Universidad de Cantabria

Facultad de Ciencias

SOLUCIÓN PARA LA IMPLANTACIÓN DE UN SISTEMA DE GESTIÓN DE INCIDENCIAS (TICKETING) AUTOMATIZADO

(Solution for the implementation of an automated incident management (ticketing) system)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Paula Muñoz Fernández
Director: Diego García Saiz
Codirector: David Irizábal Vélez
Julio - 2025

Resumen

En este proyecto se busca mejorar la gestión ineficiente de incidencias en la empresa IPS Norte, que hasta ahora solo utilizaba el correo electrónico para este fin de forma manual. Para solucionarlo se propuso un sistema de ticketing automatizado basado en Odoo, un Enterprise Resource Planning (ERP) de código abierto. Este sistema integró procesamiento de correos electrónicos y técnicas de Inteligencia Artificial (IA) para clasificar automáticamente cada incidencia, asignar prioridad y responsable, además de llevar un seguimiento del proceso.

En la práctica, se diseñaron módulos en Python para limpiar y normalizar los mensajes (extrayendo remitente, asunto, fecha, cuerpo, etc.), detectar y eliminar duplicados, y crear tickets en Odoo mediante su Application Programming Interface (API), utilizando el protocolo JavaScript Object Notation-Remote Procedure Call (JSON-RPC). También se empleó procesamiento concurrente mediante multihilo sobre el protocolo Internet Message Access Protocol (IMAP), para recibir y convertir correos entrantes en tiempo real y se adoptaron buenas prácticas basadas en el marco Information Technology Infrastructure Library (ITIL), por ejemplo, asignación de plazos según Service Level Agreement (SLA), mejorando el rendimiento del sistema.

De forma colectiva, la solución desarrollada automatiza el flujo completo desde la recepción de un correo hasta el cierre del ticket, mejorando el soporte técnico y reduciendo tareas manuales.

Palabras Clave: Gestión de incidencias, Odoo, IA, Python, clasificación automática.

Abstract

This proyect aims to improve the inefficient incident management at the company IPS Norte, which until now relied solely on manual email handling. To address this, an automated ticketing system based on Odoo, an open source Enterprise Resource Planning (ERP) platform, was proposed. The system integrated email processing and Artificial Intelligence (AI) techniques to automatically classify each incident, assign a priority and a responsible, as well as tracking the progress of the process.

In practice, Python modules were designed to clean and normalize the messages (extracting sender, subject, date, body, etc.), detect and eliminate duplicates, and create tickets in Odoo through its Application Programming Interface (API), using the JavaScript Object Notation-Remote Procedure Call (JSON-RPC) protocol. Concurrent processing using multithreading was also implemented over the Internet Message Access Protocol (IMAP), to receive and convert incoming emails in real time and best practices based on the Information Technology Infrastructure Library (ITIL) framework were also adopted, such as assigning deadlines based on Service Level Agreements (SLA), improving system performance.

Collectively, the developed solution automates the entire workflow, from receiving an email to closing the ticket, symplifying technical support and reducing manual tasks.

Keywords: Incident management, Odoo, AI, Python, automatic classification.

Índice

1 Introducción **5**

1.1 Contexto del proyecto . . . 5

1.2 Motivación . . . 5

1.3 Objetivos . . . 6

1.4 Alcance del proyecto . . . 6

2 Estado del Arte **8**

2.1 Conceptos clave en la gestión de incidencias . . . 8
2.1.1 Gestión de incidencias e ITIL . . . 8
2.1.2 SLA y métricas de calidad del servicio . . . 8
2.1.3 Automatización e IA en la gestión de incidencias . . . 8

2.2 Evolución de los sistemas de gestión de incidencias . . . 9

2.3 Comparación de herramientas actuales . . . 10

2.4 Odoo como solución escogida . . . 11

3 Metodología y Herramientas usadas **12**

3.1 Metodología de desarrollo utilizada . . . 12

3.2 Aplicación de buenas prácticas ITIL . . . 12

3.3 Herramientas utilizadas y entorno de desarrollo . . . 12
3.3.1 Lenguaje y librerías de Python . . . 13
3.3.2 Otras Herramientas . . . 13

3.4 Justificación del uso de IA . . . 14

4 Requisitos del sistema **15**
4.0.1 Requisitos funcionales . . . 15
4.0.2 Requisitos no funcionales . . . 16

5 Diseño e Implementación **17**

5.1 Arquitectura del sistema . . . 17
5.1.1 Arquitectura del funcionamiento interno . . . 17

5.2 Despliegue del entorno Odoo y módulo Helpdesk mediante Docker . . . 19

5.3 Módulos reutilizables para procesado de correos . . . 20
5.3.1 Procesamiento y limpieza de correos . . . 20
5.3.2 Tratamiento de correos duplicados . . . 22
5.3.3 Clasificación automática de categoría, prioridad y plazos del ticket . . . 23
5.3.4 Conexión y comunicación con Odoo . . . 24
5.3.5 Creación de tickets . . . 25

5.4 Procesamiento de correos entrantes en tiempo real . . . 26
5.4.1 Análisis comparativo de métodos de integración . . . 26
5.4.2 Arquitectura General del Sistema de Correos Entrantes . . . 29
5.4.3 Paralelismo del sistema . . . 30
5.4.4 Implementación del sistema multihilo . . . 31
5.4.5 Conexión y escucha en tiempo real de correos entrantes . . . 31
5.4.6 Ejecución global de los correos entrantes . . . 33

5.5 Procesamiento de correos históricos . . . 33
5.5.1 Estructura de los datos . . . 33
5.5.2 Extracción de correos desde Outlook . . . 34
5.5.3 Arquitectura General del Sistema de Correos Históricos . . . 35
5.5.4 Acceso y lectura de correos . . . 36
5.5.5 Extracción de correos embebidos . . . 37

5.5.6 Procesamiento de correos embebidos . . . 37
5.5.7 Control de duplicados . . . 38
5.6 Asignación de técnicos mediante algoritmos de IA . . . 38
5.6.1 Elección del modelo para clasificar . . . 39
5.6.2 Descripción y funciones de la API . . . 41
5.6.3 Integración de la base de datos de Odoo en la API . . . 42
5.6.4 Predicción del técnico . . . 42
6 Evaluación de Resultados **44**
6.0.1 Pruebas para los correos entrantes y la creación de históricos . . . 44
6.0.2 Pruebas de la clasificación del técnico . . . 44
6.1 Discusión de resultados y limitaciones . . . 46
7 Conclusiones **46**
8 Trabajo futuro **47**
9 Bibliografía **48**
A Apéndice **50**
A.1 Acrónimos . . . 50
A.2 Flujos de Procesamiento . . . 51
A.2.1 Flujo de procesamiento del sistema general . . . 51
A.2.2 Flujo de procesamiento del sistema de recepción de correos entrantes . . . 53
A.2.3 Flujo de procesamiento de la creación de la base de datos históricos . . . 53
A.3 Matrices de Confusión . . . 55
A.4 Manual de Usuario . . . 58
A.4.1 Manual para montar y acceder al entorno de Odoo . . . 58
A.4.2 Manual para ejecutar el script que sube base de datos de Odoo a la API . . . 60
A.4.3 Manual para ejecutar y probar el script que extrae los técnicos y documentos devueltos por la API . . . 61

Índice de Figuras

1 Diagrama de clases del sistema general 18
2 Estructura de un ticket (campos principales) 26
3 Estructura de un ticket (plazos) 26
4 Diagrama de clases del sistema de recepción de correos entrantes 30
5 Diagrama de clases del sistema de creación de la base de datos histórico 36
6 Resultado de aplicar las métricas Accuracy y Top-3 Accuracy 45
7 Flujo de procesamiento del sistema general (Apéndice) 52
8 Flujo de procesamiento del sistema de recepción de correos entrantes (Apéndice) . 53
9 Flujo de procesamiento del sistema de creación de la base de datos histórico (Apéndice) 54
10 Matriz de Confusión para texto limpio sin NLP (Apéndice) 55
11 Matriz de Confusión para texto con NLP usando tokenización + stopwords (Apéndice) 56
12 Matriz de Confusión con lematización (Apéndice) 57
13 Matriz de Confusión con stemming (Apéndice) 58
14 Pantalla de inicio tras acceder a la interfaz web de Odoo (Apéndice) 59
15 Pantalla de instalación del módulo Helpdesk (Apéndice) 59
16 Pantalla del servicio de asistencia Helpdesk (Apéndice) 60
17 Indexación de tickets de Odoo en la API (Apéndice) 60
18 Técnicos predichos por la API (Apéndice) 61

Índice de Tablas

1 Comparativa de herramientas de gestión de incidencias 10
2 Librerías de Python empleadas en el sistema 13
3 Herramientas del sistema 14
4 Requisitos funcionales del sistema 15
5 Requisitos no funcionales del sistema 16
6 Componentes del sistema interno 18
7 Componentes del sistema de correos entrantes y sus roles principales 29
8 Estructura de la base de datos para el modelo de IA 34
9 Componentes del sistema y sus roles principales 35
10 Comparativa de Enfoques de Clasificación 40

Índice de fragmentos de código

1 Archivo docker-compose.yml 19
2 Inicialización de objetos para acceder a carpetas de Outlook con win32com 34
3 Expresión regular para dividir correos 37
4 Configuración de la API privada 41
5 Ejecución del predictor con un ejemplo (Apéndice) 61

1. Introducción

En este apartado se presenta el problema que tiene la empresa IPS Norte con su sistema de gestión de incidencias y la necesidad de optimizarlo. También se detallan los motivos del proyecto, los objetivos propuestos y el alcance del sistema.

1.1. Contexto del proyecto

Hoy en día, la gestión eficiente de incidencias es fundamental para que las empresas sigan funcionando, especialmente aquellas que están adaptando sus procesos a entornos digitales. En este aspecto, los departamentos de soporte técnico y de Tecnologías de la información (TI) tienen que atender cada vez más solicitudes sin perder calidad ni control. A menudo, herramientas tradicionales como el correo electrónico no son lo ideal, ya que pueden causar retrasos y cuellos de botella.

Aunque herramientas como Outlook son útiles para comunicarse, no están pensadas para trabajar como sistemas de gestión de incidencias puesto que no permiten hacer seguimiento de los correos, asignar técnicos de forma automática, establecer prioridades basadas en reglas de negocio ni supervisar el proceso completo.

Esta situación se encuentra dentro de la empresa IPS Norte donde su proceso de gestión de incidencias consiste actualmente, en el uso del correo electrónico corporativo para recibir y resolver solicitudes. Los correos son atendidos de forma manual por los técnicos, sin un sistema centralizado de tickets ni herramientas especificas para esta función. Esto conlleva problemas de saturación, duplicación de tareas, falta de organización, y gran dependencia del conocimiento de los técnicos.

Para resolver estas limitaciones y desvincular al correo de realizar este tipo de tareas, se propone la implementación de un sistema completo basado en Odoo [44], un ERP [29] modular y de código abierto que permite gestionar desde un solo lugar los procesos fundamentales de la empresa, pudiendo desarrollar un sistema de ticketing personalizado.

Además, el proyecto incorpora técnicas de IA para analizar el contenido de los correos, asignarles una categoría técnica, una prioridad y, en caso de los correos nuevos, un técnico responsable de forma automática. Esta combinación de automatización y seguimiento busca mejorar el soporte técnico, haciéndolo más eficiente, rápido y organizado, siguiendo buenas prácticas como las del marco ITIL [8].

En resumen, este proyecto no solo resuelve una necesidad real de la empresa IPS Norte, sino que también demuestra cómo el uso de tecnologías actuales como la automatización, la IA y los sistemas ERP puede mejorar el funcionamiento de la empresa.

1.2. Motivación

La motivación de este proyecto surge de un problema real: optimizar un sistema de gestión de incidencias manual, ineficiente, lento y poco organizado.

Este cambio busca que los técnicos dejen de hacer tareas repetitivas, como clasificar y repartir incidencias, y puedan centrarse en trabajos más importantes. Al mismo tiempo, se quiere responder más rápido y dar un mejor servicio a los clientes, mejorando así la imagen y la competitividad de la empresa.

A nivel personal, este proyecto es una gran oportunidad para aplicar de forma práctica los conocimientos adquiridos durante la carrera, especialmente en el área de la IA. Trabajar con algoritmos de clasificación en un entorno real me permitirá reforzar habilidades en el diseño de soluciones informáticas.

1.3. Objetivos

El objetivo general del proyecto es desarrollar una solución técnica y sólida, basada en Odoo e IA para automatizar y optimizar la gestión de incidencias del Servicio de Asistencia Técnica (SAT) de la empresa IPS Norte. Esta solución buscará hacer el trabajo más eficiente+ y aumentar la satisfacción del cliente.

Los objetivos específicos son los siguientes:

- Diseñar e implementar un módulo personalizado de Helpdesk en Odoo que permita organizar mejor los correos, para que el equipo de soporte trabaje de forma más ordenada y pueda atender las incidencias de forma más rápida y eficaz.
- Detectar todos los correos nuevos de una cuenta de correo para poder convertirlos en tickets sin intervención manual.
- Tener una base de datos útil y bien estructurada, para entrenar la IA con ejemplos reales.
- Limpiar y procesar los correos para que la información relevante se almacene correctamente en Odoo facilitando su uso.
- Unificar los correos nuevos y ya resueltos en una misma base de datos de tickets para poder tratarlos y gestionarlos de la misma manera. Así, las solicitudes estarán mejor organizadas y será más fácil hacer seguimiento de cada caso.
- Automatizar todo el proceso de gestión de incidencias, desde que se recibe el correo hasta que se resuelve el caso, incluyendo avisos y actualizaciones del estado, para ahorrar tiempo y evitar errores.
- Clasificar los tickets según su tipo y prioridad y asignarles un plazo para resolverlos, para que cada incidencia se atienda según su prioridad y se cumplan los acuerdos establecidos (ITIL y SLA).
- Usar algoritmos de IA para clasificar automáticamente las incidencias y asignarlas al técnico más adecuado en cada caso, con el objetivo de repartir mejor el trabajo y ganar tiempo.
- Diseñar el sistema por partes (modular) para que sea más fácil de entender, mantener y mejorar en el futuro.

Estos objetivos permiten diseñar un sistema robusto y automatizado que mejora significativamente la gestión de incidencias en el entorno empresarial.

1.4. Alcance del proyecto

En este sistema se gestionan todas las etapas de la gestión de incidencias desde que llega un mensaje a la cuenta de correo hasta que se crea el ticket, se clasifica y se asigna de manera automática. Para conseguir este objetivo, se ha desarrollado un módulo personalizado en Odoo junto con otros componentes que se conectan a la cuenta de correo. Estos componentes permiten recoger tanto los correos nuevos como los resueltos, guardarlos en una base de datos y usar una API privada que ayuda a asignar automáticamente un técnico a cada caso.

Entre las funcionalidades cubiertas se incluyen:

- Detección automática de nuevos correos en cuentas de correo con IMAP.
- Limpieza y procesamiento de los cuerpos de correo y extracción de datos principales.
- Se lleva un registro de los correos ya procesados para evitar duplicación.
- Los correos resueltos se transforman en tickets válidos para crear una base de datos de entrenamiento.
- El sistema clasifica automáticamente los tickets por categoría, prioridad y plazo usando técnicas Natural Language Processing (NLP) y clustering para la categoría.

- Se predice el técnico más adecuado comparando descripciones similares mediante vectores semánticos y un sistema de votación.
- Todo el proceso, desde la llegada del correo hasta la asignación del técnico, está automatizado e incluye avisos.
- El sistema usa una estructura modular y está documentado para facilitar su mantenimiento en el futuro.
- Se han seguido buenas prácticas de seguridad con el uso de conexiones seguras como Secure Sockets Layer/Transport Layer Security (SSL/TLS) para recibir correos y claves API para servicios externos, lo que garantiza una protección adecuada de los datos.

Quedan fuera del alcance de este proyecto tareas más avanzadas por complejidad y falta de tiempo como:

- Evaluar de forma completa otros modelos de IA (por ejemplo, mediante fine-tuning supervisado).
- Conectar el sistema con varias cuentas de correo al mismo tiempo e incorporar nuevos canales (como chat o formularios).
- Asignar técnicos por área o especialidad para mejorar la predicción y tener en cuenta su disponibilidad para evitar que tengan sobrecarga de tareas.
- Mejorar el sistema de prioridad, ya que actualmente se basa solo en palabras clave, añadiendo mecanismos más precisos que permitan ajustarse a criterios avanzados de los SLA.
- Crear un portal para que los clientes puedan seguir sus tickets en tiempo real y proponerles soluciones con IA para resolver sus consultas.
- Utilizar técnicas más desarrolladas y complejas de seguridad del sistema como la encriptación de datos.

2. Estado del Arte

En este apartado se presentan los conceptos básicos relacionados con la gestión de incidencias en entornos de TI, así como su evolución a lo largo de los años. También se analizan las principales herramientas disponibles en el mercado, y se explica por qué se ha elegido Odoo Helpdesk como base para el desarrollo del sistema. Gracias al análisis realizado, se puede entender el contexto y las necesidades actuales del tema tratado.

2.1. Conceptos clave en la gestión de incidencias

Antes de comparar las distintas soluciones para gestionar las incidencias, en esta sección se explican los conceptos básicos sobre ellas, los SLA y el papel de la IA y la automatización en el soporte técnico, para entender mejor las decisiones tomadas en el desarrollo del sistema.

2.1.1. Gestión de incidencias e ITIL

Una incidencia es cualquier situación inesperada que interrumpe o afecta negativamente al funcionamiento habitual de un servicio. La gestión de incidencias tiene como finalidad restaurar ese funcionamiento normal lo antes posible, reduciendo o eliminando las consecuencias del problema en los servicios de TI [41]. En empresas donde la tecnología es crítica, una incidencia no gestionada puede provocar pérdidas de dinero, dañar la imagen de la empresa y hacer que los clientes pierdan confianza.

Para estandarizar y optimizar la gestión de servicios TI, muchas organizaciones recurren a marcos de buenas prácticas. Entre ellos, ITIL es uno de los más ampliamente adoptados por su flexibilidad y escalabilidad al adaptarse correctamente a distintos tipos de empresas. ITIL proporciona una guía estructurada para abordar la gestión de servicios de TI, incluyendo el ciclo completo de una incidencia: desde su detección, registro y categorización hasta su diagnóstico, resolución y cierre. Gracias a esto, el soporte técnico puede ser más rápido, ordenado y con mejor calidad. [41]

2.1.2. SLA y métricas de calidad del servicio

En el marco de ITIL y de la gestión de incidencias, los SLA definen de forma formal los niveles mínimos de calidad que un proveedor de servicios debe garantizar, incluyendo métricas como el tiempo máximo de respuesta o de resolución de incidencias.

Estos acuerdos no solo establecen compromisos claros entre cliente y proveedor, sino que también permiten medir el rendimiento del servicio mediante indicadores objetivos. Además, facilitan la toma de decisiones informadas para mejorar los procesos y garantizar la transparencia en la gestión. [42]

Para que un sistema de gestión sea eficaz y alineado con estos acuerdos, debe contar con funcionalidades como:

- La recopilación automática de datos sobre el rendimiento del servicio.
- La comparación en tiempo real entre los objetivos definidos en los SLA y los resultados reales.
- Sistemas de aviso que detecten posibles fallos antes de que ocurran y, si no se cumple lo pactado en el SLA, que calculen el coste económico que eso supone.

2.1.3. Automatización e IA en la gestión de incidencias

El avance de la automatización y la IA ha transformado la gestión de incidencias, evolucionando de un modelo reactivo a uno proactivo e inteligente. Gracias al uso de técnicas como el aprendizaje automático, NLP y el análisis predictivo, es posible optimizar tareas fundamentales del proceso de soporte técnico. [32]

Entre sus principales aplicaciones destacan:

- La clasificación automática de tickets.
- La asignación dinámica de técnicos en función de su disponibilidad o especialización.
- La predicción de soluciones basadas en incidencias anteriores.

Este tipo de capacidades no solo reduce los tiempos de resolución, sino que mejora la eficiencia operativa y personaliza la atención al cliente.

2.2. Evolución de los sistemas de gestión de incidencias

La gestión de incidencias ha pasado de ser un proceso manual a uno más automatizado y centrado en el usuario, gracias al avance de las TI permitiendo que las organizaciones respondan mejor ante problemas, mejorando su funcionamiento y usando IA para hacerlo de forma más rápida y eficiente.

En sus orígenes, los equipos de soporte técnico solo actuaban cuando alguien llamaba por teléfono para reportar un problema. Todo se registraba a mano, y los casos se resolvían uno a uno, en el orden en que llegaban, centrándose solo en solucionar incidencias puntuales, sin seguir un método o proceso definido. [48]

Desde los años 90, la gestión de incidencias empezó a profesionalizarse con el surgimiento de marcos de buenas prácticas como ITIL. Este enfoque ayudó a organizar mejor los servicios de TI centrándose en una restauración rápida del servicio para disminuir las consecuencias de las interrupciones en las empresas. [52]

Con la implementación de marcos como ITIL, los Help Desk, puntos de contacto donde los usuarios reportan problemas técnicos o solicitan asistencia, aportaron documentación, categorización y medición del rendimiento. Herramientas como Remedy, BMC y ServiceNow facilitaron la automatización de flujos de trabajo y el seguimiento con métricas SLA. [50]

En la década de 2010, con la extensión de metodologías ágiles y el enfoque DevOps, la gestión de incidencias evolucionó significativamente. Los equipos de desarrollo y operaciones comenzaron a colaborar de forma continua, dando lugar a herramientas como Jira Service Management o Zendesk, que permiten gestionar tickets en entornos colaborativos, con seguimiento en tiempo real y conexión con sistemas de control de versiones y monitorización. [49]

Además, se introdujeron tecnologías emergentes como la IA, el aprendizaje automático y la analítica predictiva, que permiten no solo detectar y clasificar incidencias automáticamente, sino también prever fallos antes de que ocurran. [37]

Al mismo tiempo, la experiencia del usuario pasó a primer plano. Las plataformas modernas se conectan con canales (correo, chat, apps móviles)y ofrecen interfaces intuitivas. Además, permiten adaptar la atención según el cliente o el dispositivo desde el que se envía la incidencia. [48]

Otro avance fundamental en esta evolución ha sido la extensión de uso de las herramientas de gestión de incidencias mediante software libre y de código abierto. Soluciones como OTRS y Odoo Helpdesk han permitido a empresas medianas y pequeñas la implementación de soluciones sin depender de licencias costosas, ofreciendo adaptabilidad, arquitectura modular e integración con otros sistemas [50].

