

Resultados y observaciones

Los inputs válidos fueron procesados correctamente en la mayoría de los casos, demostrando la eficacia del *EvaluatorAgent* para interpretar lenguaje natural y del *PlannerAgent* y *AdjusterAgent* para asignar y reorganizar horarios sin solapamientos. La interfaz de *Streamlit* facilitó la visualización de tareas, con mensajes de confirmación claros y un diseño intuitivo. Sin embargo, el manejo de entradas sin sentido, como "adsgsadg", evidenció una limitación: el sistema no rechaza estas entradas ni pide aclaraciones, sino que las interpreta como tareas genéricas, lo que podría generar confusión, sugiriendo que el sistema podría beneficiarse de un mecanismo para solicitar aclaraciones al usuario.

5.2 Discusión

La discusión de los resultados obtenidos en el caso de uso permite reflexionar sobre el desempeño del *Organizador de Tareas Inteligente* y compararlo con aplicaciones existentes, identificando fortalezas, limitaciones y oportunidades de mejora. Aunque no se realizaron pruebas cuantitativas exhaustivas debido a las limitaciones de un entorno académico, se ha realizado una comparativa cualitativa con herramientas populares como *Google Calendar*, *Microsoft To Do* y *Todoist* para contextualizar las capacidades del sistema.

Una de las principales fortalezas del *Organizador de Tareas Inteligente* es su capacidad para procesar instrucciones en lenguaje natural, lo que lo distingue de aplicaciones como *Google Calendar*, donde el usuario tendría que introducir manualmente "Sacar al perro el 26 de mayo de 17:15 a 17:45" mediante un formulario. En este sistema, basta con escribir "sacar al perro esta tarde", y el *EvaluatorAgent* y *PlannerAgent* gestionan la tarea automáticamente, lo que resulta especialmente útil para usuarios que prefieren una interacción más natural.

Sin embargo, aplicaciones como *Microsoft To Do* ofrecen funcionalidades que el sistema aún no incluye, como notificaciones automáticas. Por ejemplo, *Microsoft To Do* podría enviar un recordatorio antes de "Entrenamiento fútbol" el 26 de mayo a las 19:00, mientras que el *Organizador de Tareas Inteligente*, al priorizar la gestión local con archivos *JSON*, no ofrece esta opción, limitando su utilidad para usuarios que necesitan alertas. Esta autonomía también implica que no se sincroniza con dispositivos externos, a diferencia de *Google Calendar*.

Comparado con *Todoist*, el sistema destaca por su ajuste dinámico, como "bajar la basura después del entrenamiento de fútbol", interpretado correctamente por el *LLM*. *Todoist* requiere reprogramación manual, pero incluye etiquetas y prioridades (por ejemplo, "urgente" para "Entregar trabajo de matemáticas"), que el sistema no ofrece, lo que podría ser una desventaja para tareas complejas.

Un desafío identificado fue la dificultad con entradas sin sentido, como "adsgsadg". El sistema, en lugar de rechazarlas, las interpreta como tareas genéricas, lo que podría confundir al usuario. Aplicaciones como *Google Calendar* evitan esto al requerir campos obligatorios.

La arquitectura multiagente permite escalabilidad, como añadir agentes de notificaciones, pero la falta de integración con plataformas externas lo hace menos competitivo frente a soluciones comerciales. En conclusión, el sistema ofrece un enfoque innovador con lenguaje natural, pero sus limitaciones en notificaciones, robustez frente a inputs sin sentido y conectividad sugieren áreas clave para mejoras futuras.

6 Conclusiones y trabajo futuro

La evaluación del Organizador de Tareas Inteligente a través del caso de uso ha permitido comprobar su viabilidad y eficacia como una herramienta innovadora basada en inteligencia artificial y arquitectura multiagente. Este apartado sintetiza los resultados obtenidos, destacando los logros alcanzados, las limitaciones identificadas y las lecciones aprendidas durante el desarrollo y las pruebas, para sentar las bases de futuras mejoras y ampliaciones del sistema.

