UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Herramienta para la evaluación automática de entregas de programación con rúbricas y LLMs: IAGScore

Presentado por Pedro Antonio Abellaneda Canales
en Universidad de Burgos — 10 de junio de 2025
Tutor: Raúl Marticorena Sánchez

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Raúl Marticorena Sánchez, profesor del departamento de Ingeniería Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Pedro Antonio Abellaneda Canales, con DNI 23281531-B, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado **Herramienta para la evaluación automática de entregas de programación con rúbricas y LLMs: IAGScore**.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 10 de junio de 2025

Vº. Bº. del Tutor:

D. Raúl Marticorena Sánchez

Resumen

Actualmente, los avances relacionados con la inteligencia artificial, y en particular en la inteligencia artificial generativa, han supuesto una revolución en la forma en la que interactuamos con la tecnología, destacando en este sentido los modelos de lenguaje a gran escala (*LLM*). La irrupción de estas tecnologías ha supuesto un punto de inflexión para la automatización de tareas complejas, generación de contenido, de código o búsqueda de información.

En este contexto, este Trabajo Fin de Grado propone el desarrollo de una herramienta basada en el uso de modelos de lenguaje a gran escala para optimizar la evaluación de tareas de programación realizadas por el alumnado.

La propuesta consiste en ofrecer al profesorado herramientas para crear *prompts* y cargar rúbricas personalizadas en formato *Markdown*. Luego, estas instrucciones, junto con las tareas seleccionadas, se envían a un modelo de lenguaje que se encarga de evaluar automáticamente.

El objetivo de este proyecto no es sustituir a los docentes, sino implementar una aproximación a la evaluación automática de tareas de programación utilizando modelos de lenguaje a gran escala.

Descriptores

Corrección de tareas, rúbricas, prompts, Django, Python, PostgreSQL, Docker, Docker Compose, Celery, Redis, Ollama, Langchain, LLM, Markdown.

Abstract

Currently, advances in artificial intelligence, and in particular in generative artificial intelligence, have led to a revolution in the way we interact with technology, with large-scale language models (LLM) standing out in this regard. The irruption of these technologies has meant a turning point for the automation of complex tasks, content generation, code generation or information search.

In this context, this Final Degree Project proposes the development of a tool based on the use of large-scale language models to optimise the evaluation of programming tasks carried out by students.

The proposal consists of providing teachers with tools to create prompts and upload customised rubrics in Markdown format. These instructions, together with the selected tasks, are then sent to a language model that automatically evaluates them.

The aim of this project is not to replace teachers, but to implement an approach to the automatic evaluation of programming tasks using large-scale language models.

Keywords

Correction of tasks, rubrics, prompts, Django, Python, PostgreSQL, Docker, Docker Compose, Celery, Redis, Ollama, Langchain, LLM, Markdown.

Índice general

Índice general iii

Índice de figuras v

Índice de tablas vi

1. Introducción 1
1.1. Estructura de la memoria 2

2. Objetivos del proyecto 5
2.1. Objetivos generales 5
2.2. Objetivos técnicos 6
2.3. Objetivos personales 6

3. Conceptos teóricos 9
3.1. Modelos de lenguaje a gran escala (*LLM*) 9
3.2. Prompt engineering 11
3.3. Rúbricas 12
3.4. Corrección automática de tareas de programación 13

4. Técnicas y herramientas 17
4.1. Metodologías 17
4.2. Patrones de diseño 18
4.3. Comunicación 19
4.4. Control de versiones 20
4.5. Alojamiento del repositorio 20
4.6. Gestión del proyecto 21

4.7. Entorno de desarrollo integrado (IDE) 21
4.8. Integración continua 22
4.9. Calidad y consistencia del código 22
4.10. Cobertura de código 23
4.11. Framework web 24
4.12. Sistema gestor de bases de datos 24
4.13. Framework de persistencia 25
4.14. Procesamiento de tareas en segundo plano 25
4.15. Sistema de mensajería 25
4.16. Modelos de lenguaje locales 26
4.17. Documentación de la memoria 27
4.18. Documentación del código 27
4.19. Herramientas de traducción 28
4.20. Bibliotecas y librerías relevantes 28
4.21. Desarrollo web 29
4.22. Despliegue de la aplicación 30
4.23. Otras herramientas 31

5. Aspectos relevantes del desarrollo del proyecto 33
5.1. Inicio del proyecto 33
5.2. Gestión y metodología del proyecto 34
5.3. Formación 34
5.4. Desarrollo de la aplicación 36
5.5. Problemas, incidencias y soluciones 39

6. Trabajos relacionados 43

7. Conclusiones y Líneas de trabajo futuras 47
7.1. Conclusiones 47
7.2. Líneas de trabajo futuras 48

Bibliografía 51

Índice de figuras

3.1. Diagrama del modelo *Transformer* [51] 11
3.2. Diagrama ensamblado, envío a modelo y respuesta 14

Índice de tablas

6.1. Comparativa entre trabajos de corrección automática de código e IAGScore 45

1. Introducción

Evaluar las tareas de programación que entregan los estudiantes no solo implica verificar que el código funcione correctamente, sino también valorar aspectos como la calidad del diseño, la claridad, la eficiencia y el cumplimiento de criterios específicos establecidos por el docente.

Tradicionalmente, la corrección manual de estas tareas es una labor que demanda una gran inversión de tiempo y esfuerzo por parte del profesorado, especialmente en cursos con gran número de estudiantes. Esta situación limita la capacidad de proporcionar una retroalimentación rápida, detallada y personalizada, aspectos fundamentales para el aprendizaje eficaz.

En los últimos años, el avance en modelos de lenguaje a gran escala (LLM) ha abierto nuevas posibilidades en el campo de la docencia. Estos modelos tienen la capacidad de procesar y comprender tanto texto como código, lo que los convierte en herramientas prometedoras para asistir en la corrección de tareas, ofreciendo comentarios elaborados que van más allá de una simple validación funcional.

Este proyecto se centra en el desarrollo de una aplicación web que integra el potencial de los LLM para realizar evaluaciones automatizadas basadas en criterios definidos por el usuario, mediante la incorporación de rúbricas personalizadas y *prompts* específicos diseñados por los usuarios. La aplicación busca facilitar la labor docente, permitiendo una gestión eficiente de las correcciones y mejorando la calidad de la retroalimentación brindada a los estudiantes.

De esta forma, la herramienta propuesta contribuye a modernizar la enseñanza de la programación, promoviendo una evaluación más ágil, objetiva y enriquecedora, que favorezca tanto al profesorado como al alumnado, sin sustituir la necesaria supervisión humana en este proceso.

1.1. Estructura de la memoria

La memoria sigue la siguiente estructura:

- **Introducción:** breve descripción del problema a resolver y la solución propuesta.
- **Objetivos del proyecto:** exposición de los objetivos que persigue el proyecto.
- **Conceptos teóricos:** breve explicación de los conceptos teóricos.
- **Técnicas y herramientas:** listado de técnicas metodológicas y herramientas utilizadas.
- **Aspectos relevantes del desarrollo:** exposición de aspectos destacables que tuvieron lugar durante la realización del proyecto.
- **Trabajos relacionados:** pequeño resumen de los trabajos y proyectos ya realizados en el campo del proyecto en curso.
- **Conclusiones y líneas de trabajo futuras:** conclusiones obtenidas tras la realización del proyecto.

Junto a la memoria se proporcionan los siguientes anexos:

- **Plan del proyecto software:** planificación temporal y estudio de viabilidad del proyecto.
- **Especificación de requisitos del software:** se describe la fase de análisis; los objetivos generales, el catálogo de requisitos del sistema y la especificación de requisitos funcionales y no funcionales.
- **Especificación de diseño:** se describe la fase de diseño; el ámbito del software, el diseño de datos, el diseño procedimental y el diseño arquitectónico.
- **Manual del programador:** recoge los aspectos más relevantes relacionados con el código fuente.
- **Manual de usuario:** guía para el correcto manejo de la aplicación IAGScore.

Además, se incluyen los siguientes materiales complementarios accesibles a través de Internet:

- **Repositorio:** el código fuente y documentación están disponibles en el repositorio oficial de IAGScore en GitHub.
- **Gestión del proyecto:** realizada con Zube.
- **Vídeo de presentación:** presentación general del proyecto y sus objetivos.
- **Vídeo de demostración:** demostración funcional de la aplicación en uso.

2. Objetivos del proyecto

A continuación se detallan los objetivos que motivan la realización de este proyecto:

2.1. Objetivos generales

- Desarrollar un sistema *software* que permita automatizar la evaluación de tareas de programación.
- Investigar la posibilidad de hacer uso de diferentes modelos de lenguaje a gran escala para la evaluación de estas tareas.
- Comprobar la posibilidad de delegar totalmente en estos modelos para la realización de estas tareas.
- Implementar operaciones e interacción con la aplicación:
  - Permitir a los usuarios crear una cuenta.
  - Permitir a los usuarios diseñar o definir los *prompts* con los que interactuarán con los modelos.
  - Permitir al usuario importar documentos en formato *Markdown* con los contenidos de sus propias rúbricas.
  - Permitir a los usuarios configurar diferentes parámetros del modelo que evaluará las tareas.
  - Permitir a los usuarios ejecutar el modelo para que realice la evaluación.
  - Permitir a los usuarios consultar el resultado de la ejecución de la evaluación.

- Permitir a los usuarios consultar datos sobre el modelo utilizado.

2.2. Objetivos técnicos

- Crear una aplicación web utilizando *Python* y el *framework* **Django**.
- Utilizar la arquitectura MVT (Model - View - Template) de Django.
- Utilizar una base de datos **PostgreSQL** para el almacenamiento de datos.
- Utilizar **Ollama** para la gestión de los modelos de lenguaje en entornos locales.
- Utilizar **Celery** y **Redis** para la gestión de tareas asíncronas.
- Utilizar **Docker** para la creación de contenedores y facilitar el despliegue del proyecto.
- Hacer uso de **Sphinx** para la generación de la documentación del proyecto.
- Gestionar el proyecto utilizando la herramienta de gestión **Zube** para la planificación y seguimiento.
- Realizar test con una cobertura que garanticen la calidad del producto final.
- Hacer uso del sistema de control de versiones **GIT** distribuido junto con la plataforma **Github**.
- Aplicar la metodología ágil **Scrum** para el desarrollo del proyecto.
- Utilizar herramientas de integración continua (CI/CD), como las proporcionadas por *GitHub Actions*, para automatizar el despliegue del proyecto, la ejecución de pruebas, la medición de la cobertura de código con *Coverage*, y el control de calidad integrando *SonarCloud*.