En la actualidad, la gestión de incidencias ya no funciona de forma aislada, sino integrada con otras soluciones digitales, conectándose con plataformas ERP, Customer Relationship Management (CRM), sistemas de monitorización y analítica de negocio. El objetivo ya no consiste solo en solucionar los problemas cuando se presentan, sino de asegurar que todo siga funcionando sin interrupciones y que el usuario tenga una buena experiencia. Por eso, cada vez se usan más plataformas inteligentes que no solo responden automáticamente, sino que también analizan datos en tiempo real y se adaptan rápidamente a los cambios en el entorno.

Así, la evolución de los sistemas de gestión de incidencias ha pasado de lo manual a lo automatizado, y de lo técnico a lo estratégico. Hoy en día, una gestión eficaz de incidencias no solo minimiza el impacto de los errores, sino que constituye una ventaja para las empresas donde es fundamental

que el servicio esté siempre disponible.

2.3. Comparación de herramientas actuales

Actualmente, existen numerosas herramientas para gestionar incidencias adaptadas a las necesidades específicas de cada empresa, desde plataformas centradas exclusivamente en el soporte técnico (Help Desk), hasta soluciones integrales de gestión de servicios de TI, IT Service Management (ITSM).

Para este caso, se han considerado múltiples criterios a la hora de escoger las herramientas como son la integración con IA, el seguimiento de SLA, la capacidad de personalización, la disponibilidad como software de código abierto y el apoyo de la comunidad.

Para ello, se ha recurrido a rankings actualizados y de acceso público de fuentes reconocidas en el ámbito tecnológico, como PCMag. Según el artículo "*Best Help Desk Software for 2024*" publicado por PCMag (2024) [38], las soluciones más destacadas en el ámbito de Helpdesk y ITSM son:

- Jira Service Management (Atlassian)[35]: valorada por su sólida integración con herramientas de desarrollo y operaciones, y por ofrecer flujos de trabajo personalizables.
- Freshservice [39]: reconocida por su interfaz intuitiva y la facilidad de implementación, especialmente en PYMEs.
- Zendesk[53]: ampliamente utilizada en entornos corporativos por su enfoque en la experiencia del cliente y sus capacidades multicanal.
- Zoho Desk[54]: opción escalable para empresas que buscan integración con un ecosistema ERP más amplio.

Por otro lado, también es importante destacar herramientas como OTRS[45] y Odoo[44] por ser de código abierto, lo que significa que su desarrollo es público y cualquier persona puede usarlas, modificarlas o mejorarlas sin coste de licencia. OTRS[45] es una opción sólida en entornos donde la transparencia del código es prioritaria, mientras que Odoo[44] destaca por su arquitectura modular y capacidad de adaptación a distintas áreas funcionales del negocio, incluyendo Customer Relationship Management (CRM), ERP y Help Desk.

A continuación, en la siguiente tabla 1, se resumen las principales características de estas herramientas, basadas en la información proporcionada por la documentación oficial de las aplicaciones.

Herramienta	IA integrada	Gestión de SLA	Personalización	Código abierto	Enfoque Principal
Jira Service Management[35]	Sí	Sí	Alta	No	DevOps y soporte técnico
FreshService[39]	Sí	Sí	Media	No	PYMEs y automatización
Zendesk[53]	Sí	Sí	Media	No	Atención al cliente
OTRS[45]	No	Sí	Alta	Sí	Soporte técnico
Odoo Helpdesk[44]	Opcional	Sí	Muy Alta	Sí	ERP modular + Helpdesk

Tabla 1: Comparativa de herramientas de gestión de incidencias

En vista al análisis realizado en la tabla, aunque herramientas como Jira Service Management y Zendesk parecen las mejores opciones por su robustez y sus capacidades avanzadas de automatización, al ser soluciones propietarias, no permiten personalizarse fácilmente ni adaptarse a proyectos

que requieren flexibilidad e integración con otros sistemas.

Por otro lado, herramientas como Freshservice y Zendesk permiten una implementación rápida, especialmente en pequeñas y medianas empresas. Sin embargo, también incluyen limitaciones en cuanto a la personalización y adaptación a las necesidades específicas de este proyecto. En este contexto, Odoo Helpdesk destaca como una alternativa más adecuada, gracias a su carácter de código abierto y a su estructura modular.

A continuación, se explorarán en mayor detalle las ventajas específicas de Odoo Helpdesk.

2.4. Odoo como solución escogida

Tras analizar las distintas herramientas disponibles en el mercado, y teniendo en cuenta los requisitos técnicos y funcionales del proyecto, se concluye que Odoo Helpdesk es la solución que mejor se adapta a las necesidades del proyecto, pudiendo conectarla con otros sistemas y ampliarla si la organización crece. A continuación, se detallan los puntos clave que justifican esta decisión:

- Arquitectura Modular y Flexibilidad: Odoo destaca por su arquitectura modular, que permite implementar solo los componentes necesarios, optimizando recursos y facilitando la integración con sistemas como ERP y CRM. Su adaptabilidad garantiza que el sistema evolucione según las necesidades del negocio. [27]
- Código Open Source: El carácter open source de Odoo ofrece control total sobre la plataforma, facilitando su personalización según los requerimientos del proyecto. [28]
- Capacidades de personalización: A diferencia de otras herramientas propietarias que presentan limitaciones, Odoo posee una gran capacidad de personalización como la configuración de reglas y la integración de algoritmos avanzados para la gestión automatizada de incidencias. [27]
- Compatibilidad con los Objetivos del Proyecto: Odoo cumple con los requisitos técnicos y funcionales del proyecto, puesto que permite mejorar continuamente la gestión de incidencias mediante el uso de automatización y análisis de datos para poder adaptarse a nuevas necesidades.
- Coste y Accesibilidad: En comparación con herramientas como Jira Service Management o Zendesk, Odoo es una opción más económica y accesible al contar con una versión gratuita, sin perder calidad ni funciones importantes para que el proyecto funcione bien. [22]

Además, una de las mayores ventajas de Odoo Helpdesk frente a otras soluciones es su capacidad para adaptarse fácilmente cuando cambian las necesidades de la empresa. Esto es posible gracias a sus actualizaciones constantes, que incorporan mejoras continuas, y a su integración sencilla con otros módulos de Odoo.

En resumen, la decisión de elegir Odoo Helpdesk se basa en su capacidad para ajustarse a diferentes entornos, gracias a su diseño flexible y personalizable, convirtiéndolo en la opción más adecuada para este proyecto.

3. Metodología y Herramientas usadas

En esta sección se describe la metodología seguida durante el desarrollo del sistema, basada en un enfoque incremental. También, se detallan las buenas prácticas del marco ITIL aplicadas al diseño de la gestión de incidencias, así como las principales herramientas utilizadas a lo largo del trabajo.

3.1. Metodología de desarrollo utilizada

El sistema desarrollado ha seguido una metodología incremental, que consiste en dividir el proyecto en múltiples partes funcionales que se implementan y verifican de forma secuencial [9]. Esta forma de trabajo se eligió por los siguientes motivos:

- Permite contar con una versión funcional del sistema desde las primeras fases, incorporando mejoras progresivamente.
- Facilita la detección de errores al probar cada parte por separado antes de integrarla.
- Ofrece flexibilidad para adaptarse a cambios o mejoras durante el desarrollo.
- Es adecuada cuando los requisitos pueden modificarse ligeramente a lo largo del proyecto.

En este proyecto, el sistema se desarrolló por fases, empezando por la recepción de correos, luego se añadió el procesamiento del texto, la conexión con Odoo, y por último la clasificación y asignación automática de técnicos. Cada una de estas partes se probó por separado de manera independiente antes de integrarse con el resto, respetando así el enfoque incremental.

Esta metodología ha permitido construir un sistema funcional, flexible y adaptado a las necesidades reales de la empresa.

3.2. Aplicación de buenas prácticas ITIL

Durante el desarrollo del proyecto se han seguido los principios básicos de ITIL, un conjunto de buenas prácticas muy utilizado para mejorar la gestión de servicios informáticos.

En especial, se ha aplicado el proceso de gestión de incidencias, cuyo objetivo es recuperar el servicio lo antes posible cuando ocurre un problema [41]. El sistema creado sigue esta lógica, ya que permite:

- Detectar automáticamente los correos con incidencias a través de IMAP.
- Extraer de forma clara la información importante del mensaje (como asunto, remitente, contenido).
- Clasificar la incidencia, asignarle una prioridad y enviarla al técnico más adecuado.
- Registrar el ticket en Odoo para poder hacer seguimiento.

Además, se han incluido aspectos clave como la automatización de tareas repetitivas, el seguimiento de cada solicitud, la asignación por palabras clave y el cumplimiento de los SLA.

En resumen, aplicar ITIL ha ayudado a mejorar la calidad del servicio, reducir los tiempos de respuesta y hacer más eficiente el trabajo del equipo de soporte.

3.3. Herramientas utilizadas y entorno de desarrollo

A lo largo del desarrollo del proyecto se han utilizado diversas herramientas. Esta sección se organiza en dos partes: la primera está dedicada a Python y las principales librerías empleadas, que han permitido implementar diversas funcionalidades del sistema y la segunda trata sobre el resto de aplicaciones utilizadas durante el proyecto.

3.3.1. Lenguaje y librerías de Python

El sistema se basa en una serie de scripts implementados en lenguaje Python (versión 3.11.9), siendo la herramienta principal utilizada en este proyecto, por su flexibilidad y por la gran cantidad de librerías que incluye para poder trabajar con correos y aplicación de técnicas de NLP.

Para trabajar con estas librerías y comprender como funcionan, se utilizó la página oficial de paquetes de la comunidad de Python. [13]

En la siguiente tabla 2, se detallan algunas de las librerías más importantes que se han usado:

Librería	**Uso**
win32com (en Windows)	Esta librería es parte de `pywin32`, un paquete que permite a `Python` interactuar con componentes Component Object Model (COM) de Windows. Se utiliza para acceder a Microsoft Outlook desde Python.
nltk (Natural Language Toolkit)	Se utilizó para aplicar NLP, usando técnicas como la tokenización, eliminación de stopwords y lematización o stemming, para mejorar la clasificación automática del contenido de los correos.
spacy	Aunque su uso ha sido limitado, se consideró para tareas de NLP
langdetect y deep-translator	Detecta el idioma de los textos de forma automática y los puede traducir mediante Google Translate garantizando que todo el texto relevante entre los idiomas especificados se procese en español.
BeautifulSoup (bs4)	Utilizado para extraer texto limpio desde correos en formato HyperText Markup Language (HTML), eliminando etiquetas y estilos innecesarios.
dateparser	Esta librería facilita la conversión de fechas escritas en múltiples formatos e idiomas a objetos estándar en Python para normalizar los metadatos de los correos.
email y email.message	Librerías estándar de Python empleadas para analizar, extraer y manipular los contenidos Multipurpose Internet Mail Extensions (MIME) de los correos electrónicos.
imapclient	Librería que permite la conexión a servidores IMAP, facilitando la recepción y el procesamiento de correos entrantes.
requests	Librería para hacer peticiones Hypertext Transfer Protocol (HTTP) de tipo *GET, POST, etc.* Usada para poder conectarse a la API externa que predice el técnico.
logging	Librería empleada a lo largo del sistema para registrar el comportamiento del programa y facilitar la depuración y los errores.

Tabla 2: Librerías de Python empleadas en el sistema

3.3.2. Otras Herramientas

En este apartado se describen las aplicaciones y plataformas externas empleadas a lo largo del proyecto que han complementado el desarrollo técnico, facilitando tareas como la gestión de datos, la comunicación o la integración con otros sistemas.

En la siguiente tabla 3, se describe cada una de ellas:

Herramienta	Uso
Odoo	Plataforma de gestión empresarial de código abierto utilizada para la gestión de incidencias. [44]
Visual Studio Code	Editor de código gratuito que sirve para escribir y editar programas en muchos lenguajes, como Python. Se utilizó esta App para construir los distintos módulos que componen el sistema. [20]
Docker	Aunque no se ha utilizado como entorno principal de ejecución, se ha preparado un contenedor con las dependencias necesarias para facilitar el acceso a Odoo. [30]
OpenVPN GUI	Herramienta usada para conectarse de forma segura a una Virtual Private Network (VPN) y acceder de manera segura al entorno corporativo, en este caso, la API externa de la empresa. [10]
Windows Subsystem for Linux (WSL)	Se ha utilizado para disponer de un entorno Linux dentro de un sistema Windows, facilitando la instalación de dependencias como `imapclient` o `nltk` entre otros y poder ejecutar el sistema. [43]
StarUML y draw.io	Aplicaciones utilizadas para la creación de diagramas de clases y de flujo del funcionamiento de cada parte del sistema. [16]
Overleaf	Plataforma colaborativa para la redacción en LaTeX del presente documento. [11]
Git y OneDrive	Herramientas usadas para controlar el historial de versiones. [6][1]

Tabla 3: Herramientas del sistema

3.4. Justificación del uso de IA

Se ha incorporado IA al sistema para automatizar tareas complejas que antes requerían intervención humana, como entender el contenido de los correos y clasificarlos por tipo de incidencia.

Para ello, se ha usado NLP, aplicando técnicas como separar el texto en palabras (tokenización), eliminar palabras sin valor (stopwords), y simplificar términos con lematización o stemming. Esto ha permitido ordenar y estructurar los mensajes, facilitando su análisis.

Además, se ha creado un sistema que detecta palabras clave y patrones para clasificar automáticamente los tickets y asignarlos al técnico más adecuado, reduciendo el tiempo de respuesta y evitando errores al hacerlo manualmente.

En resumen, aplicar IA ha hecho el sistema más eficiente, flexible y preparado para usarse en empresas reales.

4. Requisitos del sistema

A continuación, se detallan los requisitos del sistema que son aquellos que establecen las expectativas y especificaciones para orientar a los desarrolladores en la creación de un producto que realmente cumpla con lo que necesita el usuario y con lo que busca lograr la empresa. Estos requisitos suelen dividirse en dos tipos: funcionales y no funcionales [40].

4.0.1. Requisitos funcionales

Los requisitos funcionales indican qué debe hacer el sistema para cumplir su objetivo. Es decir, describen las tareas que tiene que realizar, cómo se va a relacionar con los usuarios y cómo debe responder ante distintas acciones o situaciones. Son clave para asegurarse de que el software cumple lo que esperan tanto los usuarios como el negocio [40].

Para este proyecto, se han establecido los siguientes requisitos funcionales en la siguiente tabla 4:

Código	**Requisito**	**Descripción**
RF1	Integración con Odoo	El sistema debe comunicarse con Odoo mediante su API para crear, actualizar y gestionar tickets.
RF2	Detección de correos entrantes	El sistema debe detectar automáticamente todos los correos nuevos recibidos en una cuenta específica, garantizando eficiencia y mínimo retardo.
RF3	Creación de base de datos histórica	El sistema debe permitir importar correos antiguos como tickets válidos para su posterior análisis o entrenamiento de modelos.
RF4	Procesamiento de correos	El sistema debe extraer los campos principales de los correos (remitente, asunto, fecha, cuerpo) y aplicar limpieza textual para facilitar su análisis.
RF5	Gestión de duplicados	El sistema debe identificar y evitar la creación de tickets repetidos para un mismo caso, previniendo redundancia en la gestión.
RF6	Clasificación automática	El sistema debe asignar automáticamente prioridad, categoría y fecha límite a cada ticket para cumplir los criterios básicos de ITIL y SLA.
RF7	Asignación de técnicos	El sistema debe usar IA para predecir y asignar el técnico más adecuado.
RF8	Automatización del flujo completo	El sistema debe gestionar automáticamente todo el proceso desde la recepción del correo hasta la asignación de técnico al ticket, incluyendo avisos y notificaciones de como se va ejecutando el proceso.

Tabla 4: Requisitos funcionales del sistema

4.0.2. Requisitos no funcionales

Los requisitos no funcionales describen cómo debe funcionar el sistema, más allá de lo que hace. Se centran en aspectos como la velocidad, la facilidad de uso, la estabilidad o si puede crecer sin problemas. Ayudan a asegurar que el sistema funcione bien, sea cómodo para el usuario y cumpla con ciertos niveles de calidad [40]. Para este proyecto, se han especificado los siguientes requisitos no funcionales:

Código	**Requisito**	**Descripción**
RNF1	Rendimiento	El sistema debe procesar los correos en tiempo real y ser capaz de gestionar múltiples correos simultáneamente.
RNF2	Escalabilidad	El sistema debe seguir una estructura modular para comprender mejor el funcionamiento del sistema y ser capaz de reducir la carga de trabajo.
RNF3	Usabilidad	La interfaz debe ser fácil de usar para el equipo de soporte, aprovechando la interfaz de Odoo.
RNF4	Fiabilidad	El sistema debe ser tolerante a fallos y garantizar que los correos no se pierdan ni se procesen dos veces.
RNF5	Seguridad	La comunicación entre el sistema, Odoo y el servidor de correo debe realizarse mediante protocolos cifrados (por ejemplo, SSL/TLS o JSON-RPC seguro).

Tabla 5: Requisitos no funcionales del sistema

5. Diseño e Implementación

A continuación, en esta sección, se describe cómo se diseñó e implementó la solución para gestionar incidencias de forma automática, usando Odoo y técnicas de IA. El sistema se dividió en las siguientes fases: recepción de correos entrantes, creación de una base de datos con correos ya resueltos y automatización de tareas como la asignación de tickets.

Para la recepción de correos electrónicos y su conversión en tickets, aparte de configurar Odoo, se desarrollaron diversas clases en Python siguiendo un enfoque orientado a objetos, que interactúan directamente con la base de datos de Odoo.

Cada uno de los componentes principales, tanto de la infraestructura como del software, ha sido diseñado para garantizar una correcta integración, crecimiento y mantenibilidad del sistema. En las secciones siguientes se detallan cada uno de estos procesos, las tecnologías utilizadas, y las decisiones de diseño adoptadas.

5.1. Arquitectura del sistema

En este apartado se presenta una visión general de la arquitectura del sistema desarrollado siguiendo un enfoque modular. Esta se divide en tres bloques principales, cada uno encargado de una fase fundamental del proceso:

1. **Recepción y procesamiento de correos nuevos**: Se encarga de recibir en tiempo real los correos entrantes a través de un servidor IMAP, limpiarlos, extraer su contenido, procesarlos y generar automáticamente un ticket dentro de Odoo.
2. **Construcción de una base de datos histórica**: Recoge y procesa incidencias ya resueltas con su técnico asociado, para entrenar modelos que permitan realizar predicciones futuras.
3. **Asignación de técnico y creación del ticket**: Se predice el técnico más adecuado en base al contenido de los correos.

Este diseño modular permite separar las distintas funcionalidades, facilitar el mantenimiento del sistema y escalar cada componente de forma independiente si fuese necesario.

Aunque todas las partes forman parte del sistema global, la parte principal que está siempre en funcionamiento es la automatización del ciclo: desde que llega un correo hasta que se crea un ticket con el técnico asignado. Esta parte es la que se describe en detalle a continuación.

5.1.1. Arquitectura del funcionamiento interno

Una vez descrita la arquitectura general, en este apartado se detalla cómo se organiza el sistema internamente a nivel de software. El objetivo es mostrar cómo se ejecuta el sistema y cómo interactúan sus distintos componentes que permiten procesar correos entrantes y convertirlos en tickets con técnico asignado sin entrar en detalle de cómo se creó la base de datos ni cómo se extrajeron los correos entrantes ya que estos pasos se desarrollarán en secciones posteriores para facilitar la comprensión de cada módulo por separado.

La clase principal del sistema es `AppMain` que se encarga de conectar con Odoo (mediante Odoo-Conector), gestionar los correos ya procesados y crear los tickets con su técnico ya asignado.

En la siguiente tabla 9, se muestran los componentes del sistema:

Clase	Rol principal
AppMain	Configura la conexión a Odoo y al servidor IMAP para coger nuevos correos, inicializa componentes, arranca sistema y lo deja en ejecución hasta que se detenga manualmente.
EmailProcessor	Proporciona funciones de limpieza, normalización y traducción del texto.
TicketClassifier	Asigna prioridad, categoría y plazo de resolución al ticket.
TicketCreator	Construye el JSON del ticket y lo envía a Odoo usando OdooConector.
OdooConector	Cliente genérico de la API JSON-RPC de Odoo.
ProcessedEmailTracker	Registra hashes de mensajes ya procesados para evitar duplicados.
TaskWorker	Gestiona una cola de tareas (correos a procesar) junto con varios hilos de trabajo.
IMAPHandler	Gestiona la conexión a un servidor IMAP.
TecnicoPredictor	Predecir automáticamente qué técnico debe encargarse de un ticket a partir del texto de un correo o incidencia.

Tabla 6: Componentes del sistema interno

También, se incluyen dos representaciones complementarias: un diagrama de clases que muestra la estructura modular del código en Python y un diagrama de flujo que describe los pasos seguidos desde la recepción del correo hasta la creación del ticket en Odoo:

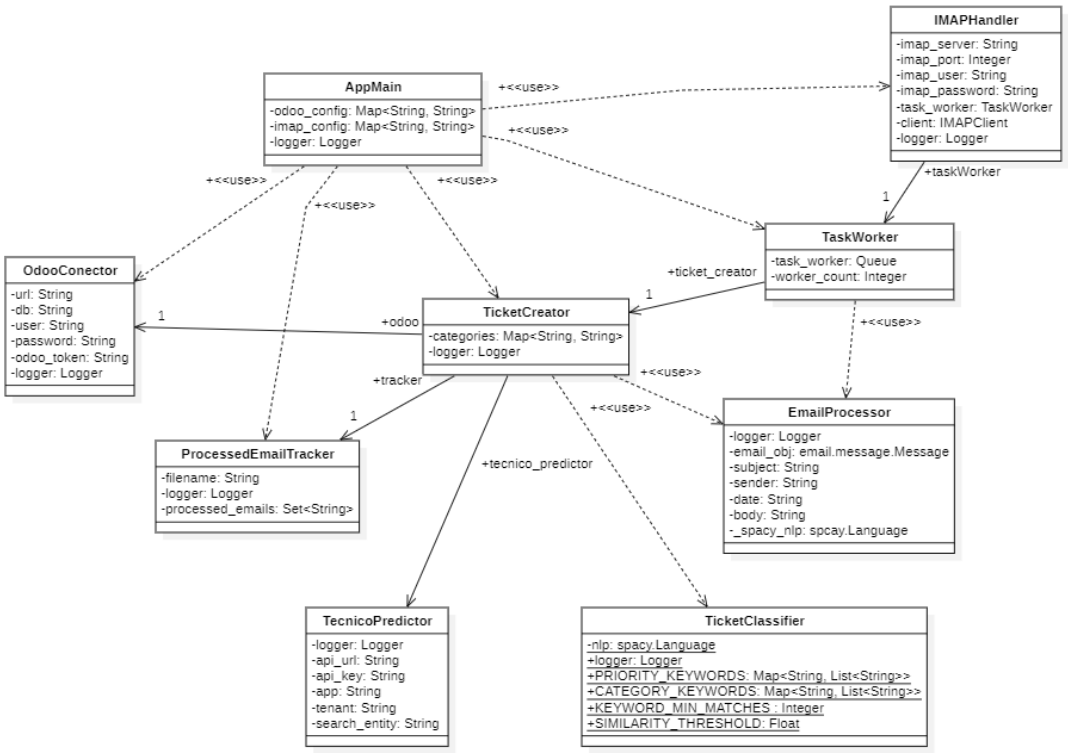

Figura 1: Diagrama de clases del sistema general

Las líneas discontinuas con el estereotipo «use» en el diagrama indican relaciones de uso o dependencia entre clases. Específicamente, significan que una clase utiliza o depende de otra para realizar alguna funcionalidad, pero no necesariamente la contiene como atributo.

