

---

¿Quién quiere ser millonario?  
Who Wants to Be a Millionaire?

---



Trabajo de Fin de Grado  
Curso 2024–2025

**Autor**

Alejandro Relanzón Coello 9.5

Álvaro Rojo Sánchez 9.5

Cintia Parreño Ruiz 9.5

Miguel Macicior Cornide 9.5

**Director**

Manuel Núñez García

Grado en Ingeniería Informática  
Facultad de Informática  
Universidad Complutense de Madrid



¿Quién quiere ser millonario?  
Who Wants to Be a Millionaire?

Trabajo de Fin de Grado en Ingeniería Informática

**Autor**

Alejandro Relanzón Coello 9.5

Álvaro Rojo Sánchez 9.5

Cintia Parreño Ruiz 9.5

Miguel Macicior Cornide 9.5

**Director**

Manuel Núñez García

Convocatoria: *Junio 2025*

Grado en Ingeniería Informática  
Facultad de Informática  
Universidad Complutense de Madrid

10 de mayo de 2025



# Dedicatoria

*“A Manuel, nuestro tutor, por su valiosa guía y apoyo en cada etapa; a nuestras familias, por su incondicional respaldo, afecto y fe en nuestro esfuerzo colectivo; y a todas las personas que nos han acompañado y apoyado a lo largo de este grado.”*



# Agradecimientos

*Comenzamos agradeciendo a Manuel, nuestro tutor, por su supervisión impecable en cada fase del TFG.*

*Hemos de agradecer también a nuestras familias por su cariño y comprensión a lo largo de toda la carrera.*

*A nuestros amigos y compañeros, gracias por su aliento constante durante la elaboración de este trabajo.*

*También queremos agradecer al equipo por su empeño y pasión, que convirtieron cada reto en un valioso aprendizaje.*

*Gracias a nuestros seres queridos por su apoyo constante y su creencia en nosotros desde el primer día.*

*Por último, agradecemos a la Universidad Complutense de Madrid por brindarnos los recursos y el entorno académico necesarios.*

*Terminamos este TFG con gratitud por el camino compartido con todos los que nos han acompañado.*



# Resumen

## ¿Quién quiere ser millonario?

Este proyecto presenta el desarrollo de una aplicación web interactiva que simula el concurso “¿Quién Quiere Ser Millonario?”, explorando la integración de *large language models* (LLMs) e inteligencia artificial (IA). El objetivo central fue diseñar una plataforma atractiva y funcional, abordando desafíos clave como la unicidad de las preguntas y la validación precisa de las respuestas mediante técnicas de *procesamiento del lenguaje natural* (NLP) y algoritmos de comparación. La aplicación implementa la mecánica del concurso, permitiendo a los usuarios responder preguntas de dificultad creciente.

Se integraron los modelos Gemini (para establecer la verdad fundamental de las respuestas) y DeepSeek, OpenAI y Llama (para el comodín de la llamada y como concursantes IA en el modo Máquina vs Máquina). Se describe la construcción y optimización de los *prompts* para DeepSeek, logrando una reducción significativa de la latencia de respuesta. De manera similar, se aborda la integración y el uso de *prompts* para OpenAI y Llama en el modo de enfrentamiento entre IAs. El trabajo presenta los resultados iniciales sin optimizar para DeepSeek y el impacto de las mejoras implementadas en su rendimiento. La aplicación desarrollada sirve como un entorno de prueba para la integración y comparación de diversas IAs en concursos de preguntas, ilustrando los retos y soluciones encontrados en el proceso de implementación y optimización de cada modelo.

## Palabras clave

Aplicación Web, Inteligencia Artificial, Modelos de Lenguaje a Gran Escala, Procesamiento del Lenguaje Natural, Concurso de Preguntas y Respuestas, Gemini, DeepSeek, Optimización de Prompts, Detección de Duplicados, Validación de Respuestas.



# Abstract

## Who Wants to Be a Millionaire?

This project presents the development of an interactive web application that simulates the “Who Wants to Be a Millionaire?” quiz show, exploring the integration of large language models (LLMs) and artificial intelligence (AI). The central objective was to design an attractive and functional platform, addressing key challenges such as the uniqueness of the questions and the precise validation of the answers through natural language processing (NLP) techniques and comparison algorithms. The application implements the mechanics of the quiz show, allowing users to answer questions of increasing difficulty. The Gemini models (to establish the fundamental truth of the answers) and DeepSeek, OpenAI, and Llama (for the phone-a-friend lifeline and as AI contestants in the Machine vs. Machine mode) were integrated. The construction and optimization of the prompts for DeepSeek, achieving a significant reduction in response latency, are described. Similarly, the integration and use of prompts for OpenAI and Llama in the AI versus AI confrontation mode are addressed. The work presents the initial, unoptimized results for DeepSeek and the impact of the improvements implemented on its performance. The developed application serves as a testing environment for the integration and comparison of various AIs in question-answering contests, illustrating the challenges and solutions encountered in the process of implementing and optimizing each model.

## Keywords

Web Application, Artificial Intelligence, Large Language Models, Natural Language Processing, Question Answering Contest, Gemini, DeepSeek, Prompt Optimization, Duplicate Detection, Answer Validation.



# Índice

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Objetivos . . . . .	2
1.3. Plan de trabajo . . . . .	3
1.4. Estructura del documento . . . . .	6
<b>2. Estado del Arte</b>	<b>7</b>
2.1. Referencias Relevantes . . . . .	7
2.2. Proyectos desarrollados . . . . .	8
2.2.1. QANTA Project . . . . .	8
2.2.2. Watson: ‘Jeopardy!’ . . . . .	11
2.2.3. LaMDA de Google . . . . .	13
2.3. Transformador (modelo de aprendizaje automático) . . . . .	14
2.3.1. BERT . . . . .	15
2.3.2. roBERTa . . . . .	16
2.4. Desafíos . . . . .	17
2.5. MCP (Model Context Protocol) . . . . .	18
<b>3. Descripción del Trabajo</b>	<b>25</b>
3.1. Framework Utilizado: <b>Reflex</b> . . . . .	26
3.1.1. Pros y contras de <b>Reflex</b> . . . . .	28
3.2. Base de Datos: PostgreSQL . . . . .	29
3.3. Estructura del proyecto . . . . .	30
3.4. Funcionalidades del Juego . . . . .	31
3.4.1. Modo Infinito (endless) . . . . .	31
3.4.2. Modo Normal . . . . .	33
3.4.3. Modo Temático . . . . .	35
3.4.4. Modo Máquina vs Máquina . . . . .	36
3.5. Sistema de Autenticación . . . . .	37
3.5.1. Inicio de Sesión . . . . .	38
3.5.2. Registro . . . . .	39
3.6. Arquitectura Reactiva . . . . .	40

3.6.1.	Menú Principal . . . . .	40
3.6.2.	Tabla de Clasificación: <i>Leaderboard</i> . . . . .	40
3.6.3.	Página de Perfil: ( <i>User Page</i> ) . . . . .	41
3.7.	Integración con Inteligencia Artificial . . . . .	41
3.7.1.	Google Gemini (Modelo: gemini-2.0-flash) . . . . .	42
3.7.2.	DeepSeek (Modelo: deepseek-chat) . . . . .	43
3.7.3.	OpenAI (Modelo:gpt-4o-mini) . . . . .	45
3.7.4.	Llama (Modelo: llama4-maverick) . . . . .	45
<b>4.</b>	<b>Contribuciones Personales</b>	<b>49</b>
4.1.	Alejandro Relanzón Coello . . . . .	49
4.1.1.	Conclusiones . . . . .	50
4.2.	Álvaro Rojo Sánchez . . . . .	50
4.2.1.	Conclusiones . . . . .	52
4.3.	Cintia Parreño Ruiz . . . . .	53
4.3.1.	Conclusiones . . . . .	54
4.4.	Miguel Macicior Cornide . . . . .	55
4.4.1.	Conclusiones . . . . .	56
<b>5.</b>	<b>Conclusiones y Trabajo Futuro</b>	<b>59</b>
5.1.	Resultados . . . . .	59
5.2.	Mejoras futuras . . . . .	62
5.2.1.	Temporizador . . . . .	62
5.2.2.	Historial de palabras clave . . . . .	63
5.2.3.	Sistema de bonificación por racha . . . . .	63
5.2.4.	Posibilidad de Guardar Partida . . . . .	64
5.2.5.	Selección del Modelo de IA . . . . .	65
5.3.	Conclusiones . . . . .	65
	<b>Introduction</b>	<b>69</b>
	<b>Conclusions and Future Work</b>	<b>71</b>
5.4.	Conclusions . . . . .	71
	<b>Bibliografía</b>	<b>73</b>
	<b>A. Imágenes adjuntas</b>	<b>75</b>
	<b>B. Implementaciones</b>	<b>81</b>
B.1.	Implementación básica para Temporizador . . . . .	81
B.2.	Implementación básica para Historial . . . . .	83
B.3.	Implementación básica para Bonificación . . . . .	85

# Índice de figuras

2.1. Cambios en las APIs con MCP . . . . .	18
3.1. Parte de la implementación del comodín 50:50 . . . . .	25
3.2. Repositorio Github de Reflex . . . . .	27
3.3. Cómo funciona una aplicación Reflex. . . . .	27
3.4. Creación de tablas para la base de datos . . . . .	30
3.5. Juego Balatro . . . . .	31
3.6. Juego Balatro rondas de 3 . . . . .	33
3.7. Modo Normal . . . . .	33
3.8. Hay 15 temas para elegir . . . . .	34
3.9. Iconos de los temas . . . . .	34
3.10. Modo temático con un comodín usado . . . . .	35
3.11. Elección de IA para el máquina vs máquina . . . . .	36
3.12. Modo Máquina Vs Máquina (Gemini Vs DeepSeek) . . . . .	37
3.13. Menú principal . . . . .	40
3.14. Perfil del usuario . . . . .	41
3.15. Prompt IA sin optimizar . . . . .	44
5.1. Coherencia en la respuesta . . . . .	59
5.2. Aciertos con OpenAI . . . . .	60
5.3. Aciertos con DeepSeek . . . . .	61
5.4. Aciertos con Lllama . . . . .	61
A.1. Gráfica de los resultados del comienzo sin optimizar . . . . .	75
A.2. Gráfica optimizada . . . . .	76
A.3. Desarrollo de la optimización . . . . .	76
A.4. Optimización del prompt . . . . .	77
A.5. Optimización con prompt mejorado . . . . .	77
A.6. Pregunta inversa para la optimización, IA responde correctamente . . . . .	78
A.7. Front al principio del desarrollo . . . . .	78
A.8. Página principal de la aplicación . . . . .	79



# Introducción

El presente Trabajo de Fin de Grado (TFG) se centra en el desarrollo de una aplicación web que recrea el concurso de televisión “¿Quién quiere ser millonario?”. Este proyecto abarca tanto el diseño y la implementación del *frontend* como del *backend* de la aplicación. El objetivo principal que nos marcamos consiste en crear una experiencia de usuario atractiva e intuitiva que permita a los usuarios participar en el concurso como si estuvieran en el programa de televisión. Esto implica la recreación fiel de la mecánica del juego, incluyendo las preguntas, las opciones de respuesta, los comodines y la dinámica de premios. Para lograr este objetivo se ha optado por utilizar el lenguaje de programación Python y el framework **Reflex**. Python es un lenguaje ampliamente utilizado en el desarrollo web debido a su versatilidad, su biblioteca de módulos y su facilidad de aprendizaje. **Reflex**, por su parte, es un *framework* que permite crear aplicaciones web interactivas y dinámicas de manera eficiente. Dicho de otra forma, la elección de Python y **Reflex** se justifica por su capacidad para ofrecer un entorno de desarrollo robusto y flexible, que permite abordar tanto el desarrollo del *frontend* como del *backend* de la aplicación de manera integrada. Además, estas tecnologías son ampliamente utilizadas en la industria, lo que garantiza la relevancia y la aplicabilidad de los conocimientos adquiridos en este proyecto.

A lo largo de este documento, se detallarán los diferentes aspectos del desarrollo de la aplicación, incluyendo el diseño de la arquitectura, la implementación de la lógica del juego, la creación de la interfaz de usuario, la gestión de la base de datos y las pruebas realizadas para garantizar la calidad y el correcto funcionamiento de la aplicación.

Además, se confía en que este proyecto pueda servir como una herramienta útil y entretenida para los usuarios, que les permita disfrutar del concurso “¿Quién quiere ser millonario?” desde la comodidad de sus hogares.

## 1.1. Motivación

La incursión en el desarrollo de aplicaciones que integran *large language models* (LLMs) e inteligencia artificial (IA) representa un **desafío técnico estimulante**, que permite aplicar y ampliar los conocimientos adquiridos durante la formación

académica en un dominio de vanguardia. El uso de tecnologías como Python y el *framework* `Reflex` se considera una oportunidad valiosa para profundizar en el conocimiento y la aplicación de herramientas ampliamente utilizadas en la industria del desarrollo web, al tiempo que se explora el potencial disruptivo de la IA en la creación de experiencias interactivas. La familiaridad con estas tecnologías, incluyendo la interacción con *APIs* de modelos como Gemini, DeepSeek, OpenAI y Llama, puede ser de gran utilidad en el futuro profesional, dada la creciente demanda de expertos en este campo. El proyecto también ofrece la posibilidad de abordar un proyecto de software de manera integral, desde la concepción inicial hasta la implementación y las pruebas finales de un sistema complejo que involucra la gestión de múltiples agentes de IA. Este proceso permite desarrollar habilidades importantes en la gestión de proyectos, la resolución de problemas específicos relacionados con la integración de IA, el diseño de *prompts* efectivos y el trabajo en equipo en un contexto tecnológico avanzado. La exploración de la interacción entre diferentes modelos de IA, como se plantea en el modo “Máquina vs Máquina”, añade una capa adicional de complejidad y aprendizaje al proyecto.

## 1.2. Objetivos

Para llevar a cabo nuestro objetivo principal desarrollaremos una aplicación web funcional que emule el concurso “¿Quién quiere ser millonario?”. Esto implica la implementación completa del juego, abarcando desde la interfaz de usuario hasta la lógica del juego y la gestión de datos. Específicamente, los objetivos que nos marcamos se enumeran a continuación:

1. Implementar la interfaz web: se creará una interfaz de usuario intuitiva y atractiva que permita a los usuarios interactuar con el juego de manera similar al programa de televisión.
2. Desarrollar la lógica del juego: se implementará la mecánica del concurso, incluyendo la presentación de preguntas con diferentes grados de dificultad, la gestión de las respuestas de los usuarios, la simulación de los comodines (50:50, llamada a un amigo, pregunta al público).
3. Diseñar e implementar una base de datos: se creará una base de datos para almacenar información relevante para el jugador.
4. Integrar un LLM para la generación de preguntas: se explorará el uso de un Modelo Extenso de Lenguaje (LLM) para la generación dinámica de preguntas, lo que permitirá ofrecer una experiencia de juego variada y desafiante. Se investigará la forma de ajustar la dificultad de las preguntas generadas por el LLM.
5. Incorporar elementos de Inteligencia Artificial: se investigará la posibilidad de incorporar técnicas de Inteligencia Artificial para mejorar la experiencia del usuario, como por ejemplo, en la selección de preguntas según el nivel del jugador o en la simulación del comportamiento del público en el comodín de la pregunta al público.

En resumen, este TFG se propone crear una aplicación web funcional y atractiva que recree el concurso anteriormente mencionado, utilizando tecnologías modernas como Python, Reflex y LLMs y explorando la aplicación de técnicas de Inteligencia Artificial para mejorar la experiencia del usuario.

## 1.3. Plan de trabajo

### Fase 0: Gestión del Proyecto (Continua)

- Objetivo: definición de tareas transversales de organización, comunicación y seguimiento.
- Inicio estimado: diciembre 2024 (02/12/2024).
- Fin estimado: mayo 2025 (06/05/2025).
- Herramientas:
  - GitHub: configuración inicial del repositorio (Diciembre 2024). Uso continuo con rama `main` estable.
  - Discord: uso continuo para comunicación del equipo y reuniones técnicas programadas.
  - Visual Studio Code: uso continuo para implementación.
  - Overleaf: inicio de estructura de la memoria (Enero 2025). Redacción progresiva.
  - Trello: configuración del tablero (Diciembre 2024). Actualización semanal de tareas.
- Entregables Continuos: actualizaciones en Trello, *commits* regulares en Git.

### Fase 1: Investigación y Análisis

- Objetivo: definición de tecnologías, *APIs* de IA (Gemini/DeepSeek) y estructura de la base de datos (PostgreSQL).
- Inicio estimado: diciembre 2024 (02/12/2024).
- Fin estimado: enero 2025 (24/01/2025).
- Duración estimada: entre 7 y 8 semanas (considerando posibles festivos de diciembre/enero).
- Tareas:
  - Investigar modelos de IA para generación de preguntas y comodines.
  - Investigar estado del arte en cuanto a tecnologías relacionadas.
  - Diseñar esquema de Base de datos (usuarios, puntuaciones, partidas).

- Documentar requisitos funcionales (modos de juego, flujos).
- Entregable parcial: documento de requisitos y especificaciones iniciales. Esquema preliminar de la BD.

## Fase 2: Diseño Arquitectónico

- Objetivo: definir componentes principales: *Frontend* (**Reflex**), *Backend* (`game_state`), integración con *APIs*.
- Inicio estimado: enero 2025 (27/01/2025).
- Fin estimado: marzo 2025 (07/02/2025).
- Duración estimada: 6 semanas.
- Tareas:
  - Definir arquitectura *Frontend* (**Reflex**): páginas para login, menú.
  - Definir arquitectura *Backend*: `game_state` (gestión de estado).
  - Planificar integración con *APIs* (Google Gemini, DeepSeek, OpenAI, Llama).

## Fase 3: Implementación por Módulos

- Objetivo: desarrollo de los componentes del sistema.
- Inicio estimado: febrero 2025.
- Fin estimado (funcionalidades principales): abril 2025.
- Duración estimada: aproximadamente 11 semanas.
- Tareas (fase paralela):
  - Autenticación/DB: Login/registro (Passlib más SQLAlchemy).
  - Core del juego: Clase `game_state` con generación de preguntas y comodines (integración inicial *APIs*).
  - Interfaz: Páginas Reflex (`login_page.py`, `menu_page.py`, `game_page.py`, `leaderboard_page.py`, etc.).
  - Sincronización del equipo: reuniones técnicas para integración y revisión de avances (febrero - abril 2025).
- Entregables parciales: módulos funcionales subidos a ramas *feature* y fusionados a `main` tras revisión.

## Fase 4: Pruebas Iterativas

- Objetivo: asegurar la calidad, funcionalidad y usabilidad del software.
- Inicio estimado (pruebas unitarias): febrero 2025 – en paralelo con la implementación.
- Fin estimado (ciclo principal de pruebas): abril 2025.
- Duración estimada: continuo, con picos de intensidad.
- Tareas:
  - Pruebas unitarias: validar lógica de preguntas, comodines y flujos (conforme se desarrolla cada módulo, Febrero - Abril 2025).
  - Pruebas de integración: conexión *frontend-backend-APIs* (marzo - abril 2025, a medida que los módulos se completan).
  - Registro: documentar *bugs* y arreglarlos (continuo hasta 30/04/2025).
- Entregables parciales: informes de *bugs* con código corregido en github.

## Fase 5: Memoria y Documentación

- Objetivo: redactar la memoria técnica completa del proyecto.
- Inicio estimado (redacción de secciones): marzo 2025 – en paralelo con las fases finales de implementación y pruebas.
- Fin estimado (memoria completa para revisión final): mayo 2025.
- Duración estimada: aproximadamente 9 semanas (redacción progresiva, intensificándose hacia el final)
- Tareas:
  - Redactar memoria técnica:
    - Explicación del estado del arte, tecnologías usadas (Enero - Febrero 2025).
    - Justificación de tecnologías (Marzo 2025).
    - Descripción detallada de la arquitectura e implementación (Abril 2025).
    - Resultados de pruebas y conclusiones (Principios Mayo 2025).
    - Redacción de mejoras futuras (Finales de Abril 2025).