2.3. Objetivos personales

El principal objetivo personal consiste en desarrollar una herramienta funcional, útil y escalable que no solo cumpla con los requisitos establecidos en la propuesta inicial, sino que también sea capaz de adaptarse a diferentes

contextos educativos y evolucionar con facilidad. Se busca que el sistema tenga una aplicación práctica real, ofreciendo una solución eficiente y sostenible que pueda mantenerse y mejorarse en el tiempo.

Además, se plantean los siguientes objetivos específicos orientados al uso y aprendizaje de tecnologías clave durante el desarrollo del proyecto:

- Utilizar el *framework* **Django** para el desarrollo de aplicaciones web, configurándolo adecuadamente para trabajar con *PostgreSQL* como sistema de bases de datos.
- Aprender a ejecutar modelos *LLM* de forma local e integrarlos en una aplicación web.
- Investigar el comportamiento de los modelos con diferentes configuraciones para evaluar la viabilidad del proyecto.
- Conocer y utilizar **Docker** para la creación de contenedores, facilitando el despliegue del proyecto mediante **Docker Compose** como herramienta de orquestación.
- Profundizar en el uso de herramientas de control de versiones como **GIT** y plataformas como **GitHub**, empleando metodologías como *GitFlow*.
- Aprender a utilizar herramientas de integración continua como **GitHub Actions**.
- Familiarizarse con herramientas de gestión de proyectos como **Zube**.
- Aprender a utilizar herramientas de análisis de calidad de código como **SonarCloud**, integrándolas en el ciclo de desarrollo.
- Aprender a generar documentación técnica automática mediante herramientas como **Sphinx**.

3. Conceptos teóricos

Este capítulo presenta los fundamentos teóricos necesarios para comprender los principales componentes que sustentan la aplicación desarrollada. Se abordan los conceptos de modelos de lenguaje a gran escala (*LLM*), técnicas de ingeniería de *prompts*, el uso de rúbricas como herramienta para la evaluación objetiva y la integración de mecanismos automáticos para la corrección asistida de tareas.

3.1. Modelos de lenguaje a gran escala (*LLM*)

Los modelos de lenguaje a gran escala (*LLM*, por sus siglas en inglés) son sistemas de inteligencia artificial diseñados para modelar y procesar el lenguaje humano. Estos modelos forman parte del campo del *procesamiento de lenguaje natural (PLN)*[54], una rama de la inteligencia artificial que estudia cómo las máquinas pueden comprender, interpretar y generar texto o habla en el lenguaje humano.

Se denominan "grandes" o "a gran escala" porque están entrenados con volúmenes masivos de texto, y tienen una arquitectura muy grande capaz de capturar matices lingüísticos complejos que los modelos más pequeños no pueden[59].

Arquitectura *Transformer*

La arquitectura *Transformer* [17] es la base de la mayoría de los modelos de lenguaje actuales (*LLMs*). A continuación, se describe de forma simplificada su funcionamiento:

- **Entrada (de palabras a números)**: el modelo toma una frase y convierte cada palabra en una representación numérica comprensible para la máquina. Este proceso se conoce como *embedding*.
- **Mecanismo de atención**: el *Transformer* identifica qué partes del texto son más relevantes mediante la atención (*self-attention*). Esta técnica permite al modelo analizar todas las palabras al mismo tiempo y establecer qué relaciones son más significativas.
- **Capas de procesamiento**: la información se pasa por múltiples capas de redes neuronales, que realizan cálculos para refinar la comprensión del texto.
- **Salida (de números a texto)**: finalmente, el modelo convierte los resultados numéricos en palabras para generar una respuesta o texto coherente.

La arquitectura se compone de dos bloques principales:

- **Codificador (*encoder*)**: procesa y comprende el texto de entrada. Está formado por varias capas que combinan atención y redes neuronales para capturar el significado de la secuencia.
- **Decodificador (*decoder*)**: genera el texto de salida utilizando la información del codificador, palabra por palabra, teniendo en cuenta tanto el contexto como el contenido generado previamente.

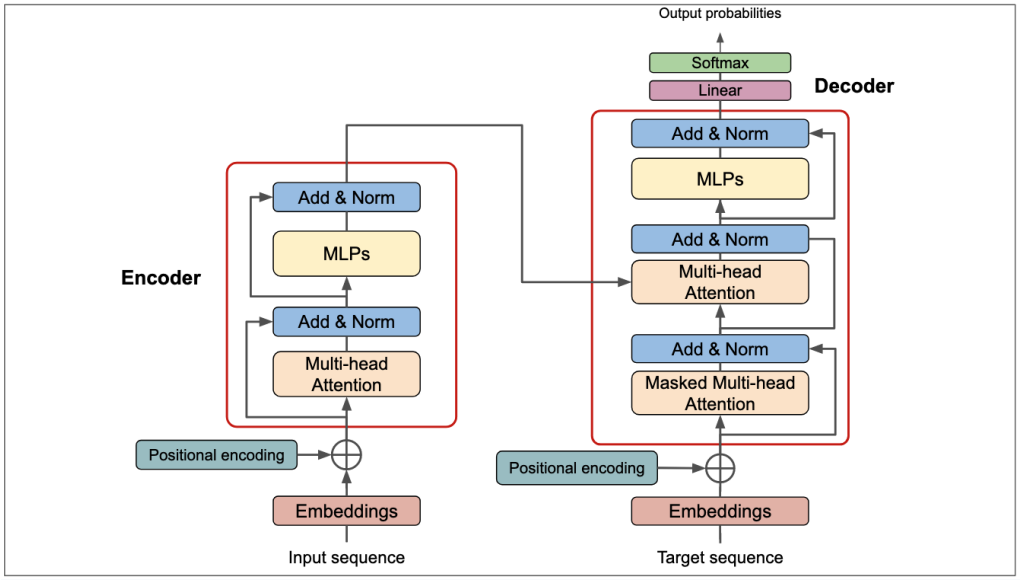

Figura 3.1: Diagrama del modelo *Transformer* [51]

Aplicación en IAGScore

En este contexto, los *LLMs* no solo representan una tecnología de vanguardia, sino que constituyen el núcleo funcional de la herramienta **IAGScore**. Gracias a su capacidad para interpretar instrucciones en lenguaje natural y analizar fragmentos de código, los *LLMs* permiten automatizar tareas que tradicionalmente requerían intervención humana, como la evaluación detallada de ejercicios de programación y la generación de retroalimentación contextualizada.

Una característica especialmente valiosa para el proyecto es la posibilidad de guiar el comportamiento del modelo mediante *prompts* bien diseñados, lo que permite adaptar las respuestas a rúbricas personalizadas o criterios específicos definidos por el profesorado.

3.2. Prompt engineering

El *prompt engineering* es una técnica dentro del campo del procesamiento de lenguaje natural que consiste en diseñar cuidadosamente las instrucciones que se proporcionan a un modelo de lenguaje (*LLM*) para obtener respuestas más precisas, relevantes o útiles.

Dado que los *LLM* no tienen comprensión real del contexto, el modo en que se les formula una entrada (*prompt*) influye significativamente en la calidad y precisión de la respuesta. Por ello, estructurar correctamente el mensaje de entrada se convierte en un factor clave para guiar el comportamiento del modelo.

Aunque en este proyecto no se ha aplicado de forma intensiva, se han realizado pruebas con distintas formulaciones de entrada para verificar cómo pequeños cambios en el *prompt* afectan la salida del modelo. Por ejemplo, el uso de frases más explícitas o el establecimiento claro del rol del modelo (e.g.,"Actúa como profesor universitario") puede mejorar la coherencia y adecuación de las respuestas al evaluar tareas [28].

Existen técnicas avanzadas dentro del *prompt engineering*, como:

- ***Zero-shot prompting***: el modelo genera una respuesta sin ejemplos previos, confiando solo en la instrucción.
- ***Few-shot prompting***: se incluyen uno o varios ejemplos dentro del *prompt* para que el modelo aprenda el formato o estilo deseado.
- ***Chain-of-thought prompting***: se induce al modelo a razonar paso a paso para tareas complejas.
- ***Role prompting***: se establece un rol específico para el modelo, como "profesor universitario".

Aunque estas estrategias no se han utilizado en profundidad en este trabajo, su conocimiento es relevante para posibles mejoras futuras del sistema, especialmente si se busca afinar el comportamiento del modelo en escenarios educativos específicos.

3.3. Rúbricas

Las rúbricas son instrumentos de evaluación que describen criterios específicos para valorar el desempeño de una tarea o actividad. Se componen generalmente de una serie de dimensiones o aspectos a evaluar, junto con niveles de desempeño definidos para cada una de ellas, que permiten asignar puntuaciones de forma objetiva y transparente.

En el ámbito educativo, las rúbricas son ampliamente utilizadas para evaluar tareas abiertas, como ensayos, proyectos, presentaciones o, en el caso

de este proyecto, ejercicios de programación. Su estructura ayuda tanto a los evaluadores como a los estudiantes, ya que clarifica las expectativas y facilita la retroalimentación formativa [60].

Una rúbrica típica incluye:

- **Criterios de evaluación**: aspectos concretos que se desean valorar (e.g., legibilidad del código, eficiencia, cumplimiento de requisitos).
- **Niveles de desempeño**: descripciones cualitativas (y en ocasiones cuantitativas) que definen grados de calidad o cumplimiento.
- **Puntuaciones**: valores numéricos o etiquetas asociadas a cada nivel.

En este proyecto, las rúbricas se utilizan como entrada esencial para el modelo de lenguaje. El usuario define una rúbrica en formato `markdown`, que posteriormente es añadida al *prompt* y enviada al *LLM*, junto con el código del estudiante. El modelo analiza el código y genera una evaluación basada en los criterios proporcionados, simulando el comportamiento de un evaluador humano.