El flujo de procesamiento de esta sección puede consultarse en el Apéndice, en la Figura 7.

5.2. Despliegue del entorno Odoo y módulo Helpdesk mediante Docker

Antes de desarrollar cualquier componente, lo primero es poder acceder a Odoo y utilizar su módulo Helpdesk que es el que gestiona los tickets. Para conseguirlo se ha utilizado Docker como herramienta principal puesto que permite crear entornos aislados, garantizando que tanto Odoo como su base de datos funcionen correctamente en cualquier equipo sin conflictos de dependencias ni configuraciones específicas.

Se ha decidido instalar Docker dentro del subsistema WSL, ya que proporciona un entorno Linux completo en máquinas Windows, facilitando la gestión de contenedores.

Para organizar los servicios necesarios (Odoo y PostgreSQL) y gestionar su arranque conjunto, se ha empleado `docker-compose`. El proyecto se estructura en una carpeta llamada `tickets2`, compuesta de la siguiente manera:

- Docker-compose.yml: Archivo de configuración que define los servicios (Odoo y PostgreSQL), los volúmenes de datos y las rutas de los módulos personalizados.
- helpdesk: Carpeta donde se encuentra el código fuente del módulo Helpdesk.
- config: Carpeta con los archivos de configuración de Odoo (Odoo.conf) ya que necesita de un fichero de configuración para arrancar.

El archivo Docker-compose.yml sirve para poner en marcha el entorno de desarrollo de Odoo, incluyendo la aplicación y su base de datos PostgreSQL:

```
services:
  odoo_tickets:
    image: Odoo:16
    depends_on:
      - db_tickets
    ports:
      - "8069:8069"
    volumes:
      - Odoo-tickets:/var/lib/Odoo
      - ./config:/etc/Odoo
      - ./helpdesk:/mnt/extra-addons
    environment:
      - HOST=db_tickets
      - USER=Odoo
      - PASWWORD=Odoo

  db_tickets:
    image: postgres:15
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_PASSWORD=Odoo
      - POSTGRES_USER=Odoo
    volumes:
      - Odoo-db-tickets:/var/lib/postgresql/data/

volumes:
  Odoo-tickets:
  Odoo-db-tickets:
```

Listing 1: Archivo docker-compose.yml

Descripción de los servicios:

- **odoo_tickets**: Servicio que ejecuta Odoo 16, muestra el puerto 8069 y monta los volúmenes necesarios para los datos, la configuración y los módulos extra (como Helpdesk).
- **db_tickets**: Servicio de base de datos PostgreSQL 15, configurado para ser usado por Odoo.

Gestión de volúmenes:

Se crean dos volúmenes para asegurar la continuidad de datos y facilitar el desarrollo:

- **Volúmenes de Docker**:
  - `Odoo-tickets`: Almacena los datos internos de Odoo (configuraciones, sesiones, archivos cargados, etc.), de forma que si el contenedor se elimina, la información no se pierde.
  - `Odoo-db-tickets`: Contiene los datos de la base de datos PostgreSQL para mantener la información de los tickets y usuarios, aunque el contenedor de la base de datos se reinicie o destruya.
- **Volúmenes montados desde carpetas locales**:
  - `./config:/etc/Odoo`: Rastrea la carpeta local `config`, donde se encuentra el archivo `Odoo.conf`, al directorio de configuración de Odoo dentro del contenedor permitiendo personalizar el comportamiento de Odoo sin modificar la imagen original.
  - `./helpdesk:/mnt/extra-addons`: Se enlaza la carpeta local "*helpdesk*", que contiene el módulo Helpdesk, a la ubicación que Odoo utiliza para cargar módulos externos facilitando el desarrollo y pruebas del módulo directamente desde el sistema de archivos del host.

Los pasos para montar y acceder al entorno de Odoo se explican en el Apéndice en el Manual de Usuario.

En esta parte, me he encargado de integrar, configurar Docker (no de desarrollar todo el sistema), y de documentar cómo se despliega y accede al sistema. Gracias a esto, cualquier persona del equipo puede poner en marcha Odoo y usar el módulo Helpdesk fácilmente.

5.3. Módulos reutilizables para procesado de correos

Esta sección describe los módulos reutilizables en el proceso de análisis, clasificación, conexión a Odoo y creación de tickets a partir de correos electrónicos.

5.3.1. Procesamiento y limpieza de correos

La clase `EmailProcessor` se utiliza tanto en el procesamiento de correos históricos como en el manejo de incidencias en tiempo real. Su objetivo es transformar un objeto `email.message.EmailMessage` en una representación estructurada y limpia del correo, extrayendo los campos clave y normalizando el contenido del texto.

Este componente forma parte del proceso de creación de tickets dentro del módulo `TicketCreator`, donde se invoca para obtener los metadatos básicos (remitente, asunto, fecha, cuerpo del mensaje) y preparar el contenido para su análisis posterior.

El sistema permite aplicar y configurar técnicas de NLP. Entre ellas, se encuentran la tokenización, la eliminación de stopwords, la lematización y el stemming. Estas técnicas pueden ejecutarse individualmente, en conjunto, o hasta omitirse completamente, dependiendo de los resultados obtenidos durante la fase de entrenamiento del modelo de clasificación.

La decisión final sobre qué técnicas aplicar depende de los resultados que se obtengan en las pruebas de evaluación que se realicen, en base a maximizar la precisión del modelo y el rendimiento del sistema.

A continuación, se detallan los pasos principales que realiza el módulo de limpieza en el siguiente orden:

Decodificación del asunto del correo:

Se define una función `decode_subject(str)` para decodificar el asunto (campo Subject) de un correo electrónico que puede encontrarse en distintas decodificaciones como por ejemplo 8-bit Unicode Transformation Format (UTF-8) o ISO-8859 para mostrar correctamente los asuntos de correos electrónicos, aunque estén en diferentes idiomas o codificaciones.

Para ello, se hace uso de la función `decode_header` de la librería de Python `email` para dividir el asunto en partes y así poder detectar su codificación. En cada parte, si se detecta una cadena de bytes, la decodifica usando la codificación detectada (o UTF-8 por defecto). Si es texto, lo añade directamente. Sin embargo, si se detecta un error de decodificación,lo registra en el log y añade la parte como texto plano.

Extraer dirección de correo del remitente

Se hace uso de la función `parseaddr` de la librería `email.utils` para separar el nombre y dirección del correo electrónico del campo *From* que constituye al remitente del correo, creando una tupla (`nombre, direccion_email`) para devolver solo el segundo elemento de ella.

Limpiar fecha

Se crea la función `clean_date` para extraer y normalizar la fecha del correo, ya que puede venir en varios formatos. Primero, se comprueba que sea un `String` y se eliminan encabezados comunes como *Enviado en* , *Sent* en los distintos idiomas en los que pueden estar los correos. Después, se eliminan los días de la semana si aparece en el formato y se intenta procesar el `String` con la fecha en un objeto `datetime`. Si se consigue, se devuelve la fecha como datetime en formato: *YYYY-MM-DD HH:MM:SS* y lo registra en el log.

Extracción y limpieza del cuerpo del correo

El método `extract_body` analiza si el correo tiene varias partes (multipart) o no. Si es multipart, recorre cada parte del mensaje y decodifica el contenido según su tipo: si es texto plano (`text/plain`) lo guarda directamente en una lista, y si es HTML (`text/html`) elimina las etiquetas para quedarse solo con el texto mediante la librería `BeautifulSoup`. Devuelve el texto plano si está disponible, o el texto extraído del HTML en su defecto. Si el correo no es multipart, decodifica el contenido y, según sea texto plano o HTML, lo devuelve limpio. Si ocurre algún error de decodificación, lo registra en el log y devuelve una cadena vacía.

Normalización de encabezados

Con el texto extraído, se aplica el método `normalize_headers` que normaliza los encabezados de los correos electrónicos traduciendo los nombres de los campos más comunes (como *From*, *To*, *Subject*, *etc.*) a un formato estándar en español. Para ello, se define un diccionario con patrones de encabezado en varios idiomas (inglés, italiano, etc) y su traducción al español. Se recorre cada patrón y se reemplazan todas las apariciones en el texto por el encabezado en español, por ejemplo: *From* a *De* y se devuelve el texto cambiado.

Limpieza de firma y pie de página

Se emplea la función `clean_signature_and_footer` que deja el cuerpo del correo sin firmas ni pies de página solo el mensaje principal. Para ello, se define una lista de patrones (expresiones regulares) comunes en firmas, avisos legales y pies de correo (como *Un saludo*, teléfonos, Uniform Resource Locator (URLs), etc.). Después, se divide el texto del correo en líneas y las va recorriendo hasta que encuentra uno de los patrones, entones deja de añadir más líneas y devuelve el texto anterior a la firma/pie, eliminando líneas vacías al final.

Eliminación del historial de respuestas y reenvíos

La función `remove_reply_history` elimina el historial de respuestas y reenvíos de un correo electrónico. Para ello, define varios patrones (en diferentes idiomas y formatos) que suelen marcar el inicio de mensajes anteriores en una cadena de correos (por ejemplo, *From: ... Sent: ..., etc.*). Para cada patrón, divide el texto por ese marcador y se queda solo con la parte anterior, es decir, el mensaje más reciente. Al final, devuelve el texto limpio, sin el historial de respuestas ni reenvíos.

Traducción del cuerpo

Aunque la mayoría de los correos están redactados en español, también se detectaron en otros idiomas como inglés, italiano y alemán, debido a los clientes internacionales de los que dispone la empresa.

El método `translate_into_spanish` traduce al español el cuerpo de un correo si detecta que está escrito en otro idioma. Para ello, detecta automáticamente el idioma mediante la librería `langdetect`. En caso de no estar en español, se divide el texto en bloques de 10 líneas para evitar errores comunes en traducciones largas. Después, la traducción de cada bloque se traduce utilizando la API de `GoogleTranslator` de `deep_translator`, y se añade al resultado solo aquellas líneas que se han traducido correctamente. Las líneas que ya están en español o que no se pueden traducir se mantienen sin cambios.

Inicialmente, el cuerpo del correo se traducía por completo aunque tuviera partes ya en español, ya que algunos correos mezclaban varios idiomas. Sin embargo, este enfoque ralentizaba mucho la ejecución y dificultaba la creación del histórico. Por eso, ahora solo se traduce si todo el cuerpo está en otro idioma ya que se reduce significativamente el tiempo de procesamiento sin perder utilidad.

Aplicación de técnicas NLP (Opcional)

La función `nlp` prepara el texto para NLP de la siguiente manera:

1. Convierte el texto a minúsculas.
2. Elimina la puntuación usando `str.translate`.
3. Tokeniza el texto (lo divide en palabras) usando `word_tokenize` de NLTK.
4. Elimina las stopwords (palabras que no aportan valor como *el, la, etc.*) y cualquier token que no sea alfabético.
5. Devuelve el texto limpio como una cadena de palabras separadas por espacios.

También, se ha implementado lematización mediante la función `lematization` reduciendo cada palabra a su forma base, lo que permite normalizar el texto y comparar mejor el significado entre mensajes. Esta función utiliza el modelo de `spaCy` para español `(es_core_news_sm)` y transforma palabras conjugadas, plurales o derivadas en su lema correspondiente (por ejemplo, *modificaciones* o *modificando* se reducen a *modificar*) mejorando la precisión en tareas de búsqueda y clasificación.

Por otro lado, se creó también una función para aplicar stemming `stemming` reduciendo cada palabra a su raíz (por ejemplo: *modificaciones*, *modificando* a *modific*) permitiendo agrupar diferentes formas de una palabra bajo una misma raíz, lo que puede ser útil para reducir la variabilidad del texto y mejorar el rendimiento de algunos modelos. Sin embargo, al acortar tanto los términos se puede perder la información.

Ambas técnicas tienen limitaciones, sobre todo en español, ya que algunas palabras no se transforman correctamente o no se reconocen como deberían.

Creación de un hash de correo

La función `get_hash` genera un identificador único para cada correo aunque no tenga un `Message-ID`, usando su asunto, remitente, cuerpo y fecha, calculando un hash de esa combinación.

Para crear ese hash, se utiliza la librería `hashlib` de Python que proporciona funciones para crear hashes criptográficos de datos. En concreto, se utilizó de ese módulo el algoritmo de hash md5.

Así se puede detectar y evitar procesar correos duplicados como mejorar la robustez en el sistema ya que si el correo cambia, el hash cambia con él.

5.3.2. Tratamiento de correos duplicados

Para garantizar que cada correo electrónico genere un ticket en Odoo una única vez, se implementa un control de duplicados mediante la clase `ProcessedEmailTracker`.

Esta clase gestiona un archivo de texto donde almacena los hashes únicos de los correos ya proce-

sados, evitando así que un mismo correo sea tratado más de una vez. Su funcionamiento se basa en los siguientes métodos:

- `load_processed_emails`: Al inicializar la clase, este método carga todos los hashes almacenados en el archivo a un conjunto (`Set`) en memoria. Si el archivo existe, lee cada línea y añade el hash correspondiente al conjunto. Si el archivo no existe, muestra una advertencia y el archivo se creará automáticamente cuando se procese el primer correo nuevo.
- `save_processed_email`: Cada vez que se procesa un correo nuevo, este método guarda el hash correspondiente en el archivo añadiéndolo como una nueva línea y lo incorpora también al conjunto en memoria. Si el archivo no existía previamente, se crea en este momento.
- `was_processed`: Comprueba si un hash de correo ya está en el conjunto de procesados, es decir, si ya ha sido procesado.

De este modo, el sistema asegura que los tickets sean únicos y evita la duplicidad en el procesamiento de correos electrónicos.

5.3.3. Clasificación automática de categoría, prioridad y plazos del ticket

Para poder cumplir con los criterios de ITL y SLA de forma automática, es imprescindible que cada ticket se clasifique en base a un tipo, una prioridad y unos plazos de resolución. Para ello, se creó una clase en el sistema que implementar esta funcionalidad, `TicketClassifier`.

Empezando por el tipo/categoría de los tickets, inicialmente los correos no traían una asignada, por lo que fue necesario analizar y limpiar su contenido para poder organizarlos. Este proceso se dividió en dos partes:

Primero, se subieron todos los tickets de los correos históricos de la empresa limpios sin emplear técnicas de NLP, a la base de datos de Odoo y se exportaron en formato Comma Separated Values (CSV) para poder realizar un análisis más rápido y eficaz de su contenido. Una vez exportados, se creó un archivo en Jupyter Notebook donde se usó ese CSV, y en él se aplicaron técnicas extra de NLP como lematización y extracción de sustantivos con el objetivo de obtener las palabras más representativas de los tickets ya que los sustantivos como por ejemplo, *factura, cliente, etc.* reflejaban mejor el contenido y el contexto de cada incidencia que los verbos como *coger, usar*, en este caso al ser más generales y ambiguos. Después de la limpieza, se aplicaron algoritmos de agrupamiento como *k-means* sobre los vectores de características obtenidos, para detectar clústeres temáticos que permitieran definir las categorías más frecuentes de los tickets históricos. Los resultados de los clústeres dieron lugar a las siguientes categorías: *Facturacion y Gestion de Clientes, Soporte General y Comunicacion, Procesos y Documentacion y Infraestructura y Ubicaciones*. En algunos clústeres había palabras que salían en más de uno porque se pueden asignar a más de una categoría dependiendo del contexto.

Luego, volviendo al sistema, en la clase `TicketClassifier` se crearon 2 funciones, además de un diccionario(`CATEGORY_KEYWORDS`) compuesto por las categorías que se definieron en los clústeres junto con las palabras que componían cada uno como valor:

- `extract_nouns_lemmas`: Procesa un texto con `spaCy`, extrae los lemmas (formas básicas de las palabras como por ejemplo, de libros se pasa a libro) de todos los sustantivos(`NOUN`), se crea un conjunto (`set`) de estas palabras para evitar repeticiones y se unen los lemmas en una cadena separada por espacios. Por ejemplo, si el texto es *Los servidores están caídos por un problema*, la función devolvería *servidor problema*.
- `assign_category`: Esta función intenta asignar la mejor categoría a un ticket usando dos métodos: coincidencia de palabras clave de las categorías definidas en el diccionario y similitud semántica con `spaCy` de la siguiente manera:
  1. Preprocesa el texto uniendo el asunto y el cuerpo, y extrae los lemmas de los sustantivos con `extract_nouns_lemmas`.
  2. Normaliza las categorías convirtiendo los nombres de estas a minúsculas y sin acentos para que coincidan con las mismas que se registran en Odoo mediante la función

`normalize` de esta misma clase.

3. Se aplica el primer método de clasificación basado en buscar coincidencias de palabras clave: el sistema cuenta cuántas palabras clave de cada una aparecen en el texto procesado. Si alguna categoría supera el umbral de al menos 2 palabras clave que aparezcan en el mensaje, se considera válida. Luego, se elige la categoría con más coincidencias, para asegurarse de que esté realmente relacionada con el contenido y evitar errores por palabras sueltas.
4. Si no hay coincidencias suficientes, se aplica el segundo método que consiste en comparar el texto del ticket con las descripciones de cada categoría usando vectores semánticos (embeddings). Se calcula la similitud y si supera un umbral 70 %, se elige la categoría con mayor similitud. Este umbral se ha elegido para asegurar que la categoría asignada realmente representa el contenido del mensaje, mejorando la precisión y evitando errores por coincidencias poco relevantes.
5. Si no hay coincidencias, se devuelve la primera categoría del diccionario como valor por defecto.

Se emplean ambas técnicas para la clasificación porque combinarlas mejora la precisión y robustez del sistema. Primero se intenta con palabras clave porque es más fiable y rápido cuando hay coincidencias claras y si no hay suficientes coincidencias, se recurre a la similitud semántica para cubrir casos donde el lenguaje es más variado o ambiguo aunque sea una técnica más lenta. Así, el sistema es más flexible y preciso, adaptándose tanto a textos muy directos como a los más complejos.

Para la prioridad, se definió un sistema de prioridades usando un diccionario con 4 niveles (del 0 al 3), ya que en Odoo este campo es de tipo `char` siendo el 3 el de mayor prioridad y a cada nivel se le asignó unas palabras clave dependiendo del grado de urgencia. Luego se creó la función `assign_priority` que analiza el asunto y el cuerpo del mensaje. Si encuentra alguna de esas palabras clave, asigna el nivel correspondiente y si no, asigna la prioridad más baja (0) por defecto.

Después, en base a la prioridad se creó una función `calculate_deadline` para asignarle a un ticket una fecha de cierre. Esto lo implementa obteniendo la hora actual en la zona horaria especificada, en este caso, Madrid. Se definen los plazos para cada prioridad (por ejemplo, 4 horas para nivel crítico) y se devuelve la suma del plazo correspondiente más la hora actual en formato *YYYY-MM-DD HH:MM:SS* para que sea aceptado por Odoo.

De esta forma se aplica de forma práctica los SLA al relacionar el nivel de prioridad con un plazo máximo, ya que permite asignar automáticamente el tiempo estimado para resolver cada incidencia según su urgencia. Aunque el sistema aún no comprueba si se cumplen esos plazos, sí guarda los datos necesarios para poder hacerlo en el futuro.

5.3.4. Conexión y comunicación con Odoo

La conexión y comunicación con la plataforma Odoo se realiza a través de la clase `OdooConector`, que encapsula toda la lógica necesaria para autentificar al usuario, gestionar la sesión y realizar operaciones sobre los tickets y sus categorías.

Autenticación y obtención del token de sesión

Al inicializar un objeto `OdooConector`, se lanza el método `get_odoo_token()`, que realiza una solicitud POST al endpoint de autentificación de Odoo utilizando el protocolo JSON-RPC. Si las credenciales son correctas, se obtiene un *token de sesión* (session_id) que se utilizará en las siguientes peticiones para mantener la sesión autentificada. Este token se extrae mayoritariamente de las cookies de la respuesta, o del propio objeto `result` de la respuesta JSON. Inicialmente surgieron errores relacionados con las credenciales y el formato de las peticiones JSON, así como problemas de timeout, que se resolvieron ajustando la configuración de red y los parámetros de la solicitud.

Gestión de categorías

La clase proporciona métodos para consultar y crear categorías de tickets en Odoo:

- `get_categories`: Recupera todas las categorías existentes mediante una petición al modelo `helpdesk.ticket.category`, devolviendo un diccionario con los nombres y sus identificadores.
- `create_category`: Crea una nueva categoría en Odoo si no existe, devolviendo el identificador de la categoría creada.
- `get_or_create_categories`: Garantiza que existan todas las categorías necesarias, creando aquellas que falten y devolviendo el conjunto actualizado.

De esta forma, el sistema puede interactuar de forma segura y eficiente con Odoo, gestionando la autentificación, la creación de tickets y la gestión de categorías de manera reutilizable.

5.3.5. Creación de tickets

La creación de tickets a partir de correos electrónicos se gestiona mediante la clase `TicketCreator`.