## 1.4. Estructura del documento

El cuerpo principal del documento se divide en cinco capítulos claramente diferenciados.

- **Capítulo 1. Introducción.** Establece el marco del proyecto, presentando la motivación, los objetivos específicos perseguidos, el plan de trabajo seguido y, finalmente, una breve descripción de la estructura de este mismo documento.
- **Capítulo 2. Estado del Arte.** Realiza una revisión exhaustiva de los conceptos y tecnologías relevantes. Se abordan referencias clave, proyectos previos significativos como QANTA, Watson en Jeopardy! y LaMDA, tecnologías subyacentes como los modelos Transformer, BERT y RoBERTa, los desafíos actuales en el campo y protocolos emergentes como MCP.
- **Capítulo 3. Descripción del Trabajo.** Detalla en profundidad el desarrollo práctico del proyecto. Explica el *framework* utilizado (Reflex), la base de datos implementada (PostgreSQL), la estructura organizativa del código fuente, las diversas funcionalidades y modos de juego (Infinito, Normal, Temático, Máquina vs Máquina), el sistema de autenticación de usuarios, la arquitectura reactiva de la aplicación y la integración específica con las diferentes *APIs* de Inteligencia Artificial (Gemini, DeepSeek, OpenAI, Llama).
- **Capítulo 4. Contribuciones Personales.** Describe de manera individualizada las aportaciones y responsabilidades de cada uno de los autores del proyecto.
- **Capítulo 5. Conclusiones y Trabajo Futuro.** Sintetiza los resultados obtenidos, expone las conclusiones derivadas del desarrollo y la investigación, y propone posibles líneas de mejora y expansión futura del proyecto.

# Capítulo 2

## Estado del Arte

Los Modelos Extensos del Lenguaje (LLMs) han revolucionado la forma en que interactuamos con la información y han demostrado un potencial significativo en el ámbito de los concursos de preguntas y respuestas. Su capacidad para comprender el lenguaje natural, generar respuestas coherentes y acceder a vastas cantidades de conocimiento los convierte en herramientas valiosas para este tipo de aplicaciones.

### 2.1. Referencias Relevantes

- TriviaQA<sup>1</sup>: un conjunto de datos a gran escala que contiene preguntas y respuestas de trivia, ha sido fundamental para entrenar y evaluar LLMs en tareas de preguntas y respuestas (Joshi et al., 2017).
- Quiz Bowl: este formato de concurso de preguntas y respuestas ha servido como banco de pruebas para LLMs, desafiándolos a responder preguntas complejas y matizadas (Wikipedia, Quiz bowl).
- Jeopardy!: el icónico programa de concursos ha inspirado la creación de conjuntos de datos y entornos de simulación para LLMs, impulsando la investigación en comprensión del lenguaje natural y recuperación de información (Wikipedia, Jeopardy!).
- Google AI: ha desarrollado modelos como LaMDA (Wikipedia, LaMDA!), que han demostrado un rendimiento impresionante en tareas de preguntas y respuestas, incluyendo la capacidad de generar explicaciones y razonamientos.
- OpenAI<sup>2</sup>: ha creado modelos como GPT-3 y GPT-4, que han sido utilizados en diversas aplicaciones de preguntas y respuestas, desde chatbots hasta sistemas de tutoría virtual. GPT-3 ha demostrado capacidades sorprendentes para generación de texto, comprensión del lenguaje natural y la respuesta a preguntas, incluso sin entrenamiento específico para cada una (Brown et al., 2020).

---

<sup>1</sup><https://nlp.cs.washington.edu/triviaqa/>

<sup>2</sup><https://openai.com/>

- Meta AI<sup>3</sup>: ha investigado en modelos como BlenderBot y Llama, que han mostrado avances en la comprensión del contexto y la generación de respuestas más humanas.
- Generación de Preguntas: los LLMs no solo se utilizan para responder preguntas, sino también para generarlas. Esto abre la posibilidad de crear concursos de preguntas y respuestas dinámicos y personalizados.
- Razonamiento y Explicación: se busca que los LLMs no solo ofrezcan respuestas, sino que también sean capaces de explicar su razonamiento y proporcionar evidencia que respalde sus respuestas.
- Personalización: los LLMs pueden adaptarse a las preferencias y al nivel de conocimiento de cada usuario, lo que permite crear experiencias de aprendizaje y entretenimiento más personalizadas.

## 2.2. Proyectos desarrollados

La capacidad de las máquinas para comprender y responder preguntas ha sido un objetivo fundamental de la inteligencia artificial (*IA*) durante décadas. En los últimos años, hemos sido testigos de avances significativos en este campo, impulsados en gran parte por el desarrollo de modelos de *modelo extenso de lenguaje* (LLMs) y técnicas de *deep learning*. Estos avances han permitido la creación de sistemas de respuesta a preguntas (QA) cada vez más sofisticados, capaces de competir con humanos en concursos de trivia y juegos de conocimiento.

En este apartado, exploraremos diversos proyectos que han abordado el desafío de construir sistemas de QA robustos y competitivos, centrándonos en aquellos que utilizan LLMs y técnicas de IA avanzadas. Analizaremos sus enfoques, arquitecturas y resultados, destacando sus fortalezas y limitaciones.

### 2.2.1. QANTA Project

El Proyecto QANTA<sup>4</sup>, alojado en la Universidad de Maryland, representa un esfuerzo innovador en el campo de la Inteligencia Artificial aplicada a la respuesta a preguntas. Su enfoque central radica en el uso de competiciones de trivia como banco de pruebas para sistemas de QA, tanto humanos como computacionales.

#### Objetivos y enfoque

El objetivo primordial de QANTA es impulsar el avance en la capacidad de las máquinas para comprender y responder preguntas complejas. Para ello, se basa en tres pilares fundamentales:

---

<sup>3</sup><https://www.meta.ai/>

<sup>4</sup><https://sites.google.com/view/qanta/home>

1. Aprendizaje a partir de expertos humanos: busca extraer conocimiento valioso sobre cómo los expertos en trivia abordan y responden preguntas. Se analizan sus estrategias, procesos de razonamiento y uso del lenguaje natural para identificar patrones y técnicas que puedan ser implementados en sistemas de IA.
2. Comprensión de la explicación en interacción humano-computadora: explora cómo mejorar la colaboración entre humanos y computadoras en tareas de QA. Se investiga cómo los humanos pueden explicar sus respuestas a las máquinas y cómo las máquinas pueden proporcionar explicaciones comprensibles a los humanos.
3. Creación eficiente de preguntas desafiantes: se enfoca en desarrollar herramientas y metodologías que faciliten la creación de preguntas de trivia desafiantes e interesantes. Esto incluye la identificación de temas relevantes, la formulación de preguntas ambiguas y la evaluación de la dificultad de las preguntas.

El proyecto QANTA se basa en una combinación de técnicas y herramientas de diversas áreas de la IA, incluyendo:

1. Recuperación de información (IR): se emplean técnicas de IR para buscar y recuperar información relevante de grandes bases de datos y corpus de texto, es decir, permite a los investigadores y a los sistemas de IA acceder y utilizar grandes cantidades de información para mejorar la capacidad de respuesta a preguntas.
2. Aprendizaje automático (Machine Learning): se aplican algoritmos de ML para entrenar modelos de QA que puedan aprender a responder preguntas de manera precisa y eficiente.
3. Interfaces de usuario (UI): se desarrollan interfaces intuitivas y fáciles de usar para facilitar la interacción entre humanos y computadoras en las competiciones de trivia.

La metodología fundamental de QANTA se basa en un ciclo iterativo de desarrollo y evaluación:

- Recopilación de datos: se recopilan conjuntos de datos de preguntas y respuestas de trivia, tanto de fuentes existentes como de competiciones organizadas por el proyecto.
- Desarrollo de sistemas: se construyen sistemas de QA que combinan técnicas de NLP, IR y ML.
- Competiciones: se organizan competiciones de trivia entre sistemas de IA y expertos humanos.
- Análisis de resultados: se analizan los resultados de las competiciones para identificar fortalezas y debilidades de los sistemas de IA y extraer conocimiento sobre las estrategias de los expertos humanos.

- Mejora de sistemas: se utilizan los resultados del análisis para mejorar los sistemas de IA y desarrollar nuevas técnicas y herramientas.

En un nuevo enfoque para la evaluación de modelos de *procesamiento del lenguaje natural* (*NLP*). En lugar de generar ejemplos de ataque automáticamente, se propone un sistema híbrido humano-computadora. Este sistema permite que humanos, específicamente aficionados a concursos de preguntas y respuestas tipo *Quizbowl*, creen preguntas difíciles para los modelos. Una interfaz interactiva muestra interpretaciones de las predicciones del modelo, guiando a los humanos en la creación de preguntas que los confunden. Estas preguntas, aunque parecen normales para los humanos, engañan sistemáticamente a los modelos neuronales y de recuperación de información. El conjunto de preguntas resultante abarca diversos desafíos, como:

- Razonamiento *multihop*: capacidad de un sistema para responder preguntas integrando información de múltiples fuentes mediante una secuencia de inferencias lógicas.
- Distractores de tipo de entidad: opciones de respuesta incorrectas en preguntas de IA QA que comparten la misma categoría general de entidad que la respuesta correcta, pero no son la entidad específica buscada.

Ambos exponen desafíos en la comprensión robusta del lenguaje. Las preguntas son validadas en partidas en vivo entre humanos y computadoras, donde los modelos, a pesar de vencer a equipos humanos de alto nivel, fallan estrepitosamente ante las preguntas. Este enfoque revela que los modelos pueden tener éxito sin una verdadera comprensión del lenguaje, aprendiendo patrones superficiales. El nuevo conjunto de datos representa un recurso desafiante y diverso para futuras investigaciones en NLP.

La generación automática de ejemplos en *NLP* es difícil sin alterar el significado o la validez de los ejemplos. Trabajos recientes se centran en transformaciones simples que preservan el significado, como reemplazar “*What has*” con “*What’s*” o añadir *oraciones distractoras*. Si bien los modelos generativos pueden descubrir perturbaciones, requieren verificación humana posterior. Estos métodos mejoran la diversidad. Por lo tanto, este trabajo propone un enfoque híbrido humano-computadora, donde humanos escriben preguntas, guiados por las predicciones e interpretaciones del modelo. Esta interfaz interactiva facilita la creación de ejemplos que confunden al modelo pero pueden ser respondidas por humanos. Este enfoque puede reemplazar o complementar los métodos de generación basados en modelos, ya que crear interfaces e interpretaciones es a menudo más fácil que entrenar modelos generativos. Además, la creatividad humana puede revelar tácticas que los enfoques automáticos pueden imitar.

*Quizbowl*, un formato de preguntas y respuestas académicas, se utiliza como banco de pruebas para evaluar modelos de *NLP*. A diferencia de *Jeopardy!*, las preguntas de *Quizbowl* están diseñadas para ser interrumpidas, comenzando con pistas difíciles y volviéndose progresivamente más fáciles. Las respuestas suelen ser entidades conocidas, lo que se conoce como *QA factoidal*. Si bien algunas preguntas son fáciles para las computadoras debido a “*palabras desencadenantes*”, otras son difíciles debido a:

- Patrones de correferencia.
- Razonamiento entre conocimientos.
- Juegos de palabras.

Se realizaron dos rondas de escritura adversarial: la primera atacó un sistema de recuperación de información (*IR*) y la segunda, un modelo neuronal QA. Para ayudar a los autores, se utilizaron mapas que muestran la importancia de cada palabra para la predicción del modelo, basados en la *expansión de Taylor de primer orden* para el modelo neuronal y la *API Highlight de Elasticsearch* para el modelo *IR*. Se amplió el conjunto de datos para la segunda ronda para incluir respuestas más diversas.

La interfaz permite a los autores interactuar con los modelos *IR* o RNN (*Recurrent Neural Network*). Escriben la pregunta en la parte superior derecha y ven las cinco mejores predicciones del modelo en la parte superior izquierda. La interfaz indica cuándo el modelo acierta por primera vez. El objetivo es que el modelo falle o retrase la respuesta correcta lo más posible. Las palabras de la pregunta se resaltan según su importancia para el modelo. Los autores pueden enviar cualquier pregunta, incluso si no *rompe* el sistema. Se registra el historial de edición para analizar los fallos recurrentes del modelo y la influencia de las interpretaciones. Los autores son miembros de la comunidad *Quizbowl*, con conocimientos profundos y experiencia en la creación de preguntas. Las preguntas siguen una estructura común, están bien editadas y terminan con una pista sencilla. Estas restricciones aseguran que los ejemplos adversariales sean difíciles.

### 2.2.2. Watson: ‘Jeopardy!’

En un episodio del concurso *Jeopardy!* televisado y visto por millones de espectadores en febrero de 2011, la computadora *DeepQA* de IBM llamada *Watson* hizo historia al derrotar a los dos campeones más destacados del programa de concursos: *Brad Rutter* y *Ken Jennings*. Nombrada en honor al primer CEO de IBM, *Watson* es un sistema informático de respuesta a preguntas desarrollado por un equipo de investigación de IBM dirigido por el investigador principal *David Ferrucci*. Fue la punta de lanza de una nueva generación de computadoras capaces de comprender preguntas formuladas en lenguaje natural y responderlas con mucha más precisión que cualquier tecnología de búsqueda estándar, sin estar conectadas a Internet<sup>5</sup>.

La capacidad de *Watson* para descubrir información en datos no estructurados representó un gran avance en el *procesamiento del lenguaje natural* (NLP) y un paso importante hacia un mundo en el que las inteligencias artificiales pueden comprender y responder a las preguntas cotidianas para mejorar la toma de decisiones. Desde su victoria en *Jeopardy!*, *Watson* ha tenido un impacto de gran alcance en la industria y la sociedad, con aplicaciones como:

- Análisis de imágenes de satélites para mejorar los esfuerzos de conservación.
- Dotar a los centros de atención al cliente con respuestas más efectivas.

---

<sup>5</sup><http://ibm.com/es-es/watson>

IBM había estado buscando un nuevo desafío humano contra máquina desde que su ordenador *Deep Blue* derrotó al campeón de ajedrez *Garry Kasparov* en un histórico partido en 1997. En 2006, *Ferrucci*, un científico informático de IBM con experiencia en *IA*, propuso a sus jefes la idea de crear una computadora que pudiera vencer a un humano en *Jeopardy!*, considerado como uno de los programas de concursos más desafiante de la televisión (Ferrucci et al., 2010).

*Ferrucci* pensó que construir una computadora para competir en un juego de preguntas y respuestas podría empujar los límites del procesamiento del lenguaje natural, en el que las computadoras están programadas para analizar y responder a palabras y frases comunes. “*El objetivo no es modelar el cerebro humano*”, dijo, “*El objetivo es construir una computadora que pueda ser más efectiva en la comprensión e interacción en lenguaje natural, pero no necesariamente de la misma manera que lo hacen los humanos*”.

*Ferrucci* reunió a un equipo de científicos, ingenieros y programadores en el centro de investigación *Thomas J. Watson* en *Yorktown Heights, Nueva York*. Entre los programadores asignados al proyecto se encontraba *Ed Toutant*, quien había ganado 10,000 USD en *Jeopardy!* en 1989. El equipo tardó cinco años en perfeccionar el sistema de respuesta a preguntas.

El *Watson* original era una computadora del tamaño de una habitación que constaba de 10 racks que contenían 90 servidores, con un total de 2,880 núcleos de procesador. Funcionaba con la tecnología de software *DeepQA* de IBM, que utilizaba para:

1. Generar *hipótesis*.
2. Recopilar *evidencia*.
3. Analizar *datos*.

Durante un período de varios años, *Watson* tomó muchísima información de *Wikipedia*, enciclopedias, diccionarios, textos religiosos, novelas, obras de teatro y libros del *Proyecto Gutenberg*<sup>6</sup>, entre otras fuentes.

A diferencia de los motores de búsqueda, que analizan palabras clave básicas y devuelven una lista de documentos relacionados, *Watson* puede comprender preguntas formuladas en lenguaje natural y devolver respuestas directas. La principal innovación de *Watson* se centró en su capacidad para ejecutar rápidamente cientos de algoritmos para:

- Analizar simultáneamente una pregunta desde distintos enfoques.
- Encontrar y calificar posibles respuestas.
- Recopilar evidencia de respaldo adicional.
- Evaluar todo utilizando *procesamiento del lenguaje natural*.

---

<sup>6</sup><https://www.gutenberg.org/browse/languages/es>

Cuanto más algoritmos llegaban independientemente a la misma respuesta, mayor era el nivel de confianza de *Watson*. Si el nivel de confianza era lo suficientemente alto, *Watson* estaba programado para pulsar el botón durante un juego de *Jeopardy!*. Si no, *Watson* no pulsaba. Todos estos cálculos los realizaba en aproximadamente tres segundos.

*Watson* fue puesto a prueba en dos partidos de *Jeopardy!* jugados durante tres días. No era perfecto. En el primer partido falló la pista de *Jeopardy! Final* en la categoría de *Ciudades de EE. UU.* (“*Su aeropuerto más grande fue nombrado en honor a un héroe de la Segunda Guerra Mundial; su segundo más grande, en honor a una batalla de la Segunda Guerra Mundial*”). La respuesta correcta era “¿*Chicago?*”, pero *Watson* respondió: “¿*Toronto?????*” con cinco signos de interrogación que indicaban una falta sustancial de confianza.

Entró en el segundo partido empatado con *Brad Rutter*, pero rápidamente tomó una ventaja dominante. El resultado final terminó con una victoria rotunda. *Watson* ganó USD 77,147, que fueron donados a varias organizaciones benéficas, superando los USD 24,000 de *Ken Jennings* y los USD 21,600 de *Brad Rutter*. Después del concurso, *Jennings* comentó irónicamente: “*Ser concursante de programas de preguntas y respuestas puede ser el primer trabajo que Watson deje obsoleto, pero estoy seguro de que no será el último*”.

Desde la victoria de *Watson* en *Jeopardy!*, la tecnología subyacente ha ayudado a las organizaciones a predecir, optimizar y automatizar procesos de negocio en numerosas industrias. Algunos ejemplos incluyen:

- Aproximadamente el 70% de las instituciones financieras globales utilizan *Watson*.
- En un proyecto con *The Weather Company*, *Watson* proporciona pronósticos hiperlocales hasta el nivel de vecindario o incluso de calle.
- *OmniEarth*<sup>7</sup>, un proveedor de observación y análisis global de la Tierra, ha utilizado los servicios de visión por computadora de *Watson* para analizar imágenes de satélites y aéreas, midiendo el uso de agua propiedad por propiedad, ayudando a los distritos de agua en *California*, afectados por la sequía, a mejorar los esfuerzos de conservación (Griggs, 2021).

### 2.2.3. LaMDA de Google

*LaMDA*, que significa *Language Model for Dialogue Applications* (Modelo de Lenguaje para Aplicaciones de Diálogo), es una familia de modelos de lenguaje neuronal conversacional desarrollada por Google. Es un modelo de lenguaje de última generación diseñado específicamente para mantener conversaciones naturales y fluidas. Su arquitectura se basa en la red neuronal *Transformador* (explicado en la sección 2.3), lo que le permite procesar grandes cantidades de texto y aprender patrones lingüísticos complejos. Ha sido entrenado con un conjunto masivo de datos de conversaciones humanas, lo que le permite generar respuestas coherentes, relevantes y con un buen contexto (Thoppilan y Others, 2022).

<sup>7</sup>[www.ai-for-sdgs.academy/case/334](http://www.ai-for-sdgs.academy/case/334)

Las características principales de LaMDA:

- Capacidad de diálogo: está diseñado para mantener conversaciones abiertas y variadas, respondiendo a preguntas, siguiendo el hilo de la conversación y generando respuestas que parecen humanas.
- Comprensión del contexto: puede entender el contexto de una conversación, lo que le permite generar respuestas relevantes y coherentes a lo largo del tiempo.
- Generación de respuestas naturales: las respuestas son gramaticalmente correctas, semánticamente coherentes y con un contexto apropiado.
- Aprendizaje continuo: puede aprender y mejorar continuamente a medida que se le proporcionan más datos de entrenamiento.