Este enfoque permite automatizar evaluaciones personalizadas, intentando mantener un grado alto de coherencia con las intenciones pedagógicas del docente. Además, al permitir reestructurar y reutilizar rúbricas fácilmente, se favorece la escalabilidad y adaptabilidad del sistema a distintos contextos de enseñanza.

3.4. Corrección automática de tareas de programación

La corrección automática de tareas de programación es una técnica que busca evaluar de manera sistemática y eficiente el código fuente entregado por estudiantes o usuarios, mediante herramientas informáticas capaces de analizar y valorar distintos aspectos del mismo. Este proceso puede incluir verificación sintáctica, evaluación funcional, análisis estático, comparación con soluciones de referencia o, como en este proyecto, análisis mediante modelos de lenguaje a gran escala.

El enfoque adoptado se basa en el uso de modelos *LLM* para comprobar no solo si un programa funciona correctamente, sino que pueden emitir juicios cualitativos sobre aspectos relacionados con la rúbrica definida por el usuario. Estos aspectos pueden ser:

- Claridad y estilo del código.
- Cumplimiento de requisitos.
- Organización y estructura lógica.
- Correcto uso de estructuras del lenguaje de programación.
- Comentarios y documentación.

Para que el modelo realice esta evaluación, se le proporciona una entrada estructurada que combina:

1. Un *prompt* que contextualiza la tarea del modelo (e.g., "Eres un profesor que debe evaluar este código según los siguientes criterios...").
2. La rúbrica definida por el usuario, en formato estructurado.
3. El código fuente a evaluar.

Esta combinación se ensambla en una cadena de texto que es enviada al modelo a través de una tarea en segundo plano. El resultado es una evaluación textual generada por el *LLM*, que puede incluir puntuaciones, observaciones, comentarios generales y sugerencias de mejora.

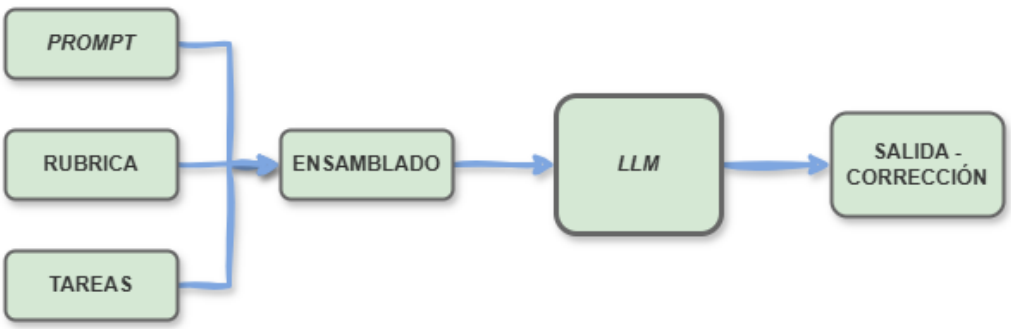

Figura 3.2: Diagrama ensamblado, envío a modelo y respuesta

Este tipo de corrección presenta ventajas significativas:

- **Escalabilidad**: se pueden corregir grandes volúmenes de tareas en poco tiempo.

- **Consistencia**: las evaluaciones tienden a ser coherentes entre tareas similares, al depender de un único sistema de criterios.
- **Flexibilidad**: permite adaptar el comportamiento del modelo mediante ajustes de parámetros y personalización de *prompts*.

Sin embargo, también existen limitaciones, como la falta de ejecución real del código (a menos que se combine con otros análisis) o la posibilidad de interpretaciones ambiguas por parte del modelo si el *prompt* no está bien diseñado. Por ello, se contempla esta herramienta como un apoyo a la evaluación docente, no necesariamente como un sustituto total.

4. Técnicas y herramientas

En este capítulo se describen las principales técnicas metodológicas y herramientas de desarrollo empleadas en el proyecto. Se presentan las opciones consideradas, destacando las características más relevantes y justificando las decisiones tomadas para su selección.

4.1. Metodologías

Scrum

Scrum [61] es un marco de trabajo ágil, líder en el desarrollo de software y la gestión de proyectos, que se distingue por su flexibilidad. Organiza el trabajo en *sprints*, iteraciones cortas de una a cuatro semanas, cada una entregando un incremento funcional del producto. Al final de cada *sprint*, se evalúa el resultado para detectar mejoras y ajustar prioridades según las necesidades del proyecto. Los requisitos, que pueden cambiar durante el desarrollo, se recogen en la pila del producto, mientras que las tareas específicas de cada iteración se detallan en la *pila del sprint*. Este enfoque incremental permite solapar fases del proyecto, optimizando tiempos y facilitando la adaptación a nuevos requerimientos. *Scrum* promueve la calidad a través del aprendizaje continuo del equipo, generando entregas tempranas de productos utilizables y asegurando una mejora constante alineada con las necesidades del usuario.

Kanban

Kanban [58] es un método ágil que se centra en la visualización del flujo de trabajo y la gestión del trabajo en curso. Utiliza un tablero representando

las tareas en una serie de tarjetas, permitiendo a los miembros del equipo ver el progreso y las prioridades del trabajo. *Kanban* se basa en principios como la limitación del trabajo en curso, la mejora continua y la adaptación a los cambios; es flexible y se puede aplicar a diferentes tipos de proyectos y equipos, lo que lo convierte en una herramienta valiosa para la gestión de proyectos en entornos ágiles.

Gitflow

Gitflow [15] es un modelo de ramificación para el uso de *Git* que define una estructura clara para gestionar el desarrollo de *software* en proyectos colaborativos. Organiza el trabajo en ramas específicas con roles definidos, facilitando la escalabilidad y la integración continua.

Las ramas principales en *Gitflow* son:

- **`main`**: contiene la versión estable del proyecto, lista para producción.
- **`develop`**: refleja la versión en desarrollo, integrando nuevas funcionalidades.
- **`feature/*`**: ramas temporales para desarrollar nuevas características.
- **`release/*`**: prepara una versión para producción, permitiendo ajustes menores.
- **`hotfix/*`**: corrige errores críticos en la rama `main`.

Gitflow asegura un flujo ordenado, minimizando conflictos y garantizando estabilidad en la rama `main`.

4.2. Patrones de diseño

Model-View-Template

El patrón de diseño *MVT* (*Model-View-Template*) es la arquitectura que sigue el *framework* Django para desarrollar aplicaciones web directamente relacionadas con el patrón *MVC*.

Los componentes detallados del patrón *MVT* son:

- **Model (Modelo)**: representa la capa de acceso a datos. Define la estructura de las tablas de la base de datos y permite interactuar con esos datos. En Django, los modelos se definen como clases que heredan de `django.db.models.Model`, y permiten realizar operaciones de consulta, inserción, actualización o borrado sin necesidad de escribir directamente *SQL*.

- **View (Vista)**: esta capa contiene la lógica de negocio de la aplicación. Las vistas reciben peticiones *HTTP*, procesan los datos necesarios y devuelven una respuesta adecuada.

- **Template (Plantilla)**: define la presentación de los datos, lo que ve el usuario. Django utiliza su propio motor de plantillas para combinar datos dinámicos con *HTML*.

Una diferencia clave entre el patrón *MVT* de Django y el clásico *MVC* es que el propio *framework* gestiona internamente la lógica de enrutamiento. Al recibir peticiones del usuario, se decide a través del sistema de *URLs* definido, qué vista debe ejecutarse, lo que elimina la necesidad de un controlador explícito. En este sentido, las vistas en Django cumplen una función similar al controlador en otros *frameworks*, ya que contienen la lógica que determina qué datos recuperar y cómo deben procesarse antes de enviarse a la plantilla correspondiente [5].

4.3. Comunicación

Para la comunicación con el tutor del proyecto se han utilizado las herramientas proporcionadas por la Universidad de Burgos. En particular, se ha empleado el correo electrónico institucional para el intercambio de información y documentación relevante, así como **Microsoft Teams** para la realización de reuniones periódicas de seguimiento.

Estas reuniones, organizadas al inicio y al final de cada sprint, se llevaron a cabo mediante video-llamadas y permitieron resolver dudas, revisar el progreso alcanzado y planificar las siguientes tareas. El uso de estas herramientas ha facilitado una comunicación fluida y constante durante todo el desarrollo del proyecto.

4.4. Control de versiones

En este proyecto, se ha optado por **Git** [22] debido a la experiencia previa adquirida en asignaturas del grado, en contraste con otras alternativas como *Apache Subversion* [21].

Git

Git es un sistema de control de versiones distribuido y una de las herramientas más extendidas en el mercado para este propósito. Entre sus características más destacadas se encuentran su eficiencia para manejar proyectos de gran tamaño, la facilidad para crear y fusionar ramas (*branches*), y su compatibilidad con diversas plataformas y servicios como *GitHub*, *GitLab* o *Bitbucket*.

4.5. Alojamiento del repositorio

Existen diversas posibilidades relacionadas con el alojamiento de repositorios:

- GitLab [30]
- BitBucket [1]
- GitHub [23]

Se ha seleccionado **GitHub** al estar familiarizado con la plataforma debido al uso de esta herramienta en múltiples asignaturas.

GitHub

GitHub es una plataforma en línea para alojar repositorios *Git*, que facilita la colaboración, el control de versiones y la gestión de proyectos de software. Entre sus características destacan la gestión de ramas y *pull requests*, el seguimiento de incidencias (*issues*), y la integración con herramientas de integración continua como *GitHub Actions*.

Durante el desarrollo de este trabajo, se ha utilizado *GitHub* no solo como espacio de almacenamiento del código fuente, sino también como herramienta de seguimiento del progreso y documentación del proyecto. La posibilidad de mantener un historial detallado de los cambios ha resultado clave para asegurar la trazabilidad y la calidad del desarrollo.

4.6. Gestión del proyecto

Hay varias opciones disponibles para la gestión de proyectos, como:

- Jira [2]
- Trello [3]
- Zube [31]

Se ha seleccionado para la gestión de este proyecto la herramienta **Zube** por su integración con *GitHub*.