La función principal de esta clase es el método `create_from_email` que toma un correo, evita duplicados, valida el cuerpo, clasifica y prepara los datos, crea el ticket en Odoo y marca el correo como procesado, todo con manejo robusto de errores y registros en el log. Esta hace lo siguiente:

1. Se asigna un técnico solo si es un correo nuevo una vez implementada esta parte del sistema.
2. Generación de un hash único para cada correo con la función `get_hash` de `EmailProcessor`.
3. Comprobación de duplicados llamando al método `was_processed` de `ProcessedEmailTracker`. Si se detecta que el correo ya ha sido procesado, se ignora el correo y no crea el ticket.
4. Se valida el cuerpo del correo comprobando si está vacío. En ese caso se ignora el correo también y no se crea el ticket.
5. Se determina la prioridad, la categoría y la fecha límite del ticket usando las funciones clasificadoras de `TicketCreator` y se prepara un diccionario con todos los datos necesarios para crear el ticket en Odoo. También, se añade una fecha
6. Se crea el ticket pasándole los datos anteriores a la función `create_ticket` de `OdooConector`.
7. Se guarda el hash del correo como procesado para evitar futuros duplicados.

En las siguiente figuras 2 y 3, se pueden ver los campos detallados de los tickets donde se introduce cada campo en Odoo. Además, en la segunda foto, se pueden observar los plazos establecidos para el ticket para cumplir con los SLA. Por otra parte, el técnico asignado se incluye en el campo *Nombre de la Empresa* en lugar de *Usuario asignado*, ya que este último es de tipo `many2one` y requiere información adicional de la que no se dispone (como el correo del técnico), mientras que el primero permite guardar directamente el nombre como texto.

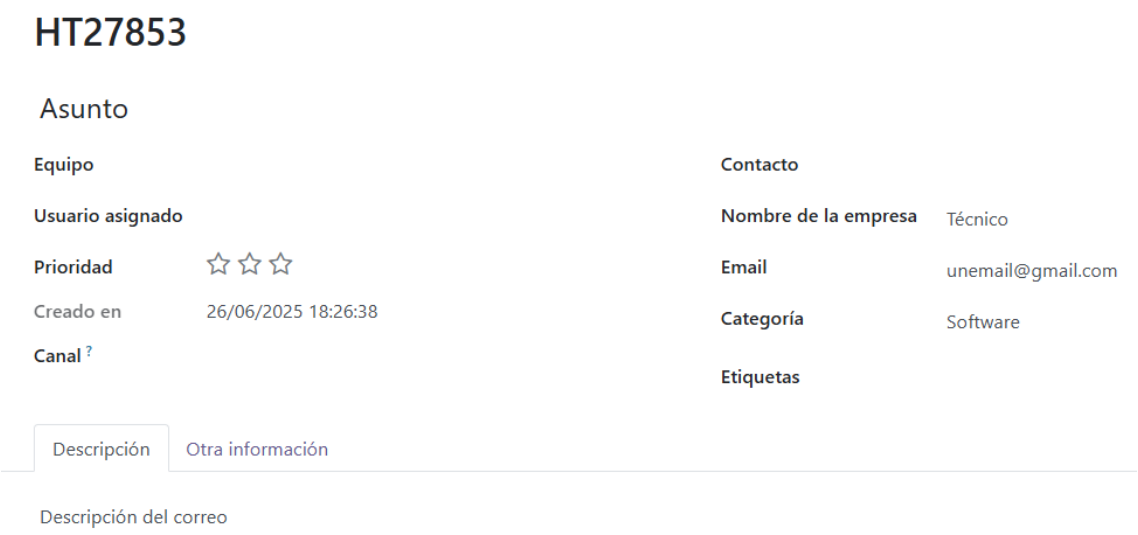

Figura 2: Estructura de un ticket (campos principales)

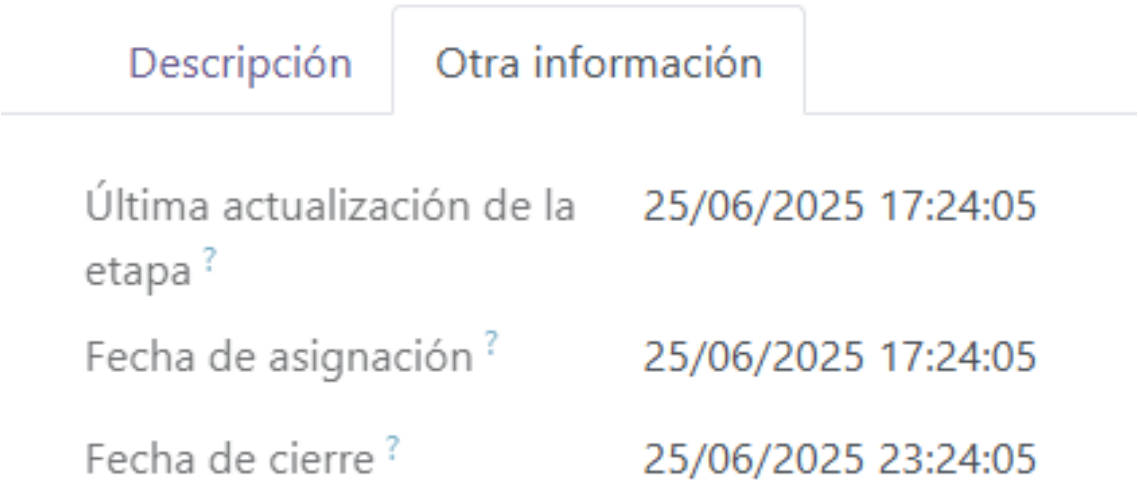

Figura 3: Estructura de un ticket (plazos)

5.4. Procesamiento de correos entrantes en tiempo real

Para automatizar la creación de tickets en la aplicación de Odoo a partir de correos electrónicos entrantes reales, la solución se basó en un sistema fiable y automático que se integró en Odoo sin depender demasiado de él ni complicar su mantenimiento.

5.4.1. Análisis comparativo de métodos de integración

Para conseguir este objetivo, se exploraron diferentes técnicas:

- **Odoo con seudónimos de correo (opción integrada en él)**: La creación de tickets se realiza automáticamente a través de direcciones de correo configuradas con seudónimos por cada equipo de soporte. Es una solución simple, sin necesidad de emplear código externo. [5]
  - **Ventajas**:
    - Integración directa en Odoo (soporte oficial).
    - No requiere desarrollo adicional.
  - **Desventajas**:
    - Requiere configuración de servidores de correo y dominios personalizados.
    - Escasa flexibilidad para añadir lógica personalizada.
    - Dificulta el preprocesamiento, clasificación y filtrado avanzado de correos.
    - Si hay un tráfico masivo, Odoo no está optimizado para manejar un gran volumen de correos y puede ralentizarse.
- **Procesamiento desde archivo Personal Storage Table (PST) (pypff + watchdog)**: Leer correos desde un archivo pst de Outlook que se va actualizando y convertirlos en tickets de Odoo de forma automática. Puede ser una opción para correos archivados sin conexión, pero introduce dependencia sobre actualizaciones manuales del archivo PST y gran complejidad en la instalación de dependencias. [24]
  - **Ventajas**:
    - Permite procesar correos sin conexión a internet.
    - Gran eficiencia tratando grandes volúmenes de correos.
    - Se puede monitorear continuamente el archivo PST en busca de cambios y procesar los correos nuevos de forma automática.
    - Se evitan duplicados mediante sistema de hash.
  - **Desventajas**:
    - Requiere instalación de librerías complejas como pypff.
    - Depende de la actualización manual del PST.
    - No permite trabajar en tiempo real.
- **Procesamiento desde MBOX (mbox + watchdog)**: Usar archivo mbox junto con la librería watchdog para monitorizar cambios en el archivo y procesar correos nuevos de forma automática enviándolos a Odoo. Puede ser útil para archivos históricos o archivados (por ejemplo, Thunderbird) y similar al PST, pero igualmente dependiente de actualizaciones externas del archivo MBOX. [31]
  - **Ventajas**:
    - Los archivos MBOX son un formato estándar de almacenamiento de correos, por lo que permite mayor compatibilidad.
    - Facilidad de uso.
    - Se puede monitorear continuamente el archivo MBOX en busca de cambios y procesar los correos nuevos de forma automática.
    - Se evitan duplicados mediante sistema de hash.
  - **Desventajas**:
    - Requiere instalación de varias librerías.
    - Depende de la actualización manual del MBOX.
    - Puede requerir de mucho espacio de almacenamiento.
- **Gateway en Python (con Flask)**: Un gateway actuaría como un intermediario para recibir

correos electrónicos y enviarlos a Odoo+. Se podría implementar un gateway simple utilizando Flask para recibir correos electrónicos y convertirlos en tickets en Odoo. Esto funciona bien entornos con múltiples fuentes de entrada de correo, pero requiere configuración avanzada de red y servidor intermedio. [25]

- **Ventajas**:
  - Centraliza el procesamiento de correos en un único punto (por si se reciben correos de diversas fuentes).
  - Hay un alto control sobre la lógica de negocio.
- **Desventajas**:
  - Depende de un servidor para gestionar los correos.
  - Se requiere configuración de red.

- **Integración mediante Dynamic-Link Library (DLL) en Python**: Una dll es una biblioteca de vínculos dinámicos y es muy reutilizable y eficiente, pero más adecuada para entornos donde se requiere rendimiento extremo o integración con software externo que consuma la DLL. [21]
  - **Ventajas**:
    - La lógica del procesamiento se encapsula en una DLL. facilitando la reutilización e integración con otros proyectos
    - Buena eficiencia y rendimiento.
    - Es flexible al poder llamar a la DLL con otros programas que soporten DLL.
  - **Desventajas**:
    - Es complejo crear y compilar la DLL.
    - Se depende de una aplicación externa para cargar y usar la DLL.
- **Integración mediante IMAP + IMAPClient**: Utilizar IMAP con la librería IMAPClient para acceder directamente a la cuenta de Outlook a través de imap y convertir los correos en Odoo. [7]
  - **Ventajas**:
    - Se puede acceder y procesar correos en tiempo real.
    - No se necesita exportar manualmente los correos ya que se accede a la cuenta de correo actualizada.
  - **Desventajas**:
    - Se requiere conexión a internet para acceder a los correos.
    - Algunos servidores IMAP pueden tener limitaciones de sincronización.

Tras analizar todas las opciones junto con sus ventajas y desventajas, se concluye que la solución basada en IMAP mediante la librería IMAPClient es la más adecuada para los objetivos del proyecto. Esta decisión se basa en los siguientes puntos:

- IMAP permite acceder a los correos entrantes inmediatamente automatizando completamente la creación de tickets sin necesidad de mantener archivos actualizados reduciendo así los errores y repeticiones, al contrario que con los métodos PST o MBOX que dependen de exportaciones y actualizaciones constantes a tiempo real.
- Es compatible con los servidores IMAP de correos comunes como Outlook o gmail y permite personalizar el sistema mientras que con los pseudónimos de Odoo no.
- IMAP ofrece una arquitectura sencilla y flexible, por su fácil integración con otros componentes del sistema (como las distintas clases de procesamiento). En cambio, alternativas como DLLs o gateways requieren implementaciones más complejas, configuración adicional

y mantenimiento más delicado.

5.4.2. Arquitectura General del Sistema de Correos Entrantes

Conociendo el método de integración a usar, se ha diseñado una arquitectura modular y multihilo basada en protocolo IMAP de forma eficiente y extensible.

La clase main donde se ejecuta todo el proceso es `AppMain` que configura y establece la conexión con Odoo, sigue a los correos procesados y activa la creación de tickets. También, gestiona la inicialización del gestor de tareas y el servidor IMAP para utilizar hilos pudiendo procesar más de un correo nuevo a la vez en tiempo real.

El sistema desarrollado se basa en una arquitectura modular compuesta por los siguientes componentes mostrados en la siguiente tabla 7:

Clase	**Rol principal**
AppMain	Configura conexión a Odoo y al servidor IMAP, inicializa componentes, arranca sistema y lo deja en ejecución hasta que se detenga manualmente.
EmailProcessor	Proporciona funciones de limpieza, normalización y traducción del texto.
TicketClassifier	Asigna prioridad, categoría y plazo de resolución al ticket.
TicketCreator	Construye el JSON del ticket y lo envía a Odoo usando OdooConector.
OdooConector	Cliente genérico de la API JSON-RPC de Odoo.
ProcessedEmailTracker	Registra hashes de mensajes ya procesados para evitar duplicados.
TaskWorker	Gestiona una cola de tareas (correos a procesar) junto con varios hilos de trabajo.
IMAPHandler	Gestiona la conexión a un servidor IMAP.

Tabla 7: Componentes del sistema de correos entrantes y sus roles principales

A continuación, se muestra el diagrama de clases:

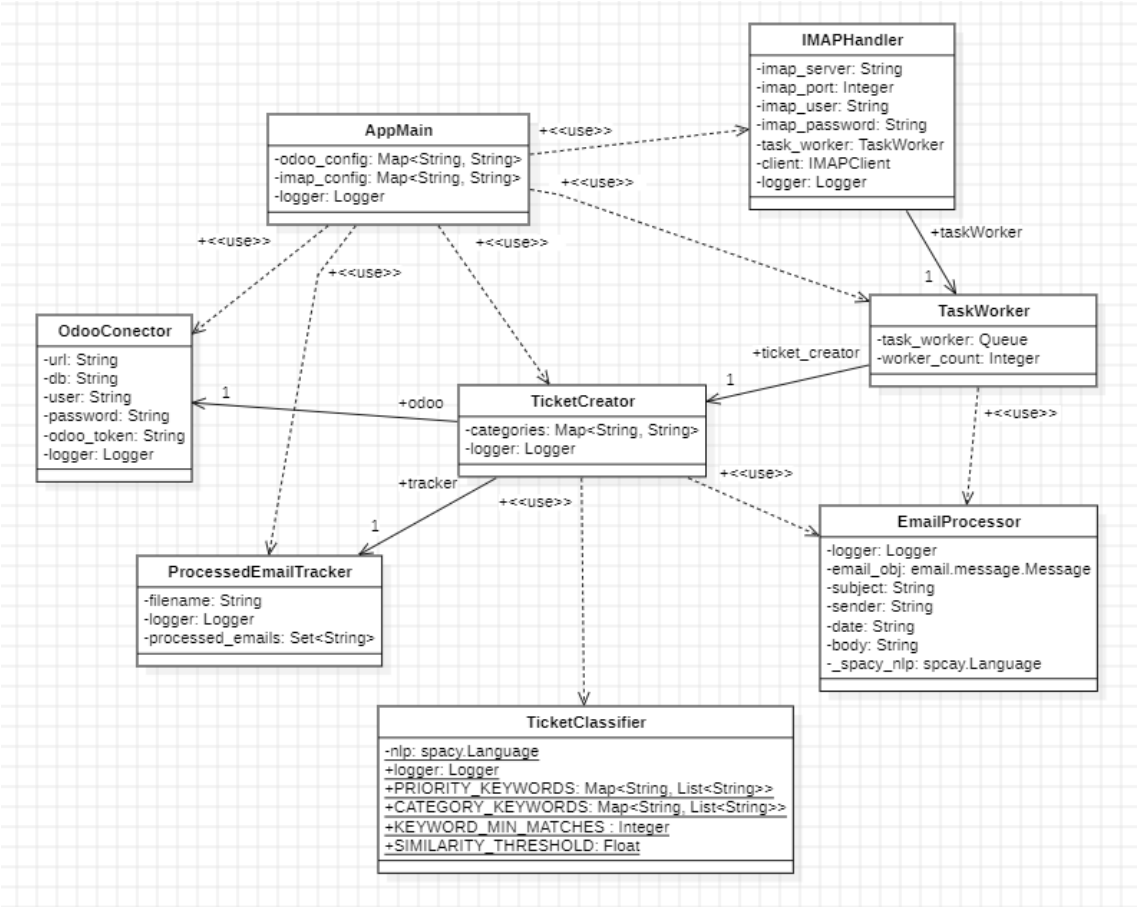

Figura 4: Diagrama de clases del sistema de recepción de correos entrantes

El flujo de procesamiento de esta sección puede consultarse en el Apéndice, en la Figura 8.

5.4.3. Paralelismo del sistema

En la versión inicial, los correos se procesaban de manera secuencial de forma que si en el minuto que la conexión IMAP espera para recibir correos, recibía más de uno, solo se procesaba el primer correo que llegó y el siguiente en el próximo minuto, haciendo al sistema más lento y provocando retrasos.

Por esta razón, se decidió incorporar paralelización en el código de manera que se redujesen los tiempos y mejorase la eficiencia. Para ello, se realizó un análisis comparativo de distintas alternativas, seleccionando aquella que permite procesar múltiples correos por minuto de forma rápida y eficaz.

- **Multithreading con el módulo `threading`**: Permite ejecutar múltiples hilos dentro de un mismo proceso. Es sencillo de implementar en Python, eficiente para operaciones I/O y funciona bien con estructuras como `Queue` para gestionar tareas. Como desventaja, requiere controlar el número de hilos para evitar sobrecargar el sistema. [17]
- **Multiprocessing**: Ejecuta tareas en procesos separados, aprovechando múltiples núcleos físicos del procesador. Aunque es útil para tareas intensivas en CPU, conlleva más consumo en memoria, dificultades para compartir datos entre procesos (por ejemplo, conexiones activas a Odoo) y es más lento en operaciones I/O, debido al coste de iniciar los procesos. [12]
- Batch Processing (Procesamiento por lotes): En lugar de procesar un correo a la vez, se agrupan varios correos en lotes y se envían a Odoo en una sola solicitud masiva. Como

ventajas, se reduce el número de llamadas a Odoo (una por lote) pero añade latencia mientras espera a que se complete el bloque y es complejo para manejar errores por mensaje dentro del lote. [12]

- Colas de Mensajes (RabbitMQ, Redis, Celery): Enviar los correos a una cola de mensajes en lugar de procesarlos inmediatamente. Un sistema como Celery con Redis puede manejar múltiples tareas en paralelo y distribuir la carga. Esta opción, separa la recepción de la ejecución y opera bien para sistemas distribuidos y balanceo de carga pero requiere de instalación y configuración de servidores de externos, es complejo para el mantenimiento. [14] [15] [3]

En base a los objetivos del proyecto, la opción que mejor se ajusta ha sido la de implementar una arquitectura basada en procesamiento paralelo asíncrono mediante hilos (*multithreading*) y colas de tareas. Esta solución permite gestionar múltiples correos a la vez sin bloquear el hilo principal, siendo eficaz en operaciones de Input/Output (I/O) como el acceso a correos electrónicos o la comunicación con Odoo. Además, permite reaccionar de inmediato ante nuevos eventos sin depender de procesos pesados ni servicios externos como Redis.

La implementación en Python es sencilla y no requiere añadir dependencias complejas, lo que favorece su integración con el flujo de eventos que ofrece IMAP y cumpliendo así el objetivo principal de procesar varios correos por minuto de forma rápida y fiable.

Además, como las tareas principales del sistema (lectura de correos, limpieza de texto y llamadas HTTP) son operaciones de I/O, el uso de hilos resulta eficiente. En este tipo de tareas, el Global Interpreter Lock (GIL) de Python, que normalmente impide que varios hilos ejecuten código Python al mismo tiempo, no representa un problema. Esto se debe a que el GIL se libera automáticamente cuando un hilo está esperando una operación de I/O, lo que permite que otros hilos avancen en paralelo sin necesidad de usar múltiples procesos. [46]

En resumen, la combinación de threading y colas de mensajes es eficiente, simple y adecuada para lograr paralelismo en el procesamiento de correos.

5.4.4. Implementación del sistema multihilo

Una vez decidida la estrategia para conseguir que múltiples correos se procesasen a la vez y de forma rápida, se creó el módulo `TaskWorker` para gestionar el procesamiento en paralelo de correos electrónicos y crear tickets en Odoo, utilizando varios hilos (`threads`).

Se comenzó el proceso inicializando el constructor de la clase `TaskWorker` creando una cola de tareas para almacenar los correos pendientes de procesar (`task_queue`), definiendo un número de hilos con los que se trabajará para procesar la cola (`worker_count`) y creando una instancia de `TicketCreator` que se usará para crear los tickets en Odoo (`ticket_creator`).

A continuación, se definen diversos métodos para emplearlos más tarde en `IMAPHandler`, estos son:

- `add_task`:Añade un correo a la cola de tareas.
- `worker`: Es el método que ejecuta cada hilo, procesando los correos de la cola uno a uno. Primero, cada hilo intenta obtener un hilo de la cola (esperando 5 segundos) y si la cola está vacía, el hilo termina su trabajo. Sin embargo, si encuentra correos en la cola, lo procesa con `EmailProcessor` y si el cuerpo está vacío, se evita crear el ticket actuando como un filtro. Sino crea el ticket con `TicketCreator`. Después, marca la tarea como completada con un método interno de la cola `task_queue.task_done()`.
- start_workers: Crea y arranca los hilos especificados en `worker_count`, cada uno ejecutando el método `worker`. Después, se espera a que todos terminen la función con el método `join`.

5.4.5. Conexión y escucha en tiempo real de correos entrantes

Teniendo definido el sistema multihilo encargado de procesar correos de forma concurrente, el siguiente paso es describir un mecanismo que permita detectar y recibir correos nuevos según llegan. Para ello, se ha desarrollado el componente `IMAPHandler` que gestiona la conexión con el

servidor de correo y actua como receptor en tiempo real de los mensajes entrantes.

Este componente está basado en el protocolo IMAP, utilizado en clientes de correo para acceder a mensajes almacenados en un servidor remoto. En especial, se ha utilizado la librería `IMAPClient`, que proporciona una interfaz sencilla para interactuar con servidores IMAP como Outlook o Gmail, permitiendo operaciones como autentificación, selección de carpetas, búsqueda de mensajes no leídos y recepción de correos.

Durante las pruebas de conexión a la cuenta de correo desde Python, se detectaron errores que impedían el acceso. Para poder conectarse correctamente, hay que tener habilitado el acceso IMAP en la configuración de la cuenta, así como la opción de permitir aplicaciones menos seguras o bien generar una contraseña de aplicación. Esto se debe a que muchos servidores de correo, por motivos de seguridad y privacidad, bloquean el acceso desde aplicaciones externas que no cumplen ciertos requisitos de verificación.

En este caso, el acceso fue denegado debido a estas restricciones, incluso al utilizar las credenciales correctas. Para solucionarlo, se generó una contraseña de aplicación. Además, el sistema implementa un mecanismo de registro (logging) que captura los errores y excepciones que puedan surgir durante el intento de conexión, facilitando el diagnóstico y solución de incidencias en el futuro.