6.1 Conclusiones

Conclusión principal

El desarrollo del *Organizador de Tareas Inteligente* ha cumplido con el objetivo principal de este Trabajo de Fin de Grado (TFG), que era desarrollar un *framework* basado en una arquitectura de agentes de servicios de IA generativa, permitiendo la integración modular de herramientas en diferentes aplicaciones. La implementación del sistema ha demostrado que es posible combinar el procesamiento de lenguaje natural con una arquitectura multiagente para automatizar la gestión de tareas de manera eficiente, adaptable y escalable. La integración del modelo *Gemini-2.0-Flash* a través de *Vertex AI* ha permitido al *EvaluatorAgent* interpretar con alta precisión entradas como "sacar al perro esta tarde" o "bajar la basura después del entrenamiento de fútbol", mientras que agentes como el *PlannerAgent* y el *AdjusterAgent* han gestionado la planificación y los ajustes dinámicos en la gran mayoría los casos analizados. Este enfoque modular facilita la incorporación de nuevas funcionalidades en el futuro, validando el potencial del *framework* para aplicaciones diversas más allá de la gestión de tareas, como la planificación de proyectos o la organización de eventos.

Conclusiones secundarias

1. **Analizar y seleccionar el entorno tecnológico más adecuado para el desarrollo del *framework***. Este objetivo se ha cumplido satisfactoriamente. Se analizaron y seleccionaron herramientas como *Python* como lenguaje principal de desarrollo por su versatilidad y amplio soporte para IA, *LangChain* para facilitar la integración y el manejo del modelo de lenguaje *Gemini-2.0-Flash*, *Streamlit* para la interfaz gráfica, *Google Cloud* con *Vertex AI* para acceder al *LLM*, y archivos *JSON* para la gestión local de datos. Estas elecciones permitieron desarrollar un sistema funcional, ya que *Python* y *LangChain* proporcionaron un entorno robusto para implementar los agentes y procesar las entradas del usuario, mientras que *Streamlit* aseguró una interfaz accesible.

2. **Diseñar la arquitectura del sistema definiendo la estructura y las interacciones entre los agentes**. Este objetivo también se ha alcanzado con éxito. La arquitectura multiagente, compuesta por el *EvaluatorAgent*, *RootAgent*, *PlannerAgent* y *AdjusterAgent*, mostró una clara división de responsabilidades, con interacciones bien definidas que permitieron el procesamiento distribuido y la planificación eficiente. La estructura modular diseñada asegura la escalabilidad del sistema, como se evidenció al ajustar tareas dinámicamente durante las pruebas.

3. **Implementar el *framework* con los agentes de IA generativa asegurando su funcionalidad y escalabilidad**. Este objetivo se ha cumplido, ya que el *framework* fue implementado con agentes funcionales que colaboran para gestionar tareas. La capacidad de interpretar referencias temporales complejas, como "mañana por la mañana" o "en 3 días", y de ajustar horarios sin solapamientos demuestra su funcionalidad. Además, la arquitectura modular garantiza escalabilidad, permitiendo la incorporación de nuevos agentes en el futuro.

4. **Desarrollar un caso de uso que valide la utilidad y aplicación del *framework* en un entorno real**. Este objetivo se ha logrado completamente. El caso de uso basado en un estudiante universitario permitió validar la utilidad del sistema en un escenario realista, gestionando tareas como asistir a clases, entrenar al fútbol y realizar tareas domésticas. Las pruebas mostraron que el sistema puede manejar un horario complejo y adaptarse a cambios, como posponer tareas y añadir nuevas, lo que confirma su aplicabilidad práctica.

5. **Realizar pruebas para evaluar el desempeño, eficiencia y facilidad de integración del sistema**. Este objetivo también se ha cumplido. Las pruebas realizadas demostraron un alto desempeño en el procesamiento de lenguaje natural y la planificación, con una gran mayoría en inputs válidos. La interfaz de *Streamlit* resultó fácil de usar y eficiente, aunque se identificaron limitaciones, como el manejo de entradas sin sentido ("adsgsadg"), que el sistema interpreta como tareas genéricas en lugar de rechazarlas.