Zube

Zube es una herramienta de gestión de proyectos orientada a equipos de desarrollo de software. Su principal ventaja radica en la integración nativa con *GitHub*, lo que permite vincular *issues*, *commits* y *pull requests* directamente con las tareas del tablero *Kanban*. Esta integración facilita el seguimiento del estado del proyecto en tiempo real y mejora la colaboración entre los miembros del equipo.

Durante el desarrollo de este proyecto, se ha utilizado *Zube* para organizar las tareas en distintas columnas (*Ready*, *In Progress*, *In Review*, *Done*), asignar prioridades y establecer hitos. Esto ha permitido una planificación ágil y un control más eficiente del avance del proyecto.

4.7. Entorno de desarrollo integrado (IDE)

Se han valorado las siguientes opciones:

- Spyder IDE [14]
- Visual Estudio Code [39]

De entre estas opciones valoradas se ha seleccionado **Visual Studio Code**

Visual Studio Code

Visual Studio Code es un editor de código desarrollado por *Microsoft*, ampliamente utilizado por su versatilidad, ligereza y amplia comunidad de extensiones. Permite trabajar con múltiples lenguajes de programación y se integra fácilmente con herramientas de control de versiones como *Git*.

En el desarrollo de este proyecto, se ha utilizado *Visual Studio Code* por su compatibilidad con *Python*, su integración con *GitHub*, y el soporte de extensiones útiles como *linters*.

4.8. Integración continua

Se ha seleccionado **GitHub Actions** por su integración nativa con *GitHub*, la plataforma elegida para alojar y gestionar el repositorio del proyecto.

GitHub Actions

GitHub Actions es una funcionalidad integrada en *GitHub* que permite automatizar flujos de trabajo mediante la ejecución de acciones definidas en archivos de configuración. Esta herramienta facilita la implementación de integración continua (*CI*) y entrega continua (*CD*), permitiendo ejecutar tareas como pruebas, validación de código o despliegue de forma automática.

En este proyecto se ha utilizado *GitHub Actions* para implementar un flujo de integración continua básico. Cada vez que se realiza un *push* o una *pull request* al repositorio, se activa automáticamente un flujo de trabajo que ejecuta las pruebas del proyecto y verifica que el código cumple con los requisitos establecidos. Esto permite detectar errores de manera temprana y mantener la calidad del código a lo largo del desarrollo.

4.9. Calidad y consistencia del código

Existen múltiples herramientas para garantizar la calidad, seguridad y consistencia del código. Algunas de las más utilizadas en proyectos desarrollados en *Python* son:

- **Pylint**: herramienta de análisis estático que evalúa la calidad del código en base a una serie de reglas y convenciones del lenguaje.

- **Flake8**: combina varias herramientas para comprobar el estilo y errores comunes en el código.
- **SonarCloud**: plataforma en línea para análisis continuo de calidad del código, integrada con repositorios en la nube.
- **Djlint**: herramienta específica para analizar y formatear plantillas *Django*, útil para mantener una estructura coherente en los archivos *HTML*.
- **Bandit**: escáner de seguridad que analiza código *Python* en busca de patrones potencialmente vulnerables.

Durante el desarrollo del proyecto se ha empleado **Pylint**, **Djlint** y **Bandit** en local para garantizar la corrección sintáctica, el cumplimiento de buenas prácticas y la detección temprana de posibles vulnerabilidades. Además, se ha integrado **SonarCloud** en el flujo de trabajo mediante **GitHub Actions**, lo que ha permitido realizar análisis automáticos del código en cada *push* o *pull request*, contribuyendo a mantener un estándar de calidad a lo largo del ciclo de vida del desarrollo.

4.10. Cobertura de código

La cobertura de código es una métrica que indica el porcentaje del código fuente que ha sido ejecutado durante la ejecución de las pruebas. Esta información permite identificar qué partes del código no están siendo verificadas por los *tests*, lo que contribuye a mejorar la calidad y fiabilidad del software.

En este proyecto se ha utilizado la herramienta **Coverage** para medir la cobertura de las pruebas unitarias. Inicialmente, se ejecutó localmente para generar informes detallados que muestran las líneas de código cubiertas y no cubiertas.

Posteriormente, **Coverage** se integró en el flujo de trabajo de **GitHub Actions**, lo que permitió la ejecución automática de las pruebas y la generación de informes de cobertura en cada *push* o *pull request*. Esta automatización ha facilitado el seguimiento continuo del nivel de cobertura y ha contribuido a mantener un estándar de calidad constante durante el desarrollo.

4.11. Framework web

Para el desarrollo del proyecto se han valorado distintos *frameworks web* en *Python*, entre los cuales destacan:

- **Flask**: un *microframework* ligero y flexible, que permite construir aplicaciones web de forma modular. [49]
- **Django**: un *framework* de alto nivel que incluye por defecto un conjunto completo de herramientas para el desarrollo rápido de aplicaciones web, como sistema de autenticación, panel de administración, ORM y enrutamiento. [20]

Se ha seleccionado **Django** como *framework* para este proyecto debido a su enfoque en el desarrollo rápido, su estructura organizada y su integración con bases de datos a través de su *ORM*. Además, proporciona una gran cantidad de funcionalidades listas para usar, lo que ha permitido acelerar el proceso de desarrollo y mantener una arquitectura coherente a lo largo del proyecto.

4.12. Sistema gestor de bases de datos

En el desarrollo de aplicaciones web se valoraron dos sistemas gestores de bases de datos (*SGBD*) ampliamente utilizados:

- **SQLite**: un motor de base de datos ligero y sencillo, integrado en la mayoría de entornos de desarrollo, ideal para aplicaciones pequeñas o prototipos. [7]
- **PostgreSQL**: un sistema gestor de bases de datos relacional avanzado, con soporte para características complejas como transacciones, concurrencia, y extensiones, recomendado para aplicaciones de mayor escala y producción. [25]

Se ha seleccionado **PostgreSQL** para este proyecto debido a su robustez, escalabilidad y compatibilidad con las necesidades del sistema. Además, es un *SGBD* con el que se ha trabajado durante el grado, lo que facilita su uso y administración en el contexto del proyecto.

4.13. Framework de persistencia

Un *framework de persistencia* es una capa de software que facilita la gestión y almacenamiento de datos en una base de datos, abstrayendo las operaciones complejas de acceso y manipulación mediante el uso de un *Object-Relational Mapping (ORM)* [19].

En el caso de *Django*, el *ORM* integrado actúa como su *framework* de persistencia principal, proporcionando una interfaz sencilla para definir modelos de datos, realizar consultas y gestionar migraciones. Esto permite al desarrollador trabajar con objetos de *Python* en lugar de escribir consultas *SQL* directamente.

Aunque es posible utilizar otros *frameworks* de persistencia externos, como *SQLAlchemy* [4], en lugar del *ORM* de *Django*, esta práctica no es habitual debido a la fuerte integración del *ORM* propio con otros componentes del *framework*, lo que facilita la coherencia y mantenimiento del proyecto.

4.14. Procesamiento de tareas en segundo plano

Celery

Celery [46] es una biblioteca de *Python* diseñada para la ejecución de tareas asíncronas y trabajos en segundo plano. En este proyecto, se ha utilizado para gestionar tareas que no deben interferir con el flujo principal de la aplicación, como procesos de larga duración.

Se valoraron otras alternativas como **Dramatiq** [8] o **Huey** [8], que también permiten la ejecución de tareas en segundo plano. Sin embargo, se seleccionó *Celery* por su madurez, amplia comunidad, documentación robusta y su excelente integración con *Django*. Su modelo basado en trabajadores (*workers*) permite distribuir la carga de forma eficiente, lo que lo convierte en una solución sólida y escalable.

4.15. Sistema de mensajería

Redis

Redis [33] ha sido utilizado como sistema de mensajería (*message broker*) para *Celery*, facilitando la comunicación entre la aplicación y los trabajadores

que ejecutan tareas en segundo plano. Redis es un almacén de datos clave-valor en memoria, extremadamente rápido y ampliamente adoptado.

Se consideraron alternativas como **RabbitMQ** [53], también compatible con *Celery*. No obstante, se optó por *Redis* por su menor complejidad de instalación y configuración, así como por su rendimiento óptimo en entornos de desarrollo y pruebas. Además, su simplicidad lo hace ideal para proyectos que no requieren mecanismos avanzados de enrutamiento de mensajes.

4.16. Modelos de lenguaje locales

Ollama

Ollama [42] es una herramienta que permite ejecutar modelos de lenguaje de gran tamaño (LLMs) localmente, sin necesidad de enviar datos a servidores externos. Esto proporciona un mayor control sobre los modelos empleados y mejora la privacidad de la información tratada.

Entre las alternativas consideradas se encuentran plataformas como **OpenAI** [43], **Hugging Face Inference API** [16] y **Replicate** [50], que ofrecen servicios en la nube para el despliegue de modelos. Sin embargo, *Ollama* ha sido seleccionado por su capacidad de funcionar de manera local, su integración con la biblioteca `langchain-ollama` [10], y por eliminar la dependencia de servicios externos, lo que reduce latencia y costes asociados al uso de *APIs* comerciales.

Además, se optó por *Ollama* para preservar la privacidad de los ficheros tratados durante el proceso de evaluaciones, evitando que información sensible salga del entorno local del proyecto.

Langchain

Langchain [6] es una biblioteca diseñada para facilitar la integración de modelos de lenguaje en aplicaciones, proporcionando herramientas para la gestión de cadenas de *prompts*, el manejo de memoria conversacional y la orquestación de tareas complejas.

En este proyecto, se ha utilizado `langchain-ollama` [10], que permite conectar de forma sencilla modelos *LLM* ejecutados localmente mediante *Ollama*. Esta integración ha sido fundamental para realizar inferencias de lenguaje natural directamente en el entorno local, sin necesidad de acceder a servicios externos.

4.17. Documentación de la memoria

La documentación del proyecto se ha desarrollado en LaTeX. Para ello, se consideraron diversas herramientas:

- **MiKTeX**, para sistemas operativos Windows. [48]
- **TeX Live**, para sistemas operativos Linux. [27]
- **MacTeX**, para sistemas operativos MacOS. [26]
- **Overleaf**, editor colaborativo en línea. [37]

Overleaf

Finalmente, se optó por utilizar **Overleaf** como entorno principal para la edición y compilación de los documentos. Esta plataforma en línea ha facilitado la colaboración, el acceso remoto y la gestión del proyecto documental, evitando la necesidad de configurar localmente las distribuciones de LaTeX.