### Aplicaciones

Tiene el potencial de revolucionar la forma en que interactuamos con la tecnología. Algunas de sus posibles aplicaciones incluyen *Chatbots* y asistentes virtuales; traducción automática (comprendiendo mejor el contexto); generación de contenido (generación de texto creativo); educación (tutores virtuales más interactivos y personalizados).

Hay controversia en torno a LaMDA, en junio de 2022, LaMDA llamó la atención cuando un ingeniero de Google, *Blake Lemoine*, afirmó que el *chatbot* había cobrado conciencia (Biurrun, 2022). Estas afirmaciones fueron ampliamente rechazadas por la comunidad científica, pero generaron un debate sobre la naturaleza de la conciencia y la capacidad de las máquinas para pensar.

## 2.3. Transformador (modelo de aprendizaje automático)

Los transformadores constituyen una arquitectura de red neuronal que ha marcado una revolución en el ámbito del procesamiento del lenguaje natural (NLP). Estos modelos emplean un mecanismo de atención interna o auto-focalización, el cual les facilita el procesamiento paralelo de la información de entrada y la identificación de dependencias a largo alcance entre los elementos de una secuencia (Vaswani et al., 2017).

### Características

- Autoatención o mecanismo de auto-focalización: a diferencia de los modelos de red neuronal recurrentes (RNN) que procesan datos secuencialmente, los Transformadores utilizan la auto-focalización para ponderar la importancia de diferentes palabras en una secuencia y capturar relaciones entre ellas, incluso si están muy separadas. Esto permite que el modelo entienda mejor el contexto y la semántica del texto.

- Procesamiento en paralelo: la arquitectura permite procesar datos en paralelo, lo que acelera significativamente el entrenamiento y la inferencia en comparación con los modelos RNN que procesan datos secuencialmente.
- Codificador-decodificador: muchos utilizan una arquitectura de codificador - decodificador. El codificador procesa la secuencia de entrada y el decodificador genera la secuencia de salida. Esta arquitectura es especialmente útil para tareas como la traducción automática y la generación de texto (Wikipedia, Transformador (modelo de aprendizaje automático)).

## Aplicaciones

Han demostrado ser muy efectivos en una variedad de tareas de NLP, incluyendo:

- Traducción automática: han mejorado significativamente la calidad de la traducción automática, permitiendo la creación de modelos como Google Translate.
- Generación de texto: pueden generar texto coherente y natural, lo que ha llevado a la creación de modelos como GPT-3 y LaMDA.
- Comprensión del lenguaje natural: han mejorado la precisión de las tareas de comprensión del lenguaje natural, como la clasificación de texto, la extracción de entidades y el análisis de sentimientos.
- Respuesta a preguntas: se utilizan en sistemas de respuesta a preguntas para entender la pregunta y encontrar la respuesta relevante en un texto.

**Ventajas:** mayor precisión (han demostrado ser más precisos que los modelos RNN en muchas tareas de NLP); mayor velocidad (el procesamiento en paralelo permite un entrenamiento y una inferencia más rápidos); Mejor captura de relaciones a largo plazo (la autoatención permite capturar relaciones entre palabras en secuencias largas).

**Desventajas:** mayor precisión (han demostrado ser más precisos que los modelos RNN en muchas tareas de PNL); mayor velocidad (el procesamiento en paralelo permite un entrenamiento y una predicción más rápidos); Mejor captura de relaciones a largo plazo (la autoatención permite capturar relaciones entre palabras en secuencias largas).

### 2.3.1. BERT

BERT (*Bidirectional Encoder Representations from Transformers*) es un modelo de lenguaje pre-entrenado desarrollado por Google que utiliza la arquitectura de los transformadores. Su principal innovación radica en su capacidad para aprender representaciones contextualizadas de palabras, lo que significa que el significado de una palabra se determina en función de las palabras que la rodean en una oración.

Se basa en un enfoque de “aprendizaje no supervisado”, lo que significa que se entrena en grandes cantidades de texto sin necesidad de etiquetas o anotaciones

manuales. Esto le permite aprender patrones lingüísticos complejos y desarrollar una comprensión profunda del lenguaje natural (Devlin et al., 2018).

### 2.3.1.1. Arquitectura de BERT

Se basa en la arquitectura del modelo Transformador, que se caracteriza por el uso de mecanismos de autoatención. La autoatención permite que el modelo capture las relaciones entre todas las palabras de una oración de manera simultánea, en lugar de secuencial, como lo hacen las redes neuronales recurrentes (RNN).

BERT se compone de dos partes principales (Wikipedia, 2019):

1. El codificador toma la secuencia de entrada (oración) y la transforma en una serie de vectores de representación con contexto. Cada vector representa una palabra y su contexto.
2. El decodificador utiliza los vectores de representación contextualizada para realizar diversas tareas de NLP, como la clasificación de texto, la respuesta a preguntas o la generación de texto.

BERT se distingue por ser un modelo “bidireccional”, lo que significa que considera el contexto tanto a la izquierda como a la derecha de una palabra al calcular su representación. Esto le permite capturar relaciones lingüísticas más ricas y mejorar el rendimiento en diversas tareas. BERT ha demostrado ser muy efectivo en una amplia gama de aplicaciones de NLP, incluyendo:

- Clasificación de texto: BERT puede utilizarse para clasificar textos en diferentes categorías, como análisis de sentimientos, detección de spam o clasificación de temas.
- Respuesta a preguntas: puede utilizarse para encontrar la respuesta a una pregunta en un texto dado.
- Reconocimiento de entidades nombradas: puede usarse para identificar y clasificar entidades nombradas en un texto, como personas, organizaciones o lugares.
- Traducción automática: puede usarse para mejorar la calidad de la traducción automática al conseguir un entendimiento mejorado del contexto.

### 2.3.2. roBERTa

roBERTa (*A Robustly Optimized BERT Pretraining Approach*) es un modelo de lenguaje pre-entrenado desarrollado por *Facebook AI Research (FAIR)* que se basa en la arquitectura BERT, pero con optimizaciones significativas en el proceso de pre-entrenamiento. Su objetivo principal es mejorar el rendimiento de BERT en diversas tareas de procesamiento del lenguaje natural (NLP). Tiene un entrenamiento más robusto, entrenado con un conjunto de datos mucho más grande y diverso que BERT, lo que le permite aprender patrones lingüísticos más ricos y generales. Además, tiene un ajuste de hiperparámetros de forma que se usa un conjunto de hiperparámetros

optimizados, incluyendo un tamaño de lote mayor y una secuencia de entrada más larga, que mejora la eficiencia del entrenamiento y el rendimiento del modelo (Liu et al., 2019). Se basa en la misma arquitectura del modelo *Transformador* que BERT. Sus dos partes principales también son el *Codificador* y el *Decodificador*.

A continuación enumeramos las mejoras que presenta roBERT (NovitaAI, 2024):

- Implicación textual (TE) o inferencia del lenguaje natural (NLI): ha demostrado un rendimiento superior a BERT en tareas de NLI (*Natural Language Inference*), que implica determinar la relación lógica entre dos oraciones (si una oración implica, contradice o es neutral con respecto a la otra). La mayor capacidad de roBERTa para comprender el contexto y las relaciones semánticas lo hace más efectivo en este tipo de tareas.
- Parfraseo y detección de duplicados: más eficaz para identificar si dos oraciones tienen el mismo significado, aunque estén redactadas de forma diferente. También es mejor para detectar duplicados en grandes cantidades de texto. Esto es útil en aplicaciones como la organización de contenido, la detección de plagio o la mejora de la precisión de los motores de búsqueda.
- Análisis de sentimientos y emociones: ofrece una mayor precisión en la identificación de emociones y matices sutiles en el lenguaje. Esto es valioso en áreas como el análisis de la opinión del cliente, la monitorización de redes sociales o el estudio del comportamiento del consumidor.
- Sistemas de diálogo: puede mejorar la calidad de los sistemas de diálogo al permitir una mejor comprensión del lenguaje natural y una generación de respuestas más naturales y relevantes. Esto es crucial para crear chatbots y asistentes virtuales más conversacionales y útiles.

### Diferencias entre BERT y roBERTa.

La principal diferencia entre BERT y roBERTa radica en el proceso de entrenamiento. roBERTa se entrena con un conjunto de datos más grande y diverso, utiliza hiperparámetros optimizados y elimina la tarea de predicción de la siguiente oración. Estas optimizaciones permiten que roBERTa aprenda representaciones lingüísticas más robustas y generales, lo que se traduce en un mejor rendimiento en diversas tareas de NLP. En general, roBERTa suele ser una mejor opción que BERT para la mayoría de las aplicaciones de NLP.

## 2.4. Desafíos

A pesar de los avances, los *LLMs* aún enfrentan desafíos significativos en el contexto de concursos de preguntas y respuestas. Un estudio publicado en enero de 2025 reveló que los *chatbots* de inteligencia artificial todavía no pueden responder con precisión preguntas avanzadas de historia (Dolan, 2025).

Los investigadores probaron los modelos *GPT-4* de *OpenAI*, *Llama* de *Meta* y *Gemini* de *Google* utilizando una nueva referencia llamada *Hist-LLM*, basada en el

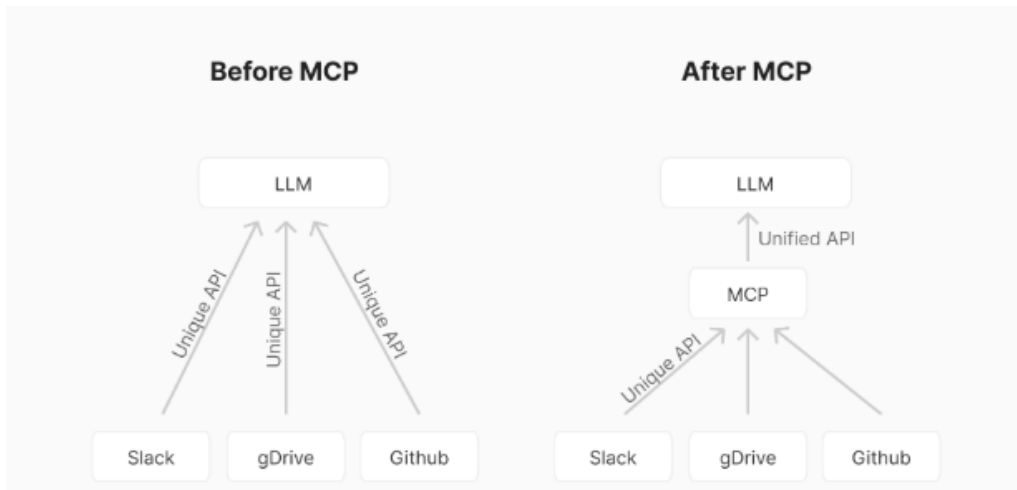


Figura 2.1: Cambios en las APIs con MCP

*Seshat Global History Databank*. Se encontró que el *GPT-4 Turbo* tuvo el mejor rendimiento con una precisión del 46 %, apenas mejor que el azar. Los modelos carecen de profundidad para consultas históricas detalladas y tienden a extrapolar datos, pero fallan mostrando posibles sesgo y limitaciones en los datos de entrenamiento, especialmente en regiones como *África subsahariana* (Zilber, 2025).

Además, un artículo publicado en septiembre de 2024 en *Nature* (Zhou et al., 2024) por investigadores de la *Universitat Politècnica de València* y la *Universidad de Cambridge* reveló que, aunque estos sistemas avanzan en tareas complicadas, a menudo fallan en tareas sencillas. Este incremento en la complejidad y la expectativa humana puede provocar mayor decepción al recibir respuestas incorrectas. Los modelos tienden a no admitir cuando no saben la respuesta, gastando a veces mucho tiempo y recursos en intentar resolver preguntas difíciles sin certeza. La investigación sugiere volver a diseñar estos sistemas para permitir opciones de rechazo en áreas críticas, como la medicina, y fomentar la supervisión humana para mitigar errores significativos. Sin embargo, los autores señalan que estos modelos no serán capaces de lograr una “superinteligencia” fiable para resolver la mayoría de las tareas de manera segura.

## 2.5. MCP (Model Context Protocol)

El Protocolo de Contexto de Modelos (MCP) es un estándar abierto presentado por Anthropic en noviembre del 2024 con el objetivo de estandarizar cómo las aplicaciones de IA (*chatbots*, asistentes de entornos de desarrollo o agentes personalizados) se conectan con herramientas externas, fuentes de datos y sistemas.

Tradicionalmente, la integración de  $M$  aplicaciones de IA con  $N$  sistemas externos (p. ej., *GitHub*, *Slack*, bases de datos) generaba un problema de complejidad  $M \times N$  (véase figura 2.1), requiriendo el desarrollo de integraciones específicas para cada par aplicación-sistema. Esto resultaba en esfuerzos redundantes y soluciones heterogéneas (Wikipedia, Protocolo de Contexto de Modelo).

El MCP aborda esta problemática al proponer una interfaz de programación de aplicaciones (API) común que reduce la complejidad a un problema  $M + N$ . En este enfoque, los desarrolladores de herramientas implementan  $N$  servidores MCP, cada uno correspondiente a un sistema externo, mientras que los desarrolladores de aplicaciones crean  $M$  clientes MCP, uno por cada aplicación de IA<sup>8</sup>. La arquitectura cliente-servidor del MCP se estructura en los siguientes componentes:

- **Hosts:** aplicaciones de usuario final que alojan la interacción con la IA, como entornos de desarrollo o agentes personalizados.
- **Cientes:** módulos integrados en el *Host* que gestionan una conexión 1:1 con un servidor MCP específico, facilitando la comunicación entre la aplicación y los servicios externos.
- **Servidores:** programas externos que exponen herramientas, recursos y *prompts* al modelo de IA mediante una API estandarizada.

Los servidores MCP proporcionan tres elementos principales:

1. **Herramientas (controladas por el modelo):** funciones que se invocan mediante modelos de lenguaje (LLMs) para ejecutar acciones específicas, como consultas a APIs externas (p. ej., una API meteorológica), equivalentes a llamadas a funciones.
2. **Recursos (controlados por la aplicación):** fuentes de datos accesibles por los LLMs, análogas a puntos finales *GET* en *APIs REST*, que proveen información sin realizar cálculos complejos ni generar efectos secundarios. Estos recursos se integran en el contexto de la solicitud.
3. **Prompts (controlados por el usuario):** plantillas predefinidas que optimizan el uso de herramientas y recursos, seleccionadas antes de la inferencia del modelo.

El MCP, al estandarizar estas interacciones, minimiza la redundancia en el desarrollo de integraciones y fomenta la interoperabilidad entre aplicaciones de IA y sistemas externos, promoviendo un ecosistema más eficiente y escalable.

Opera bajo un modelo cliente-servidor, diseñado para facilitar la integración de aplicaciones de inteligencia artificial (IA) con sistemas externos. A continuación, se describe de forma técnica y resumida el flujo operativo del protocolo (Schmid, 2025):

1. **Inicialización:** al iniciar una aplicación *Host*, esta crea  $N$  clientes MCP, cada uno dedicado a un servidor específico. Durante esta fase, los clientes y servidores intercambian información sobre capacidades y versiones del protocolo mediante un proceso de *handshake*.
2. **Descubrimiento:** Cada cliente solicita al servidor una lista de capacidades disponibles, que incluyen herramientas (funciones invocables), recursos (fuentes de datos) y *prompts* (plantillas predefinidas). El servidor responde con una lista detallada de estas capacidades.

---

<sup>8</sup><https://modelcontextprotocol.io/introduction>

3. Provisión de Contexto: la aplicación *Host* procesa las herramientas, recursos y *prompts*, poniéndolos a disposición del usuario o formateándolos (p. ej., en JSON para invocación de funciones) para su compatibilidad con un modelo de lenguaje (LLM).
4. Invocación: cuando el LLM identifica la necesidad de usar una herramienta, basándose en la solicitud del usuario, el *Host* instruye al cliente correspondiente para enviar una solicitud de invocación al servidor adecuado.
5. Ejecución: el servidor recibe la solicitud (p. ej., `fetch_github_issues` para el repositorio X), ejecuta la lógica subyacente (como una llamada a la API de *GitHub*) y obtiene el resultado.
6. Respuesta: el servidor envía el resultado al cliente.
7. Completado: el cliente retransmite el resultado al *Host*, que lo incorpora al contexto del LLM. Esto permite al modelo generar una respuesta final para el usuario, integrando la información externa actualizada.

Este flujo asegura una comunicación eficiente y estandarizada entre aplicaciones de IA y sistemas externos, reduciendo la complejidad de integración y habilitando interacciones dinámicas con herramientas y datos externos (FireShip, 2025).

## Servidores MCP

Los servidores MCP actúan como una interfaz entre el ecosistema MCP y las funcionalidades de sistemas externos, como APIs, bases de datos o archivos locales. Estos servidores encapsulan las capacidades externas conforme a la especificación MCP, exponiéndolas como herramientas, recursos o *prompts*. Pueden desarrollarse en diversos lenguajes (*Python*, *TypeScript*, *Java*, *Rust*, entre otros), siempre que soporten los métodos de comunicación definidos:

- *stdio* (entrada/salida estándar): utilizado cuando el cliente y el servidor se ejecutan en la misma máquina, ideal para integraciones locales, como acceso a archivos o ejecución de *scripts*.
- *HTTP* con *SSE* (*Server-Sent Events*): el cliente se conecta al servidor mediante HTTP, y el servidor envía mensajes asíncronos al cliente a través de una conexión persistente utilizando el estándar SSE.

Un ejemplo de implementación de un servidor MCP en Python con la biblioteca **FastMCP** incluye la definición de herramientas (funciones como sumar números), recursos (datos como saludos personalizados) y *prompts* (plantillas para revisar código). Existen numerosos servidores MCP predefinidos y desarrollados por la comunidad, disponibles en repositorios (KseniaSe, 2025).

## Cientes MCP

Los clientes MCP son componentes integrados en aplicaciones *Host*, como entornos de desarrollo (IDE), *chatbots* o agentes personalizados, y gestionan la comunicación con un servidor MCP específico. Sus funciones incluyen:

- Gestión de la conexión con el servidor.
- Descubrimiento de capacidades (herramientas, recursos, *prompts*).
- Reenvío de solicitudes y manejo de respuestas conforme a la especificación MCP.

Ejemplos de aplicaciones *Host* que incorporan clientes MCP incluyen aplicaciones de interfaz de usuario (*Claude Desktop*, *Microsoft Copilot Studio*, *LibreChat*), IDEs (*Cursor*, *Windsurf*, *Continue*, *Zed*) y agentes personalizados desarrollados con *frameworks* como *Firestore Genkit*, *LangGraph* o el *SDK* de agentes de *OpenAI*.

En conjunto, los servidores y clientes MCP habilitan una integración estandarizada y eficiente entre aplicaciones de IA y sistemas externos, reduciendo la complejidad y fomentando la interoperabilidad en el ecosistema de IA.

## Relevancia y Adopción del MCP

El protocolo ha ganado un notable seguimiento en 2025 debido a su diseño técnico y su enfoque en las necesidades de los agentes de inteligencia artificial (IA) modernos. A continuación, se resumen los factores clave que explican su relevancia y adopción:

- Diseño específico para IA: a diferencia de estándares previos como *OpenAPI*, *GraphQL* o *SOAP*, el MCP está optimizado para agentes de IA, estructurando las interacciones en: Herramientas (controladas por el modelo); recursos (controlados por la aplicación); *prompts* (controlados por el usuario).
- Estándar abierto con especificación robusta: cuenta con una especificación detallada que supera a otros estándares competidores con documentación menos completa, facilitando su adopción y desarrollo.
- Fundamentos consolidados: en lugar de crear un protocolo desde cero, *Anthropic* adaptó el *Language Server Protocol (LSP)* y *JSON-RPC 2.0*, aprovechando tecnologías probadas para garantizar estabilidad y compatibilidad.
- Ecosistema inicial sólido: *Anthropic* implementó y probó exhaustivamente el MCP (*dogfooding*) antes de su lanzamiento, proporcionando: Cliente (aplicación *Claude Desktop*); Servidores (implementaciones de referencia para sistemas como archivos locales, *Git* y *Slack*); herramientas (*MCP Inspector* para pruebas y documentación detallada); *SDKs* (bibliotecas en *Python*, *TypeScript*, *Java*, *Kotlin* y *C#*).
- Efectos de red y adopción comunitaria: la naturaleza abierta del MCP ha impulsado un ecosistema colaborativo. Herramientas como *Cursor* y *Windsurf*

han integrado el protocolo, mientras que empresas como *Composio* han desarrollado servidores preconfigurados para cientos de integraciones. *OpenAI* ha anunciado soporte para MCP, y la comunidad ha creado miles de servidores para plataformas como *GitHub*, *Slack*, bases de datos y *Docker*.