El proceso comienza en el constructor de la clase `IMAPHandler`, donde se recibe la configuración de conexión (servidor (`imap_server`), puerto (imap_port), usuario (imap_user) y contraseña (imap_password)) y una instancia de `TaskWorker` (`task_worker`). Esta última permite delegar los correos recibidos directamente al sistema multihilo para su procesamiento paralelo.

Durante la inicialización, se establece una conexión segura SSL/TLS con el servidor IMAP. Este protocolo cifra la comunicación entre el cliente y el servidor, evitando que terceros puedan interceptar información sensible como credenciales o correos electrónicos. Tras establecer esta conexión cifrada, se realiza el login con las credenciales proporcionadas y se selecciona la bandeja de entrada (INBOX). Si la conexión falla, se marca el cliente como inactivo.

El proceso funciona de la siguiente manera:

La función `start_idle()` es el método principal, encargado de mantener al sistema atento a nuevos correos. Lo hace iniciando un hilo en segundo plano que se queda escuchando continuamente al servidor IMAP mediante la función interna de este método `idle`, donde se activa el modo Integrated Development and Learning Environment (IDLE) que permite que el servidor avise al cliente cuando llega un nuevo correo, sin necesidad de que el cliente ande revisando constantemente. Así se evita sobrecargar el servidor o generar retrasos innecesarios.

El funcionamiento es el siguiente:

1. Se lanza un hilo en segundo plano que entra en un bucle infinito.
2. En cada iteración, se activa el modo `idle()` durante 60 segundos (tiempo de espera antes de forzar una comprobación).
3. Después, se sale del modo IDLE (`idle_done()`) y se invoca el método `fetch_emails()` que selcciona todos los correos no leídos de la carpeta indicada y se descarga su contenido mediante el método `fetch()` de IMAPClient, se transforman esos correos en objetos `email.message.Message`, y se pasan a `TaskWorker` mediante `task_worker.add_task()` para añadirlos a la cola de tareas.
4. Finalmente, se invoca a `task_worker.start_workers()` para lanzar los hilos que procesarán los correos en paralelo.

Gracias al modo `IDLE`, el sistema reacciona automáticamente ante la llegada de nuevos correos sin necesidad de realizar consultas periódicas ni intervención manual, se reduce la carga tanto en el cliente como en el servidor al evitar *polling*(encuestas) innecesario, se pueden procesar múltiples correos simultáneamente mejorando los tiempos de respuesta y aumentar o disminuir el número de hilos para soportar más o menos carga. Además, si ocurre algún error, se intenta volver a recuperar la conexión.

5.4.6. Ejecución global de los correos entrantes

La clase `AppMain` es el núcleo principal de la aplicación de ticketing automatizado. Su función es inicializar y conectar todos los componentes necesarios para que el sistema funcione de forma autónoma, desde la recepción de correos electrónicos hasta la creación de tickets en Odoo.

Esta clase se construye a partir de dos bloques de configuración principales:

- **Configuración de Odoo**: Incluye los parámetros `url`, `db`, `user` y `password`.
- **Configuración de IMAP**: Contiene los parámetros `imap_server`, `imap_port`, `imap_user` y `imap_password`.

Se ejecuta el método `run()` de la clase `AppMain`, siguiendo la siguiente secuencia:

1. **Inicialización de componentes**: Se crean las instancias necesarias de:
   - `OdooConector`: Conecta con la API de Odoo[44].
   - `ProcessedEmailTracker`: Gestiona el control de correos ya procesados.
   - `TicketCreator`: Transforma los correos en tickets.
   - `TaskWorker`: Gestiona la cola de tareas y el procesamiento multihilo.
   - `IMAPHandler`: Establece la conexión con el servidor de correo IMAP.
2. **Inicio del modo escucha (IDLE)**: Se lanza el método `start_idle()` de `IMAPHandler`, que mantiene una conexión activa con el servidor de correo, quedando a la espera de nuevos correos en tiempo real.
3. **Recepción y procesamiento de correos**: Cuando llega un nuevo correo, se detecta automáticamente, se descarga y se convierte en un objeto `email.message.Message`. Este objeto se añade a la cola de tareas, y se lanza el método `start_workers()` de `TaskWorker`, que activa varios hilos en paralelo para procesarlos.
4. **Transformación del correo en ticket**: Cada hilo toma correos de la cola y los procesa usando `EmailProcessor`. Si el cuerpo del correo es válido y no ha sido procesado anteriormente (según el `tracker`), se transforma en un ticket mediante `TicketCreator`, que además asigna su prioridad y categoría. Sin embargo, la asignación automática de técnicos no se aplica aún en esta parte del sistema,sino que se desarrolla más adelante.
5. **Ejecución continua**: Finalmente, la aplicación se mantiene activa dentro de un bucle infinito, escuchando y procesando nuevos correos de forma indefinida, hasta que sea detenida manualmente mediante un comando como `Ctrl + C`.

5.5. Procesamiento de correos históricos

Para entrenar el modelo de IA que asigna automáticamente un técnico responsable a cada nueva incidencia, se necesita una base de datos sólida y bien organizada que contenga incidencias reales ya resueltas teniendo como etiqueta al técnico que las resolvió.

Para que el modelo funcione bien, los datos con los que se entrena deben tener variedad y calidad para que el sistema sea preciso y confiable.

5.5.1. Estructura de los datos

Con este objetivo, se ha decidido construir dicha base de datos a partir de los correos electrónicos históricos gestionados por la empresa, los cuales se encuentran almacenados en un archivo Offline Storage Table (OST) vinculado a una cuenta corporativa de Microsoft Outlook.

Este archivo incluye tanto los mensajes ubicados en la carpeta *Bandeja de entrada* (42 correos) como aquellos archivados en la carpeta *Archivo* (7000 correos aprox). En total, se recopilan más de 8.000

mensajes intercambiados entre los años 2021 y finales de 2024, los cuales contienen información relevante y confidencial.

Debido a la gran cantidad de datos, se decidió limitar el análisis a 3000 correos para disponer de mejor control sobre los datos y minimizar la variabilidad y los cambios en la gestión para no afectar a la consistencia del modelo.

A continuación, en la tabla 8 se presenta la estructura de los correos:

Campo	**Descripción**
Asunto	Título del correo que resume el contenido principal
Remitente	Persona que envía el correo
Destinatarios (To)	Personas a las que va dirigido el mensaje
Cc	Copia informativa a otros receptores
Fecha	Fecha y hora de envío del correo
Descripción	Cuerpo del mensaje que detalla la incidencia
Técnico asignado	Etiqueta personalizada que indica quién resolvió la incidencia

Tabla 8: Estructura de la base de datos para el modelo de IA

5.5.2. Extracción de correos desde Outlook

Una vez comprendida la estructura de los datos, se procedió a la creación de la base de datos. Sin embargo, el formato OST presentó diversas dificultades en relación a la extracción de la información, especialmente en campos como las etiquetas Messaging Application Programming Interface (MAPI) personalizadas, que en este caso contienen el nombre del técnico responsable del incidente.

Debido a este problema, se probaron de forma práctica distintas alternativas que pudieran extraer esos campos:

- MBOX: Se probó exportar los correos a formato MBOX estándar. Este formato permite extraer la mayoría de campos básicos del mensaje (asunto, remitente, cuerpo), pero no conserva las etiquetas MAPI personalizadas de Outlook. Por lo tanto, la información del técnico asignado (almacenada en esas etiquetas) se perdía en la conversión. [31]
- EML: Similar al MBOX, el formato EML (archivos independientes por correo) extrae los campos esenciales del mensaje, pero tampoco retiene las categorías personalizadas de Outlook solo las que tiene Outlook por defecto. [2]
- Librería pypff (para OST/PST): Usándola desde WSL se utilizó el módulo libpff de Python[34], intentando acceder directamente al archivo OST. Aunque pypff puede leer estructuras básicas de OST/PST, tampoco recupera los campos MAPI personalizados ni el contenido de categorías definidas en Outlook. [33]

Dado que las etiquetas personalizadas eran uno de los campos esenciales para la base de datos (contienen el técnico asignado en cada incidencia), se optó finalmente por la siguiente solución: convertir el archivo de OST a PST, importar las carpetas relevantes en una cuenta de Outlook, y luego utilizar Python para acceder a Outlook mediante la librería `win32com.client`, un módulo de Python que permite acceder a objetos COM (Component Object Model) expuestos por aplicaciones de Microsoft Office.

La conexión se establece mediante la siguiente secuencia básica:

```
import win32com.client

outlook = win32com.client.Dispatch("Outlook.Application")
namespace = outlook.GetNamespace("MAPI")
cuenta = namespace.Folders["Carpeta"]
bandeja = cuenta.Folders["Archivo"]
```

Listing 2: Inicialización de objetos para acceder a carpetas de Outlook con win32com

Una vez conectada la cuenta, se puede acceder a cualquier carpeta dentro de Outlook, recorrer los mensajes (Items) y obtener las propiedades de cada uno.

Este enfoque permite obtener las propiedades MAPI, incluyendo etiquetas personalizadas, lo cual no era posible mediante otras soluciones como formatos MBOX o EML. Además, al tratarse de un acceso directo a Outlook instalado, no requiere servidores intermedios ni transformación de formatos, facilitando su integración.

5.5.3. Arquitectura General del Sistema de Correos Históricos

Con la conexión a Outlook y el acceso a los datos solucionado, se diseñó una arquitectura modular capaz de procesar automáticamente una serie de correos, transformándolos en tickets estructurados y funcionales para ser integrados en el sistema Odoo. El diseño asegura que cada parte del sistema se encargue de una tarea concreta y se conecte con las otras de manera ordenada obteniendo un mejor funcionamiento y mantenimiento del sistema.

La clase principal es OutlookProcessor. Este componente gestiona la conexión a Outlook, recorre las carpetas indicadas (por ejemplo, Bandeja de entrada o Archivo) y extrae los campos relevantes de cada mensaje. En cada iteración colabora con otros módulos del sistema, dando forma al flujo completo de procesamiento. La estructura modular es la siguiente:

Clase	**Rol principal**
OutlookProcessor	Recorre las carpetas de Outlook, extrae y organiza los datos de cada mensaje.
EmailProcessor	Proporciona funciones de limpieza, normalización y traducción del texto.
TicketClassifier	Asigna prioridad, categoría y plazo de resolución al ticket.
TicketCreator	Construye el JSON del ticket y lo envía a Odoo[44] usando OdooConector.
OdooConector	Cliente genérico de la API JSON-RPC de Odoo[44].
ProcessedEmailTracker	Registra hashes de mensajes ya procesados para evitar duplicados.

Tabla 9: Componentes del sistema y sus roles principales

A continuación, se muestra el diagrama de clases:

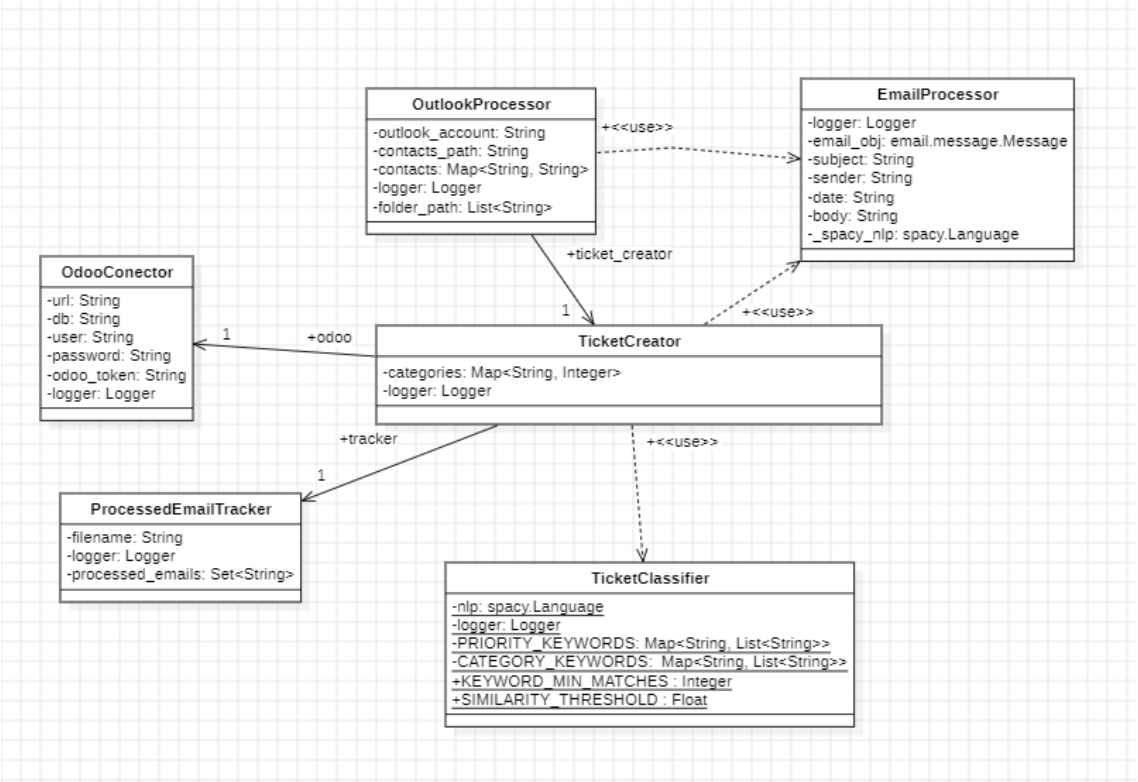

Figura 5: Diagrama de clases del sistema de creación de la base de datos histórico

El flujo de procesamiento de esta sección puede consultarse en el Apéndice, en la Figura 9.

Todo el sistema funciona gracias a este diseño y a cómo se conectan sus componentes. En los apartados siguientes, se muestra como funciona cada parte del sistema con más detalle.

5.5.4. Acceso y lectura de correos

En esta sección, se detalla como se accede a las bandejas de entrada de Outlook mediante la interfaz COM proporcionada por la librería `win32com.client`, se recorren los mensajes almacenados y se extraen de cada uno los campos necesarios para la creación de tickets. Todo este proceso se ha implementado en la clase `OutlookProcessor`.

Tras revisar varios correos electrónicos, se detectó que al acceder a cada correo, se observaba un flujo de mensajes compuesto por el mensaje principal y una cadena de respuestas o reenvíos asociados en algunos correos.

Teniendo esto en cuenta, se creó el método principal `process_emails()`, que procesa todos los correos electrónicos de una bandeja específica de Outlook, extrayendo su información relevante y creando tickets en Odoo a partir de ellos. Además, también se encarga de gestionar posibles correos embebidos por ponerles un nombre común (respuestas o reenvíos incluidos como texto dentro del cuerpo del mensaje).

Inicialmente, se lanza Outlook, se accede al espacio de nombres MAPI e intenta abrir la cuenta y bandejas especificadas si existen. A continuación, se obtienen todos los mensajes ordenados por fecha de recepción (más recientes primero).

Para cada mensaje se extraen los siguientes metadatos:

- Asunto del correo (`Subject`)
- Remitente (`SenderName`, `SenderEmailAddress`)

- Destinatarios principales (`To`) y en copia (`Cc`)
- Fecha de envío (`SentOn`)
- Categorías de Outlook (`Categories`) que constituye al técnico
- Cuerpo del mensaje (`Body`)
- Identificador de hilo (`EntryID`), para distinguir correos embebidos

A partir de estos datos, se construye un objeto de tipo `email.message.EmailMessage` mediante la función `build_email`, que encapsula el contenido completo del correo. Al mismo tiempo, se registra al remitente y los destinatarios en la lista de contactos, si no estaban previamente almacenados.

Este objeto `EmailMessage` se pasa a la clase `EmailProcessor`, que se encarga de su limpieza, traducción y estructura final del cuerpo del mensaje.

De este modo, es posible comprobar rápidamente si el mensaje es repetido y si el cuerpo del mensaje es válido `(processor.body)` . Si es así, se asigna una categoría y prioridad al correo en base a su contenido (`TicketClassifier`) y se crea el ticket llamando a `create_from_email` de la clase `TicketCreator`, pasándole ese objeto procesado. De esta forma, se evita enviar correos vacíos o irrelevantes al sistema de tickets.

Una vez procesado el correo principal, se analiza el cuerpo del mensaje en busca de posibles respuestas o reenvíos anteriores que se encuentren en texto plano. Para ello, se procede a extraerlos y procesarlos:

5.5.5. Extracción de correos embebidos

Para extraerlos, se aplica el método `extract_embedded_emails`, que sigue los siguientes pasos:

1. Comprobar tipo de entrada: Si el texto no es `String`, se devuelve lista vacía.
2. Normalizar saltos de línea: Convierte todos los saltos de línea a `\n`, puesto que hay correos con mucho espaciado entre líneas y puede dificultar el procesamiento.
3. División del texto en bloques: Se usa la expresión regular multilingüe:

```
r"\n(?=(De|From|Da|Von|Envoyé):)"
```

Listing 3: Expresión regular para dividir correos

   para detectar inicios de hilos y separar el texto cada vez que encuentra una línea que empieza por *De, From, etc.* que suelen marcar el inicio de un correo embebido.
4. Reconstruir bloques del correo
5. Añadir último bloque: Si queda texto en actual, lo añade a la lista.
6. Filtrar bloques: Se cogen los bloques que contienen un campo *Asunto* para asegurarse que son bloques completos y se devuelven.

5.5.6. Procesamiento de correos embebidos

A continuación, se detalla como se procesan estos correos mediante la función `process_embedded_text`.

Primero se aplica el método `normalize_headers` de `EmailProcessor` para estandarizar los encabezados de los mensajes embebidos, ya que suelen venir en varios idiomas (como inglés, francés, alemán) y con formatos distintos: *From, De, Envoyé, etc.* Esta normalización los convierte al español, lo que facilita su detección mediante expresiones regulares.

En cambio, los correos principales no necesitan esta normalización puesto que la información principal (remitente, asunto, fecha...) ya viene bien estructurada gracias a las propiedades del objeto `message` que ofrece la API de Outlook (`win32com.client`), sin importar el idioma del correo. Por

eso, la normalización solo se aplica cuando los encabezados vienen como texto libre dentro del cuerpo, es decir, en los mensajes embebidos.

Extracción de metadatos embebidos

Después de normalizar los encabezados, se crea un diccionario `extraidas` para guardar los metadatos principales del correo extrayendo los campos clave mediante expresiones regulares específicas para varios idiomas. Se recopila:

- El remitente (from), extrayendo la dirección de correo electrónico del texto con `clean_address` de la clase `EmailProcessor` y registra el contacto con `registrar_contacto` extrayendo el nombre en el texto con formato 'Nombre Apellido' en la función `extract_name_from_text`. Si no se encontró el remitente, intenta buscar un email en el texto.
- Los destinatarios (to, cc), mediante `extract_outlook_emails` que extrae todas las direcciones de correo en el texto y, si no se encuentra el correo, mediante `find_contact_by_name` intentando buscar un nombre en el texto.
- La fecha (date), que se normaliza con `clean_date` o se intenta extraer desde otras líneas si el formato es irregular.
- Para el resto de campos, se guarda el valor tal cual.

Como estos correos no tienen asociados objetos `message` de Outlook desde los que obtener propiedades estructuradas, se hereda la categoría del correo principal al llamarse a `process_embedded_text`. Esta categoría se pasa como `user_id` al creador de tickets, con el mismo objetivo que en los mensajes principales: identificar el técnico responsable o etiquetar la incidencia con información contextual.

Una vez localizados los encabezados, se identifica el comienzo del cuerpo del mensaje y se construye un nuevo objeto `EmailMessage`, que se procesa exactamente igual que un correo principal.

5.5.7. Control de duplicados

Aunque todos los correos, tanto normales como embebidos, se procesan finalmente mediante la clase TicketCreator, y esta ya incluye un control de duplicados, en el caso de los correos embebidos se realiza una verificación previa del hash antes de llamar a `create_from_email` porque muchos de estos correos no disponen de identificadores únicos (Message-ID), por lo que se calcula un hash manualmente a partir de sus campos principales para evitar el procesamiento innecesario de mensajes ya tratados.

En cambio, para los correos principales, esta comprobación se omite en `OutlookProcessor` porque `TicketCreator` ya la realiza de forma centralizada, y en estos casos sí suele disponerse de Message-ID fiable para el seguimiento. De esta forma, si el correo ya fue tratado anteriormente, se omite automáticamente, garantizando la unicidad de los tickets generados. Finalmente, se crea el ticket al igual que un correo principal.

En resumen, procesar todos esos correos antiguos ha permitido crear una base de datos clara y bien organizada, esencial para entrenar el modelo que asigna técnicos de forma automática. Gracias al diseño modular y a técnicas específicas para manejar correos duplicados, embebidos y metadatos, el sistema se integra bien con Odoo y replica el comportamiento de los correos reales.

5.6. Asignación de técnicos mediante algoritmos de IA

Con la base de datos ya creada en Odoo, el siguiente paso consiste en la asignación del técnico más adecuado a cada nueva incidencia.

5.6.1. Elección del modelo para clasificar

Lo primero es seleccionar el modelo que permita, a partir del contenido textual de la incidencia y sus metadatos, predecir qué técnico es el más adecuado para resolverla.

Para que el modelo pueda funcionar, hay que preparar los datos, empleando varias técnicas, según el tipo de datos que se use y el modelo a aplicar.

- **Vectorización de texto**: Transforma descripciones en vectores usando técnicas como:
  - **Term Frequency-Inverse Document Frequency (TD-IDF)**: Calcula la relevancia de cada palabra en un documento con respecto a un conjunto de documentos. [18]
  - **Embeddings(Word2Vec, GloVe, Bidirectional Encoder Representations from Transformers (BERT))**: Son representaciones semánticas más avanzadas que capturan el significado y contexto de las palabras. [51]
- **Codificación Categórica**: Convierte variables categóricas(técnicos, clientes, categorías), aquellas que representan cualidades o categorías, en formatos numéricos para ser utilizados en modelos predictivos ya que estos trabajan mejor con números (One-Hot, Label Encoding, Target Encoding). [47]
- **Formato tabular**: Cada fila representa un ticket y las columnas contienen: variables numéricas (como tiempo de resolución o número de incidencias previas), variables categóricas (como técnico asignado o tipo de problema), y variables de texto vectorizadas (como la descripción del problema procesada con TF-IDF o embeddings).

Estos datos pueden ser utilizados con distintos modelos, desde algoritmos como *Random Forest* o *Naive Bayes* para variables estructuradas, hasta modelos más complejos como *Redes Neuronales* o *BERT* que ofrecen resultados más precisos en tareas complejas, pero necesitan muchos datos y mucha potencia de cálculo.