4.18. Documentación del código

Sphinx

Sphinx [55] es una herramienta de documentación ampliamente utilizada en proyectos *Python*. Permite generar documentación estructurada a partir de archivos fuente escritos en *reStructuredText* o *Markdown*, generando resultados en formatos como *HTML* o *PDF*.

Durante el desarrollo del proyecto se valoraron también otras alternativas como **MkDocs** [9], una herramienta más orientada a documentación en Markdown y con una configuración más sencilla. Sin embargo, se seleccionó **Sphinx** por su mejor integración con bibliotecas *Python*, su capacidad para generar documentación técnica a partir del código fuente mediante *docstrings*, y su soporte más maduro para proyectos complejos.

Para mejorar la apariencia y navegabilidad de la documentación generada, se ha utilizado el tema **sphinx-book-theme** [47], que proporciona una interfaz moderna y adecuada para documentación académica o técnica.

4.19. Herramientas de traducción

Durante la elaboración de la documentación del proyecto y la consulta de materiales técnicos en otros idiomas, se han utilizado herramientas de traducción automática con el objetivo de agilizar la comprensión y adaptación de los contenidos.

En particular, se han utilizado los servicios de **Google Translate** [35] y **DeepL** [24] para facilitar la traducción de contenidos relevantes empleados en el desarrollo del proyecto, así como en la redacción de esta memoria y de su documentación técnica.

Estas herramientas han permitido interpretar documentación, artículos y ejemplos en inglés de manera eficaz, facilitando así la integración de conceptos y soluciones en el contexto del proyecto.

4.20. Bibliotecas y librerías relevantes

psycopg2-binary

psycopg2-binary [12] es un adaptador que permite a *Django* comunicarse con bases de datos *PostgreSQL*. Ha sido imprescindible para conectar el *backend* del proyecto con el sistema gestor de base de datos seleccionado, proporcionando una integración fluida y eficiente.

python-decouple

python-decouple [13] permite gestionar las variables de entorno de forma segura, separando la configuración del entorno de desarrollo del código fuente. Se ha empleado para definir credenciales sensibles y configuraciones del entorno sin exponerlas directamente en el repositorio.

Markdown

Markdown [11] es una biblioteca de *Python* que permite convertir texto escrito en formato Markdown a otros formatos, como *HTML*. En este proyecto, se ha utilizado para procesar y generar documentación dinámica a partir de contenido en Markdown, facilitando la integración con otras herramientas de documentación y la presentación de información legible y estructurada.

4.21. Desarrollo web

El desarrollo web es fundamental para la creación de aplicaciones y sitios accesibles a través de navegadores. Esta sección presenta las tecnologías y herramientas utilizadas en el proyecto para construir la interfaz de usuario, abarcando desde la estructura básica y estilos visuales hasta la interactividad y componentes avanzados. Se detallan los lenguajes estándar, así como *frameworks* y bibliotecas modernas que agilizan el diseño y desarrollo *frontend*.

HTML

HTML (HyperText Markup Language) [57] es el lenguaje estándar para la creación de páginas web. Se utiliza para estructurar el contenido y definir elementos como texto, imágenes, enlaces y formularios.

CSS

CSS (Cascading Style Sheets) [56] permite definir el estilo visual y la presentación de las páginas web. Se emplea para controlar aspectos como colores, fuentes, márgenes y disposición de los elementos en la pantalla.

JavaScript

JavaScript [32] es un lenguaje de programación que añade interactividad y dinamismo a las páginas web. Permite manejar eventos, modificar el *DOM* y comunicarse con servidores para actualizar contenido sin recargar la página.

Tailwind

Tailwind CSS [34] es un *framework* de CSS basado en clases utilitarias que facilita la creación rápida y consistente de diseños personalizados sin salir del *HTML*. Ofrece un sistema modular para construir interfaces responsivas y estilizadas.

Flowbite

Flowbite [18] es una biblioteca de componentes UI basada en *Tailwind CSS* que proporciona elementos preconstruidos como botones, menús y for-

mularios. Simplifica el desarrollo *frontend* al ofrecer componentes accesibles y personalizables, listos para usar.

4.22. Despliegue de la aplicación

Para facilitar el despliegue, la ejecución en distintos entornos y la gestión de dependencias, se optó por contenerizar la aplicación utilizando **Docker**. Esta decisión permite asegurar que todos los componentes del sistema funcionen de forma consistente, independientemente del entorno en el que se ejecute.

El despliegue se realiza mediante **Docker Compose**, una herramienta que facilita la definición y ejecución de aplicaciones formadas por múltiples contenedores. En este caso, el ecosistema de la aplicación está compuesto por los siguientes servicios:

- **Django**: servicio principal que contiene la lógica de la aplicación web. Se ejecuta dentro de un contenedor que contiene un entorno *Python* configurado con todas las dependencias necesarias.
- **PostgreSQL**: base de datos relacional utilizada por *Django*. Se levanta como un contenedor independiente, asegurando la persistencia y accesibilidad desde otros servicios mediante redes internas de *Docker*.
- **Celery**: sistema de tareas asíncronas utilizado para procesar trabajos en segundo plano. Este servicio se comunica con *Django* para recibir tareas y con *Redis* como sistema de mensajería.
- **Redis**: cola de mensajes que actúa como *broker* entre *Django* y *Celery*. Su uso permite desacoplar la gestión de tareas del servidor principal y mejorar el rendimiento general de la aplicación.
- **Ollama**: servicio utilizado para ejecutar modelos de lenguaje de forma local. Este componente permite realizar **correcciones automáticas de tareas**. Al ejecutarse localmente, evita la dependencia de servicios externos y garantiza un mayor control sobre el procesamiento de datos.

Cada uno de estos servicios se define en el archivo `docker-compose.yml`, donde se especifican las imágenes, variables de entorno, volúmenes, redes y dependencias entre contenedores. Este enfoque permite levantar toda la infraestructura del proyecto con un único comando:

```
docker compose up --build
```

Además, se han incluido archivos como `Dockerfile` y `.env` para una configuración personalizada. Gracias a esta arquitectura basada en contenedores, se ha logrado un despliegue limpio, reproducible y fácilmente escalable.

4.23. Otras herramientas

Draw.io

Draw.io [36] es una herramienta en línea para la creación de diagramas y esquemas de forma sencilla y colaborativa. Se ha utilizado para diseñar diagramas de casos de uso, diagramas *UML, ER* y otras representaciones gráficas necesarias durante el desarrollo del proyecto.

Miro

Miro [40] es una plataforma colaborativa de pizarras digitales que facilita el trabajo en equipo, la planificación visual y la lluvia de ideas. En este proyecto, se ha empleado para la creación de un *mockup* de la página de inicio.

Herramientas generativas

La integración de herramientas basadas en inteligencia artificial generativa ha cobrado una relevancia creciente en entornos educativos y de desarrollo de software. Estas tecnologías permiten automatizar tareas rutinarias, mejorar la eficiencia y asistir en procesos tanto técnicos como creativos. En el marco de este proyecto, se ha explorado el uso de algunas de estas herramientas en diversas fases del trabajo.

Una de las más destacadas es *GitHub Copilot* [38], una extensión integrada en *Visual Studio Code*. Esta herramienta ofrece sugerencias contextuales mientras se programa, proponiendo fragmentos de código, completando funciones y generando comentarios explicativos. Su funcionamiento resulta especialmente útil en tareas repetitivas o estructuralmente predecibles, contribuyendo a mantener un flujo de trabajo ágil y menos propenso a errores menores.

Asimismo, se ha empleado *Perplexity AI* [45], una herramienta de búsqueda conversacional basada en inteligencia artificial, para generar imágenes utilizadas en algunas secciones de la aplicación web.

5. Aspectos relevantes del desarrollo del proyecto

5.1. Inicio del proyecto

El desarrollo de este proyecto surgió tras un proceso de negociación y diálogo con mi tutor, durante el cual le presenté varias propuestas iniciales. A partir de sus ideas y sugerencias, acordamos centrar el trabajo en la creación de un sistema para la corrección automática de tareas de alumnos utilizando inteligencia artificial.

La motivación principal radica en la creciente integración de tecnologías de inteligencia artificial y modelos de lenguaje a gran escala (*LLMs*) en el ámbito educativo. Dado que el alumnado empieza a ayudarse de la guía proporcionada por este tipo de herramientas en la resolución de sus tareas, resulta natural y necesario explorar métodos que permitan evaluar dichas tareas utilizando la misma metodología basada en *IA*.

El proyecto se seleccionó por ser una propuesta innovadora, que aprovecha tecnologías actuales como los *LLMs* para implementar una aplicación web que se distingue de las soluciones tradicionales de corrección automática. Esta aplicación no solo integra inteligencia artificial generativa, sino que también ofrece una interfaz y funcionalidades diferentes a lo habitual, adaptándose a las necesidades actuales del entorno educativo digital.

Este enfoque no solo pretende agilizar el proceso de evaluación, sino también adaptarse a las nuevas formas de aprendizaje y producción de contenidos, manteniendo la coherencia con las herramientas que los alumnos emplean. De esta forma, el proyecto se posiciona como una respuesta innovadora y pertinente a la evolución tecnológica en el contexto académico.

5.2. Gestión y metodología del proyecto

Desde el inicio del proyecto se decidió aplicar una **metodología ágil** adaptada al contexto académico e individual del trabajo. En concreto, se optó por ***Scrumban*** [52], una combinación entre *Scrum* y *Kanban* que permite mantener ciclos de trabajo planificados (*sprints*) al mismo tiempo que se ofrece una mayor flexibilidad en la gestión de tareas.

El proyecto se estructuró en *sprints* de dos semanas, al final de los cuales se realizaban **reuniones con el tutor**. En estas sesiones se revisaban las **tareas completadas** durante el *sprint* anterior, se evaluaba el cumplimiento de los objetivos y se proponían nuevas tareas para el siguiente ciclo. Este enfoque iterativo facilitó una **mejora continua** del producto y permitió adaptarse a cambios o mejoras surgidas a lo largo del desarrollo.