Conclusión final

El TFG ha permitido explorar el potencial de los sistemas multiagentes y los modelos de lenguaje de gran escala en un contexto práctico, validando la hipótesis de que estas tecnologías pueden automatizar la gestión de tareas de manera eficiente. La experiencia adquirida en la selección de tecnologías como *Streamlit* y *Google Cloud*, así como en el diseño de agentes especializados, ha enriquecido el conocimiento técnico y ha confirmado la viabilidad de integrar soluciones de IA generativa en aplicaciones cotidianas. Aunque el sistema no alcanza la madurez de aplicaciones comerciales debido a sus limitaciones actuales, ha cumplido con los objetivos específicos de analizar el entorno tecnológico, diseñar la arquitectura, implementar el framework y validar su utilidad en un caso de uso real, proporcionando un punto de partida sólido para investigaciones y desarrollos posteriores.

6.2 Trabajo futuro

El desarrollo del *Organizador de Tareas Inteligente* ha sentado las bases para un sistema prometedor, pero existen múltiples oportunidades para su mejora y expansión en futuros trabajos. Una de las prioridades sería mejorar la interfaz gráfica, actualmente implementada con *Streamlit*, para ofrecer una experiencia más atractiva y personalizable. Esto podría incluir un calendario visualmente más elaborado, con colores personalizables para diferentes tipos de tareas (por ejemplo, azul para clases, verde para deportes, rojo para urgencias) y animaciones suaves que faciliten la navegación, como transiciones al cambiar entre días o al añadir una nueva tarea. Además, integrar la posibilidad de sincronizar con calendarios externos, como *Google Calendar* o *Microsoft Outlook*, permitiría al usuario combinar las tareas generadas por el sistema con eventos ya existentes en estas plataformas, mejorando la interoperabilidad y la utilidad en entornos profesionales o académicos.

Otra área clave para el futuro sería la incorporación de tareas concurrentes, permitiendo al sistema gestionar actividades que puedan realizarse simultáneamente, como escuchar un podcast mientras se pasea al perro o revisar notas durante un trayecto en transporte público. Esto requeriría al *PlannerAgent* desarrollar un algoritmo más sofisticado que analice la naturaleza de las tareas y determine si son compatibles, basándose en información proporcionada por el usuario (por ejemplo, "escuchar podcast mientras camino" o "leer mientras espero"). Además, añadir notificaciones sería un paso esencial para competir con aplicaciones como *Microsoft To Do*, implementando alertas personalizables (por ejemplo, un recordatorio 15 minutos antes de "Entrenamiento fútbol" el 28 de mayo a las 19:00) que puedan enviarse a través de notificaciones *push* en dispositivos móviles o correos electrónicos, mejorando la experiencia del usuario al mantenerlo al tanto de sus compromisos.

Un aspecto importante para trabajos futuros sería la capacidad de crear tareas recurrentes, permitiendo al sistema programar actividades que se repiten regularmente. Por ejemplo, si el usuario introduce "sacar al perro todos los lunes", el *PlannerAgent* podría añadir esta tarea automáticamente cada lunes, ajustándola según los horarios disponibles y evitando conflictos con otras tareas fijas, como entrenamientos de fútbol. Esto requeriría implementar una lógica adicional para manejar patrones de repetición (diarios, semanales, mensuales) y permitir al usuario editar o cancelar estas recurrencias, aumentando la flexibilidad del sistema para rutinas habituales.

Otra mejora significativa sería transformar el sistema en uno basado en usuarios, similar a los calendarios de *Google* o *Microsoft*, donde cada usuario tenga un perfil individual con su propio calendario. Esto habilitaría la posibilidad de compartir tareas con otros usuarios, fomentando la colaboración. Por ejemplo, si el usuario introduce "reunión el viernes con Juan García a las 16:00", el sistema podría identificar a Juan García como otro usuario registrado, enviarle una notificación para que acepte o rechace la tarea, y, de aceptarla, añadirla automáticamente a su calendario personal con el mismo horario. Esto requeriría desarrollar un módulo de gestión de usuarios con autenticación, un sistema de notificaciones entre usuarios (por ejemplo, mediante correo o notificaciones *push*), y un mecanismo para sincronizar tareas compartidas, asegurando que ambos usuarios reciban actualizaciones si la reunión cambia de horario o se cancela.