En todos los casos, estos modelos deben entrenarse, validarse y ajustarse constantemente con datos bien etiquetados y actualizados. Por ello, pueden no ser la mejor opción ya que los datos en este caso, pueden variar al aparecer problemas distintos en la base de datos con nuevos técnicos que se han incorporado a la empresa.

Por otro lado, existen otros modelos que permiten realizar predicciones basadas en similitud semántica, sin tener que entrenar un modelo desde cero. Algunas de estas estrategias son:

- **Búsqueda vectorial con Elasticsearch**: los textos se codifican en vectores mediante embeddings (por ejemplo, BERT o SentenceTransformers) y se almacenan en una base de datos vectorial. Cuando aparece una nueva descripción, se recuperan los tickets más similares y se asigna el técnico más frecuente entre ellos. [36]
- **Estrategia híbrida Retrieval-Augmented Generation (RAG)**: Combina la búsqueda semántica con un modelo de lenguaje Large Language Model (LLM), generando una predicción basada en los ejemplos recuperados. Aunque es más potente, implica más latencia y consumo computacional. [23]

Para elegir la mejor estrategia de clasificación, se ha realizado un análisis comparativo de los distintos modelos en la siguiente tabla 5.6.1 para ver cuál es mejor usar:

Aspecto	**Modelos de Machine Learning**	**Búsqueda Vectorial en Elasticsearch**	**RAG (Búsqueda + LLM)**
Enfoque	Clasificación supervisada	Recuperación por similitud semántica	Recuperación + razonamiento textual contextual
Necesita entrenamiento	Sí (entrenar con datos históricos)	No (usa comparaciones directas)	No (usa modelo preentrenado + ejemplos)
Requiere pipeline de Machine Learning (ML) externo	Sí (entorno de entrenamiento, validación...)	No (embedding + Elasticsearch)	No (solo embeddings + LLM)
Actualización con nuevos datos	Reentrenar el modelo	Solo indexar los nuevos vectores	Solo indexar + enviar al LLM
Velocidad de predicción	Muy rápida tras entrenar	Muy rápida si Elasticsearch está bien configurado	Media: depende del modelo LLM y latencia
Interpretabilidad	Menor (según el modelo)	Alta: ves incidencias similares concretas	Muy alta: el LLM puede razonar en lenguaje natural
Escalabilidad	Alta, pero depende del entorno de despliegue	Alta, usando Hierarchical Navigable Small World (HNSW) u otro Artificial Neural Network (ANN)	Media: limitado por tokens, coste y latencia
Precisión y control	Alta si se entrena bien y se ajusta el modelo	Dependiente de la calidad del embedding	Alta si se formatea bien el prompt y contexto
Adaptabilidad a nuevos técnicos	Puede requerir reentrenamiento	Se adapta automáticamente si hay embeddings en el histórico	Se adapta automáticamente, sin reentrenar
Dependencia del embedding	Opcional (puede usar TF-IDF, one-hot, etc.)	Esencial: el vector define todo	Esencial: el vector permite recuperar los ejemplos

Tabla 10: Comparativa de Enfoques de Clasificación

Tras comparar las diferentes opciones de la tabla, se llega a la conclusión que el mejor modelo para este proyecto es el que se basa en **búsqueda vectorial semántica**, por las siguientes razones:

- **No necesita entrenamiento previo**: No requiere de un pipeline completo de entrenamiento, validación y ajustes, al contrario que los modelos de Machine Learning. De esta manera, se puede trabajar directamente con los datos históricos, sin necesidad de procesos adicionales.
- **Se adapta fácilmente a nuevos datos y técnicos**: Solo hay que indexar los nuevos tickets a los correos históricos para que el sistema sea capaz de utilizarlos en predicciones futuras. En cambio, los modelos tradicionales necesitan ser reentrenados completamente cada vez que se amplía la base de datos.
- **Facilidad de comprensión**: Recupera incidencias parecidas ya resueltas por los técnicos recomendados facilitando la propuesta de soluciones.
- **Bajo coste computacional**: Utiliza embeddings ligeros (por ejemplo, de `SentenceTransformers`) y no requiere infraestructuras pesadas como GPUs o servidores

especializados, como sí ocurre con enfoques más complejos como los modelos RAG o redes neuronales profundas.

- **Cumple con los objetivos del proyecto**: El objetivo principal no es generar texto ni responder preguntas complejas, sino simplemente asignar el técnico más adecuado basándose en la similitud con incidencias pasadas. Por tanto, la búsqueda vectorial ofrece una solución directa, simple y efectiva.

5.6.2. Descripción y funciones de la API

Para aplicar este modelo de IA, se utiliza una API que consiste en una aplicación web basada en FastAPI que permite realizar búsquedas en documentos almacenados en Elasticsearch. Utiliza un modelo de transformadores para generar representaciones vectoriales de los documentos y mejorar la relevancia de los resultados. Es privada y ha sido desarrollada internamente por la empresa. Esta ofrece funcionalidades de búsqueda semántica y RAG, y está desplegada en una red interna accesible únicamente mediante VPN.

El acceso a la API requiere:

- Un enlace interno proporcionado por la empresa.
- Configuración de una conexión VPN para habilitar el acceso seguro.

Aunque la interfaz gráfica de la API incluye funcionalidades como cargar documentos en formatos Word y chat asistido con LLMs, para este proyecto se han utilizado únicamente una serie de endpoints disponibles que permiten interactuar con Elasticsearch y el motor de embeddings. Sin embargo, para algunos endpoints se requiere de una API key como parámetro para ejecutarlos ya que permiten modificar los índices de Elasticsearch.

A continuación, se detallan los principales endpoints empleados:

- `delete_doc_or_index`: Elimina un índice completo o documentos específicos dentro de un índice en Elasticsearch y requiere de API key.
- `insert_doc`: Inserta un documento troceado desde JSON o archivo codificado en base64, usando Tika si es necesario permitiendo extraer texto y metadatos de archivos de muchos tipos diferentes y requiere de API key.
- `search`: Busca documentos semánticamente y requiere de API key.
- `get_indices`: Muestra los índices creados en Elasticsearch.

Estos endpoints se pueden usar desde Python sin dar problema ya que es donde tenemos todo el sistema y de donde debemos sacar la información a pasarle a la API. Para ello, me proporcionaron un archivo config.json con los parámetros necesarios para poder ejecutar todos estos endpoints y controlar la autenticación.

```
{
    "urlAPI": "url",            % URL del servicio de la API
    "apiKey": "clave",          % Clave de autenticación
    "app": "Odoo",              % Nombre de la aplicación que está enviando los
                                % datos
    "tenant": "tickets",        % Identificador del cliente o entorno
    "searchEntity": "Tickets",  % Tipo de entidad que se va a indexar
    "delete_index": false
}
```

Listing 4: Configuración de la API privada

El campo delete index indica si se debe borrar el índice actual antes de guardar los nuevos datos. Si es true, antes de indexar se borra el índice existente en el sistema externo y si es false, no se borra nada y solo se añaden nuevos datos.

5.6.3. Integración de la base de datos de Odoo en la API

Antes de realizar la búsqueda para predecir al técnico, hay que subir la base de datos de Odoo a la API para poder predecir y mostrar los documentos similares del técnico que llega.

Con ese fin, se creó un script (`main.py`) con ayuda de los desarrolladores de la API de la empresa. El script hace lo siguiente:

1. Carga la configuración del archivo config.json y la registra en un archivo llamado sync.log.
2. Si en la configuración se indica que `delete_index` es true, se envía una petición POST a la API para borrar todos los datos indexados previamente para ese tenant y app con la función `delete_doc_or_index`.
3. Se buscan los tickets en Odoo `(env['helpdesk.ticket'].search([]))`.
4. Se procesa cada ticket buscando los mensajes.
5. Para predecir el técnico nos basaremos en la descripción, por eso, se construye un diccionario con la descripción, el técnico asignado y el ID del ticket, se crea un JSON con los datos del ticket y se envía a la API usando una petición PUT con la función `insert_doc`.
6. Se indexan los mensajes del chatter del ticket. Para cada mensaje del ticket, si tiene cuerpo, se codifica en base64 y se envía a la API como un documento tipo *Message* y para cada adjunto, se intenta extraer el texto (decodificando si es texto o extrayendo si es PDF). Si se obtiene texto, se envía a la API como documento tipo *Attachment* junto con el nombre y la fecha de creación.
7. Se informa por log y por consola si la operación fue exitosa o si hubo errores.

En el Apéndice se muestra como ejecutar este script y su resultado en el Manual de Usuario.

5.6.4. Predicción del técnico

Con los correos históricos ya indexados en la API de la empresa, se implementó una nueva clase en el proyecto denominada `TecnicoPredictor` que predice automáticamente el técnico más adecuado para una nueva incidencia, basándose en descripciones similares ya almacenadas en la API mediante una búsqueda vectorial semántica donde compara el nuevo texto con los históricos almacenados en Elasticsearch y extrae qué técnico resolvió las incidencias más parecidas.

Durante la implementación de la clase, se plantearon dos formas distintas de asignar el técnico responsable:

- Asignar el técnico del documento más relevante, el que más score tenga.
- Contar cuántas veces aparece cada técnico de los 10 documentos en ellos, y el que mayor votos tenga, es el que se asignará.

Se eligió la segunda opción porque resulta más fiable puesto que, al considerar varios resultados en lugar de uno solo, se obtiene una predicción más estable, siendo menos probable que un error puntual afecte a la elección del técnico y se selccionaron los 10 primeros documentos para asegurar que las predicciones se basaran en resultados relevantes y no se alejaran demasiado del problema original.

Para conectarse a la API, la clase reutiliza los parámetros definidos en el archivo `config.json`, el mismo que se empleó para subir los correos históricos.

Primero, en el constructor de la clase, se carga y lee la configuración de ese archivo y se extraen los datos para poder conectarse a la API.

Después, se define la función `predict_technician`, encargada de asignar el técnico más frecuente entre los 10 documentos más relevantes de la siguiente manera:

1. Construye el payload, es decir, prepara los datos que se enviarán a la API, incluyendo el

texto a analizar y pidiendo que en los resultados se añada el campo "Tecnico".

2. Se preparan los headers (API key y el tipo de contenido).
3. Se realiza petición POST con la función `search` enviando el texto y recibiendo los 100 documentos más relevantes en relación a él con una puntuación (`score`) de similitud, cuanto más alta más similar.
4. Si la respuesta no es correcta, se devuelve el string .ºtro".
5. Cuando la respuesta es correcta, se convierte a JSON. Si no hay resultados, se devuelve el string .ºtroz en caso contrario, se extraen los técnicos de los primeros `max_results` documentos (10 en este caso).
6. Se cuenta cuantas veces aparece cada técnico y el que tenga más votos, se asigna. Si hay empate, se escoge el primero.

En el Ápendice se muestra un ejemplo de prueba para comprobar que se asigna bien un técnico y se explica como ejecutar el script en el Manual de Usuario.

6. Evaluación de Resultados

En esta sección se recogen los resultados obtenidos tras ejecutar el sistema completo, se analiza si las funciones desarrolladas funcionan correctamente, cómo rinde el sistema y cómo se clasifican las incidencias por técnico. También se comentan las principales limitaciones detectadas durante el desarrollo, con el objetivo de plantear posibles mejoras para el futuro.

6.0.1. Pruebas para los correos entrantes y la creación de históricos

Se han implementado una serie de test para comprobar que los métodos de las clases `EmailProcessor`, `TicketClassifier` y `ProcessedEmailTracker` funcionan correctamente, ya que estas incluyen funciones importantes para el sistema, por lo que es fundamental asegurarse de que su comportamiento sea correcto en diferentes situaciones y con distintos tipos de datos. Gracias a estas pruebas, es más fácil detectar posibles errores y garantizar que el sistema sea estable y fiable.

Por otro lado, las clases como `OdooConector`, `TicketCreator`, `IMAPHandler`, `TaskWorker`, `AppMain` y `OutlookProcessor` trabajan con varios componentes a la vez y dependen de servicios externos, como los servidores IMAP o el entorno de Odoo. Por eso, su comprobación se hizo a través de pruebas funcionales desde la terminal de WSL o de Python de la aplicación de Visual Studio Code. Se comprobó, por ejemplo, que se podía acceder a Odoo desde el navegador, que los correos se recibían correctamente y que se generaban tickets con todos sus campos en la base de datos de Odoo, etc.

6.0.2. Pruebas de la clasificación del técnico

Para predecir el técnico con la mayor precisión posible mediante búsqueda semántica a través de la API, se han realizado varias pruebas empleando diferentes técnicas.

Para conseguir este objetivo, se dividieron los correos en dos conjuntos: un conjunto de entrenamiento (train set) y un conjunto de prueba (test set). En el train set se añadieron correos reales ya clasificados, para que la API pudiera entrenarse y mejorar sus predicciones, mientras que el test set sirvió para comprobar si el sistema era capaz de predecir correctamente el técnico a partir de correos reales nuevos no vistos.

Empezando por el train set, se crearon cuatro versiones distintas, cada una aplicando una técnica de procesamiento diferente al contenido de los correos, los cuales fueron limpiados y procesados anteriormente siguiendo los pasos en orden descritos en la sección de *Procesamiento y limpieza de correos*, sin incluir aún las técnicas específicas de NLP.

A continuación, se describen las técnicas empleadas:

- **Texto limpio sin técnicas NLP**: Se aplica limpieza básica (eliminación de firmas, normalización de encabezados, etc.)
- **Procesamiento con NLP (tokenización + stopwords)**: Se dividen las palabras (tokens) y se eliminan aquellas más comunes que no aportan información útil (stopwords).
- **Procesamiento con lematización**: Se transforma cada palabra a su forma base o raíz gramatical (por ejemplo, “comprando” se convierte en “comprar”).
- **Procesamiento con stemming**: Se reducen las palabras a su raíz morfológica (por ejemplo, “comprador” y “comprando” se convierten en “compr”).

En todas las versiones se procesaron los mismos correos (los 3000 primeros), aplicando la técnica correspondiente en cada caso para poder comparar posteriormente los resultados de manera igualitaria.

Después, en cuanto al test set, se procesaron 100 correos nuevos reales ya resueltos. Estos correos no fueron incluidos en ninguna de las bases de datos anteriores y se empleó la misma limpieza que para los datos crudos (traducción de correos, etc.) ya que los pies de página por ejemplo pueden

ocupar medio cuerpo del correo causando mucho ruido, sin aplicar técnicas NLP, para simular un caso real de predicción conociendo el resultado.

Esto se puso a prueba mediante la clase `ModelEvaluator` que realiza predicciones para cada train set recorriendo los correos del test set, y se recogen los documentos más similares devueltos por la API junto con la predicción del técnico elegido por votación. También, se obtuvieron los tres técnicos más frecuentes para calcular una métrica top-3 comprobando si el modelo se acerca a la respuesta correcta aunque no sea su primera opción. Para medir la calidad de las predicciones, se evaluaron las siguientes métricas:

- Accuracy: Porcentaje de predicciones totalmente acertadas (el técnico predicho coincide con alguno de los técnicos reales ya que la incidencia puede haber sido resuelta por más de un técnico). [4]
- Top-3 Accuracy: Porcentaje de veces que el técnico correcto está entre los tres más sugeridos por el sistema. [26]
- Precisión, Recall y F1-Score por técnico: Estas métricas permiten evaluar cómo responde el modelo con cada persona individualmente. Indican la proporción de aciertos al predecir un técnico, cuántos correos se logran identificar correctamente para cada uno y una combinación de ambas medidas. [4]
- Matriz de confusión: Es una matriz que muestra cuántas veces el modelo acierta o se equivoca al hacer predicciones. Las filas representan los resultados reales, y las columnas lo que el modelo ha predicho. Así se puede ver fácilmente en qué casos el sistema confunde técnicos entre sí. [19]

Además, la API devuelve un "score de similitud" que indica lo parecido que es un mensaje nuevo respecto a los ya indexados. Aunque no se usó como métrica principal porque no comprueba directamente si la predicción del técnico es correcta, se observó que cuánto más bajo es, se devuelven coincidencias con peores predicciones.

Tras aplicar las métricas, en la siguiente Figura 6 pueden verse los resultados del Accuracy y del Top-3 Accuracy:

```
Resultados finales:
Texto Crudo: 62.0% de acierto (62/100), Top-3: 87.0%
Con NLTK: 64.0% de acierto (64/100), Top-3: 90.0%
Con Lematización: 58.0% de acierto (58/100), Top-3: 84.0%
Con Stemming: 49.0% de acierto (49/100), Top-3: 86.0%
```

Figura 6: Resultado de aplicar las métricas Accuracy y Top-3 Accuracy

La técnica basada en tokenización junto con stopwords alcanzó el mayor rendimiento, con un 64 % de *accuracy* y un 90 % *top-3 accuracy*, aunque también se obtuvieron buenos resultados con el texto limpio sin técnicas NLP, sus porcentajes fueron algo inferiores. En cambio, técnicas como la lematización y el stemming consiguieron porcentajes muchos menores posiblemente debido a la pérdida de información al reducir demasiado el contenido de las palabras.

Por otro lado, respecto a las métricas individuales por técnico (precisión, recall y F1-score), se observó que los técnicos que más aparecen en el conjunto de entrenamiento mostraron mejor rendimiento como Néstor, Alba o Guille. Por ejemplo, el técnico Néstor alcanzó un F1-score superior al 0.8 en varios modelos. En cambio, técnicos como José Félix o Gerardo, con pocos casos en los datos de entrenamiento, obtuvieron resultados casi nulos, lo que refleja que el modelo necesita suficientes casos previos para poder aprender a identificarlos correctamente.

Por último, se analizaron matrices de confusión para identificar en qué casos el sistema confunde técnicos entre sí más detalladamente. Estas se pueden ver completas en el Anexo en las Figuras 10, 11, 12 y 13.

En ellas, se puede observar que en general, los modelos basados en texto crudo y con NLTK mostra-

ron una diagonal más definida, lo que indica mayor precisión y menos errores al clasificar técnicos como Néstor o Guille. En cambio, los modelos con lematización y especialmente con stemming presentaron mayor dispersión fuera de la diagonal, lo que evidencia más errores y confusiones frecuentes entre técnicos. Esto refuerza que un procesamiento más agresivo del lenguaje puede deteriorar la calidad de la predicción.

Como conclusión, los resultados muestran que el preprocesamiento del lenguaje es importante para mejorar las predicciones. La técnica de NLP usando tokenización y stopwords ofrecieron el mejor rendimiento, mientras que técnicas como lematización o stemming, al ser más agresivas, redujeron la precisión. Además, el análisis de las métricas por técnico mostró que, cuanto más aparece una persona en los datos de entrenamiento, más fácil es para el sistema identificarla correctamente. Esto resalta la importancia de contar con una base de datos extensa, variada y actualizada. Por todo esto, se eligió la técnica de procesamiento con NLP (tokenización y stopwords) para el contenido de los correos en las predicciones del sistema final.

6.1. Discusión de resultados y limitaciones

Los resultados del sistema han sido en general positivos. Se ha logrado automatizar completamente el flujo de gestión de incidencias, desde la recepción del correo hasta la creación del ticket y la asignación del técnico que era el objetivo principal del proyecto y se cumple con los requisitos funcionales y no funcionales que se definieron. Además, el uso de técnicas de NLP ha permitido mejorar la clasificación de mensajes y la precisión del sistema.

Sin embargo, se identificaron algunas limitaciones:

- Predicción del técnico: El sistema de votación empleado para predecir al técnico demostró ser funcional, pero depende en gran medida de la calidad de los textos procesados y del tamaño del histórico indexado. En mensajes muy cortos o ambiguos, la predicción puede no ser tan precisa.
- Dependencia de servicios externos: El sistema depende de una API privada para la búsqueda semántica. Si esta sufre problemas o cambios, el sistema puede verse afectado.

Sin embargo, el sistema cumple con los objetivos principales del proyecto y representa una solución eficaz y extensible, aunque se pueden aañdir mejoras.

7. Conclusiones

Este Trabajo de Fin de Grado ha consistido en crear un sistema automatizado para gestionar incidencias en la empresa IPS Norte. El objetivo consiste en solucionar el problema de depender únicamente del correo electrónico, para atender las incidencias técnicas sin contar con un sistema especifico para este fin. Gracias a este proyecto, se ha conseguido quitar una funcion que no correspondia al servicio de mensajeria que tenia la empresa gestionado por la aplicacion outlook , ejecutando la tarea especifica de asignar una etiqueta con el nombre de los tecnicos para hacerse cargo de dicha incidencia. De esta manera, se han mejorado los procesos del área de soporte aplicando en un entorno real, gran parte de los conocimientos adquiridos durante el Grado en Ingeniería Informática.

Durante este trabajo se ha analizado cómo funcionaba el sistema anterior en la empresa, detectando sus limitaciones y buscando una solución mejor que utiliza tecnologías actuales como Odoo, técnicas de NLP, IA y automatización con Python. Además, se ha diseñado una estructura basada en módulos para facilitar su desarrollo, pruebas, mantenimiento y cambios o mejoras futuras siguiendo una metodología incremental.

Uno de los logros más importantes del proyecto ha sido automatizar en gran medida el proceso de gestión de incidencias: desde que llega un correo hasta que se crea el ticket con el técnico asignado, incluyendo su clasificación y prioridad. Esto se ha conseguido gracias a la integración de distintos elementos, como la recepción de correos en tiempo real con IMAP y el uso de paralelismo, el

procesamiento del texto para poder limpiar y extraer mejor los campos y el contenido de cada correo, la detección de mensajes duplicados para evitar redundancia en la base de datos y la predicción automática del técnico más adecuado mediante búsqueda semántica usando la API externa proporcionada por la empresa.

También se ha construido un sistema para generar tickets de incidencias ya solucionadas anteriormente con su técnico correspondiente en la aplicación de Odoo, incluyendo en estos toda la conversación del correo junto con las respuestas o los reenvíos derivados de los mismos. De esta forma, se pudo generar una base de datos útil para entrenar y validar los modelos de clasificación.

Respecto a la aplicación de las buenas prácticas ITIL, estas se cumplen en el proceso de automatización desde la detección hasta el cierre del ticket.

En cuanto a los SLA, aunque se asignan plazos y prioridades siguiendo unas reglas definidas, el sistema aún no realiza un seguimiento completo de su cumplimiento, como controlar los tiempos o generar alertas automáticas. Por tanto, está alineado con el enfoque de los SLA, pero todavía no los gestiona de forma completa.