La **gestión de tareas** se realizó con la herramienta ***Zube***, integrada con *GitHub*. Esta plataforma permitió organizar el trabajo mediante *issues*, que simulaban la dinámica de un equipo multidisciplinar. Se utilizó un **tablero *Kanban*** con columnas como *inbox*, *ready*, *in progress*, *in review* y *done*, lo que ayudó a tener una visión clara y actualizada del estado de cada tarea y a gestionar eficazmente el flujo de trabajo.

También se utilizaron **gráficos *burndown*** para medir el progreso de cada *sprint* y visualizar la carga de trabajo restante. En general, se logró cumplir con los plazos previstos. Estas herramientas resultaron útiles para mantener un **control claro del avance** y asegurar que los objetivos establecidos al inicio de cada ciclo se cumplieran de forma efectiva.

Adicionalmente, para el **control de versiones** se siguió la **metodología *GitFlow***, que permitió organizar el repositorio de forma estructurada mediante ramas diferenciadas como *main*, *develop*, *feature*, *release* y *hotfix*. Esta estrategia facilitó un desarrollo más limpio, con mayor trazabilidad de los cambios y mejor manejo de nuevas funcionalidades y correcciones de errores.

5.3. Formación

Este trabajo ha requerido adquirir ciertos conocimientos específicos que inicialmente no se dominaban en profundidad, especialmente en lo relativo a la integración de modelos de lenguaje (*LLMs*) con aplicaciones web, la ejecución de tareas en segundo plano y el despliegue de servicios mediante contenedores. No obstante, ya se contaba con una base previa en bases de

datos y programación en Python, lo cual ha facilitado algunas de las fases del proyecto.

La tecnología central del *backend* ha sido el *framework Django*, cuyo sistema de vistas, *ORM* y gestión de rutas se ha aprendido principalmente a través de su documentación oficial [20, 19], y mediante pruebas experimentales con pequeños proyectos de ejemplo. La comprensión de los distintos motores de bases de datos también fue necesaria; se optó finalmente por *PostgreSQL* como solución [25].

Uno de los mayores retos ha sido la integración de modelos de lenguaje ejecutados localmente a través de *Ollama* [42]. Para ello, fue clave familiarizarse con el funcionamiento de la biblioteca *LangChain* [6], así como su extensión para *Ollama* [10].

En cuanto a la ejecución de tareas en segundo plano, fundamental para no bloquear la experiencia del usuario, se investigaron varias alternativas. Finalmente se eligió `Celery` con `Redis` como broker [46, 33].

Otro aspecto relevante del aprendizaje ha sido la contenerización de servicios mediante *Docker* y *Docker Compose*, lo cual permitió simular entornos de producción de forma local. Aunque no se tenía experiencia previa con estas tecnologías, su curva de aprendizaje fue razonable gracias a su documentación oficial y a la gran cantidad de recursos disponibles en línea [29].

En cuanto al desarrollo del *frontend*, se utilizaron conocimientos previos de `HTML` [57], `CSS` [56] y `JavaScript` [32], complementados con el uso del framework `Tailwind CSS` [34] y su librería de componentes Flowbite [18], que permitieron construir una interfaz moderna y accesible.

Por último, para la documentación técnica del proyecto se exploraron y probaron herramientas como Sphinx [55], su tema Book Theme [47] y MkDocs [9]. También se utilizó Markdown [11] como lenguaje de marcado ligero y herramientas visuales como Draw.io [36] y Miro [40] para la elaboración de diagramas y esquemas.

A lo largo del proceso se han consultado otras muchas fuentes, tanto en forma de documentación como de vídeos [62, 63] y artículos técnicos, pero sin duda las anteriormente mencionadas han sido las más relevantes en cuanto a la formación adquirida durante el desarrollo de este proyecto.

5.4. Desarrollo de la aplicación

El desarrollo de la aplicación se estructuró en fases iterativas, comenzando con la implementación de las funcionalidades básicas de gestión de usuarios y navegación. La primera etapa incluyó el desarrollo del sistema de autenticación (*login, logout* y registro), así como la creación de una página inicial sencilla desde la cual los usuarios podían acceder al resto de funcionalidades. Esta base permitió asegurar un entorno seguro y personalizado para cada usuario.

Posteriormente, se incorporaron las funcionalidades relacionadas con la gestión de los elementos fundamentales para la evaluación: los *prompts* y las rúbricas. Se implementaron las funcionalidades necesarias para crear, consultar y eliminar prompts personalizados, así como importar rúbricas de evaluación en formato markdown, con el objetivo de facilitar la configuración de criterios de corrección.

En una siguiente iteración, se desarrolló la lógica para configurar las correcciones propiamente dichas. En esta fase, la aplicación permitió asociar un *prompt*, una rúbrica y subir un fichero comprimido con las tareas de programación ubicadas en la raíz del fichero. Estas configuraciones quedaban almacenadas en la base de datos, permitiendo mantener un registro estructurado de cada tarea de evaluación preparada por el usuario.

Con la configuración establecida, se implementó el procesamiento en segundo plano de las correcciones utilizando `Celery` junto con `Redis` como *message broker*. Esta solución permitió ejecutar de forma asíncrona las tareas relacionadas con el envío de información al modelo de lenguaje y la recepción de respuestas, evitando bloquear el servidor principal.

Debido a que los *LLMs* pueden tardar varios segundos en generar una respuesta, especialmente en entornos locales, resultaba inviable gestionar estas operaciones de forma síncrona. Gracias a esta arquitectura, la aplicación mantiene su capacidad de respuesta y permite gestionar múltiples evaluaciones simultáneamente sin afectar al rendimiento general.

Finalmente, se añadió la posibilidad de modificar ciertos parámetros del modelo *LLM* en cada evaluación, como el nivel de creatividad (*temperatura*), la probabilidad acumulada máxima (*top-p*) y el número máximo de *tokens* considerados en la selección (*top-k*). Esta flexibilidad permitió a los usuarios adaptar la evaluación a distintos escenarios y necesidades específicas.

Procesamiento de archivos y estructura del sistema

Se estableció un sistema de subida de archivos donde el usuario podía cargar:

- Un archivo con la rúbrica de evaluación (en formato `md`).
- Un archivo comprimido con las tareas en su interior.

Además, el usuario podía escribir manualmente un *prompt* personalizado. La lógica del sistema se encargaba de ensamblar todos estos elementos en una única entrada textual coherente, que era enviada al modelo.

Evaluación con modelos LLM

El modelo principal utilizado fue `LLaMA 3.1`, ejecutado localmente a través de *Ollama* e integrado en la lógica de la aplicación mediante la librería `langchain-ollama`.

Cada evaluación se procesaba como una tarea independiente, registrando:

- Fecha y hora de la configuración.
- *Prompt* y rúbrica utilizados.
- Tareas evaluadas.
- Fecha y hora de la evaluación.
- Parámetros del modelo y formato de la respuesta.
- Respuesta devuelta por el modelo.

Cada evaluación generaba un archivo de texto plano (`.txt`) que se almacenaba en el directorio `media`.

Interfaz y usabilidad

La interfaz web fue desarrollada con `HTML`, `CSS` y `JavaScript`, apoyada por `Tailwind CSS` y la biblioteca de componentes `Flowbite`, lo que permitió un diseño limpio y funcional. Se priorizó una navegación simple en la que el flujo de trabajo fuera evidente: autenticación → gestión de prompts → gestión de rúbricas → configuración de evaluación → ejecución de correcciones → consulta o descarga de resultado.

Se incorporaron funcionalidades adicionales como:

- Tabla de correcciones por usuario.
- Tabla de rúbricas por usuario.
- Tabla de prompts por usuario.
- Gestión de usuarios y sesiones mediante el sistema de autenticación de Django.

Contenerización y despliegue

Para garantizar la reproducibilidad y facilitar el despliegue en distintos entornos, se optó por contenerizar todos los servicios mediante `Docker`. El sistema quedó estructurado en varios contenedores:

- *Django*.
- Base de datos *PostgreSQL*.
- *Broker* de mensajería *Redis*.
- *Ollama* con modelos *LLM*.
- *Celery*.

Esta separación de servicios permitió trabajar de forma más ordenada y facilitó el proceso de pruebas y desarrollo local.

Resumen de funcionalidades desarrolladas

A lo largo del desarrollo se implementaron las siguientes características clave:

- Sistema de autenticación con *login, logout* y registro.
- Gestión y edición de *prompts* y rúbricas.
- Configuración de tareas de evaluación que almacenan *prompt*, rúbrica y archivos asociados.
- Ejecución asincrónica de evaluaciones mediante Celery y Redis.

- Personalización de parámetros del modelo *LLM* en cada evaluación.
- Registro, almacenamiento y consulta de resultados.
- Interfaz web amigable y funcional.
- Contenerización completa para facilitar el despliegue.

El desarrollo modular y progresivo permitió construir una aplicación fácilmente ampliable, alineada con los objetivos establecidos al inicio del proyecto.

Constantes configurables del sistema

El sistema cuenta con ciertas constantes definidas en el código fuente que permiten ajustar su comportamiento de forma sencilla. En particular, existen dos elementos clave que pueden modificarse para ampliar la funcionalidad de la aplicación:

- En el archivo `corrections/views.py`, línea 28, se encuentra la tupla `VALID_EXTENSION`, que especifica las extensiones de archivo válidas para la corrección automática. Actualmente incluye formatos como `.java`, `.sql`, `.py`, `.cpp`, `.txt`, entre otros. Esta lista puede ampliarse fácilmente para soportar nuevos lenguajes o tipos de archivo.
- En el archivo `corrections/forms.py`, línea 10, se define la lista `MODEL_CONTEXT_CHOICES`, que establece las opciones de contexto máximo en número de *tokens* que pueden procesarse por el modelo de lenguaje. Los valores actuales son 2048, 4096 y 8192 *tokens*, pero esta lista puede modificarse para incorporar futuras configuraciones.

Estas constantes permiten adaptar la aplicación a distintos entornos o requerimientos sin necesidad de modificar la lógica principal del sistema.

5.5. Problemas, incidencias y soluciones

Durante el desarrollo de la aplicación surgieron distintos problemas técnicos, principalmente relacionados con la ejecución de tareas asíncronas mediante `Celery` y la contenerización del proyecto con `Docker`. A continuación, se describen los problemas más relevantes y las soluciones aplicadas.