Otras ideas para el trabajo futuro incluyen la integración de un sistema de aprendizaje por refuerzo para que los agentes mejoren sus decisiones con el tiempo. Por ejemplo, el *AdjusterAgent* podría aprender de los hábitos del usuario, como el hecho de que prefiere posponer tareas por la mañana al mediodía, y sugerir horarios óptimos de forma proactiva. Asimismo, se podría explorar la incorporación de un agente de priorización que clasifique las tareas según su urgencia o importancia, similar a las etiquetas de *Todoist* ("urgente" o "personal"), permitiendo al usuario definir prioridades como "Entregar trabajo de matemáticas" por encima de "Ir de compras". Esto enriquecería la funcionalidad del sistema y lo haría más competitivo frente a herramientas comerciales.

Otra posibilidad sería añadir soporte multilingüe al *EvaluatorAgent*, permitiendo que el sistema procese instrucciones en otros idiomas, como inglés o francés, lo que ampliaría su alcance a usuarios internacionales. Esto requeriría integrar modelos de lenguaje adicionales o adaptar *Gemini-2.0-Flash* para manejar múltiples lenguas, junto con una interfaz que detecte automáticamente el idioma del usuario. Además, se podría desarrollar un módulo de análisis de productividad que genere informes semanales o mensuales, mostrando al usuario cuánto tiempo dedicó a cada tipo de tarea (estudio, deportes, hogar) y sugiriendo ajustes para optimizar su rutina, como reducir el tiempo dedicado a tareas domésticas si se detecta un exceso.

La seguridad y la privacidad también son aspectos a considerar. Dado que el sistema actual depende de *Google Cloud*, un trabajo futuro podría centrarse en implementar una versión completamente local, utilizando modelos de lenguaje de código abierto como *LLaMA* o modelos optimizados para hardware local, lo que eliminaría la necesidad de conexión a internet y protegería mejor los datos personales del usuario. Esto podría complementarse con un sistema de cifrado para los archivos *JSON*, asegurando que la información sensible, como horarios de clases o entrenamientos, permanezca confidencial.

Finalmente, se podría explorar la integración con dispositivos inteligentes del hogar, como asistentes virtuales (*Amazon Alexa*, *Google Home*) o sensores que detecten la ubicación del usuario (por ejemplo, al salir de casa, activar una tarea como "Comprar pan"). Esto requeriría desarrollar un agente de conexión *IoT* que traduzca las señales de estos dispositivos en tareas para el sistema, ampliando su funcionalidad a un entorno conectado. Estas mejoras no solo resolverían las limitaciones actuales, como la falta de robustez con entradas sin sentido, sino que también posicionarían al *Organizador de Tareas Inteligente* como una solución innovadora y versátil en el campo de la gestión personal asistida por IA.

7 Análisis de Impacto

El *Organizador de Tareas Inteligente* desarrollado en este Trabajo de Fin de Grado (TFG) tiene el potencial de generar impactos significativos en diversos contextos, tanto a nivel individual como colectivo. Este capítulo analiza el impacto esperado de los resultados obtenidos en los ámbitos personal, empresarial, social, económico, medioambiental y cultural, destacando los beneficios anticipados, los posibles efectos negativos y las decisiones tomadas durante el desarrollo que consideraron estos impactos. Además, se vinculan los resultados con los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030 [20], identificando cómo el sistema puede contribuir a metas globales de sostenibilidad.