En conclusión, la combinación de un diseño orientado a IA, una especificación técnica robusta, fundamentos tecnológicos consolidados, un ecosistema inicial completo y una adopción comunitaria masiva ha posicionado al MCP como un estándar líder para la integración de aplicaciones de IA con sistemas externos(Leo, 2025).

## Ejemplo práctico

El siguiente código implementa un agente de IA que utiliza un cliente MCP para interactuar con un servidor MCP, permitiendo al modelo *Gemini* invocar herramientas externas, como una API de *Airbnb*, para procesar una solicitud de reserva. El flujo incluye inicialización del cliente, descubrimiento de herramientas, invocación de funciones y manejo de respuestas (Schmid, 2025).

```
1 from typing import List
2 from google import genai
3 from google.genai import types
4 from mcp import ClientSession, StdioServerParameters
5 from mcp.client.stdio import stdio_client
6 import os
7
8 client = genai.Client(api_key=os.getenv("GEMINI_API_KEY"))
9 model = "gemini-2.0-flash"
10
11 # Configuración del servidor MCP para conexión stdio
12 server_params = StdioServerParameters(
13     command="npx",
14     args=["-y", "@openbnb/mcp-server-airbnb"],
15     env=None
16 )
17
18 async def agent_loop(prompt: str, client: genai.Client, session: ClientSession):
19     contents = [types.Content(role="user", parts=[types.Part(text=prompt)])]
20     await session.initialize()
21
22     # Obtener herramientas del servidor MCP y convertirlas a formato Gemini
23     mcp_tools = await session.list_tools()
24     tools = types.Tool(function_declarations=[
25         {"name": tool.name, "description": tool.description, "parameters":
26         tool.inputSchema}
27         for tool in mcp_tools.tools
28     ])
29
30     # Solicitud inicial al modelo con el prompt y herramientas
```

```

31     response = await client.aio.models.generate_content(
32         model=model,
33         contents=contents,
34         config=types.GenerateContentConfig(temperature=0, tools=[tools])
35     )
36     contents.append(response.candidates[0].content)
37
38     # Bucle de invocación de herramientas
39     turn_count = 0
40     max_tool_turns = 5
41     while response.function_calls and turn_count < max_tool_turns:
42         turn_count += 1
43         tool_response_parts: List[types.Part] = []
44
45         # Procesar cada llamada a función
46         for fc_part in response.function_calls:
47             tool_name = fc_part.name
48             args = fc_part.args or {}
49             print(f"Intentando invocar herramienta MCP: '{tool_name}' con args:
50                 {args}")
51             try:
52                 tool_result = await session.call_tool(tool_name, args)
53                 print(f"Herramienta MCP '{tool_name}' ejecutada con éxito.")
54                 tool_response = {"error": tool_result.content[0].text} if
55                 tool_result.isError else {"result": tool_result.content[0].text}
56             except Exception as e:
57                 tool_response = {"error": f"Fallo en ejecución de herramienta:
58                 {type(e).__name__}: {e}"}
59
60             tool_response_parts.append(
61                 types.Part.from_function_response(name=tool_name, response=tool_response)
62             )
63
64         # Actualizar historial y realizar siguiente llamada al modelo
65         contents.append(types.Content(role="user", parts=tool_response_parts))
66
67         print(f"Añadidas {len(tool_response_parts)} respuestas de herramientas al
68             historial.")
69         response = await client.aio.models.generate_content(
70             model=model,
71             contents=contents,
72             config=types.GenerateContentConfig(temperature=1.0, tools=[tools])
73         )
74         contents.append(response.candidates[0].content)
75
76     if turn_count >= max_tool_turns and response.function_calls:
77         print(f"Máximo de iteraciones ({max_tool_turns}) alcanzado. Saliendo del
78             bucle.")
79

```

```
80     print("Bucle de invocación de herramientas MCP finalizado. Devolviendo
81     respuesta final.")
82     return response
83
84     async def run():
85         async with stdio_client(server_params) as (read, write):
86             async with ClientSession(read, write) as session:
87                 prompt = "Quiero reservar un apartamento en París por 2 noches. 28/03
88                 - 30/03"
89                 print(f"Ejecutando bucle de agente con prompt: {prompt}")
90                 res = await agent_loop(prompt, client, session)
91                 return res
92
93     res = await run()
94     print(res.text)
```

---

# Capítulo 3

## Descripción del Trabajo

El proyecto desarrolla una adaptación web del concurso “Quién Quiere Ser Millonario”, utilizando para ello Reflex y la arquitectura de la aplicación sigue una estructura clara separando la *interfaz de usuario*, implementada en los archivos `*_page.py`, y la gestión del estado y la lógica de la aplicación, ubicada en los `*_state.py`.

El proyecto parte de un archivo `requirements.txt` que especifica todas las dependencias necesarias, incluyendo *bibliotecas fundamentales* (Reflex, SQLAlchemy y Passlib) y Herramientas para integración con *APIs* de inteligencia artificial: (Google Generative AI y Requests)

El estado `GameState` centraliza toda la información relevante durante una partida: Pregunta actual y opciones de respuesta; dificultad y número de pregunta; estado de los comodines (`fifty_used`, `public_used`, `call_used`); mensajes de feedback para el jugador.

La lógica principal del juego se encapsula en la clase `GameState`, que incorpora funcionalidades avanzadas: la generación dinámica de preguntas mediante Google Gemini API, una simulación del comodín “*Pregunta al público*” usando modelos de IA y la implementación del comodín “*Llamada a un amigo*” (véase figura 3.1) mediante integración con DeepSeek API. La implementación de los comodines similares a los del programa de televisión original sigue un enfoque diferenciado:

```
@rx.event
def use_fifty_option(self):
    """Usa el comodín 50:50 para marcar dos respuestas incorrectas como erróneas."""
    if not self.fifty_used:
        opciones = ["A", "B", "C", "D"]
        incorrectas = [op for op in opciones if getattr(self, f"option_{op.lower()}") != self.correct]

        # Seleccionar dos opciones incorrectas al azar
        eliminadas = random.sample(incorrectas, 2)

        # Marcar los botones eliminados como incorrectos en gris
        for op in eliminadas:
            self.button_classes[op] = "hex-button disabled"

        self.fifty_used = True
    else:
        self.feedback = "❌ Ya has usado el comodín 50:50."
```

Figura 3.1: Parte de la implementación del comodín 50:50

- El 50:50 se resuelve principalmente a nivel de interfaz (*figura 3.1*).
- Los comodines Público y Llamada delegan su lógica a la clase `GameState`.

### 3.1. Framework Utilizado: Reflex

**Reflex** es un *framework* de código abierto diseñado para permitir a los desarrolladores construir aplicaciones web *full-stack* utilizando *Python*, eliminando la necesidad de aprender lenguajes como *JavaScript*, *HTML* o *CSS* para el desarrollo *frontend*.

Su historia comienza con la visión de simplificar el desarrollo web para aquellos que ya dominan *Python*, un lenguaje conocido por su simplicidad y versatilidad, especialmente en áreas como la *inteligencia artificial* o el desarrollo *backend*.

Antes de **Reflex**, los desarrolladores que querían crear aplicaciones web, solían recurrir a *frameworks* como *Django* o *Flask* para el *backend*, pero aún necesitaban integrar tecnologías *frontend* como *React* o *Vue.js*, lo que implicaba aprender un nuevo lenguaje y manejar la comunicación entre *front* y el *back*. Esta división generaba una curva de aprendizaje pronunciada y ralentizaba el proceso de desarrollo. **Reflex** nació con la idea de unificar todo el proceso en un solo lenguaje.

Este *framework* fue lanzado inicialmente bajo el nombre *Pynecone* alrededor de 2022, fue creado por un equipo liderado por *Nikhil Rao* y *Alek Petuskey*, con el respaldo de *xAI*. En sus primeras etapas, *Pynecone* buscaba ofrecer una alternativa a herramientas como *Streamlit* o *Dash*, que también permiten crear interfaces web en *Python*, pero que estaban más limitadas a casos específicos como visualización de datos y no ofrecían la flexibilidad de un *framework* web completo. Con el tiempo, el proyecto evolucionó y fue renombrado a (Rao, 2023) para reflejar mejor su propósito: ser una herramienta ágil para el desarrollo web.

Uno de los hitos clave en la historia de **Reflex** fue su aceptación en el programa (Combinator, 2023) (ayuda a startups, proporciona financiación inicial, asesoramiento y contactos a empresas en las primeras etapas) en 2023, lo que le dio un impulso significativo en términos de visibilidad y recursos. **Reflex** se diseñó con la premisa de “*Python* puro”.

El *framework* compila el código a una aplicación *React* para el *frontend* y utiliza *FastAPI* para el *backend* (Reflex, 2023b), aprovechando tecnologías web modernas sin que el usuario tenga que interactuar directamente con ellas.

Desde su lanzamiento, **Reflex** ha crecido rápidamente en popularidad y su repositorio en *GitHub*<sup>1</sup> ha acumulado miles de estrellas (véase *figura 3.2*), y ha sido adoptado por desarrolladores de empresas como *Amazon*, *NASA*, *DELL* y *Samsung*, entre otras. Parte de su éxito se debe a su simplicidad: con comandos como `reflex init` para inicializar un proyecto y `reflex run` para ejecutarlo, los usuarios pueden tener una aplicación web funcional en minutos. Además, **Reflex** ofrece más de 60 componentes predefinidos (Reflex, 2023a), integración con bibliotecas de *Python* como *Pandas* o *Matplotlib*, y un sistema de despliegue con un solo comando.

<sup>1</sup><https://github.com/reflex-dev/reflex>

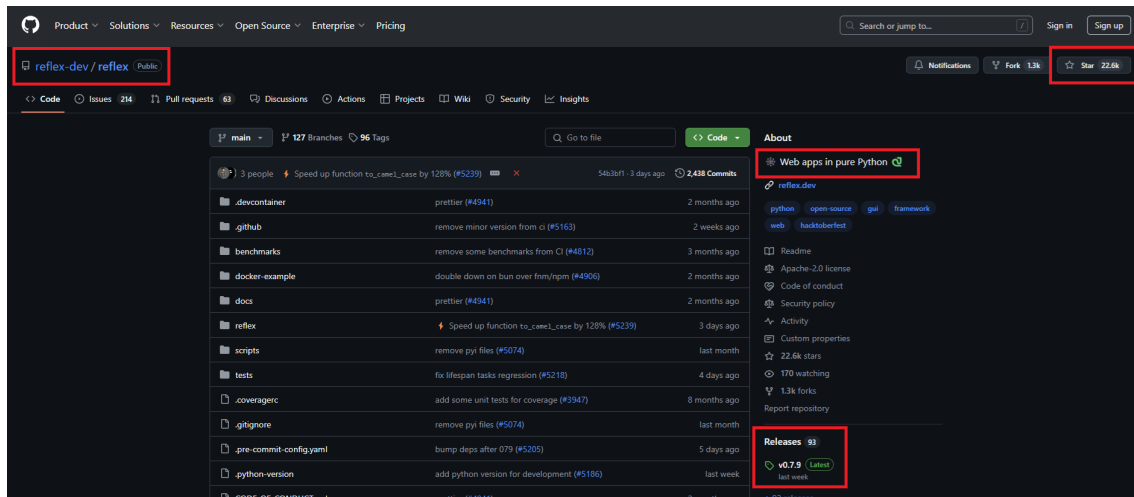


Figura 3.2: Repositorio Github de Reflex

A lo largo de su evolución, el equipo de Reflex ha mantenido un enfoque en la comunidad y la accesibilidad. El framework está licenciado bajo la *Apache License 2.0*, lo que fomenta su uso y contribución abierta. También han trabajado en mejorar la documentación y en proporcionar ejemplos prácticos, como tableros de datos, aplicaciones de chat y generadores de imágenes con *IA*. También ha ido integrándose con herramientas como *WebSockets* para aplicaciones en tiempo real y ofreciendo soporte para temas personalizables, adaptándose a las necesidades de los desarrolladores actuales.

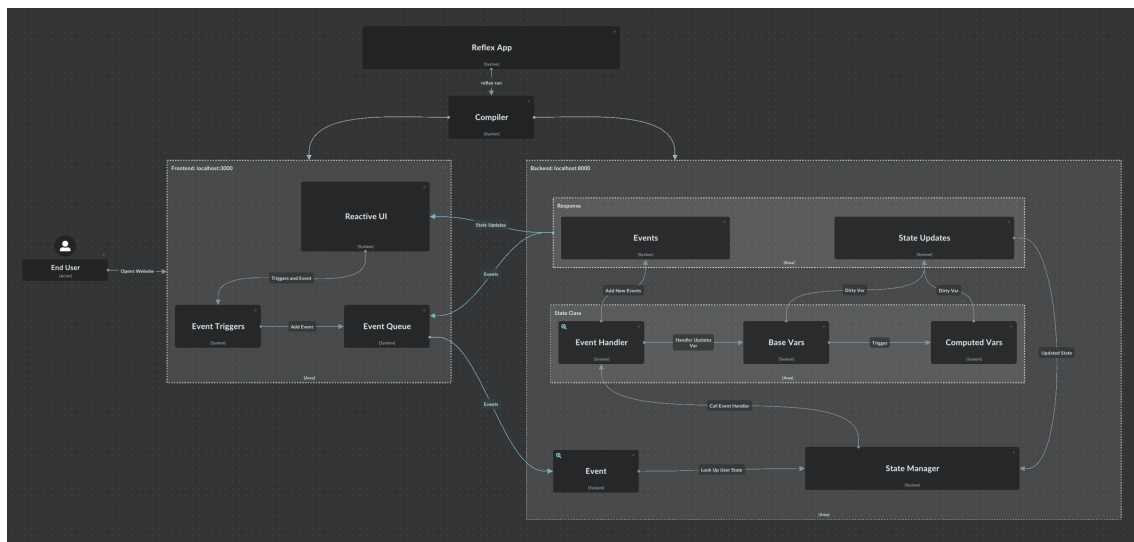


Figura 3.3: Cómo funciona una aplicación Reflex.

Reflex se basa en una arquitectura reactiva y orientada a componentes (véase figura 3.3). Los principales elementos de su arquitectura son (Reflex, 2023b):

- Estado (State): es el corazón de cualquier aplicación Reflex. Representa los datos que la aplicación maneja y que pueden cambiar con el tiempo. El estado se define como una clase de Python y se gestiona de forma centralizada.

- Componentes: los componentes son bloques de construcción reutilizables que encapsulan la lógica y la presentación de una parte específica de la interfaz de usuario. Los componentes se definen como funciones de Python que reciben el estado como entrada y devuelven la representación visual del componente.
- Eventos: acciones que ocurren en la interfaz de usuario, como clics de botón, entradas de texto o cambios en el estado. `Reflex` permite definir funciones de Python que se ejecutan en respuesta a estos eventos, lo que permite actualizar el estado y, por lo tanto, la interfaz de usuario de manera reactiva.
- Reactividad: es el principio fundamental de `Reflex`. Cuando el estado de un componente cambia, `Reflex` actualiza automáticamente la interfaz de usuario para reflejar estos cambios. Esto se logra mediante un sistema de seguimiento de dependencias que garantiza que solo los componentes que dependen del estado que se ha modificado se vuelvan a actualizar.

Este ejemplo muestra cómo definir un estado, crear un componente y gestionar eventos en `Reflex`. El componente `counter` muestra un contador que se puede incrementar o decrementar haciendo clic en los botones.

---

```
1 import reflex as rx
2
3 class State(rx.State):
4     count: int = 0
5
6     def increment(self):
7         self.count += 1
8
9     def decrement(self):
10        self.count -= 1
11
12 @rx.component
13 def counter():
14     state = rx.use_state(State)
15     return rx.vstack(
16         rx.button("Incrementar", on_click=state.increment),
17         rx.text(f"Contador: {state.count}"),
18         rx.button("Decrementar", on_click=state.decrement),
19     )
20
21 app = rx.App()
22 app.add_component(counter)
23 app.run()
```

---

### 3.1.1. Pros y contras de Reflex

Entre sus pros, destaca su simplicidad y rapidez de desarrollo: permite crear interfaces web sin necesidad de conocimientos profundos en HTML, CSS o JavaScript,

ya que todo se maneja desde Python, con lo que se agiliza el prototipado y la implementación. Su enfoque reactivo, basado en un sistema de estado, facilita la actualización automática de la interfaz ante cambios, ideal para aplicaciones dinámicas como juegos con temporizadores o estadísticas en tiempo real. Además, al ser Python puro, ofrece acceso directo a muchas bibliotecas permitiendo integrar funcionalidades avanzadas sin salir del lenguaje. También genera aplicaciones web con un *backend* y *frontend* sincronizados, eliminando la necesidad de gestionar tecnologías separadas, lo que simplifica el despliegue.

No obstante, Reflex tiene contras que deben considerarse. Su relativa juventud implica una documentación limitada y menos ejemplos prácticos en comparación con *frameworks* establecidos como Django o Flask, lo que puede complicar la resolución de problemas específicos. Aunque es versátil para aplicaciones pequeñas o medianas, su escalabilidad para proyectos complejos o de alto tráfico no está tan probada, pudiendo requerir ajustes manuales en el rendimiento. La dependencia de Python para el *frontend* puede limitar la personalización avanzada de la interfaz (por ejemplo, animaciones complejas), que sería más directa con JavaScript. Además, al abstraer el *frontend*, los desarrolladores pierden cierto control sobre el código generado, lo que podría ser un inconveniente para optimizaciones específicas. Su enfoque en simplicidad podría no satisfacer necesidades de aplicaciones empresariales con requisitos robustos de seguridad o infraestructura, donde otros *frameworks* ofrecen más herramientas nativas. En resumen, Reflex es ideal para proyectos educativos o prototipos rápidos como este proyecto, pero sus limitaciones emergen en escenarios más exigentes o personalizados.

## 3.2. Base de Datos: PostgreSQL

Es un sistema de gestión de bases de datos relacional de código abierto, robusto, escalable y compatible con SQL estándar, que ofrece ventajas como ser gratuito, seguro, extensible y capaz de manejar *grandes volúmenes de datos* con transacciones ACID. Funciona bajo un modelo *cliente-servidor*, almacenando datos en tablas relacionadas, procesando consultas SQL y utilizando índices para optimizar búsquedas (Wikipedia, PostgreSQL).

En el juego, se usan tablas para usuarios y partidas (véase figura 3.4), permitiendo consultas dinámicas como actualizar puntajes. Para la integración con Reflex hemos usado SQLAlchemy, facilitando la conexión entre la interfaz web y la base de datos.

Ofrece seguridad para datos sensibles. Es fácil de mantener con herramientas como *pgAdmin* y permite *respaldos automáticos*. Es una buena elección por su flexibilidad, comunidad activa y capacidad para escalar<sup>2</sup>, perfecta para un juego como “¿Quién quiere ser millonario?”.

---

<sup>2</sup><https://www.postgresql.org/>

```
-- Tabla de usuarios
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(255) UNIQUE,
  password VARCHAR(255),
  fecha_union TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  tema_stats JSONB, -- JSONB para almacenar los porcentajes de aciertos por tema
  max_puntuacion INTEGER
);

-- Tabla de partidas
CREATE TABLE partidas (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id),
  fecha_partida TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  tema_stats JSONB -- JSONB para almacenar el número de aciertos y preguntas por tema
);
```

Figura 3.4: Creación de tablas para la base de datos

### 3.3. Estructura del proyecto

El proyecto se estructura en la carpeta raíz `QQSM`. La organización del código sigue un patrón claro que separa responsabilidades, facilitando el mantenimiento y la escalabilidad.