Desde el punto de vista académico y personal, este trabajo ha sido una experiencia profundamente enriquecedora. Me ha permitido consolidar conocimientos en programación, sistemas ERP, arquitectura de software, gestión de servicios TI e IA. Gracias a este proyecto, he aprendido a abordar problemas reales surgidos en un entorno laboral, investigar sus posibles soluciones aplicando mis conocimientos adquiridos en esta carrera y llegando a solucionar dicho problema.

En definitiva, este proyecto me ha hecho ver lo que tendré que afrontar en un futuro laboral.

8. Trabajo futuro

Aunque el sistema es funcional, existen varias áreas donde se podrían introducir mejoras o cambios:

- **Aprendizaje continuo**: Actualmente, el sistema no incorpora feedback de si la predicción fue correcta. Incluir aprendizaje supervisado (por ejemplo, con modelos entrenables en Odoo) permitiría mejorar progresivamente la precisión.
- **Interfaz gráfica de gestión**: Se podría crear un panel de control o dashboard donde visualizar el estado de los tickets, ajustar parámetros del clasificador o visualizar estadísticas del rendimiento del sistema.
- **Valoraciones de los clientes**: Permitir que los usuarios valoren la atención recibida, retroalimentando así el sistema con datos útiles para evaluación y mejora de calidad.
- **Optimización del rendimiento**: El proceso de lematización con spaCy, aunque preciso, puede resultar lento con correos muy largos o en grandes volúmenes. Aplicar paralelismo o procesamiento por lotes podría mejorar la velocidad.
- **Mejoras en la traducción**: Aunque se incluye un traductor automático, sería mejor incluir idiomas comunes de la empresa de forma nativa, ya que eso ayudaría a que las clasificaciones sean más precisas.
- **Clasificación multitarea**: Extender el clasificador para que no solo prediga al técnico, sino también el nivel de urgencia o la posibilidad de resolución de incidencias de más de un técnico.
- **Mejora del control de SLA**: Incorporar mecanismos de supervisión y seguimiento del cumplimiento de los plazos asignados a los tickets, mediante alertas automáticas, generación de métricas y análisis de tiempos de resolución, alineando así el sistema con los estándares avanzados de ITIL.

Estos cambios y mejoras permitirían llevar el sistema a un nivel más avanzado, mejorando el rendimiento y eficacia del sistema junto con el servicio ofrecido al cliente.

9. Bibliografía

Referencias

[1] Almacenamiento personal en la nube: Microsoft OneDrive.

[2] Archivo EML: Cómo abrirlo y consejos para utilizarlo.

[3] Celery - Distributed Task Queue — Celery 5.5.3 documentation.

[4] Clasificación: Exactitud, recuperación, precisión y métricas relacionadas | Machine Learning.

[5] Empezar a recibir tickets — documentación de Odoo - 16.0.

[6] Git.

[7] Internet Message Access Protocol (IMAP). Section: Computer Networks.

[8] ITIL | IT Service Management | Axelos.

[9] Metodologías de desarrollo de software: ¿qué son?

[10] Open Source Community.

[11] Overleaf, Editor de LaTeX online.

[12] An Overview of Batch Processing & Multiprocessing in Odoo 16.

[13] PyPI · El Índice de paquetes de Python.

[14] RabbitMQ tutorial - "Hello world!"| RabbitMQ.

[15] Redis Pub/Sub. Section: develop.

[16] StarUML.

[17] threading — Thread-based parallelism.

[18] Understanding TF-IDF (Term Frequency-Inverse Document Frequency) - GeeksforGeeks.

[19] Understanding the Confusion Matrix in Machine Learning. Section: GBlog.

[20] Visual Studio Code - Code Editing. Redefined.

[21] What is dynamic link library (DLL)?

[22] Zendesk Suite y Odoo.

[23] ¿Qué es RAG?: explicación de la IA de generación aumentada por recuperación, AWS.

[24] ¿Qué es un Archivo PST? Todo Lo Que Debes Saber.

[25] Python Flask: pros and cons, November 2019.

[26] Top-N Accuracy Metrics | Baeldung on Computer Science, August 2020.

[27] Modularidad y escalabilidad en Odoo ¿Qué son estos conceptos? ¿Qué ventajas aportan para sui empresa?, May 2024.

[28] Odoo el ERP de Código Abierto (Open Source), January 2024.

[29] ¿Qué es un ERP?, June 2024.

[30] Docker: Accelerated Container Application Development, April 2025.

[31] The G2 on MBOX File Converter Tool, June 2025.

[32] IA para Gestión de Incidentes: casos de uso y ventajas, May 2025.

[33] libyal/libpff, June 2025. original-date: 2014-10-02T20:21:52Z.

[34] Welcome to Python.org, June 2025.

[35] Atlassian. Jira service management documentation, 2024.

[36] Valentin CrettazOn. How to set up vector search in Elasticsearch, February 2025.

[37] EasyVista. Herramientas y técnicas esenciales para una gestión eficiente de incidentes de ti, 2021.

[38] PCMag Editors. The best help desk software for 2024. *PCMag*, 2024.

[39] Freshworks. Freshservice itsm features, 2024.

[40] Anushtha Jain and Visure Solutions. Requisitos funcionales y no funcionales (con ejemplos), May 2025.

[41] Amal Latrache, El Habib Nfaoui, and Jaouad Boumhidi. Multi agent based incident management system according to itil. In *2015 Intelligent Systems and Computer Vision (ISCV)*, pages 1–7, 2015.

[42] Byung-Yun Lee and Gil-Haeng Lee. Service oriented architecture for sla management system. In *The 9th International Conference on Advanced Communication Technology*, volume 2, pages 1415–1418, 2007.

[43] mattwojo. Documentación del Subsistema de Windows para Linux.

[44] Odoo S.A. Odoo helpdesk module documentation, 2024.

[45] OTRS AG. Otrs help desk software overview, 2024.

[46] Real Python. What Is the Python Global Interpreter Lock (GIL)? – Real Python.

[47] Shipra Saxena. What are Categorical Data Encoding Methods | Binary Encoding, August 2020.

[48] Sosmatic. La evolución del help desk: de la resolución de incidencias a la experiencia del usuario, 2022.

[49] SysAid. The 30-year evolution of the it help desk – and where it's going next, 2022.

[50] TOPdesk. The history of the help desk, 2023.

[51] Ruiyu Wang. Revisiting GloVe, Word2Vec and BERT: On the Homogeneity of Word Vectors.

[52] Ángel Zavala Quiñones. Aprovechando la historia de itil para una gestión moderna de ti, 2020.

[53] Zendesk. Zendesk for service teams, 2024.

[54] Zoho. Software de seguimiento de incidencias, 2025.

A. Apéndice

A.1. Acrónimos

TI: Tecnologías de la Información

IT: Information Technology

ERP: Enterprise Resource Planning

ITIL: Information Technology Infrastructure Library

SAT: Servicio de Asistencia Técnica

SLA: Service Level Agreement

API: Application Programming Interface

IMAP: Internet Message Access Protocol

NLP: Natural Language Processing

SSL: Secure Sockets Layer

TLS: Transport Layer Security

IA: Inteligencia Artificial

AI: Artificial Intelligence

ITSM: IT Service Management

CRM: Customer Relationship Management

HTML: HyperText Markup Language

MIME: Multipurpose Internet Mail Extensions

HTTP: Hypertext Transfer Protocol

GIL: Global Interpreter Lock

WSL: Windows Subsystem for Linux

VPN: Virtual Private Network

JSON: JavaScript Object Notation

RPC: Remote Procedure Call

UTF-8: 8-bit Unicode Transformation Format

URL: Uniform Resource Locator

CSV: Comma Separated Values

I/O: Input/Output

IDLE: Integrated Development and Learning Environment

OST: Offline Storage Table

MAPI: Messaging Application Programming Interface

PST: Personal Storage Table

DLL: Dynamic-Link Library

COM: Component Object Model

TD-IDF: Term Frequency-Inverse Document Frequency

RAG: Retrieval-Augmented Generation

BERT: Bidirectional Encoder Representations from Transformers

ML: Machine Learning

LLM: Large Language Model

HNSW: Hierarchical Navigable Small World

ANN: Artificial Neural Network

A.2. Flujos de Procesamiento

A.2.1. Flujo de procesamiento del sistema general

A continuación se detalla el flujo general del sistema propuesto, desde la recepción de correos hasta la creación y gestión de tickets.

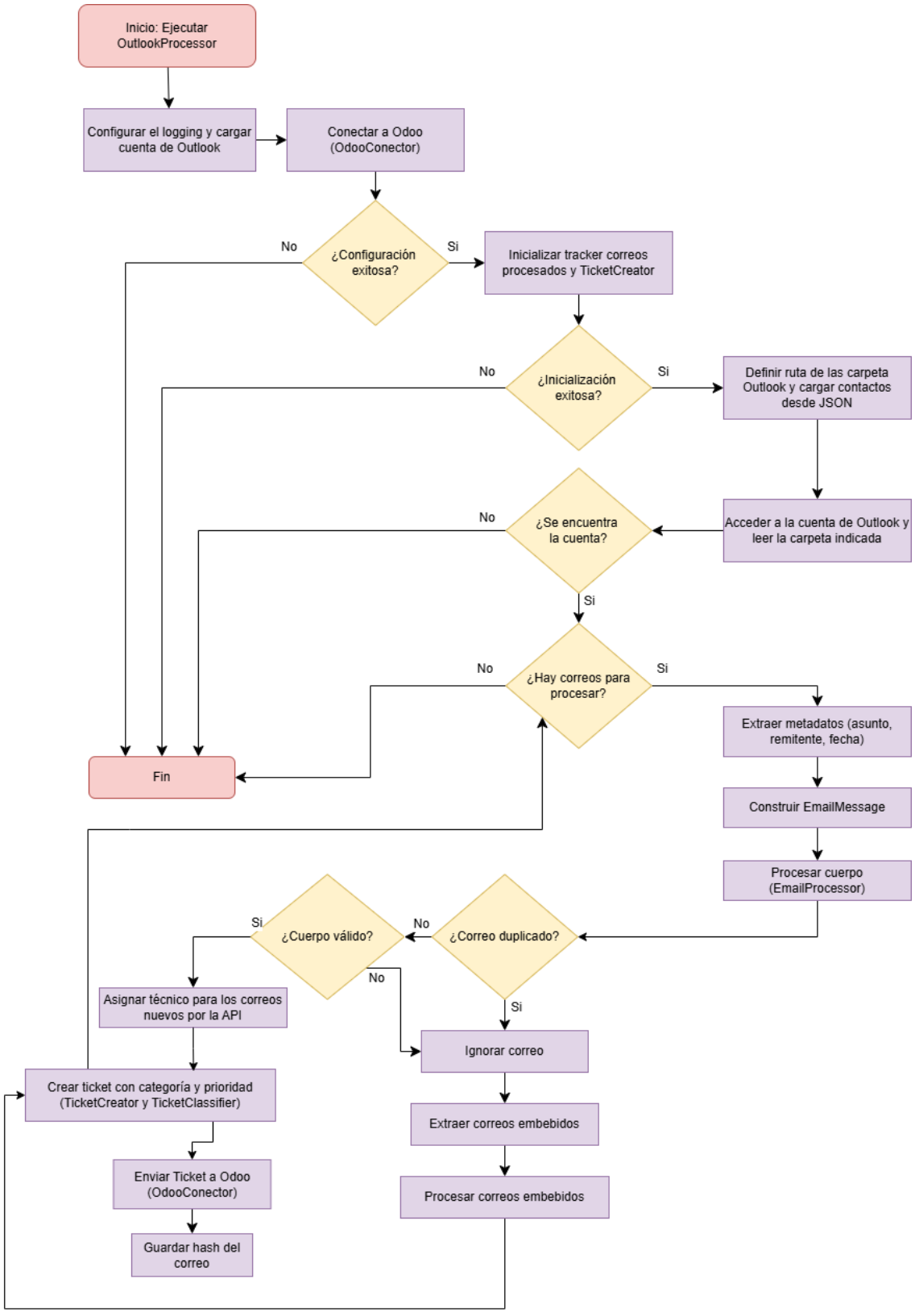

Figura 7: Flujo de procesamiento del sistema general (Apéndice)

A.2.2. Flujo de procesamiento del sistema de recepción de correos entrantes

Este flujo muestra el procesamiento que realiza el sistema al recibir nuevos correos electrónicos y transformarlos en tickets.

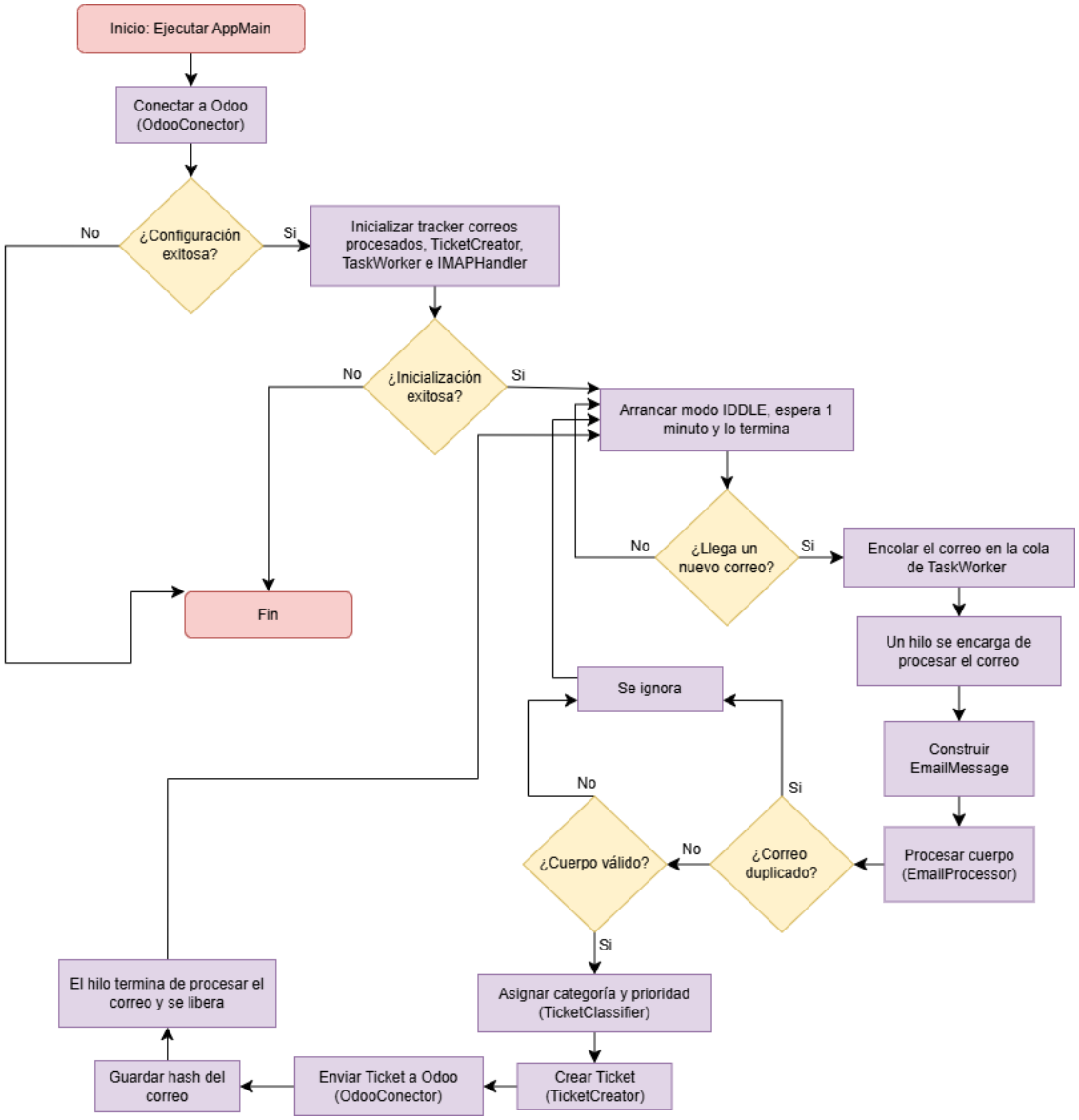

Figura 8: Flujo de procesamiento del sistema de recepción de correos entrantes (Apéndice)

A.2.3. Flujo de procesamiento de la creación de la base de datos históricos

El siguiente diagrama muestra el flujo correspondiente a la creación de la base de datos de correos históricos y su transformación en tickets para la base de datos de Odoo.

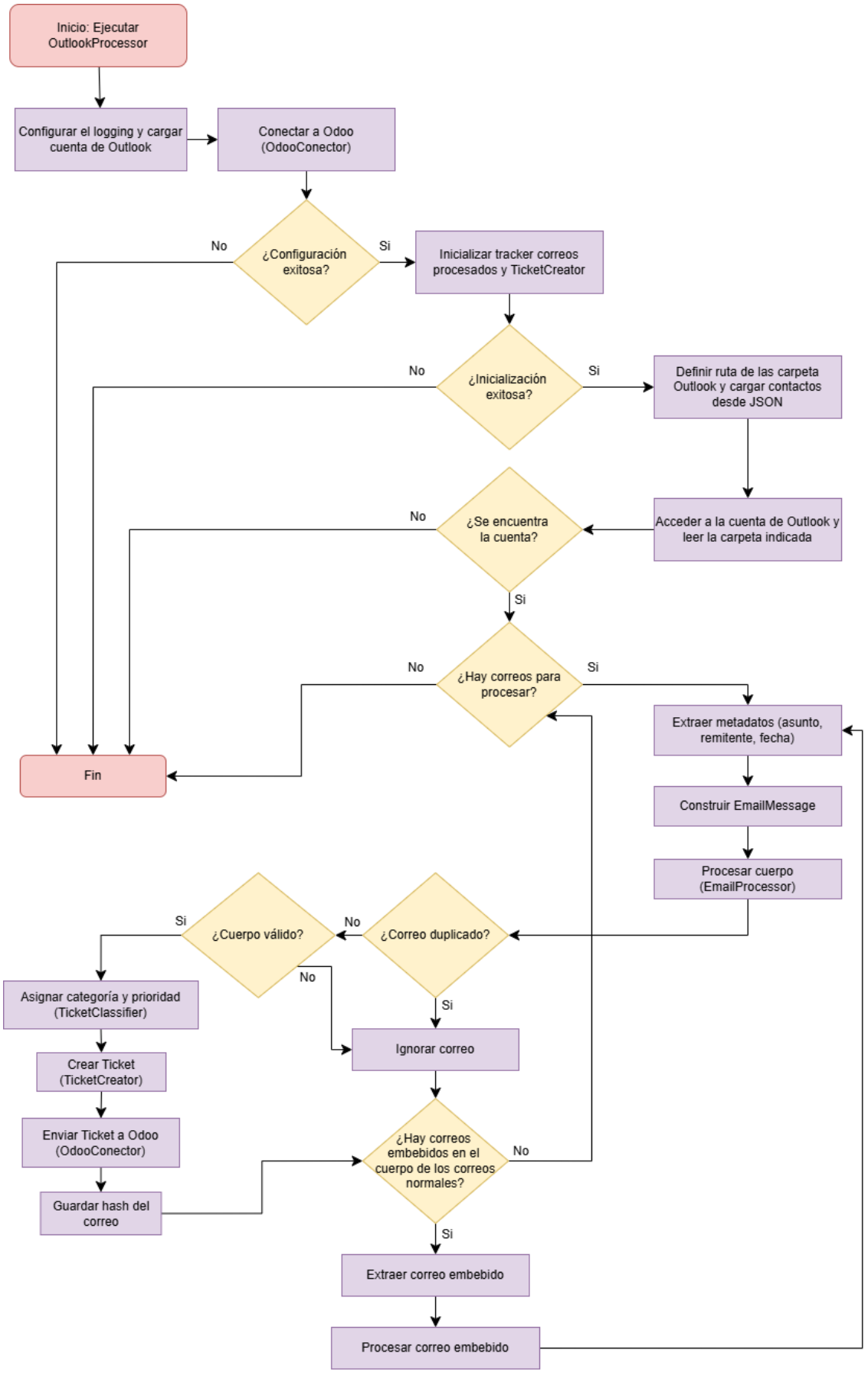

Figura 9: Flujo de procesamiento del sistema de creación de la base de datos histórico (Apéndice)

A.3. Matrices de Confusión

La siguiente matriz muestra los resultados de esta métrica para la técnica de texto limpio sin aplicar NLP.

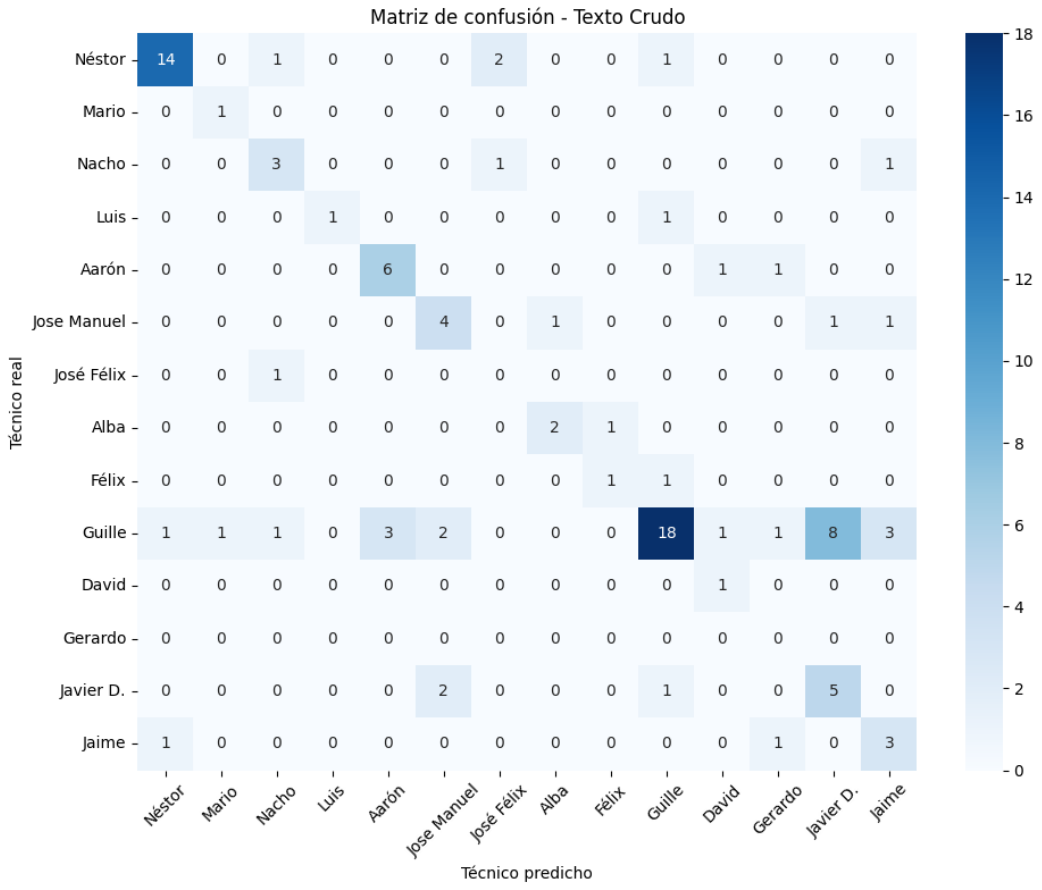

Figura 10: Matriz de Confusión para texto limpio sin NLP (Apéndice)

La siguiente matriz muestra los resultados de esta métrica para la técnica de texto aplicando NLP.