Impacto personal

A nivel personal, el Organizador de Tareas Inteligente ofrece una herramienta práctica para mejorar la gestión del tiempo y reducir el estrés asociado con la organización de actividades diarias. Para un usuario como el estudiante universitario del caso de uso, que maneja distintas tareas, como clases (10:00-14:00 de lunes a viernes), entrenamientos de fútbol (lunes y miércoles 19:00-20:00), y tareas domésticas como sacar al perro o comprar pan, el sistema proporciona una solución intuitiva que automatiza la planificación mediante instrucciones en lenguaje natural, como "sacar al perro esta tarde". Esto permite al usuario dedicar menos tiempo a la organización manual y más a actividades productivas o de ocio, mejorando su bienestar mental y su productividad personal. La interfaz gráfica de *Streamlit*, con su diseño claro y mensajes de confirmación, también facilita una experiencia de usuario accesible, incluso para personas no expertas en tecnología.

Impacto Empresarial

En un contexto empresarial, el sistema tiene el potencial de optimizar la gestión de tareas en equipos pequeños o profesionales independientes, como autónomos o startups. Por ejemplo, un freelance podría usar el sistema para programar reuniones, gestionar entregas de proyectos y organizar tareas administrativas mediante instrucciones como "programar una reunión el viernes por la tarde". La arquitectura multiagente permite escalabilidad, lo que podría adaptarse para coordinar tareas entre varios usuarios, asignando roles a los agentes para gestionar agendas compartidas. Esto podría mejorar la eficiencia operativa y reducir el tiempo dedicado a la planificación, permitiendo a las empresas centrarse en sus objetivos principales.

Un posible impacto negativo sería el costo asociado con el uso de servicios en la nube como *Google Cloud*, que podría limitar su adopción por empresas con presupuestos ajustados. Además, la falta de integración con herramientas empresariales como *Microsoft Outlook* o *Slack* podría dificultar su implementación en entornos corporativos más grandes. Durante el desarrollo, se priorizó la simplicidad y el uso de tecnologías accesibles como *Streamlit* y *Python*, lo que facilita la adopción por pequeñas empresas, aunque se reconoce la necesidad de integración con herramientas externas en el futuro.

Impacto social

Socialmente, el *Organizador de Tareas Inteligente* puede fomentar una mayor inclusión al ofrecer una herramienta accesible para personas que no están familiarizadas con tecnologías complejas. Al permitir instrucciones en lenguaje natural, como "comprar pan mañana por la mañana", el sistema elimina barreras técnicas, beneficiando a usuarios de diferentes edades y niveles educativos, desde estudiantes hasta personas mayores. Esto podría promover una sociedad más conectada y productiva, al facilitar la organización personal en comunidades diversas.

Sin embargo, la dependencia de conexión a internet para acceder a Gemini-2.0-Flash podría excluir a usuarios en áreas con acceso limitado a la red, incrementando la brecha digital. Durante el desarrollo, se tomó la decisión de usar un *LLM* avanzado para garantizar precisión en el procesamiento del lenguaje, pero esto limitó la accesibilidad en entornos offline. En el futuro, implementar una versión local con modelos de código abierto podría solventar este impacto.

Impacto económico

Económicamente, el sistema tiene el potencial de generar ahorros al reducir el tiempo invertido en la planificación manual, lo que podría traducirse en mayor productividad para individuos y empresas. Por ejemplo, un estudiante que automatiza su horario con el sistema puede dedicar más tiempo al estudio o al descanso, mejorando su rendimiento académico, mientras que una pequeña empresa podría optimizar sus operaciones sin necesidad de invertir en software comercial costoso. Además, el uso de tecnologías accesibles como *Python* y *Streamlit* reduce los costos de desarrollo, haciendo que el sistema sea una solución económica para usuarios y desarrolladores.

Un posible efecto negativo sería el costo recurrente de servicios en la nube, que podría ser una barrera para usuarios o empresas con recursos limitados. Durante el desarrollo, se decidió usar *Google Cloud* para acceder a *Gemini-2.0-Flash* debido a su disponibilidad y capacidad, pero esto implica un costo que podría limitar la escalabilidad comercial del sistema si no se explora una alternativa local.