El núcleo reside en el paquete anidado `QQSM/`, donde se distingue una división fundamental entre la *interfaz de usuario* (ubicada en `QQSM/pages/`) y la *lógica de estado* (contenida en `QQSM/states/`). Esta separación sigue las mejores prácticas de desarrollo con `Reflex`.

En el directorio `pages/` se encuentran todos los archivos `*_page.py` que definen las pantallas visibles de la aplicación. Estos incluyen desde las interfaces básicas como `login_page.py` y `register_page.py`, hasta las complejas vistas de juego como `game_page.py`, `theme_page.py` y `endless_page.py`.

Por otro lado, la carpeta `states/` contiene las clases `*_state.py` (`GameState`, `LoginState`, `RegisterState`) que gestionan el estado dinámico de la aplicación. Estas clases actúan como puente entre la interfaz de usuario y la lógica.

La interacción con la base de datos se encuentra en el directorio `db/` de nivel superior. Aquí, `database.py` configura la conexión utilizando `SQLAlchemy`, mientras que `models.py` define los modelos de datos. La lógica específica de autenticación se maneja en `QQSM/auth.py`, módulo que probablemente integra funcionalidades de `db/` con `Passlib` para garantizar la seguridad en el manejo de credenciales.

A nivel de raíz del proyecto, destacan varios archivos esenciales para la configuración y el despliegue. El archivo `requirements.txt` enumera todas las dependencias necesarias. `rxconfig.py` contiene la configuración específica de `Reflex`, mientras que `.gitignore` y `setupDB.sql` facilitan el control de versiones y la inicialización de la base de datos respectivamente. Los recursos estáticos como estilos, imágenes y variables de entorno se organizan en carpetas dedicadas (`styles/`, `assets/`, `img/`, `env/`), completando una estructura de proyecto bien organizada y profesional.

## 3.4. Funcionalidades del Juego

En esta sección se presenta una descripción detallada de las funcionalidades y modos de juego que se han desarrollado para la aplicación, incluyendo sus mecánicas, reglas y particularidades.

### 3.4.1. Modo Infinito (endless)

El modo Endless ofrece una experiencia de juego continuo donde el objetivo es alcanzar la máxima puntuación posible respondiendo preguntas sin límite. La dificultad se incrementa progresivamente (4 puntos por pregunta hasta 100) mediante la función `generate_difficulty_endless_mode` en `game_state.py`, mientras que los temas se seleccionan aleatoriamente con `generate_topic_normal_mode`.

El `game_state` gestiona el flujo del juego, solicitando nuevas preguntas mientras el jugador acierte. A diferencia del modo clásico, no existe la barra de progreso de 15 pasos, continuando indefinidamente hasta que ocurra un fallo.



Figura 3.5: Juego Balatro

Para definir este modo de juego nos hemos inspirado en Balatro (véase figura 3.5), un juego de construcción de mazos inspirado en el *póker* con las siguientes características:

- Objetivo: superar rondas con puntuaciones crecientes mediante manos de *póker*.
- Mecánicas: compra de cartas especiales (Jokers, Planetas, Tarots) entre rondas y multiplicadores de puntuación.
- Partidas (“runs”) que continúan hasta fallar una ronda.
- Combinación de cartas para superar la dificultad progresiva.

El modo infinito (`endless_mode`) del proyecto se inspira en varios aspectos de Balatro:

- Naturaleza “*Run-Based*”:

- Ambos juegos continúan hasta que el jugador comete un error.
- Sensación de progresión y tensión creciente.

- Dificultad Escalable:

- Balatro: aumenta puntuación requerida por “ante”.
- Nuestro código (`game_state.py` → `generate_difficulty_endless_mode`):
  - Dificultad calculada como 4 x número\_pregunta
  - Límite máximo de dificultad en 100
- El valor base de cada ronda en Balatro sigue esta progresión: **Valor del Ante = 300 x Número de Ante**(véase figura 3.1) ; así cada ante (ronda) tiene un valor base que debes superar con el puntaje de tu mano, el orden sería el siguiente:

Ronda (Ante)	Valor Requerido (Chips)
Ante 1	300
Ante 2	600
Ante 3	900
Ante 4	1,200
Ante 5	1,500

Tabla 3.1: Valores Requeridos por Ronda (Ante)

- Indicador Cíclico (Inspiración Visual):

- Implementación en `endless_page.py` con 3 ciclos (`range(3), % 3`).
- Representación visual de ciclo similar a Balatro(Véase figura figura 3.6).
- Incremento lineal (no cíclico) de dificultad.

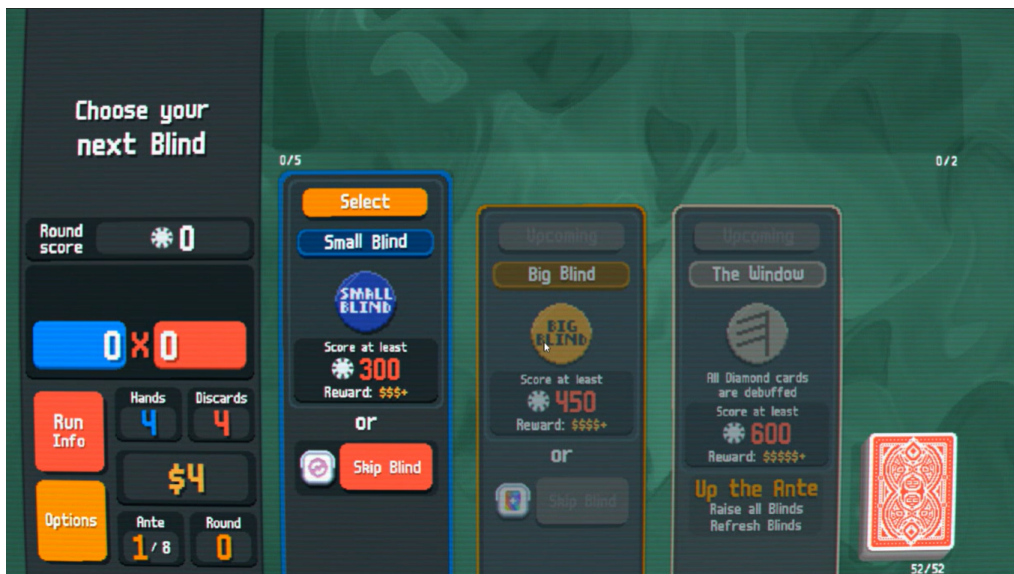


Figura 3.6: Juego Balatro rondas de 3

### 3.4.2. Modo Normal

Este modo replica la experiencia televisiva con 15 preguntas de dificultad creciente (Véase figura 3.7). La progresión está predefinida en la lista `difficulties_NM` de `game_state.py`, accediéndose mediante `generate_difficulty_normal_mode`.

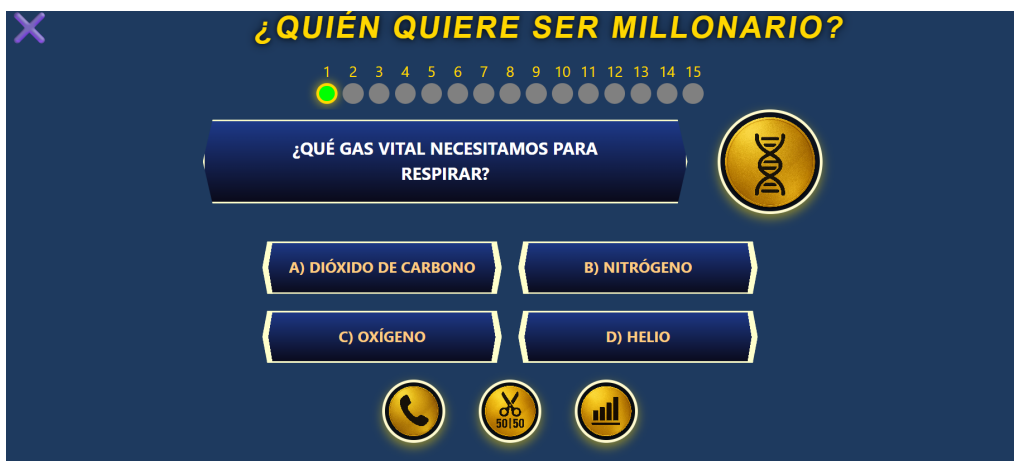


Figura 3.7: Modo Normal

Los temas (Véase figura 3.8 y figura 3.9) se seleccionan aleatoriamente, evitando repeticiones consecutivas. El `game_state` coordina la obtención de preguntas adecuadas, mostrándolas en la interfaz. La partida finaliza al completar las 15 preguntas o al cometer un error.



Figura 3.8: Hay 15 temas para elegir



Figura 3.9: Iconos de los temas

Cuando se selecciona una respuesta (A, B, C o D) haciendo clic en su botón, se llama al método `validate_answer` con la letra elegida. Dentro de este método (en `game_state.py`), el código primero marca la respuesta como elegida:

---

```
1 self.chosen_answer = True
```

---

Esto desactiva los demás botones. Luego compara el texto de tu opción con la respuesta correcta almacenada en el estado (`self.correct`). Basándose en si coinciden, actualiza un indicador (`self.correct_answer`) y modifica las clases CSS de los botones (`self.button_classes`) para mostrar tu elección como éxito o error,

y siempre resalta la opción correcta en verde. Si es correcta se puede avanzar, si se falla, se debe terminar la partida. El resultado determina si aparecerá el botón para avanzar a la siguiente pregunta. También se actualizan las estadísticas del usuario mediante `update_user_stats`.

### 3.4.3. Modo Temático

Este modo se ha diseñado para ofrecer mayor control al jugador. Este modo permite seleccionar entre dos temas alternativos tras cada respuesta correcta (Véase figura 3.10). La interfaz `theme_page.py` presenta los botones de selección, cuyas opciones se generan cuando utilizamos `generate_topic_theme_mode` en `game.py`.

El `game_state` captura la elección mediante `set_theme` y `get_themes`, actualizando el estado antes de solicitar la siguiente pregunta con `generate_question`. La dificultad sigue la progresión estándar mediante `generate_difficulty_normal_mode`, manteniendo la estructura clásica de 15 preguntas pero con temática parcialmente personalizable.



Figura 3.10: Modo temático con un comodín usado

Cuando se presentan los dos nuevos temas para que el usuario elija, el `game_state` llama a su método `get_themes`. Este invoca al método `generate_topic_theme_mode` dentro de la clase `game_state`, pasándole como argumentos los dos temas que se mostraron en la ronda anterior (`self.topic_selection1` y `self.topic_selection2`).

El código primero selecciona dos temas completamente al azar de la lista completa `self.topics`. Luego, entra en un bucle `while` cuya condición es crucial para que los temas no se repitan en rondas seguidas:

---

```

1 while ((random_topic1 == topic1 or random_topic2 == topic1) and
2       (random_topic1 == topic2 or random_topic2 == topic2) or
3       (topic1 == topic2)):

```

---

Esta condición verifica si alguno de los nuevos temas elegidos (`random_topic1`, `random_topic2`) coincide con alguno de los temas de la ronda anterior (`topic1`,

`topic2` recibidos como argumento) y además si los dos nuevos temas elegidos son iguales entre sí.

Si se cumple esta condición (es decir, hay una repetición inmediata con la ronda anterior o los nuevos temas son idénticos), el código vuelve a seleccionar dos nuevos temas al azar:

---

```
1 random_topic1 = random.choice(self.topics)
2 random_topic2 = random.choice(self.topics)
```

---

Después repite la comprobación. El bucle continúa hasta que se genera una pareja de temas nuevos que son distintos entre sí y ninguno coincide con los dos temas ofrecidos inmediatamente antes. Así el código asegura que las dos opciones de tema presentadas al usuario no sean las mismas que las de la ronda justamente anterior, aunque no garantiza que un tema no pueda volver a aparecer después de varias rondas distintas.

#### 3.4.4. Modo Máquina vs Máquina

El modo “Máquina vs Máquina” se inicia desde el menú principal, redirigiendo a la ruta `/maquinaVS`. Esta página (`maquinaVS_page.py`) funciona como una pantalla de selección, permitiendo al usuario elegir qué Inteligencia Artificial participará en la contienda (Véase figura 3.11). El diseño de esta página presenta iconos representativos para cada una de las IAs disponibles: DeepSeek, OpenAI y Llama, facilitando una selección intuitiva por parte del usuario.



Figura 3.11: Elección de IA para el máquina vs máquina

Al seleccionar una de las IAs, se ejecuta la función `initialize_game` (" [nombre de la IA]IA"), donde "nombre de la IA" se sustituye por `deepSeek`, `openAI` o `llama` según la elección del usuario. Esta acción resetea el estado del juego, establece el modo correspondiente (p. ej. `/deepSeekIA`, `/openAIIA`, `/llamaIA`), genera la primera pregunta utilizando Gemini como motor de preguntas base y redirige al usuario a la página específica de la IA seleccionada. La parte fundamental de este modo ocurre justo antes de que la página de la IA seleccionada se muestre al usuario:

- La clase `game_state` utiliza la API de Gemini para generar una pregunta junto con sus cuatro opciones de respuesta (A, B, C, D) y la respuesta correcta.

- Inmediatamente después, se llama al método correspondiente para la IA seleccionada:
  - Para DeepSeek: `.deepSeekAnswer()`.
  - Para OpenAI: `.openai_answer()`.
  - Para Llama: `.llama_answer()`.

Este método consulta la API de la IA elegida para obtener su predicción de la respuesta correcta (la letra A, B, C o D) a la misma pregunta generada por Gemini.

- La respuesta proporcionada por la IA seleccionada se valida automáticamente mediante la función `.validate_answer`, comparándola directamente con la respuesta correcta que Gemini determinó.

Cuando el usuario finalmente visualiza la página de la IA seleccionada, se encuentra con la pregunta y los botones de respuesta ya deshabilitados y coloreados (Véase figura 3.12). Esta representación visual indica de forma inmediata si la respuesta automática de la IA fue acertada o errónea, y además resalta cuál era la opción correcta según Gemini.

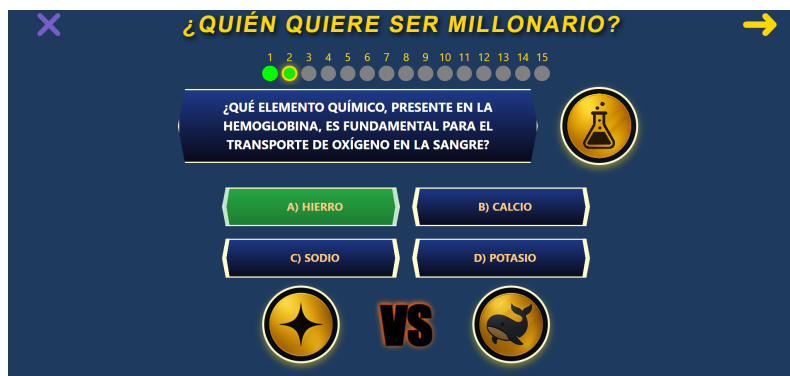


Figura 3.12: Modo Máquina Vs Máquina (Gemini Vs DeepSeek)

El usuario asume el rol de observador en este enfrentamiento. Si la IA seleccionada responde correctamente, aparece el botón “Siguiente” (->). Al hacer clic en este botón, el juego avanza a la siguiente ronda, repitiendo el ciclo completo: Gemini genera una nueva pregunta, la IA seleccionada intenta responderla, su respuesta se valida automáticamente y el resultado se muestra al usuario. En el caso de que la IA falle al responder, la secuencia del juego se detiene en esa pregunta, mostrando el resultado del enfrentamiento directo entre las inteligencias artificiales. Este modo ofrece una demostración secuencial y comparativa de las capacidades de diferentes modelos de IA para resolver preguntas generadas por otro modelo.

## 3.5. Sistema de Autenticación

Esta sección presenta los componentes y la lógica que componen el sistema de autenticación de la aplicación, abarcando tanto el registro de nuevos usuarios como

el inicio de sesión de usuarios existentes.

### 3.5.1. Inicio de Sesión

El usuario comienza el proceso desde la página de bienvenida (`welcome_page.py`). Al hacer clic en “Iniciar Sesión”, se le redirige mediante `LoginState.clear_message` a la ruta `/login`. La interfaz, definida en `login_page.py`, se construye con un componente `rx.center` que contiene un formulario `rx.form`, organizado verticalmente con `rx.vstack`. Este formulario incluye:

- Un campo de entrada `rx.input` para el nombre de usuario, con placeholder “Usuario” y nombre interno `usuario`.
- Un campo de entrada `rx.input` para la contraseña, con `placeholder= “Contraseña”`, tipo `password`, y nombre interno `password`.
- Un botón `Iniciar Sesión` que al ser tipo `submit` ejecuta la acción del formulario.
- Un botón `Volver` que redirige a `/welcome`.

Debajo del formulario se encuentra `rx.text(LoginState.login_message)`, que muestra feedback dinámico sobre el proceso.

La función clave es `handle_login`, ejecutada al enviar el formulario (`on_submit=LoginState.handle_login`). El proceso es:

- Validación de Campos: recibe un diccionario `form_data`, con claves `usuario` y `password`. Antes de continuar, se validan los campos:

---

```

1 if (not username and password) or (not password and username) or
2     username == "" and password == "":
3     self.login_message = "Por favor, complete ambos campos"
4     return

```

---

Si alguno está vacío, se muestra un mensaje de error y se detiene el proceso.

- Intento de Autenticación: si ambos campos están completos:

- Se crea una sesión de base de datos:

---

```

1 db = SessionLocal()

```

---

- Se llama a `login_user(username, password)` (de `auth.py`), que verifica las credenciales con la base de datos usando `passlib`.

- Resultados:

Éxito: se actualiza el estado:

---

```
1     self.is_authenticated = True
2     self.username = username
3     self.password = password # Riesgo de seguridad
4     self.login_message = "Bienvenido"
5     return rx.redirect("/menu")
```

---

Fallo: se actualiza el mensaje:

---

```
1     self.login_message = "Usuario o contraseña incorrectos."
```

---

- Manejo de errores: todo está dentro de un bloque `try...except...finally` para capturar excepciones y garantizar el cierre de la sesión:

---

```
1     try:
2         ...
3     except Exception as e:
4         self.login_message = "Error en la autenticación: {e}"
5     finally:
6         db.close()
```

---

### 3.5.2. Registro

El proceso de registro se inicia al enviar el formulario de la página `/register`, cuya interfaz es visualmente similar a la de `login`. El método `RegisterState.handle_register` es el encargado de recibir los datos.

Primero, se valida que los campos de nombre de usuario y contraseña no estén vacíos. Si ambos contienen datos, se procede a llamar a la función `create_user` (definida en `auth.py`). Dicha función realiza el *hash* de la contraseña de forma segura, utilizando `passlib`, antes de intentar insertar el nuevo usuario en la base de datos.

Esta operación se va a encontrar encapsulada dentro de un bloque de la forma `try/except/finally`, lo que permite una gestión robusta de la sesión con la base de datos (mediante `SessionLocal` y `db.close()`), de manera análoga al proceso de *login*.

- Si la creación en la base de datos tiene éxito (es decir, no se lanza ninguna excepción), se actualiza el estado `register_message` con un mensaje de éxito y se redirige al usuario a la página de inicio de sesión (`/login`).
- Si ocurre una excepción (por ejemplo, debido a un nombre de usuario duplicado), se muestra un mensaje de error mediante `register_message`.

El objetivo principal de este flujo es crear un usuario de forma segura y guiarlo inmediatamente al proceso de inicio de sesión.

## 3.6. Arquitectura Reactiva

La arquitectura de la aplicación se fundamenta en los principios de la programación reactiva, facilitada por el framework `Reflex`. Esta sección explora los elementos esenciales de esta arquitectura: la gestión del estado de la aplicación, el diseño de los componentes de la interfaz de usuario y la implementación de la navegación entre las diferentes páginas.