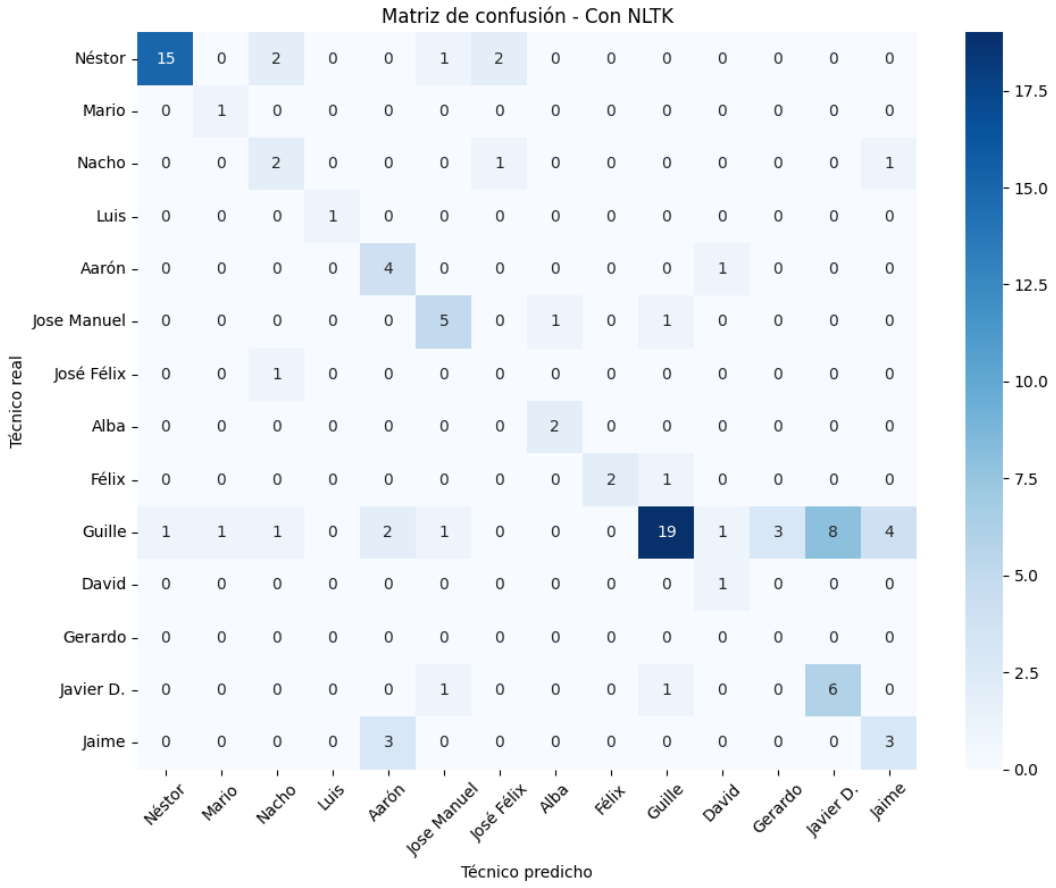

Figura 11: Matriz de Confusión para texto con NLP usando tokenización + stopwords (Apéndice)

La siguiente matriz muestra los resultados de esta métrica para la técnica de lematización.

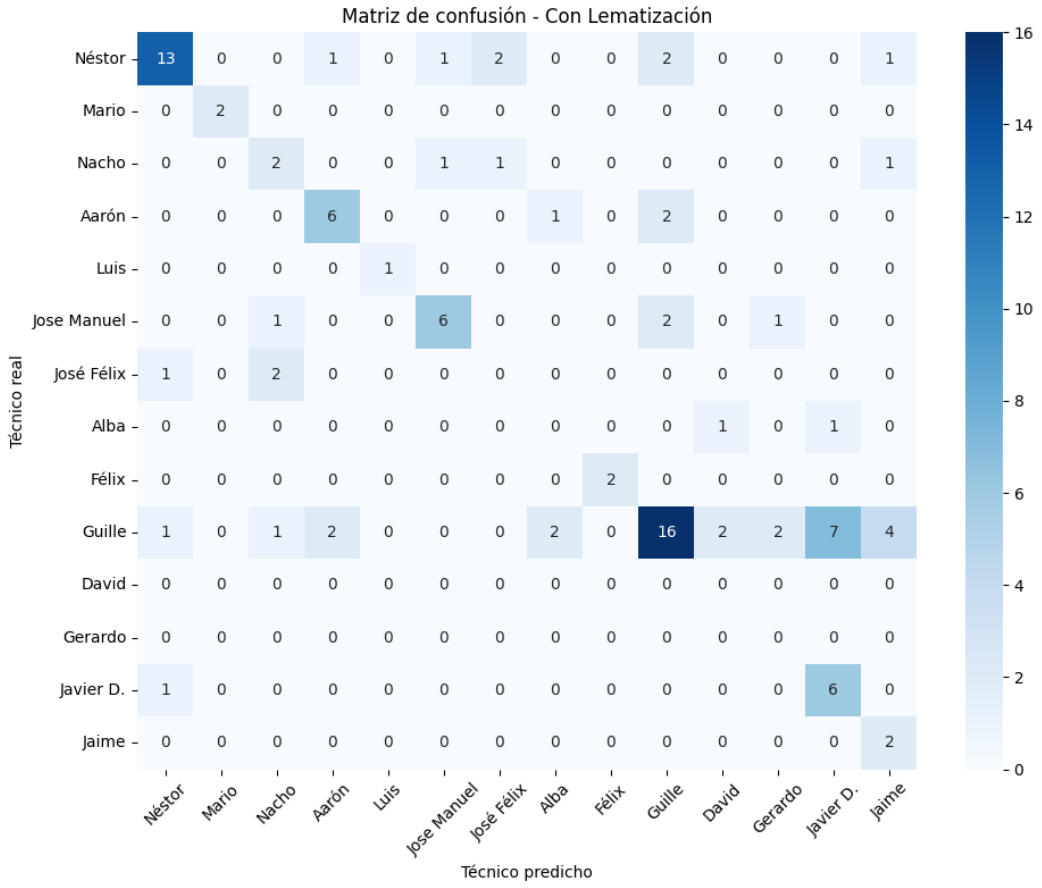

Figura 12: Matriz de Confusión con lematización (Apéndice)

La siguiente matriz muestra los resultados de esta métrica para la técnica de stemming.

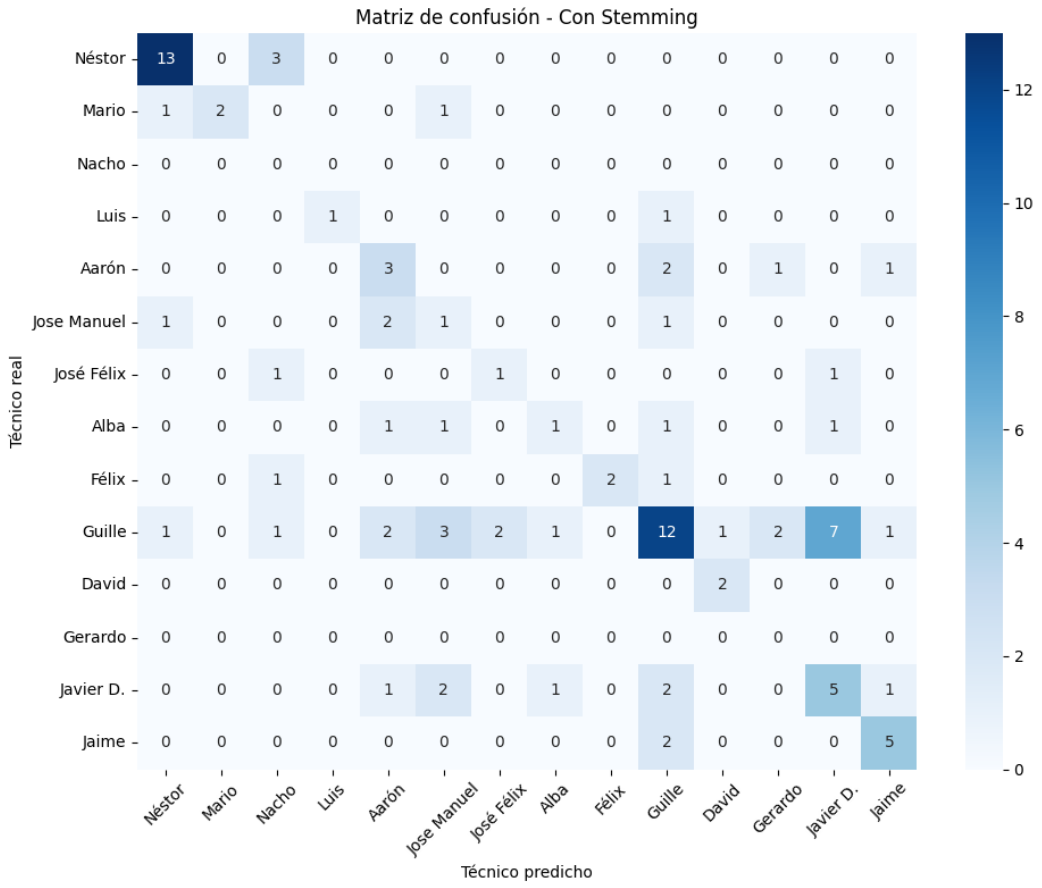

Figura 13: Matriz de Confusión con stemming (Apéndice)

A.4. Manual de Usuario

A.4.1. Manual para montar y acceder al entorno de Odoo

A continuación, se explican los pasos para montar el entorno de Odoo:

1. Abrir una terminal en la carpeta donde se encuentre Odoo.
2. Ejecutar el siguiente comando para iniciar los servicios en segundo plano: `docker-compose up -d`
3. Docker descargará las imágenes necesarias y montará los contenedores definidos.

Una vez que los contenedores están en funcionamiento, se puede acceder a la instancia de Odoo desde cualquier navegador web introduciendo la siguiente dirección: *"http://localhost:8069"*

Desde esta url se puede crear la base de datos inicial, iniciando sesión con las credenciales dadas.

Después, según inicias sesión, te encuentras con esta ventana de inicio de Odoo:

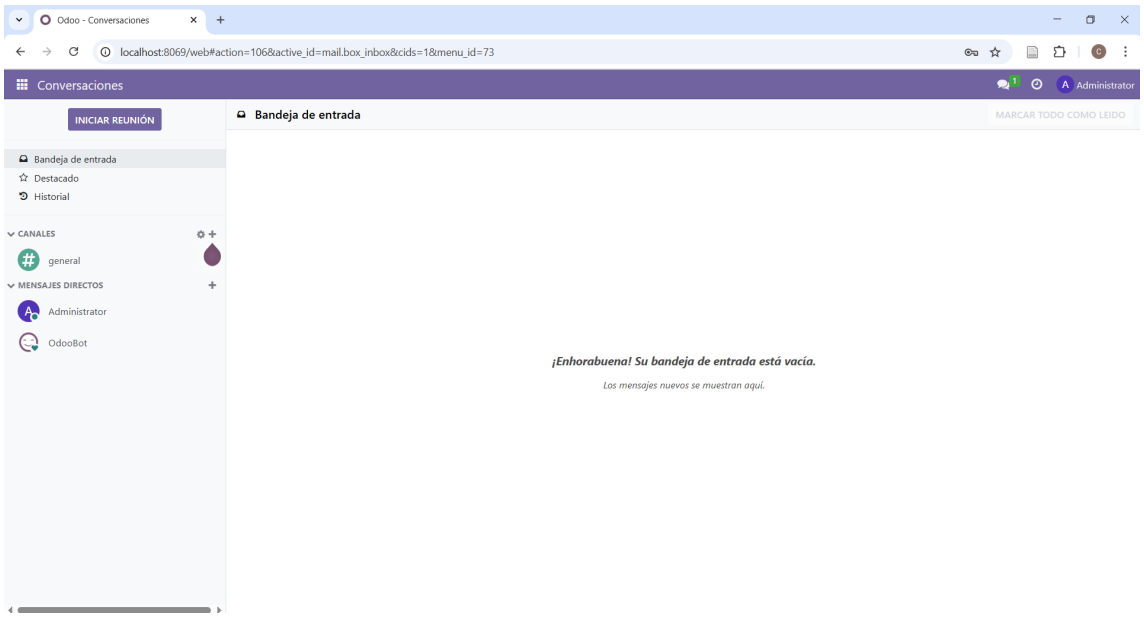

Figura 14: Pantalla de inicio tras acceder a la interfaz web de Odoo (Apéndice)

Después, habría que instalar el módulo Helpdesk (que se encuentra disponible gracias al volumen `./helpdesk:/mnt/extra-addons`):

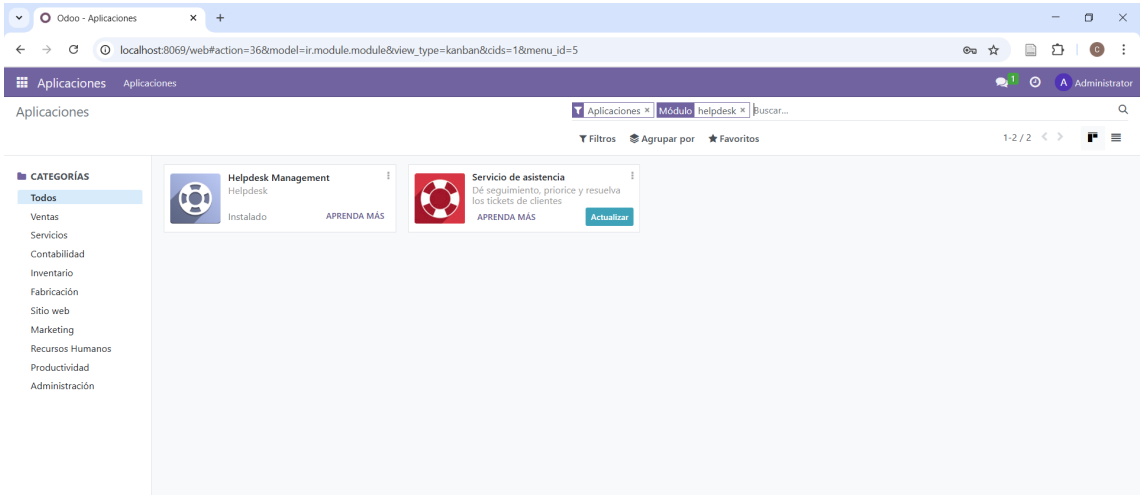

Figura 15: Pantalla de instalación del módulo Helpdesk (Apéndice)

Y una vez que lo has instalado, aparece la opción de servicio de asistencia en el menú. Se selecciona para poder trabajar con el sistema de tickets:

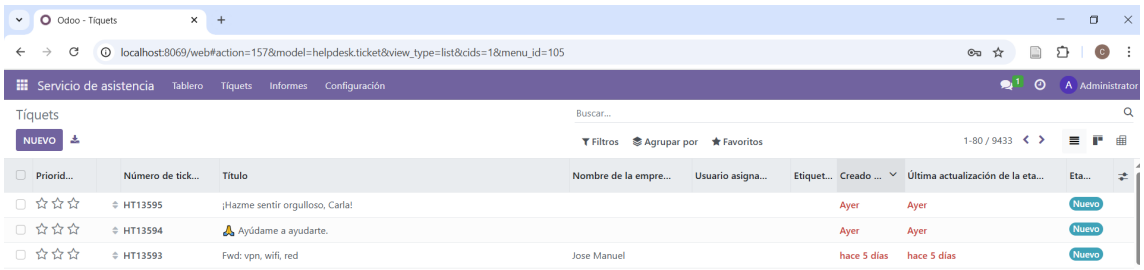

Figura 16: Pantalla del servicio de asistencia Helpdesk (Apéndice)

Para disponer de ajustes personalizados como ver que variable se asigna a cada campo del ticket y así poder subir los campos desde scripts en Python, se debe activar el modo desarrollador desde el menú: Ajustes → Opciones Generales → Activar modo desarrollador.

A.4.2. Manual para ejecutar el script que sube base de datos de Odoo a la API

Para poder ejecutar el script para subir la base de datos de Odoo a la API y comprobar su funcionamiento, hay que contar con dos requisitos: tener Odoo(versión 16), acceso a la API y disponer del módulo `helpdesk_mgmt` y la instalación de la librería `PyMuPDF`.

Esto se realizó creando una carpeta en WSL con los archivos (main.py y config.json). Se creó un `README.md` que indica como ejecutarlo.

Tras ejecutarlo, los tickets se iban indexando como se ve en la siguiente imagen:

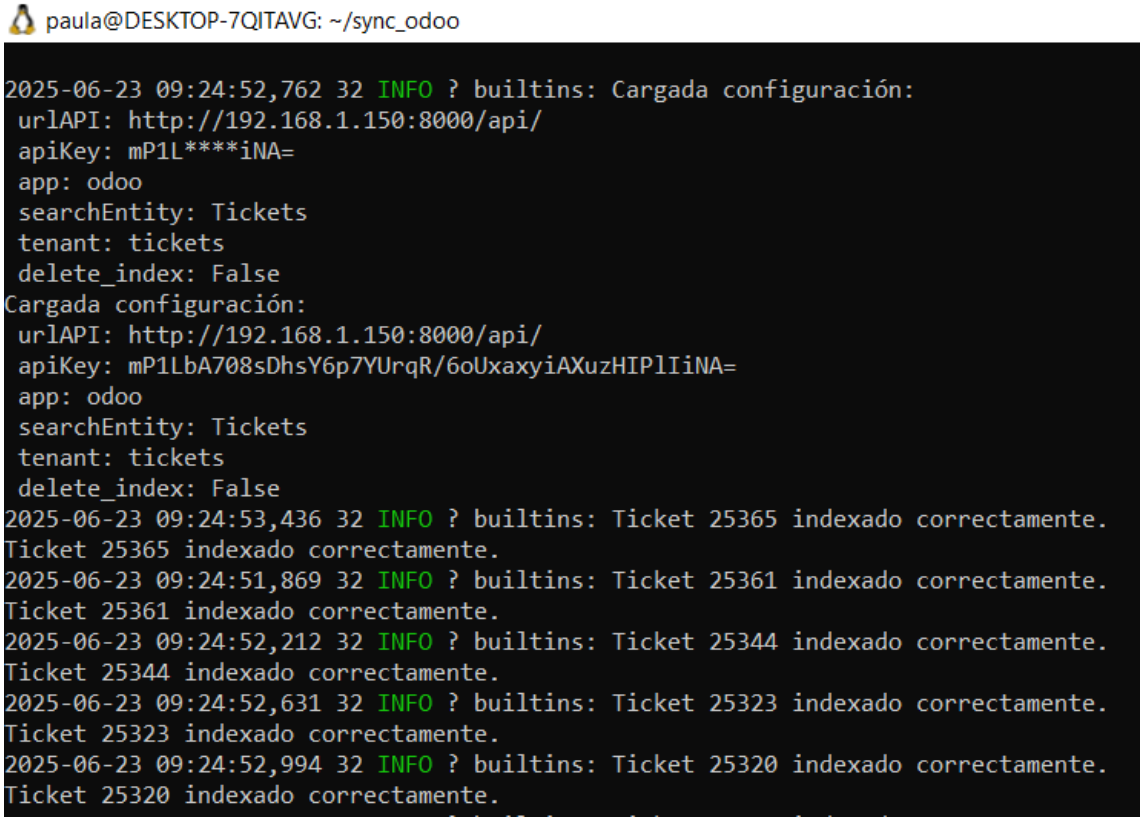

Figura 17: Indexación de tickets de Odoo en la API (Apéndice)

A.4.3. Manual para ejecutar y probar el script que extrae los técnicos y documentos devueltos por la API

Al final del archivo de `TecnicoPredictor`, se creó un ejemplo de uso creando un método main para su ejecución. Se crea un objeto de la clase y un texto sin información confidencial pero con contenido similar respecto a algunos correos históricos, y se imprime el técnico recomendado.

```
if __name__ == "__main__":
    predictor = TecnicoPredictor("config.json")
    texto = "Buenos días, Haciendo pruebas referentes a lo de los almacenes de
        certificación, he visto lo siguiente: Al filtrar en Consulta pedido
        venta, solo con el almacén de certificación, nos sale un único pedido,
        que es de agosto, cuando hemos empezado con esto hace menos de un mes:"
    tecnico = predictor.predecir_tecnico(texto)
    print(f"\nTécnico sugerido: {tecnico}")
```

Listing 5: Ejecución del predictor con un ejemplo (Apéndice)

Para ejecutar este código en la WSL, basta con tener Python instalado y ejecutar el comando: `python3 TecnicoPredictor.py`. Se obtienen los siguientes resultados:

```
INFO:__main__:Votación entre técnicos: Counter({'Jaime': 3, 'Nacho': 2, 'Néstor': 2, 'Luis': 1, 'Alba': 1, 'Guille': 1})
INFO:__main__:Técnico predicho por la mayoría: Jaime

Técnico sugerido: Jaime
```

Figura 18: Técnicos predichos por la API (Apéndice)

Tras ver que la clase hace su función, se comentó el método `main` junto con las instrucciones `print` y se procedió a llamar a la función de `predict_technician` dentro del método `create_ticket` de la clase `TicketCreator` para asignar un responsable de la incidencia. Este método se ejecuta solo si el ticket no tiene un técnico como es el caso de los nuevos tickets. En cambio, los correos históricos ya contienen ese campo definido, por lo que no es necesario predecirlo en esos casos.