Impacto medioambiental

Desde el punto de vista medioambiental, el *Organizador de Tareas Inteligente* tiene un impacto mixto. Por un lado, al optimizar la gestión del tiempo, puede reducir desplazamientos innecesarios; por ejemplo, al programar "comprar pan" y "sacar al perro" de manera eficiente, el usuario podría combinar estas actividades en un solo viaje, disminuyendo su huella de carbono. Esto se alinea con el ODS 11 (Ciudades y Comunidades Sostenibles) [20], que promueve una planificación eficiente para reducir el impacto ambiental. Por otro lado, la dependencia de *Google Cloud* implica un consumo energético significativo, ya que los centros de datos que soportan estos servicios generan emisiones de carbono. Durante el desarrollo, no se priorizó la optimización energética debido a las limitaciones del proyecto, pero se reconoce que una versión local con modelos de IA más ligeros podría reducir este impacto, contribuyendo al ODS 13 (Acción por el Clima) [20] al minimizar el uso de recursos computacionales intensivos.

Impacto cultural

Culturalmente, el sistema puede influir en las prácticas de organización personal, promoviendo una cultura de productividad y eficiencia. En contextos donde la planificación manual es predominante, como en comunidades con menor acceso a tecnología, el sistema podría introducir un cambio hacia la automatización, fomentando una mayor adopción de herramientas digitales. Sin embargo, esto podría generar resistencia en culturas que valoran las tradiciones de organización manual, como el uso de agendas físicas. Además, la falta de soporte multilingüe limita su alcance a usuarios que hablen español, lo que podría excluir a comunidades culturales diversas. Durante el desarrollo, se decidió priorizar el idioma español para garantizar precisión en el procesamiento del lenguaje, pero se identifica como un área de mejora para incluir más idiomas en el futuro, promoviendo la inclusión cultural y alineándose con el ODS 10 (Reducción de las Desigualdades) [20].

Relación con los Objetivos de Desarrollo Sostenible (ODS)

El *Organizador de Tareas Inteligente* se alinea con varios ODS de la Agenda 2030 [20]. En primer lugar, contribuye al ODS 3 (Salud y Bienestar) al reducir el estrés asociado con la gestión del tiempo, mejorando la calidad de vida de los usuarios. También se relaciona con el ODS 4 (Educación de Calidad), ya que estudiantes como el del caso de uso pueden optimizar su tiempo para dedicar más horas al aprendizaje, como programar "estudiar pasado mañana". Como se mencionó, el ODS 11 y el ODS 13 se ven impactados positivamente por la potencial reducción de desplazamientos, pero negativamente por el consumo energético de la nube. Finalmente, el ODS 10 podría beneficiarse si se implementa soporte multilingüe y accesibilidad offline, reduciendo desigualdades en el acceso a la tecnología.

Decisiones basadas en el impacto

A lo largo del desarrollo, varias decisiones se tomaron considerando el impacto potencial. Se eligió *Streamlit* para la interfaz gráfica debido a su simplicidad y bajo costo, promoviendo la accesibilidad (impacto social y económico). La gestión local con archivos *JSON* se implementó para minimizar la dependencia de la nube, reduciendo preocupaciones de privacidad (impacto personal) y consumo energético (impacto medioambiental), aunque no se eliminó por completo debido a la necesidad de usar *Gemini-2.0-Flash*. Finalmente, se priorizó el procesamiento de lenguaje natural en español para garantizar precisión, pero esto limitó la inclusividad cultural, una decisión que se planea revertir en futuros desarrollos.

Conclusión del análisis de impacto

El *Organizador de Tareas Inteligente* tiene un impacto potencialmente positivo en los ámbitos personal, empresarial y social al mejorar la productividad, la accesibilidad y la calidad de vida, mientras que su impacto económico y medioambiental presenta oportunidades y desafíos. El sistema podría contribuir significativamente a los ODS 3, 4, 10, 11 y 13 si se abordan las

limitaciones actuales, como la dependencia de la nube y la falta de multilingüismo. Este análisis destaca la importancia de considerar estos impactos en el diseño y desarrollo de soluciones tecnológicas, asegurando que los beneficios superen los posibles efectos negativos y que el sistema evolucione hacia una herramienta más inclusiva y sostenible.