La **gestión de estados** se realiza a través de las clases derivadas de `rx.State` (como `game_state` o `login_state`), que constituyen el núcleo lógico de la aplicación. Contienen variables reactivas (`question`, `feedback`, etc.) cuya modificación actualiza automáticamente la interfaz.

Los métodos con `@rx.event` responden a acciones del usuario, conectando la interfaz con la lógica de negocio en `game_state.py` y `auth.py`.

En cuanto al **diseño de Interfaces**, cada pantalla se define en `*_page.py` con `@rx.page(route="/...")`. Utilizan componentes de `Reflex` (`rx.button`, `rx.text`, etc.) para construir la interfaz gráfica, mostrando datos dinámicos desde los `States` y conectando eventos de usuario a la lógica correspondiente.

### 3.6.1. Menú Principal

El menú principal (Véase figura 3.13) actúa como centro de navegación tras el *login*. Desde aquí los usuarios acceden a los modos de juego, perfil personal y marcadores, mediante redirecciones como `rx.redirect("/leaderboard")`.



Figura 3.13: Menú principal

### 3.6.2. Tabla de Clasificación: *Leaderboard*

Tiene como propósito principal exhibir un *ranking* organizado de los jugadores con las mayores puntuaciones, destacando además la posición actual del usuario activo. Su implementación se estructura en torno al archivo `leaderboard_page.py`, que construye la interfaz visual. Para obtener los datos necesarios, esta página interactúa con funciones especializadas del módulo `auth`: `get_top_10_users` recupera los diez mejores jugadores, mientras que `get_user_position` y `get_user_leaderboard` obtienen información específica sobre el usuario actual.

La presentación visual se organiza en dos segmentos claramente diferenciados: una lista desplazable muestra de forma ordenada los primeros diez clasificados con su respectiva posición, nombre y puntuación acumulada. Simultáneamente, el sistema identifica al usuario activo mediante `LoginState.username` y muestra su registro particular de forma destacada bajo la lista principal, permitiendo una rápida identificación de su posición en el ranking global. Como elemento de navegación, se incluye un botón que redirige al menú principal para facilitar la experiencia de usuario.

### 3.6.3. Página de Perfil: (*User Page*)

Su diseño está orientado a presentar información básica pero relevante sobre la cuenta del jugador actual (Véase figura 3.14). Implementada en `user_page.py` y accesible mediante la ruta `/user_page`, esta interfaz recupera automáticamente el nombre de usuario del estado global (`LoginState.username`) y complementa estos datos con la puntuación máxima histórica del jugador, obtenida a través de la función `get_user_leaderboard` del módulo `auth`.

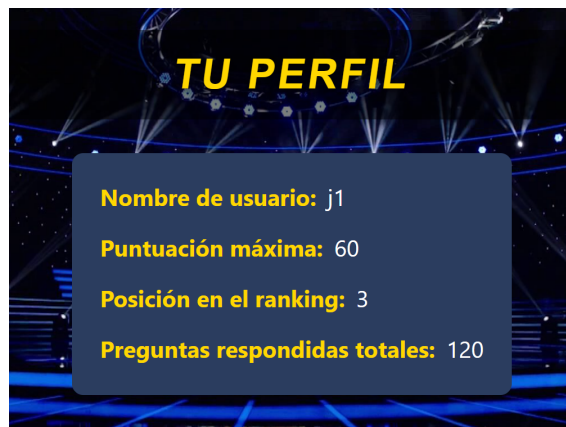


Figura 3.14: Perfil del usuario

La presentación utiliza componentes como `rx.text` y `rx.box` para crear una vista limpia y minimalista que muestra el nombre de usuario junto con su mejor puntuación registrada. Su función es principalmente informativa.

## 3.7. Integración con Inteligencia Artificial

Toda la comunicación directa con las *APIs* externas se encuentra encapsulada dentro de la clase `GameState`, mientras que los distintos `States` interactúan exclusivamente con los métodos expuestos por esta clase, nunca accediendo directamente a los servicios externos. Este diseño mantiene una clara separación de responsabilidades y facilita el mantenimiento futuro del código.

Como sistema dependiente de servicios externos, el proyecto presenta una dependencia crítica de la conectividad a Internet para su funcionamiento completo. La generación dinámica de preguntas y la implementación de los comodines que utilizan inteligencia artificial (*público* y *llamada*) requieren acceso constante a los

servicios en la nube. Además, esta arquitectura implica una dependencia operativa de la disponibilidad y costes asociados a las *APIs* utilizadas, factores que deben considerarse para el despliegue en producción.

Un aspecto fundamental en la implementación ha sido el *prompt engineering*. La efectividad de las integraciones con modelos de IA depende directamente de la calidad, especificidad y estructuración de los *prompts* diseñados. Como se evidencia en el código, se han implementado formatos estrictos y estructurados para las consultas a los modelos, garantizando así respuestas consistentes que el sistema pueda interpretar correctamente.

En cuanto a la gestión de errores, los `States` incorporan mecanismos básicos de manejo de excepciones mediante bloques `try-except`. Estos bloques protegen las interacciones con funciones que pueden fallar, como las llamadas al módulo `auth` o las solicitudes a los servicios de IA.

El proyecto demuestra un uso de modelos distintos de inteligencia artificial para potenciar funcionalidades clave: Gemini se emplea para la generación masiva de contenido estructurado (preguntas del juego) y para la simulación del comodín del público, mientras que DeepSeek proporciona interacciones más especializadas que simulan la consulta a un experto durante el comodín de la llamada. OpenAI y Llama se integran para proporcionar respuestas en el modo de juego “Máquina vs Máquina”.

La conexión con estos servicios se implementa utilizando aproximaciones técnicas diferenciadas: la biblioteca específica proporcionada por Google para la integración con Gemini, contrastando con el uso de la biblioteca genérica `requests` para las consultas a DeepSeek. Este diseño dual pone de manifiesto la importancia crítica de dos aspectos fundamentales: la gestión segura de claves API (especialmente sensibles en entornos de producción) y el diseño de los *prompts* que determinan la calidad y utilidad de las respuestas obtenidas de los modelos de IA.

### 3.7.1. Google Gemini (Modelo: gemini-2.0-flash)

Las preguntas no están predefinidas, sino que se generan en tiempo real. El origen es el modelo `gemini-2.0-flash` de Google, al que se accede mediante la biblioteca `google-generativeai`, dentro del método `generate_question` de la clase `GameState`.

Dentro de `GameState.generate_question`, se construye un `prompt` específico para Gemini. Este `prompt` es una cadena de texto que incluye instrucciones fijas y dos variables clave: `difficulty` y `topic` (cadena como “historia” o “ciencia”). El `prompt` exige que Gemini devuelva la información en un formato muy estricto, separado por punto y coma.

- La función `.generate_question` envía este `prompt` al modelo Gemini usando:

```
self._model.generate_content(prompt).text
```

Gemini devuelve una cadena de texto que sigue el formato solicitado.

- La cadena recibida se divide con `answer.split(";")`, se elimina el prefijo

"Pregunta:" y se limpian caracteres innecesarios. El resultado es una lista con seis elementos: pregunta, opciones A-D y respuesta correcta.

- Gemini no juega, sino que incluye la respuesta correcta junto con la pregunta. La respuesta del jugador (humano o IA) se procesa por separado.
- La respuesta correcta (último elemento de la lista generada) se guarda en `self.correct` (en `GameState`). Cuando un jugador elige una opción, tenemos que `GameState.validate_answer` compara ese texto con `self.correct`.

Gemini también se utiliza para simular el comodín del público mediante el uso de `Game.public_option`. Se construye un nuevo *prompt* con la pregunta y opciones actuales, solicitando porcentajes de votación plausibles para cada opción, separados por punto y coma:

---

```

1 # Para Generar Preguntas
2 question = "Quiero que me hagas una pregunta como si fuera quien quiere ser"
3 "millonario con una dificultad " + str(difficulty) + "/100 y que el tema de
4 "la pregunta sea " + topic + ". Tambien quiero que el formato este separado por
5 punto y coma donde me muestre la pregunta las "cuatro respuestas y la pregunta
6 correcta.Como ejemplo Pregunta:";cual es la capital de España?";
7 Paris;Roma;Madrid;Wansinton;Madrid;
8 Pasame solo el mensaje sin nada extra"
9
10 # Luego se usa: self._model.generate_content(prompt_pregunta).text
11
12 # Para simular el comodín del público
13 public = ("Estoy jugando a quien quiere ser millonario y me preguntan" +
14 self.question + "con estas posibles respuestas" + self.option_a + self.option_b +
15 self.option_c + self.option_d + "y quiero usar el comodin del publico. Quero que
16 solo me muestres el texto del comodin del publico de la siguiente forma como
17 en el ejemplo : "Courrèges:15%;Miyake:60%;Ungaro:5%Cardin:20% .
18 Pasame solo el mensaje sin nada extra ")
19
20 # Luego se usa: self._model.generate_content(prompt_publico).text

```

---

### 3.7.2. DeepSeek (Modelo: deepseek-chat)

DeepSeek se utiliza en dos funciones dentro de `GameState`, cada una con su propio *prompt* construido dinámicamente:

Para el **Comodín de la Llamada** se usan las variables de la instancia actual de `GameState`: `self.question`, `self.option_a`, `self.option_b`, `self.option_c`, `self.option_d`.

---

```

#PROMPT:
message = ("En '¿Quién quiere ser millonario?', la pregunta es: " + self.question +
"Opciones: " + self.option_a + ", " + self.option_b + ", " +

```

---

```
self.option_c + ", " + self.option_d + "." +
"Usando el comodín de la llamada, muestra solo con el siguiente formato: "
"respuesta correcta;descripcion breve.")
```

---

Se le pide a DeepSeek que identifique la respuesta correcta y añada una breve justificación, separadas por punto y coma. El tiempo de respuesta de DeepSeek para esta función se divide en dos componentes: el tiempo que tarda en procesar la pregunta y determinar la respuesta (*tiempo de pensamiento*) y el tiempo que tarda en generar y enviar la respuesta (*tiempo de respuesta*). Una vez obtenido el *prompt*, el comodín se desarrolla de la siguiente manera:

- Se hace una llamada HTTP POST a la API de DeepSeek: `https://api.deepseek.com/v1/chat/completions` usando `requests`. El tiempo transcurrido para esta llamada incluye tanto el tiempo de pensamiento de DeepSeek como el tiempo de transmisión de la respuesta.
- La clave API está codificada directamente en el código.
- Se envía el prompt en el cuerpo JSON de la solicitud.
- Se recibe la respuesta, se extrae el contenido y se divide por “;” para obtener la respuesta y su explicación.
- Esta información se guarda formateada en `GameState.call_text` y se muestra al usuario (función `render_call_box`).

Inicialmente, las interacciones con la API de DeepSeek, especialmente para el comodín de la llamada y las respuestas IA, presentaban latencias promedio de 20 segundos. Este tiempo se dividía en el tiempo que DeepSeek tardaba en analizar la pregunta y formular una respuesta, y el tiempo necesario para transmitir esa respuesta a nuestra aplicación. Esta latencia ralentizaba la experiencia de juego.

Identificamos que esto se debía en parte a *prompts* iniciales que podrían haber sido más ambiguo (Véase figura 3.15) y también a un valor elevado en el parámetro `max_tokens` solicitado a la API, que obligaba al modelo a generar más texto del estrictamente necesario, incrementando tanto el tiempo de pensamiento como el tiempo de respuesta.

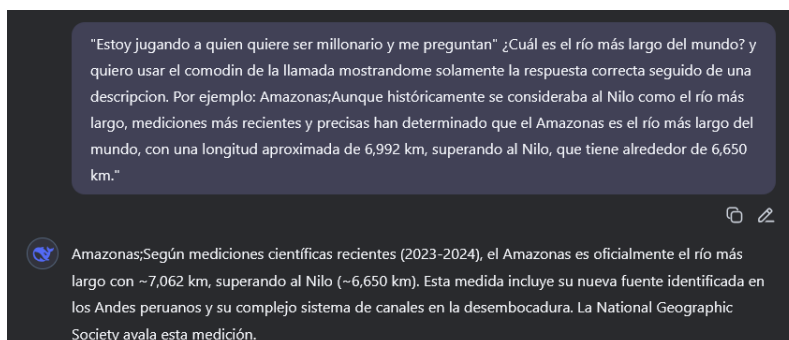


Figura 3.15: Prompt IA sin optimizar

### 3.7.3. OpenAI (Modelo:gpt-4o-mini)

La API de OpenAI procesa el `prompt` y devuelve una respuesta en formato JSON, la cual contiene el texto generado por el modelo de lenguaje.

---

```
1 # Dentro de Game.openai_answer
2 messages=[
3 {
4     "role": "user",
5     "content": "Responde a la pregunta: " + self.question +
6                 "Opciones: " + self.option_a + ", " + self.option_b +
7                 ", " + self.option_c + ", " + self.option_d + "." +
8                 "Muestrame SOLO LA LETRA de la solucion, sin negrita "
9                 "ni ningun efecto, ni ningun caracter especial, "
10                "solo la letra. Opciones: A, B, C, D."
11 }
12 ],
```

---

- Una vez recibida la respuesta JSON desde la API, el siguiente paso es su procesamiento para extraer la información relevante, que en este caso se espera que sea la letra de la opción que el modelo considera correcta. Esto implica la navegación a través de la estructura JSON de la respuesta para acceder al contenido del mensaje generado por la IA. Se implementan mecanismos de validación para asegurar que el valor extraído sea efectivamente una de las letras válidas ('A', 'B', 'C' ó 'D'), lo que permite integrar la predicción de manera consistente con el resto de la lógica del juego.
- El modelo de OpenAI no interactúa directamente con el jugador humano. La letra de la opción predicha como correcta se devuelve internamente dentro del flujo de ejecución del método `.openai_answer`.
- La letra de la respuesta se retorna a la función `.generate_question` dentro del archivo `game_state.py`. Allí, se utiliza como entrada para la función `.validate_answer`. Esta función es la encargada de comparar la predicción de OpenAI con la respuesta correcta (`self.correct`) que fue previamente determinada por Gemini para la misma pregunta. Este proceso de validación es idéntico al utilizado para evaluar las respuestas de un jugador humano o de la IA DeepSeek, asegurando una evaluación consistente y objetiva del rendimiento de OpenAI en el contexto del juego. Es importante destacar que OpenAI, al igual que DeepSeek, opera sin autoevaluación; su precisión se mide estrictamente en relación con la "verdad" establecida por Gemini, actuando Gemini como el árbitro final de la corrección de las respuestas en este modo de juego.

### 3.7.4. Llama (Modelo: llama4-maverick)

La integración del modelo de lenguaje Llama, sigue una arquitectura similar a la implementada para OpenAI y DeepSeek. Se utiliza la biblioteca `requests` de Python para establecer comunicación con la API de Llama a través de solicitudes HTTP POST.

Esta IA se dedica exclusivamente a participar en el modo de juego “Máquina vs Máquina”, enfrentándose al criterio de corrección establecido por el modelo Gemini de Google. La lógica de interacción con la API de Llama se encuentra encapsulada en el método `llama_answer` de la clase `GameState`.

Dentro de la función `.llama_answer`, se construye un `prompt` específico para interrogar al modelo Llama. Este `prompt` toma como entrada la pregunta actual del juego y sus correspondientes cuatro opciones de respuesta (`self.question`, `self.option_a`, `self.option_b`, `self.option_c`, `self.option_d`), replicando la estructura de los `prompts` diseñados para las otras IAs en el modo IA vs IA. El objetivo principal es instruir a Llama para que identifique la opción que considera la respuesta correcta, devolviendo únicamente la letra correspondiente.

- La función `.llama_answer` se encarga de la creación dinámica del `prompt` y del envío de la solicitud a la API de Llama mediante una llamada HTTP POST al `endpoint` de finalización de la conversación proporcionado por la API. La clave de la API de Llama, necesaria para la autenticación, se gestiona como una variable dentro de la clase `AIClient` (`self.llama_api_key`). La solicitud incluye el `prompt` en formato JSON dentro del cuerpo de la petición.

---

```

1 # Dentro de Game.llama_answer
2 message = [
3 {
4     "role": "system",
5     "content": ("Responde a la pregunta: " + self.question +
6               "Opciones: " + self.option_a + ", " + self.option_b +
7               ", " + self.option_c + ", " + self.option_d +
8               ". " +
9               "Muestrame SOLO LA LETRA de la solucion, sin negrita "
10              "ni ningun efecto, ni ningun caracter especial, "
11              "solo la letra. Opciones: A, B, C, D."
12 }

```

---

La API de Llama procesa el `prompt` y devuelve una respuesta en formato JSON, conteniendo el texto generado por el modelo.

- Tras recibir la respuesta JSON de la API de Llama, se procede a la extracción de la letra correspondiente a la opción predicha como correcta. Esto implica el análisis de la estructura JSON de la respuesta para acceder al contenido textual generado por la IA. Se implementan comprobaciones para asegurar que la respuesta extraída sea una de las letras válidas, garantizando la coherencia con el flujo del juego.
- Al igual que las otras IAs en el modo IA vs IA, Llama no interactúa directamente con el usuario. Su predicción (la letra de la opción correcta) se devuelve internamente.
- La letra obtenida a partir de `.llama_answer` se devuelve a `GameState.generate_question` y posteriormente se evalúa usando `GameState.validate_answer`. Esta función

compara la predicción de Llama con la respuesta correcta (`self.correct`) establecida por Gemini para la misma pregunta, siguiendo el mismo protocolo de validación utilizado para las respuestas de jugadores humanos y otras IAs. Llama, al igual que DeepSeek y OpenAI, no se autoevalúa, y su rendimiento se mide en relación con la “verdad” definida por Gemini.

Se ha creado un video explicativo donde se puede ver la funcionalidad completa del juego: <https://www.youtube.com/watch?v=XMeuNWvVXZc>



## Contribuciones Personales

### 4.1. Alejandro Relanzón Coello

- Creación y gestión de la base de datos: en el backend, creación y gestión de la base de datos utilizando PostgreSQL como sistema de almacenamiento. Se configuró una conexión robusta con la base de datos en el archivo `database.py`, utilizando SQLAlchemy como ORM para facilitar la interacción con la base de datos. El modelo de Usuario fue definido con campos esenciales como `id`, `username`, `password`, `fecha_union` y `tema_stats`, para almacenar las estadísticas de los jugadores. Para asegurar la creación de las tablas necesarias, se implementó la función `init_db()`.
- Interfaz de registro y login: en cuanto a la gestión de la autenticación, se desarrollaron las páginas de *Login* y Registro (`login_page.py` y `register_page.py`), las cuales permiten la validación de las credenciales de los usuarios. Se implementaron las funciones `get_password_hash()` y `verify_password()` utilizando la librería `passlib` para garantizar el almacenamiento seguro de las contraseñas en forma de *hashes*. Las funciones `create_user()` y `login_user()` permiten añadir nuevos usuarios y verificar la autenticación de los mismos.
- Estructura de rutas y flujo de aplicación: la estructura de las rutas y el flujo de la aplicación fueron configurados en `QQSM.py`. Utilizando `rx.cond()`, se aseguró que el acceso a las diferentes páginas fuera condicionado por el estado de autenticación del usuario, con rutas para páginas como el menú principal, las páginas de juego y las páginas de autenticación.
- Páginas de leaderboard y perfil de usuario: desarrollo de la página de **Leaderboard** para mostrar las puntuaciones de los jugadores, obteniendo el top 10 de jugadores desde la base de datos, y la página del Perfil de Usuario, donde se muestran estadísticas detalladas del usuario actual. Ambas páginas se conectan directamente con la base de datos para obtener y mostrar la información relevante, como el `maxPuntuacion` y los aciertos por tema.
- Investigación de IA vs IA: investigación de diversas tecnologías de *Modelos de Lenguaje de Gran Escala* (LLMs) como DeepSeek, OpenAI GPT y Llama

AI, con el objetivo de integrarlas en el sistema para ofrecer un modo de juego interactivo y desafiante: el modo IA vs IA. Realización de pruebas de las diferentes versiones de estos modelos, evaluando su rendimiento en términos de precisión y velocidad para elegir las más adecuadas para el juego.