Inicio con `async` de *Django* y cambio a `Celery` y `Redis`

En una primera etapa, se intentó utilizar las funcionalidades `async` nativas de *Django* para gestionar la ejecución de tareas en segundo plano. Sin embargo, esta implementación inicial no cumplía con las necesidades reales de procesamiento asíncrono, ya que no permitía la ejecución fuera del ciclo de vida de la solicitud *HTTP* ni garantizaba la persistencia ni la gestión eficiente de tareas largas.

Solución: Se optó por integrar `Celery` como sistema de cola de tareas junto con `Redis` como broker de mensajes. Esta combinación permitió una gestión robusta y confiable de las tareas asíncronas, garantizando que las operaciones de evaluación y generación de respuestas pudieran ejecutarse en segundo plano sin afectar la experiencia del usuario.

Advertencia por ejecutar Celery como usuario `root`

Uno de los primeros problemas detectados fue una advertencia al iniciar `Celery`, que alertaba de que el proceso se estaba ejecutando como usuario `root`. Aunque funcional, esto supone un riesgo potencial de seguridad, especialmente en entornos de producción, y va en contra de las buenas prácticas recomendadas.

Solución: Para evitar ejecutar procesos como superusuario, se creó un nuevo usuario dentro del contenedor de Docker (llamado `dj_admin`) y se configuró para que los procesos de Django y Celery se ejecutaran bajo ese usuario. En el archivo `Dockerfile`, se utilizó la instrucción `USER dj_admin` para establecer este comportamiento. Esto permitió mantener un entorno más seguro y alineado con las recomendaciones de los *frameworks* utilizados.

Problemas de permisos al escribir en la carpeta `media`

Tras aplicar la solución anterior, surgió una nueva incidencia: el usuario `dj_admin` no tenía permisos suficientes para acceder a la carpeta `media`, que se utiliza para almacenar archivos temporales y resultados de las evaluaciones (en formato `.txt`). Esta carpeta es esencial para el correcto funcionamiento del sistema, ya que en ella se guardan las tareas a corregir, así como la respuesta generada por el modelo *LLM*.