8 Bibliografía

[1] S. Ahmed, "*AI Agents in 2025: A Comprehensive Review and Future Outlook*," *Medium*, 2025. [En línea] Disponible: https://medium.com/@sahin.samia/current-trends-in-ai-agents-use-cases-and-the-future-ahead-1026c4d753fd

[2] Rannaberg, C. (2025). *State of AI Agents in 2025: A Technical Analysis*. Medium. Disponible: https://medium.com/@carlrannaberg/state-of-ai-agents-in-2025-5f11444a5c78

[3] Google Cloud Documentation. Disponible: https://cloud.google.com/docs?hl=es-419. Fecha de consulta: 25/05/2025

[4] S. J. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 3.ª ed. Upper Saddle River, NJ, USA: Pearson, 2020.

[5] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2.ª ed. Chichester, UK: Wiley, 2020.

[6] A. Dorri, S. S. Kanhere y R. Jurdak, "*Multi-agent systems: A survey*," *IEEE Access*, vol. 6, pp. 28573–28593, 2018.

[7] T. Brown et al., "*Language models are few-shot learners*" en *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[8] Modelos de Gemini, *Google AI for Developers* [En línea]. Disponible: https://ai.google.dev/gemini-api/docs/models?hl=es-419

[9] K. Zhang, Z. Yang y T. Başar, "*Multi-agent reinforcement learning: A selective overview of theories and algorithms*" en *Handbook of Reinforcement Learning and Control*, pp. 321–384

[10] "*LangChain* Documentation," *LangChain* Inc., 2023. [En línea]. Disponible: https://python.*LangChain*.com/docs/get_started/introduction. Fecha de consulta: 27/05/2025

[11] "Haystack Documentation," deepset, 2023. [En línea]. Disponible: https://haystack.deepset.ai/docs/latest/intro. Fecha de consulta: 27/05/2025

[12] "Rasa Documentation", Rasa, 2025. [En línea]. Disponible: https://rasa.com/docs/. Fecha de consulta: 27/05/2025

[13] "LlamaIndex Documentation", [En línea]. Disponible: https://docs.llamaindex.ai/en/stable/. Fecha de consulta: 27/05/2025

[14] "AutoGen Documentation", Microsoft. [En línea]. Disponible: https://microsoft.github.io/autogen/0.2/docs/contributor-guide/documentation/. Fecha de consulta: 27/05/2025

[15] “Streamlit Documentation”, Streamlit. [En línea]. Disponible: https://docs.streamlit.io/. Fecha de consulta: 27/05/2025

[16] “Dash Documentation”, Plotly. [En línea]. Disponible: https://dash.plotly.com/. Fecha de consulta: 27/05/2025

[17] “Flask Documentation”, [En línea]. Disponible: https://flask.palletsprojects.com/en/stable/. Fecha de consulta: 27/05/2025

[18] “Gradio Documentation”, [En línea]. Disponible: https://www.gradio.app/docs. Fecha de consulta: 27/05/2025