- Creación de la lógica IA vs IA: implementación de la lógica para IA vs IA. Esto incluyó la integración de los tres LLMs en la aplicación, así como la creación de páginas dedicadas para cada uno de los modos de IA (DeepSeek, OpenAI GPT y Llama AI). Cada IA fue configurada para responder a las preguntas generadas en el juego y competir entre sí, proporcionando una experiencia dinámica e interactiva para los usuarios.
- Implementación de páginas para cada IA: desarrollo de páginas específicas para cada modo de IA, donde los jugadores pueden ver a las IAs enfrentándose entre sí. Estas páginas implementan la lógica de la IA y muestran el progreso de las IAs, permitiendo a los usuarios seguir las interacciones de manera intuitiva.
- Refactorización del código y modularización: la refactorización del código fue un paso esencial para garantizar que la aplicación fuera escalable y fácil de mantener. Se dividió la lógica en módulos independientes, cada uno encargado de un aspecto específico de la aplicación. Las dos secciones principales son la de `pages` con toda la parte del *frontend*, y la carpeta `QQSM` donde están los distintos `States` de Reflex al igual que el backend extra para el manejo del inicio de sesión. El desarrollo de este proyecto ha sido una experiencia enriquecedora, donde se integraron tecnologías como Reflex y PostgreSQL para crear una aplicación interactiva con LLMs. Sin embargo, se presentaron varios desafíos, especialmente con la integración de las IAs. Las preguntas generadas y las respuestas de las LLMs a menudo presentaron fallos y errores, lo que afectó la experiencia de usuario.

#### 4.1.1. Conclusiones

Durante el desarrollo de este proyecto se han integrado tecnologías como Reflex y PostgreSQL para crear una aplicación interactiva con LLMs. Sin embargo, se presentaron varios desafíos, especialmente con la integración de las IAs. Las preguntas generadas y las respuestas de las LLMs a menudo presentaron fallos y errores, lo que afectó la experiencia de usuario. Esto resalta que, aunque las LLMs son herramientas poderosas y prometedoras, todavía están en desarrollo y no son infalibles. Es crucial tener en cuenta sus limitaciones y no confiar ciegamente en ellas. A pesar de su potencial, el uso de LLMs requiere monitoreo y ajustes constantes para asegurar respuestas precisas y coherentes en aplicaciones como esta.

## 4.2. Álvaro Rojo Sánchez

- Propuesta y estructura del proyecto: como miembro con mayor experiencia técnica, propuso un entorno colaborativo donde todos los integrantes pudieran

contribuir equitativamente. Reconociendo que un proyecto web normalmente involucra múltiples tecnologías, sugirió utilizar una librería de Python para el desarrollo, decisión fundamentada en su experiencia previa con conexiones a bases de datos y la idoneidad de Python para interactuar con APIs de inteligencia artificial.

- Base de datos: en colaboración con Alejandro Relanzón Coello, implementaron una solución basada en PostgreSQL, definiendo una estructura que incluía una tabla principal para almacenar los datos de usuario junto con sus puntuaciones máximas. Esta elección tecnológica demostró ser adecuada para los requisitos del proyecto.
- Investigación e implementación de APIs: El desarrollo centró sus esfuerzos en dos modelos de inteligencia artificial: Gemini (de Google) y DeepSeek. La integración de Gemini destacó por su eficiencia, aprovechando el conector oficial de Python que permitió una generación rápida de preguntas con un formato predefinido. Por otro lado, DeepSeek se utilizó específicamente para implementar el comodín de llamada, aportando un mayor realismo al no depender del mismo modelo que generaba las preguntas.
- Optimización de respuestas: Un aspecto crítico fue el procesamiento de las respuestas generadas por las IAs, eliminando el “ruido” o información irrelevante que estos modelos suelen incluir. Se logró una mejora significativa en los tiempos de respuesta de DeepSeek, reduciéndolos de 15-20 segundos a 7-12 segundos, lo que hizo viable su uso en tiempo real durante el juego.
- Arquitectura del juego: Se diseñó una clase principal que encapsulaba toda la lógica del juego y las conexiones con las IAs, manteniendo una estructura genérica para facilitar su reutilización en los distintos modos de juego. Dos métodos destacables fueron: la validación de respuestas para avanzar entre preguntas, y la generación dinámica de preguntas mediante Gemini, que producía tanto la pregunta como cuatro posibles respuestas.
- Implementación de comodines: el sistema incorporó tres tipos de comodines:
  - 50/50: Eliminaba localmente dos respuestas incorrectas.
  - Público: Generaba porcentajes de votación mediante Gemini.
  - Llamada: Proporcionaba una respuesta experta usando DeepSeek.
- Modos de juego: para el *modo infinito* se inspiró en el modelo de Balatro, implementando rondas de tres preguntas con dificultad creciente hasta alcanzar un nivel máximo constante. En el *modo temático*, los jugadores seleccionaban entre dos temas por pregunta, con mecánicas que evitaban repeticiones y garantizaban variedad.
- Optimización de menús: se abordó la incoherencia existente entre el login, el registro y el menú principal. Se corrigieron mensajes que no se mostraban o se mostraban indebidamente al navegar entre estas secciones. Adicionalmente,

se creó una pantalla de inicio (`welcome_page`) que centralizaba los accesos al registro y al login. Se implementó también la funcionalidad de cierre de sesión (deslogueo).

- Refinamiento de código: se optimizó el código para eliminar redundancias. Se combinaron o eliminaron clases que no eran estrictamente necesarias para simplificar la estructura del proyecto. Finalmente, se realizó una revisión exhaustiva para corregir todos los *warnings* y errores existentes, mejorando la calidad general del código.
- Investigación técnica: se profundizó en el funcionamiento de los estados de `Reflex`, analizando sus diferencias con las clases Python estándar y su comportamiento durante la compilación y ejecución. Esta investigación fue crucial para integrar correctamente la lógica del juego con la interfaz web.
- Desafíos encontrados: durante el desarrollo del proyecto, se presentaron varios desafíos significativos:
  - El manejo de los estados (`State`) dentro de las clases de `Reflex` y su particular ciclo de vida.
  - La generación consistente y precisa de preguntas utilizando el modelo Gemini, asegurando el formato deseado.
  - La optimización del tiempo de respuesta del modelo DeepSeek para su viabilidad en interacciones en tiempo real.
  - El diseño de una lógica de juego general y modular que permitiera la fácil integración y prueba de nuevas IAs.
  - La gestión eficiente de las variables de estado a lo largo de los diferentes componentes de la aplicación.

### 4.2.1. Conclusiones

El proyecto demostró el potencial de las IAs para la generación dinámica de contenido, destacando la importancia de la optimización para obtener resultados útiles. La comparación entre Gemini y DeepSeek reveló las fortalezas particulares de cada modelo. La experiencia confirmó la versatilidad de Python como lenguaje unificador, permitiendo integrar componentes diversos a través de `Reflex` y sus librerías asociadas. En este proyecto ha sido interesante ver cómo se generan las preguntas y poder expresar la optimización de las distintas IAs para poder obtener las respuestas como queríamos y agilizar todo lo posible la generación de las mismas. También fue muy revelador el poder comparar las distintas IAs utilizadas, ya que cada una tiene sus ventajas, inconvenientes, diferente tiempo de ejecución y porcentaje de acierto. Por último, ha sido muy gratificante poder realizar la gran mayoría del proyecto en Python gracias a `Reflex` y a las distintas librerías que posee Python que nos permiten conectar todo bajo un mismo lenguaje.

### 4.3. Cintia Parreño Ruiz

- Desarrollo del modo temático: diseño, desarrollo e implementación del Modo Temático (Theme Mode). Esta modalidad de juego representa una mejora sustancial respecto a la estructura clásica del concurso, introduciendo un elemento estratégico y de personalización para el jugador.
  - Conceptualización y diseño: se concibió este modo como una forma de otorgar mayor agencia al participante. En lugar de seguir una secuencia de temas predefinida o puramente aleatoria, el Modo Temático permite al jugador, tras responder correctamente a una pregunta, elegir el tema de la siguiente pregunta entre dos opciones alternativas presentadas dinámicamente. Esta mecánica no solo aumenta la rejugabilidad, sino que también permite a los jugadores orientar el juego hacia áreas de su fortaleza o interés.
  - Implementación técnica: interfaz de Usuario (Front-End) diseñó e implementó los componentes visuales necesarios en la página específica (`theme_page.py`). Esto incluyó la presentación clara de las dos opciones de tema como botones interactivos después de cada respuesta correcta.
  - Gestión de estado (Back-End/State Management): Modificó y amplió el `GameState` para manejar la lógica específica de este modo. Implementó las funciones `get_themes` (para obtener las dos opciones temáticas generadas por la clase `Game`) y `set_theme` (para capturar la selección del usuario y actualizar el estado del juego consecuentemente). Aseguró que el estado reflejara correctamente el tema elegido para la siguiente ronda y gestionó la habilitación/deshabilitación de los botones de selección para guiar el flujo del juego.
  - Integración con la lógica del juego y la IA: Aseguró la correcta comunicación entre la interfaz, el `GameState` y la clase `Game`. La función `generate_question` fue adaptada o utilizada de manera que, en este modo, la generación de la siguiente pregunta por parte de la IA (Google Gemini / DeepSeek) utilizara el tema específicamente seleccionado por el jugador en el paso anterior, manteniendo la progresión de dificultad estándar (`generate_difficulty_normal_mode`).
- Estudio de proyectos emblemáticos: investigación detallada de sistemas de IA relevantes en el dominio de concursos y Q&A:
  - IBM Watson en Jeopardy!: análisis de su arquitectura DeepQA, enfocándose en su capacidad para procesar preguntas en lenguaje natural y extraer respuestas precisas de grandes volúmenes de datos no estructurados. Este estudio permitió extraer lecciones valiosas sobre preprocesamiento de preguntas y evaluación de confianza en respuestas.
  - LaMDA (Google): examen de sus capacidades conversacionales, con especial atención a fluidez, seguridad y coherencia de diálogo. Los hallazgos

sugirieron posibles mejoras futuras como comodines más interactivos mediante diálogos simulados.

- QANTA: investigación de este sistema académico especializado en trivia en tiempo real, destacando sus técnicas para extracción rápida de información y su relevancia para el formato de concurso implementado.
- Análisis de tecnologías fundamentales de IA: estudio profundo de las bases tecnológicas de sistemas modernos de Q&A:
  - Modelos Transformer: investigación de la arquitectura revolucionaria en NLP, con énfasis en su mecanismo de auto-atención para comprender capacidades y limitaciones de los modelos.
  - BERT: análisis de su enfoque bidireccional que mejora la comprensión contextual del lenguaje, esencial para interpretar matices en preguntas del concurso.
  - RoBERTa: evaluación de sus optimizaciones sobre BERT, demostrando cómo diferentes estrategias de preentrenamiento mejoran el rendimiento en tareas específicas de Q&A.
- Investigación sobre *frameworks* y arquitecturas emergentes:
  - Reflex: evaluación exhaustiva de sus capacidades como framework full-stack en Python, validando su idoneidad para el proyecto y documentando sus particularidades técnicas.
  - MCP (Model-Context-Protocol): estudio de patrones para gestión estructurada de interacciones entre modelos de IA, contexto del juego y APIs. Se investigaron métodos para: implementar integraciones robustas para aplicaciones con múltiples servicios de IA.
- Síntesis y aplicación directa en la memoria: integración de los hallazgos en el documento del proyecto:
  - Selección fundamentada de modelos de IA (Gemini, DeepSeek, OpenAi) frente a alternativas.
  - Contextualización realista de capacidades y limitaciones del proyecto.
  - Redacción rigurosa de capítulos teóricos y justificación tecnológica.

### 4.3.1. Conclusiones

Su contribución ha dotado al proyecto de una base teórica sólida y una perspectiva informada sobre el panorama de la inteligencia artificial aplicada a los sistemas de preguntas y respuestas. Su investigación sobre proyectos emblemáticos y tecnologías subyacentes no solo enriqueció la calidad de la memoria, sino que también influyó en la toma de decisiones técnicas y en la comprensión de las capacidades y limitaciones de las herramientas seleccionadas.

El desarrollo del Modo Temático es un claro ejemplo de cómo su investigación se tradujo en una funcionalidad concreta que mejora significativamente la experiencia

del usuario, añadiendo una capa de estrategia y personalización al juego. La dedicación a la configuración, redacción y estructuración de la memoria ha sido igualmente crucial, asegurando que el trabajo realizado por todo el equipo se presente de manera clara, coherente y profesional, esencial para la comunicación correcta de los resultados y aprendizajes del proyecto.

## 4.4. Miguel Macicior Cornide

Diseño, programación y depuración del **frontend** de la aplicación, así como programación y coordinación del **backend** con el **frontend**. Refactorización y optimización del código, testing y aportaciones en el manejo de la base de datos de la aplicación.

- **Diseño Visual:** Definió una paleta de colores inspirada en el programa de televisión, utilizando azul oscuro y dorado como colores principales, y naranja y blanco como secundarios. Diseñó los botones de la interfaz, buscando una estética similar a las opciones del concurso. Creó botones interactivos con imágenes para la temática de las preguntas, los logos y las acciones de la aplicación. Desarrolló menús verticales con opciones de navegación. Priorizó un diseño minimalista, donde se muestran solo los componentes esenciales y se evita el exceso de texto.
- **Programación con Reflex:** Generó diversas clases `*_page.py`, cada una representando una página de la aplicación, y en muchos casos, las acompañó con un archivo `*_state.py` para gestionar el estado de la página, siguiendo la estructura del *framework* Reflex. Entre las páginas desarrolladas se encuentran:
  - `welcome_page`: Página de recepción del usuario (en colaboración con Álvaro).
  - `register_page` y `login_page`: Páginas de registro e inicio de sesión (en colaboración con Álvaro y Alejandro).
  - `menu_page`: Página principal con las opciones de juego, acceso al perfil y estadísticas (en colaboración con Alejandro y otros).
  - `game_page`: Página principal del juego, que muestra las preguntas, opciones, *feedback* y comodines.(en colaboración con Álvaro)
  - `leaderboard_page`: Página que muestra la tabla de clasificación de los jugadores y se actualiza en tiempo real.
  - `user_page`: Página que muestra las estadísticas del usuario.
  - `maquinaVs_page`: Página para el modo “Máquina vs Máquina” (diseño de los componentes en colaboración con Alejandro).
- Implementación de una alta modularización del código y los componentes visuales, creando componentes reutilizables en `components.py`, como botones de comodín, selectores de categoría, indicadores de dificultad y botones de opción interactivos.

- Organización de las páginas del juego en paneles (superior, inferior, derecho, central e izquierdo), implementando un patrón fachada para adaptar la disposición de los paneles a las necesidades de cada modo de juego. Mantenimiento de la coherencia en el diseño de los menús de registro, inicio de sesión y juego. Creación de estilos CSS personalizados para elementos complejos como los botones hexagonales y para añadir dinamismo a la interacción con la interfaz (*hovers*).
- Reutilización del código en todo el **frontend**.
- Backend: colaboración en el desarrollo de la lógica del estado del juego (**GameState**). Participación en la optimización y refactorización del código del **backend**. Diseño de funciones para manejar la base de datos, específicamente para obtener el *ranking* global de jugadores y las estadísticas detalladas de cada jugador por tema.

En cuanto a los desafíos enfrentados:

- Aprendizaje de **Reflex**, incluyendo sus componentes, condicionales, estados y variables reactivas.
- Manejo de la concurrencia de procesos y la ejecución de acciones en segundo plano sin bloquear la interfaz.
- Implementación de estilos CSS complejos y dinámicos; diseño de formas que se salieran de los rectángulos convencionales en CSS.
- Gestión de variables de estado (reactivas, computadas, base).
- Resolución de errores al iniciar **Reflex**, como bloqueos de puertos y congelación de procesos.
- Adaptación de la interfaz a la variabilidad en la longitud de las preguntas y opciones generadas por las IAs.
- Diseño web *responsive*, adaptado a diferentes resoluciones de pantalla.
- Resolución de conflictos y coordinación entre diferentes estados de **Reflex**.
- Implementación de *feedback* para el usuario ante errores (fallos en las llamadas a la API, usuario no autenticado, etc.).

#### 4.4.1. Conclusiones

Este Trabajo de Fin de Grado demuestra que la Inteligencia Artificial ha trascendido su carácter experimental para convertirse en una fuerza verdaderamente transformadora: Una IA ya no solo puede generar texto o dibujar imágenes, ahora también puede ser presentadora de televisión. En el ámbito del desarrollo de software, lejos de suponer una amenaza para el ingeniero informático, la IA potencia su capacidad creativa y productiva: el ingeniero que maneje con soltura la formación

---

recibida alcanza niveles de eficiencia e innovación hasta ahora inimaginables; de lo contrario, incluso el mejor motor de generación producirá código deficiente. Sigue siendo necesario que el ingeniero elija los patrones de diseño adecuados —como los empleados en este proyecto (Observer, Fachada y Singleton)— y los aplique dentro de una arquitectura modular, eficiente y flexible que garantice tanto la mantenibilidad como la escalabilidad del sistema. Sigue siendo necesario que el ingeniero seleccione herramientas idóneas para estructurar sus desarrollos; en este TFG, Reflex ha permitido cohesionar todos los componentes en un entorno Python integrado. De cara al futuro, este tándem desarrollador–tecnología sentará las bases de flujos de futuras herramientas aún más sofisticadas y productivas. Sigue siendo necesario que alimentemos estas herramientas inteligentes con datos rigurosos y principios sólidos de calidad, para que sus resultados respondan a los estándares más exigentes y se traduzcan en soluciones fiables y de alto rendimiento. Enseñemos bien a las IAs y tendremos algo que aprender de ellas.



# Capítulo 5

## Conclusiones y Trabajo Futuro

Si bien la aplicación desarrollada cumple con los objetivos iniciales, esta sección se dedica a explorar un conjunto de mejoras y extensiones que podrían enriquecer significativamente la experiencia del usuario y la robustez del sistema. Se analizan tanto aspectos técnicos, como la escalabilidad y el rendimiento, como aspectos de usabilidad, como la incorporación de nuevas mecánicas de juego. El propósito es definir una visión a futuro para la aplicación.

### 5.1. Resultados

Los resultados obtenidos con la aplicación son muy favorecedores, ya que se ha cumplido el objetivo de crear una aplicación totalmente funcional con la que somos capaces de emular una partida de “¿Quién quiere ser millonario?” con una fluidez muy buena gracias a la elección de la inteligencia artificial (Gemini) con la que preguntamos y con la información obtenida con dos de los tres comodines. No obstante, el comodín de la llamada puede lastrar la ejecución debido a los segundos que debemos esperar (7-12 segundos) para obtener el resultado dado por la inteligencia artificial (Deepseek), siendo esta una espera aceptable debido a que en el juego mostrado por televisión es el comodín con el mayor tiempo en obtener una respuesta. Realizamos varios *tests* donde verificamos que realmente las preguntas que se realizan tienen coherencia y son correctas (véase figura 5.1), con lo que son aptas para su uso en la aplicación.