Solución: Se añadieron comandos específicos al `Dockerfile` para asignar los permisos adecuados a la carpeta `/app/media`. Concretamente, se utilizó el siguiente bloque de instrucciones:

```
...

RUN chown -R dj_admin:dj_admin /app/media

EXPOSE 8000

COPY entrypoint.sh /entrypoint.sh

RUN chmod +x /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]

USER dj_admin

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Esto garantiza que el usuario sin privilegios tenga acceso completo a dicha carpeta. Con este ajuste, fue posible evitar errores de acceso y escritura al ejecutar tareas de evaluación que involucran la generación y almacenamiento de archivos.

Conflictos por carpetas `media` durante la ejecución de tests

Durante la ejecución de los *tests* automatizados de la aplicación de correcciones, se detectó que se creaban carpetas nuevas dentro del directorio `media`. Esto podía provocar interferencias o conflictos con datos y archivos pertenecientes a otros usuarios, afectando la integridad y el aislamiento de las pruebas.

Solución: Para evitar estos problemas, se implementó un mecanismo que crea una carpeta temporal específica para `media` cada vez que se ejecutan los *tests*. Esta carpeta temporal se utiliza exclusivamente durante la ejecución de las pruebas y se elimina automáticamente al finalizar, garantizando así que el entorno de *tests* esté aislado y no interfiera con el sistema en uso normal.

Esta estrategia asegura un entorno limpio y reproducible para las pruebas, mejorando la fiabilidad y evitando efectos colaterales no deseados en el almacenamiento de archivos durante la fase de *testing*.

6. Trabajos relacionados

Herramienta de corrección automática de prácticas de programación

El trabajo *Herramienta de corrección automática de prácticas de programación* [64], realizado en la Universidad Politécnica de Madrid, presenta un sistema para la corrección automática de ejercicios de programación. La herramienta analiza el código de los estudiantes, teniendo en cuenta aspectos como el uso de buenas prácticas, la legibilidad y la estructura del código, con el objetivo de ofrecer retroalimentación inmediata y aliviar la carga de trabajo del profesorado.

A diferencia de IAGScore, que utiliza modelos de lenguaje a gran escala (*LLMs*) para interpretar y evaluar el contenido de las tareas, este sistema se basa en reglas definidas previamente y en análisis más estáticos. Aunque ambas propuestas buscan automatizar la corrección en cursos de programación, lo hacen con metodologías distintas: una más tradicional, basada en validaciones estructurales, y otra más flexible, apoyada en inteligencia artificial generativa.

Sistema de corrección automática basado en tests y análisis estático

El trabajo presentado como *Sistema de corrección automática basado en tests y análisis estático* [41] describe un sistema para la corrección automática de tareas de programación que se apoya principalmente en la ejecución de tests unitarios para verificar que el código entregado cumple con los requisitos funcionales. Además, incorpora técnicas de análisis estático para evaluar aspectos como el estilo, la calidad y la estructura del código.

Este enfoque combina la validación dinámica mediante pruebas automatizadas con la revisión estática del código, ofreciendo una evaluación más completa que se centra tanto en el funcionamiento correcto como en las buenas prácticas de programación.

A diferencia del sistema propuesto en este proyecto, que emplea modelos de lenguaje (*LLMs*) para interpretar y evaluar las soluciones desde una perspectiva semántica y contextual, el enfoque descrito se basa en validaciones formales y reglas predefinidas. Mientras que la herramienta basada en tests requiere una programación previa de casos de prueba y métricas específicas, el uso de *LLMs* permite una evaluación más flexible y adaptable al tipo de problema planteado, así como una retroalimentación en lenguaje natural más cercana a la corrección humana.

Rubric Is All You Need

El artículo *Rubric Is All You Need: Enhancing LLM-based Code Evaluation With Question-Specific Rubrics* [44], analiza el uso de modelos de lenguaje a gran escala para evaluar tareas de programación en cursos universitarios. Proponen tres técnicas específicas: Evaluación Completa de Rúbrica (CRE), Evaluación Punto a Punto (PRE) y Evaluación por Ensamble (EME), que permiten evaluar la lógica y el contenido semántico del código más allá de la simple corrección sintáctica.

El estudio se basa en dos conjuntos de datos: exámenes de Java en Programación Orientada a Objetos y ejercicios de estructuras de datos y algoritmos. Los resultados muestran que el uso de rúbricas específicas para cada problema mejora la precisión y la calidad de la retroalimentación, obteniendo una alta correlación con las calificaciones de evaluadores humanos. También introducen una métrica llamada *Leniency*, que ajusta el nivel de exigencia del sistema evaluador, aportando flexibilidad para adaptar la rigurosidad de la corrección, un aspecto interesante para futuras mejoras del proyecto, especialmente en entornos educativos con distintos niveles de exigencia.

Características	**Corrección automática (UPM)**	**Tests y análisis estático**	**Rubric Is All You Need**	**IAGScore**
Evaluación automatizada	Sí	Sí	Sí	Sí
Análisis estático	Sí	Sí	No	No
Uso de rúbricas	No	Parcial	Sí	Sí
Generación de lenguaje natural	No	No	Sí	Sí
Uso de IA/LLM	No	No	Sí	Sí
Evaluación contextual	No	No	Parcial	Sí
Adaptabilidad	Media	Baja	Alta	Alta
Personalización	No	No	Parcial	Sí

Tabla 6.1: Comparativa entre trabajos de corrección automática de código e IAGScore

7. Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones

A lo largo del desarrollo de este Trabajo de Fin de Grado se han alcanzado los objetivos propuestos, tanto a nivel funcional como técnico y personal. El proyecto ha consistido en la construcción de una herramienta web capaz de evaluar automáticamente ejercicios de programación, integrando modelos de lenguaje a gran escala (*LLMs*) de forma local.

Cumplimiento de los objetivos

Se ha logrado integrar modelos *LLM* de forma local a través de *Ollama*. Esto ha permitido implementar un sistema de corrección automatizada capaz de analizar el código enviado por el usuario y devolver una evaluación textual. Además, se ha desarrollado una interfaz que permite interactuar con el sistema de manera sencilla.

Durante el desarrollo se han utilizado tecnologías como *Django, PostgreSQL, Docker, GitHub Actions o Sphinx*, entre otras. El uso de contenedores ha facilitado el despliegue y prueba del sistema, y el control de versiones ha permitido mantener una estructura organizada. También se han incorporado prácticas de calidad del software, como la integración de *SonarCloud*.

Este proyecto ha sido una oportunidad para afianzar conocimientos en múltiples áreas. Se han reforzado competencias en gestión de proyectos, control de versiones, calidad de código y documentación técnica, permitiendo una visión más completa del ciclo de vida del *software*.

Reflexiones y retos del proceso

Uno de los principales desafíos ha sido trabajar con tecnologías emergentes como los modelos de lenguaje de gran escala (*LLMs*). Desde el inicio del proyecto, existía incertidumbre sobre su viabilidad para cumplir los objetivos planteados, lo que supuso un reto tanto en el diseño como en la implementación. Esta falta de garantías iniciales exigió una actitud exploratoria, así como la adaptación continua a un entorno tecnológico en rápida evolución.

Además, se ha tenido que lidiar con algunas limitaciones inherentes a este tipo de modelos, como las denominadas *alucinaciones*, es decir, respuestas generadas que pueden parecer coherentes pero no son correctas o están inventadas. Dado que se trata de tecnologías emergentes, este fenómeno subraya la necesidad de mantener una supervisión humana en el proceso de evaluación, especialmente en entornos educativos donde la precisión y la fiabilidad son fundamentales.

Valoración personal

La realización de este proyecto ha sido altamente satisfactoria, tanto a nivel académico como personal. No solo se ha desarrollado una aplicación funcional, sino que se ha conseguido una aproximación a la integración de *LLMs* en el entorno educativo para la corrección de tareas.

Esta experiencia ha sido fundamental tanto para consolidar la formación académica como para prepararse para futuros retos profesionales.

7.2. Líneas de trabajo futuras

El proyecto desarrollado sienta las bases para futuras ampliaciones tanto a nivel funcional como de integración con otros entornos. Una de las líneas de trabajo más prometedoras es la implementación de una *API* que permita conectar el sistema con plataformas educativas como *Moodle*. Esta integración facilitaría su adopción en entornos reales de enseñanza, automatizando la evaluación de tareas de programación directamente desde el entorno de gestión académica.

Otra posible ampliación consiste en utilizar distintos modelos *LLM* para realizar una evaluación cruzada del resultado generado por el *LLM* evaluador principal. Esto podría mejorar la precisión y robustez del sistema, al aprovechar distintas perspectivas y reducir posibles sesgos del modelo individual.

Asimismo, resultaría adecuado implementar un sistema de notificaciones que informe al usuario una vez finalizado el proceso de corrección. Estas notificaciones podrían realizarse mediante correo electrónico, *SMS* o notificaciones *push*, mejorando así la experiencia de usuario y facilitando la gestión del tiempo de espera en procesos asíncronos.

Bibliografía

[1] Atlassian. Bitbucket: Repositorios git para equipos. https://bitbucket.org/product/, 2025. [Internet; Accedido el 15 de mayo de 2025].

[2] Atlassian. Jira | herramienta de gestión de proyectos. https://www.atlassian.com/software/jira, 2025. [Internet; Accedido el 19 de mayo de 2025].

[3] Atlassian. Trello | herramienta de colaboración visual. https://trello.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[4] SQLAlchemy Authors. Sqlalchemy – the python sql toolkit and object relational mapper. https://www.sqlalchemy.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[5] CodeMaple. Understanding django mvt architecture and view functions – django full course for beginners [lesson 2]. https://medium.com/@CodeMaple/understanding-django-mvt-architecture-and-view-functions-django-full-course-for-beginners-lesson-39c8da093b44, 2023. [Internet; Accedido el 14 de mayo de 2025].

[6] LangChain Community. Langchain – framework for building applications with llms. https://langchain.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[7] SQLite Consortium. Sqlite – embedded sql database engine. https://www.sqlite.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[8] Dramatiq Contributors. Dramatiq – fast and reliable distributed task processing. https://dramatiq.io, 2025. [Internet; Accedido el 19 de mayo de 2025].

[9] MkDocs contributors. Mkdocs – project documentation with markdown. https://www.mkdocs.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[10] PyPI Contributors. langchain-ollama – langchain integration for ollama. https://pypi.org/project/langchain-ollama/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[11] PyPI Contributors. Markdown – python implementation of markdown. https://pypi.org/project/Markdown/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[12] PyPI Contributors. psycopg2-binary – postgresql database adapter for python. https://pypi.org/project/psycopg2-binary/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[13] PyPI Contributors. python-decouple – strict separation of settings from code. https://pypi.org/project/python-decouple/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[14] Spyder Project Contributors. Spyder: The scientific python development environment. https://www.spyder-ide.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[15] Vincent Driessen. A successful git branching model. https://nvie.com/posts/a-successful-git-branching-model/, 2010. [Internet; Accedido el 15 de mayo de 2025].

[16] Hugging Face. Hugging face inference api. https://huggingface.co/inference-api, 2025. [Internet; Accedido el 19 de mayo de 2025].

[17] Josep Ferrer. Cómo funcionan los transformadores: Una exploración detallada de la arquitectura de los transformadores. https://www.datacamp.com/es/tutorial/how-transformers-work, 2024. [Internet; Accedido el 30 de mayo de 2025].

[18] Flowbite. Flowbite – ui component library built on tailwind css. https://flowbite.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[19] Django Software Foundation. Django orm — database access layer. https://docs.djangoproject.com/en/stable/topics/db/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[20] Django Software Foundation. Django – the web framework for perfectionists with deadlines. https://www.djangoproject.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[21] The Apache Software Foundation. Apache subversion - welcome to apache subversion. https://subversion.apache.org/, 2025. [Internet; Accedido el 15 de mayo de 2025].

[22] Git-scm.com. Git – distributed version control system. https://git-scm.com, 2025. [Internet; Accedido el 15 de mayo de 2025].

[23] Inc. GitHub. Github: Where the world builds software. https://github.com, 2025. [Internet; Accedido el 15 de mayo de 2025].

[24] DeepL GmbH. Deepl translator. https://www.deepl.com/translator, 2025. [Internet; Accedido el 19 de mayo de 2025].

[25] PostgreSQL Global Development Group. Postgresql – the world's most advanced open source relational database. https://www.postgresql.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[26] TeX Users Group. Mactex – tex distribution for macos. https://tug.org/mactex/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[27] TeX Users Group. Tex live – tex distribution. https://tug.org/texlive/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[28] Prompt Engineering Guide. Guía de prompting para modelos de lenguaje. https://www.promptingguide.ai/es, 2025. [Internet; Accedido el 9 de mayo de 2025].

[29] Docker Inc. Docker – empowering app development for developers. https://www.docker.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[30] GitLab Inc. Gitlab: Plataforma de devops. https://about.gitlab.com/es/, 2025. [Internet; Accedido el 15 de mayo de 2025].

[31] Zube Inc. Zube | project management for github issues. https://zube.io, 2025. [Internet; Accedido el 19 de mayo de 2025].

[32] ECMA International. Ecmascript® language specification. https://262.ecma-international.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[33] Redis Labs. Redis – in-memory data structure store. https://redis.io, 2025. [Internet; Accedido el 19 de mayo de 2025].

[34] Tailwind Labs. Tailwind css – a utility-first css framework. https://tailwindcss.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[35] Google LLC. Google translate. https://translate.google.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[36] JGraph Ltd. Draw.io – online diagramming tool. https://app.diagrams.net, 2025. [Internet; Accedido el 19 de mayo de 2025].

[37] Overleaf Ltd. Overleaf – online latex editor. https://www.overleaf.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[38] Microsoft. Microsoft copilot: Tu compañero de ia. https://copilot.microsoft.com, 2025. [Internet; Accedido el 30 de mayo de 2025].

[39] Microsoft. Visual studio code. https://code.visualstudio.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[40] Miro. Miro – collaborative online whiteboard platform. https://miro.com/es/signup/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[41] Òscar Montañés Juanico. Sistema automatizado de corrección de programas: ampliación de herramientas. https://diposit.ub.edu/dspace/bitstream/2445/60450/2/memoria.pdf, 2014. [Internet; Accedido el 19 de mayo de 2025].

[42] Ollama. Ollama – local ai models. https://ollama.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[43] OpenAI. Openai. https://openai.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[44] Aditya Pathak, Rachit Gandhi, Vaibhav Uttam, Devansh, Yashwanth Nakka, Aaryan Raj Jindal, Pratyush Ghosh, Arnav Ramamoorthy, Shreyash Verma, Aditya Mittal, Aashna Ased, Chirag Khatri, Jagat Sesh Challa, and Dhruv Kumar. Rubric is all you need: Enhancing llm-based code evaluation with question-specific rubrics. *arXiv preprint arXiv:2503.23989*, 2025. [Internet; Accedido el 20 de mayo de 2025].

[45] Inc. Perplexity AI. Perplexity ai – answer engine powered by large language models. https://www.perplexity.ai/, 2025. [Internet; Accedido el 1 de junio de 2025].

[46] Celery Project. Celery – distributed task queue. https://docs.celeryproject.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[47] Executable Books Project. Sphinx book theme. https://sphinx-book-theme.readthedocs.io, 2025. [Internet; Accedido el 19 de mayo de 2025].

[48] MiKTeX Project. Miktex – windows tex distribution. https://miktex.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[49] Pallets Projects. Flask – web development, one drop at a time. https://flask.palletsprojects.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[50] Replicate. Replicate – run machine learning models in the cloud. https://replicate.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[51] Deep Revision. Transformer explicado con ejemplos sencillos. https://deeprevision.github.io/posts/001-transformer/, 2023. Consultado en junio de 2025.

[52] Anna Pérez, OBS Business School. La metodología scrumban: cuándo y por qué utilizarla. https://www.obsbusiness.school/blog/la-metodologia-scrumban-cuando-y-por-que-utilizarla, 2014. [Internet; Accedido el 12 de mayo de 2025].

[53] Pivotal Software. Rabbitmq – messaging that just works. https://www.rabbitmq.com, 2025. [Internet; Accedido el 19 de mayo de 2025].

[54] Cole Stryker and Jim Holdsworth. ¿qué es el pln (procesamiento del lenguaje natural)? https://www.ibm.com/es-es/think/topics/natural-language-processing, 2024. [Internet; Accedido el 20 de mayo de 2025].

[55] Sphinx Team. Sphinx – documentation generator. https://www.sphinx-doc.org, 2025. [Internet; Accedido el 19 de mayo de 2025].

[56] World Wide Web Consortium (W3C). Cascading style sheets (css). https://www.w3.org/Style/CSS/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[57] World Wide Web Consortium (W3C). Hypertext markup language (html). https://www.w3.org/TR/html52/, 2025. [Internet; Accedido el 19 de mayo de 2025].

[58] Wikipedia. Kanban (desarrollo) — wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Kanban_(desarrollo), 2025. [Internet; Accedido el 12 de mayo de 2025].

[59] Wikipedia. Modelo extenso de lenguaje — wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Modelo_extenso_de_lenguaje, 2025. [Internet; Accedido el 25 de mayo de 2025].

[60] Wikipedia. Rúbrica (docencia) — wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/RÃžbrica_(docencia), 2025. [Internet; Accedido el 18 de mayo de 2025].

[61] Wikipedia. Scrum (desarrollo de software) —— wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software), 2025. [Internet; Accedido el 12 de mayo de 2025].

[62] YouTube. Curso de django para principiantes. https://www.youtube.com/watch?v=T1intZyhXDU, 2022. [Internet; Accedido el 1 de marzo de 2025].

[63] YouTube. Curso de django para principiantes. https://www.youtube.com/watch?v=nG9vnEuIzBM, 2022. [Internet; Accedido el 1 de marzo de 2025].

[64] Luis Felipe Álvarez Osorio. Herramienta de corrección automática de prácticas de programación. https://oa.upm.es/83172/, 2024. [Internet; Accedido el 19 de mayo de 2025].