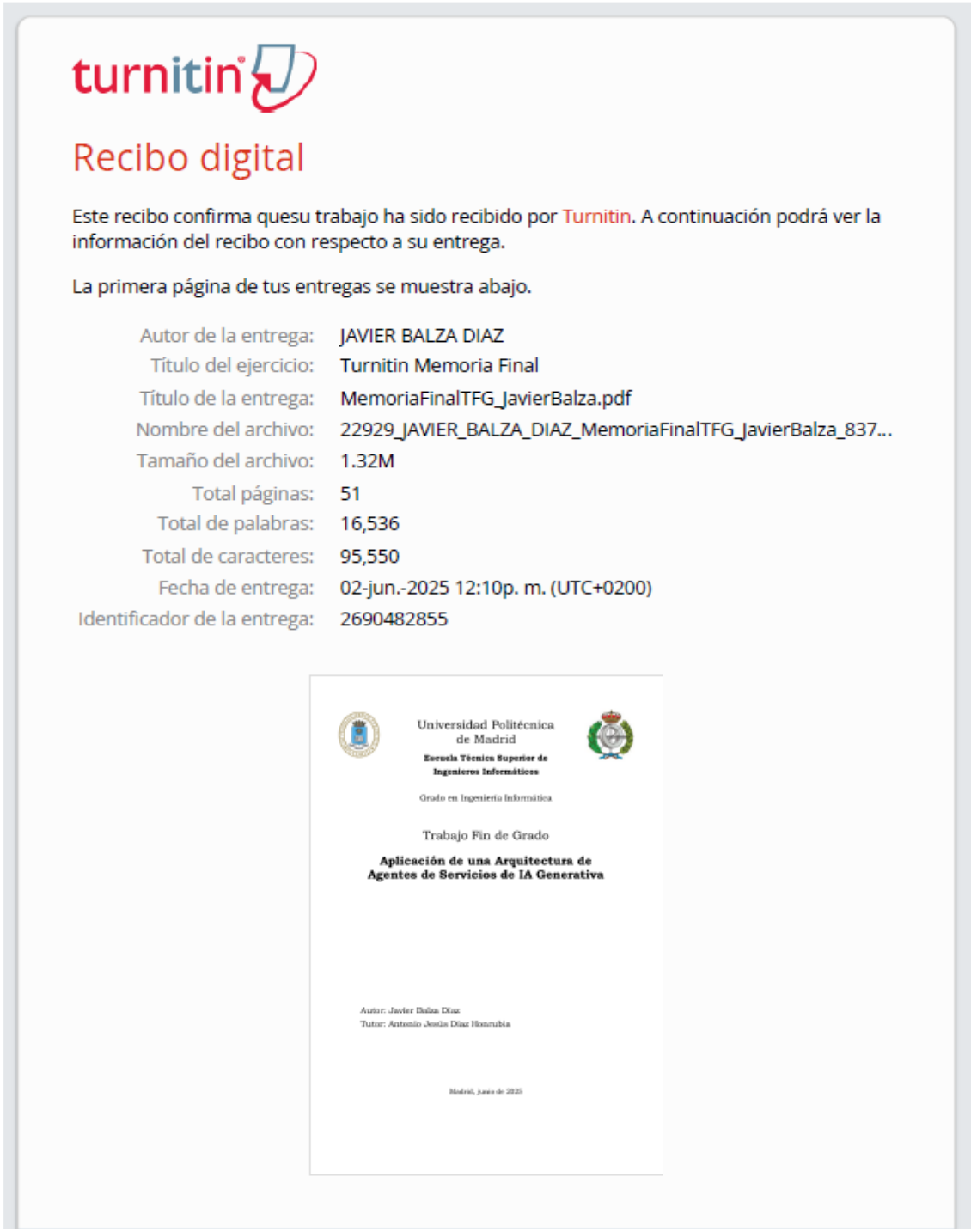
[19] “Shiny for Python Documentation”, [En línea]. Disponible: https://shiny.posit.co/py/docs/overview.html. Fecha de consulta: 27/05/2025

[20] “Agenda 2030 del Desarrollo Sostenible”, UAM, [En línea]. Disponible: https://ods.uam.es/agenda-2030-y-ods/. Fecha de consulta: 28/05/2025

9 Anexo

Comprobante digital

En la Figura 9-1 se muestra el comprobante digital de la presente memoria del TFG obtenido mediante la página web de *Turnitin.*


turnitin

Recibo digital

Este recibo confirma quesu trabajo ha sido recibido por Turnitin. A continuación podrá ver la información del recibo con respecto a su entrega.

La primera página de tus entregas se muestra abajo.

Autor de la entrega:	JAVIER BALZA DIAZ
Título del ejercicio:	Turnitin Memoria Final
Título de la entrega:	MemoriaFinalTFG_JavierBalza.pdf
Nombre del archivo:	22929_JAVIER_BALZA_DIAZ_MemoriaFinalTFG_JavierBalza_837...
Tamaño del archivo:	1.32M
Total páginas:	51
Total de palabras:	16,536
Total de caracteres:	95,550
Fecha de entrega:	02-jun.-2025 12:10p. m. (UTC+0200)
Identificador de la entrega:	2690482855

Figura 9-1. Comprobante digital de Turnitin

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jun 04 13:23:56 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)