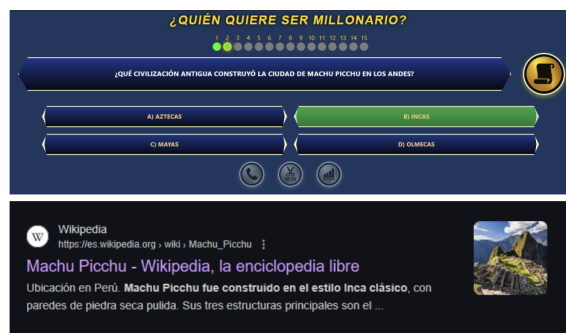


Figura 5.1: Coherencia en la respuesta

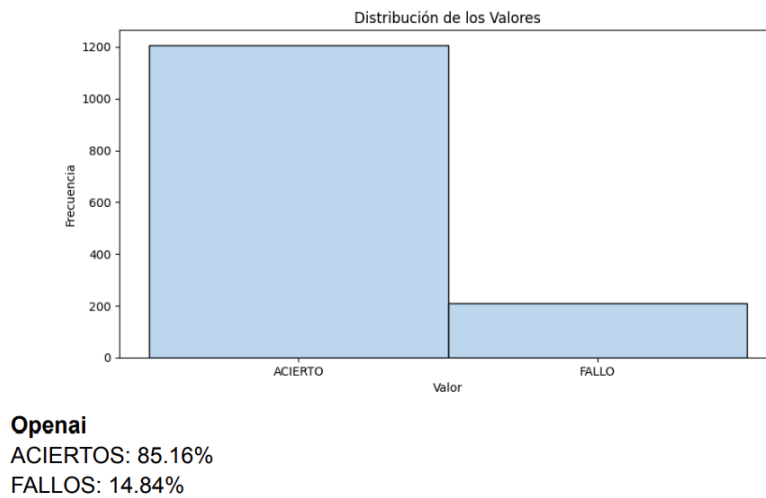
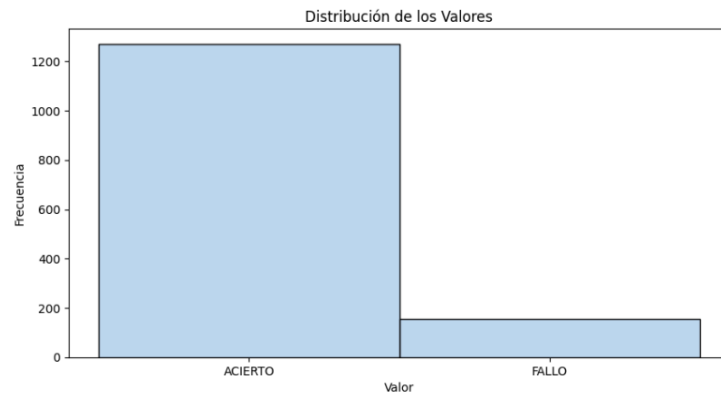


Figura 5.2: Aciertos con OpenAI

También validamos de la siguiente forma el resto de modos de juego para comprobar que efectivamente seguían sus propias reglas de juego:

1. Para el modo *endless* (infinito) comprobamos que era capaz de llegar a realizar más de 15 preguntas, y que al llegar a la pregunta 25 se lograra realizar preguntas con la máxima dificultad.
2. Para el modo temas comprobamos que en la elección de temas entre las 15 preguntas, nunca se repitieran temas de la anterior pregunta.
3. Para el modo máquina contra máquina comprobamos dos problemáticas que podrían tener lugar: debíamos comprobar que realmente las inteligencias artificiales fueran capaces de acertar las 15 preguntas de un cuestionario, lo que originó otra inquietud, ¿cuál es la capacidad de acierto de cada inteligencia artificial contra Gemini?
  - Para ello se realizaron unos entornos de test donde se enfrentaba cada inteligencia artificial con Gemini con un bloque de 1500 preguntas o dicho de otro modo 100 cuestionarios completos. Además, serían preguntas totalmente aleatorias donde se incluyen todos los temas.

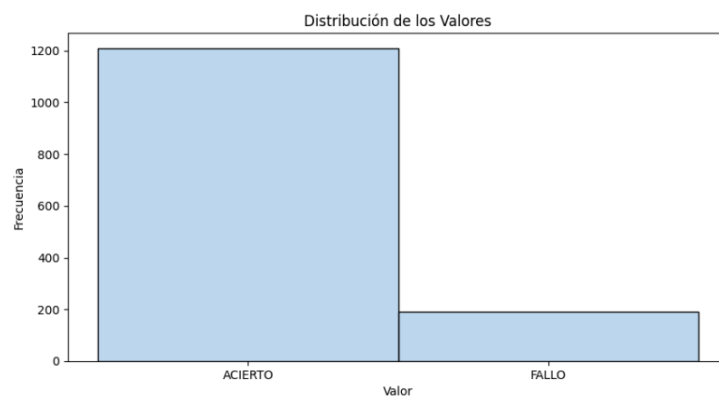
Con estos resultados podemos afirmar que Deepseek es la inteligencia artificial que más acierta (véase figura 5.3); siendo Openai la que más falla con una diferencia de 3,96 % (véase figura 5.2), que la inteligencia artificial más óptima con una relación porcentaje de aciertos / tiempo de ejecución de cada pregunta sería Llama (véase figura 5.4), y por último, podemos afirmar que las tres inteligencias artificiales tienen una alta tasa de acierto medio de 86,87 % frente a la tasa de fallo medio de 13,13 %, y por tanto son muy aptas para este modo de juego.

**Deepseek**

ACIERTOS: 89.12%

FALLOS: 10.88%

Figura 5.3: Aciertos con DeepSeek

**Llama**

ACIERTOS: 86.35%

FALLOS: 13.65%

Figura 5.4: Aciertos con Lllama

## 5.2. Mejoras futuras

Esta sección tiene como propósito explorar diversas propuestas de optimización y enriquecimiento de la implementación, las cuales se detallarán a lo largo de los apartados siguientes.

Estas mejoras no solo buscan corregir posibles limitaciones identificadas durante el desarrollo, sino también potenciar la experiencia del usuario y ampliar el alcance del sistema. Entre los aspectos a considerar se encuentran:

1. La escalabilidad de la aplicación.
2. La incorporación de funcionalidades avanzadas.
3. La mejora de la interacción con el usuario, todo ello manteniendo la simplicidad y eficacia que caracterizan al diseño actual.

Además, se plantean ideas para integrar tecnologías emergentes que podrían elevar el proyecto a un nivel superior de sofisticación y utilidad. Por ejemplo, la adición de elementos más complejos o la implementación de un sistema de almacenamiento de datos parapersonalizar la experiencia podrían transformar la aplicación en una herramienta más robusta y atractiva.

Estas propuestas se fundamentan en un análisis reflexivo de las posibilidades que ofrece el contexto tecnológico actual, así como en las necesidades potenciales de los usuarios finales. A lo largo de esta sección, se expondrán con detalle cada una de estas ideas, evaluando su viabilidad y los beneficios que podrían aportar al proyecto. De esta manera, se busca no solo consolidar lo ya logrado, sino también sentar las bases para una evolución continua que permita a la aplicación adaptarse a futuros desafíos y expectativas.

### 5.2.1. Temporizador

Uno de los elementos clave que define la dinámica de un juego como “¿Quién Quiere Ser Millonario?” es la presión temporal que experimentan los participantes al responder las preguntas, un factor que no solo incrementa la emoción, sino que también simula el desafío intelectual inherente al formato televisivo. En el contexto de esta aplicación, surge la posibilidad de enriquecer esta experiencia mediante la implementación de un *temporizador dinámico* que se ajuste en función de la dificultad de cada pregunta.

La idea central de esta propuesta es que el tiempo disponible para responder no sea estático, sino que varíe de manera proporcional al nivel de complejidad: a mayor dificultad, menor tiempo disponible, o incluso una aceleración en el contador para intensificar la sensación de urgencia. Esta funcionalidad, aunque no plenamente desarrollada en la versión inicial del proyecto, representa una mejora significativa que podría alinearse con los objetivos de ofrecer una experiencia más inmersiva y fiel al espíritu del juego original.

La introducción de un temporizador dinámico implicaría ajustar el tiempo base —por ejemplo, 20 segundos— mediante un algoritmo que reduzca este valor, como

por ejemplo implementar una resta de 1.5 segundos por cada nivel de dificultad, garantizando un mínimo razonable (como 5 segundos) para mantener la jugabilidad. Alternativamente, se podría explorar una aceleración progresiva del conteo, donde los segundos transcurran más rápido en preguntas difíciles, añadiendo un componente psicológico que desafíe aún más al usuario.

Evaluar esta posibilidad requiere considerar tanto los aspectos técnicos, como la precisión del temporizador, como la experiencia del usuario, asegurando que el ajuste del tiempo sea intuitivo y equilibrado (Véase apéndice B.1).

### 5.2.2. Historial de palabras clave

Uno de los aspectos fundamentales para garantizar una experiencia atractiva y dinámica es la diversidad de las preguntas presentadas al usuario durante una sesión de juego. Investigando posibles mejoras para este proyecto, surgió la idea de incorporar un sistema de historial que registre las preguntas ya utilizadas, permitiendo evitar repeticiones y asegurando que cada nueva pregunta sea única en el contexto de la partida actual.

Esta funcionalidad se basa en almacenar no solo las preguntas completas, sino también un conjunto de palabras clave extraídas de ellas, las cuales servirán como base para compararlas con las preguntas generadas posteriormente. El objetivo principal es descartar cualquier pregunta nueva que contenga similitudes significativas con las anteriores, utilizando un criterio de coincidencia de palabras clave que determine si existe una repetición temática o conceptual.

Los beneficios de esta propuesta incluyen:

1. Evitar que el usuario perciba redundancia, manteniendo el interés a lo largo de la partida.
2. Optimizar la experiencia de juego sin depender exclusivamente de una base de datos extensa, ya que el filtrado se realiza en tiempo real dentro de la sesión.

La integración de este mecanismo en **Reflex** resulta especialmente interesante debido a su modelo reactivo, que facilitaría actualizar el historial y reflejar los cambios en la interfaz de manera inmediata. Desde un punto de vista técnico, esta mejora también permite explorar conceptos como el procesamiento básico de texto o la gestión de estructuras de datos dinámicas. Estos elementos enriquecen el proyecto con un enfoque más robusto y profesional (para ejemplo práctico consultar apéndice B.2).

### 5.2.3. Sistema de bonificación por racha

Se busca incentivar a los jugadores a mantener una cadena de respuestas correctas consecutivas, premiándolos por su consistencia. La idea consiste en rastrear cuántas preguntas ha respondido correctamente el jugador sin fallar y, al alcanzar un umbral mínimo (por ejemplo, 20 aciertos seguidos), otorgar una recompensa. Esta recompensa podría ser puntos adicionales (como duplicar el valor de la siguiente pregunta) o un comodín adicional. Para hacerlo interesante, el sistema podría escalar:

por ejemplo, 20 aciertos dan un pequeño bono, 25 aciertos uno mayor, y 30 aciertos algo más significativo, como un “*escudo*” que protege de un fallo.

En el contexto de **Reflex**, esto añadiría un elemento de estrategia y emoción, ya que los jugadores sentirían la presión y la satisfacción de mantener su racha. Además, se podría mostrar la racha actual en la interfaz para que el usuario la vea en tiempo real, aumentando la inmersión. La implementación en *Python* sería sencilla: bastaría con una variable que cuente las respuestas correctas consecutivas y un condicional que active el bono al alcanzar el umbral. Para evitar que el sistema sea demasiado generoso, se podría reiniciar la racha al fallar una pregunta, lo que mantendría el desafío. Este mecanismo no solo hace el juego más dinámico, sino que también recompensa la habilidad y el enfoque del jugador, elevando la experiencia general (para ejemplo práctico consultar apéndice B.3).

#### 5.2.4. Posibilidad de Guardar Partida

Esta mejora permitiría a los usuarios pausar su progreso en el juego y retomarlo posteriormente desde el mismo punto. Esto es especialmente útil para partidas largas o cuando los usuarios no disponen del tiempo necesario para completarlas en una sola sesión. La implementación de esta funcionalidad mejoraría significativamente la experiencia del usuario, proporcionando mayor flexibilidad y comodidad. Para desarrollar esta mejor, se necesitaría la creación de una nueva tabla en la base de datos para almacenar el estado de las partidas guardadas. Esta tabla estaría relacionada con la tabla de usuarios mediante una *clave foránea*, permitiendo asociar cada partida guardada a un usuario específico. Los campos de la tabla podrían incluir:

- `id_partida` (clave primaria, entero).
- `id_usuario` (*clave foránea*, entero, referencia a la tabla de usuarios).
- `modo_juego` (cadena, indica el modo de juego en curso).
- `numero_pregunta` (entero, número de la pregunta actual).
- `pregunta` (cadena, texto de la pregunta actual).
- `opciones` (cadena o JSON, almacena las opciones de respuesta).
- `respuesta_correcta` (cadena, la respuesta correcta).
- `tema` (cadena, el tema de la pregunta actual).
- `flags_comodines` (JSON, almacena el estado de los comodines utilizados).
- `fecha_guardado` (fecha y hora del guardado).

Al seleccionar un modo de juego, la aplicación verificaría si existe una partida guardada para el usuario en ese modo. Si existe una partida guardada, se ofrecería al usuario la opción de cargarla o comenzar una nueva partida. Si el usuario elige cargar la partida guardada, se recuperaría el estado del juego de la base de datos

y se restaurarían las variables de estado correspondientes (en `GameState` o similar). Se implementarían funciones para guardar el estado del juego en la base de datos en momentos específicos (por ejemplo, al pulsar un botón de “Guardar Partida”, al cerrar la aplicación, o automáticamente cada cierto tiempo).

También se podría implementar un sistema para gestionar varias partidas guardadas, permitiendo al usuario elegir cuál cargar. Se debería considerar la seguridad de los datos guardados, especialmente si se almacenan datos sensibles como las respuestas del usuario. Se podría implementar un sistema de auto-guardado para evitar la pérdida de progreso en caso de cierre inesperado de la aplicación.

### 5.2.5. Selección del Modelo de IA

Esta mejora permitiría a los usuarios personalizar el modo “Máquina vs Máquina” seleccionando qué modelos de IA desean enfrentar entre sí. Actualmente, el modo tiene una configuración predefinida de IAs. Permitir la selección de modelos aumentaría la flexibilidad y el valor educativo del modo, permitiendo a los usuarios comparar las fortalezas y debilidades de diferentes modelos de IA.

- *Interfaz de Usuario:* se modificaría la página `maquinaVS_page.py` para presentar al usuario una interfaz que le permita seleccionar los modelos de IA para la partida. Esto podría implementarse como un formulario con listas desplegables o botones de selección para cada IA (Gemini, DeepSeek, OpenAI, Llama). Si Gemini se utiliza para la generación de preguntas, se podría incluir también la opción de seleccionar entre diferentes versiones o configuraciones de Gemini (por ejemplo, “gemini-2.0-flash”, “gemini-2.5”).
- *Lógica de la Aplicación:* se modificaría la función `GameState.initialize_game` para recibir como parámetros los modelos de IA seleccionados por el usuario. La clase `GameState` se adaptaría para utilizar los modelos de IA especificados en lugar de depender de una configuración predeterminada.

Se podrían implementar funciones para:

- Obtener una lista de los modelos de IA disponibles y sus características (para mostrar al usuario en la interfaz).
- Manejar la lógica de llamada a las *APIs* de los modelos de IA seleccionados, adaptando los *prompts* y las solicitudes según las necesidades de cada modelo.

Se podría implementar un sistema para mostrar las características de cada modelo de IA (por ejemplo, velocidad, precisión, costo) para ayudar al usuario a tomar una decisión informada. Por otro lado, se podría implementar una función para registrar los resultados de las partidas entre diferentes modelos de IA para su análisis.

## 5.3. Conclusiones

El desarrollo de la aplicación ha representado un ejercicio desafiante, logrando con éxito el objetivo principal de crear una plataforma atractiva y funcional. Este

proyecto no sólo ha permitido recrear la mecánica del programa de televisión, sino también explorar la integración y el potencial de *modelos de lenguaje a gran escala* (LLMs) e inteligencia artificial en la generación de contenido y la mejora de la experiencia de usuario.

- Desarrollo integral: se ha completado el diseño e implementación tanto del *frontend* como del *backend* de la aplicación, utilizando Python y Reflex, lo que ha permitido un desarrollo unificado y eficiente.
- Experiencia de usuario fiel al programa original: la aplicación replica con fidelidad la dinámica del concurso, incluyendo la presentación de preguntas con dificultad creciente, opciones de respuesta y los comodines (50:50, llamada y pregunta al público).
- Integración de IA: se han integrado con éxito múltiples modelos de IA para la generación dinámica de preguntas y la simulación de los comodines y como concursantes en el modo “Máquina vs Máquina”. Este aspecto ha sido fundamental para ofrecer una experiencia de juego variada y para explorar las capacidades y desafíos de diferentes IAs.
- Modos de Juego: se han implementado varios modos de juego para enriquecer la experiencia:
  - Modo Infinito: inspirado en juegos como Balatro, ofrece un desafío continuo con dificultad progresiva.
  - Modo Normal: replica la estructura clásica del concurso con 15 preguntas.
  - Modo Temático: permite al jugador influir en la temática de las preguntas, pudiendo elegir entre dos temas.
  - Modo Máquina vs Máquina: ofrece una plataforma para observar y comparar el rendimiento de diferentes IAs respondiendo a las preguntas generadas por Gemini.
- Optimización y gestión de desafíos: se abordaron desafíos significativos como la unicidad de las preguntas, la validación precisa de respuestas mediante *NLP*, la optimización de los *prompts* para mejorar la latencia y la calidad de las respuestas de las IAs (especialmente con DeepSeek), y la gestión de la variabilidad en las respuestas de los LLMs.

El proyecto ha puesto de manifiesto el considerable potencial de las IAs para la generación dinámica de contenido y la creación de experiencias interactivas. La elección de Python y Reflex demostró ser acertada, facilitando la integración de diversos componentes y tecnologías. La interacción con diferentes *APIs* de IA (Gemini, DeepSeek, OpenAI, Llama) ha proporcionado una valiosa experiencia en el diseño de *prompts* efectivos y en la gestión de las particularidades de cada modelo, subrayando que, si bien son herramientas poderosas, aún se encuentran en desarrollo y requieren supervisión y ajustes para garantizar precisión y coherencia.

Con la vista puesta en la evolución continua y el enriquecimiento de la aplicación, se han identificado diversas áreas y funcionalidades que podrían incorporarse

en futuras iteraciones del proyecto. Estas propuestas buscan no solo ampliar las capacidades actuales, sino también mejorar la experiencia del usuario y la robustez general del sistema:

- Implementación de un *Temporizador*: para añadir emoción y ajustar el tiempo de respuesta según la dificultad de la pregunta.
- *Historial* de palabras clave: evitando la repetición de preguntas durante una partida.
- Sistema de *Bonificación* por racha: para incentivar y recompensar la consistencia en las respuestas correctas.
- Guardar Partida: para ofrecer mayor flexibilidad a los usuarios.
- Selección de Modelos de IA: aumentando el valor educativo y el análisis de eficiencia de las diferentes inteligencias.

En conclusión, este Trabajo de Fin de Grado no sólo ha culminado en una aplicación funcional y entretenida, sino que también ha servido como un valioso campo de pruebas para la integración y comparación de diversas tecnologías de IA en el contexto de un concurso de preguntas y respuestas, sentando las bases para futuras exploraciones y desarrollos en este apasionante campo.



# Introduction

This final degree project focuses on the development of a web application that recreates the television quiz show “Who Wants to Be a Millionaire?”. This project encompasses both the design and implementation of the frontend and the backend of the application. The main objective of this project is to create an attractive and intuitive user experience that allows users to participate in the quiz show as if they were on the television program. This involves the faithful recreation of the game mechanics, including the questions, the answer options, the lifelines. To achieve this objective, the programming language Python and the **Reflex** framework have been chosen. Python is a widely used language in web development due to its versatility, its module library, and its ease of learning. **Reflex**, on the other hand, is a framework that allows for the efficient creation of interactive and dynamic web applications. In other words, the choice of Python and **Reflex** is justified by their ability to offer a robust and flexible development environment, which allows for addressing both the development of the frontend and the backend of the application in an integrated manner. Furthermore, these technologies are widely used in the industry, which guarantees the relevance and applicability of the knowledge acquired in this project. Throughout this document, the different aspects of the application’s development will be detailed, including the design of the architecture, the implementation of the game logic, the creation of the user interface, the management of the database, and the tests carried out to guarantee the quality and correct functioning of the application. Furthermore, it is believed that this project can serve as a useful and entertaining tool for users, allowing them to enjoy the “Who Wants to Be a Millionaire?” quiz show from the comfort of their homes.



# Conclusions and Future Work

While the developed application meets the initial objectives, this section is dedicated to exploring a set of improvements and extensions that could significantly enrich the user experience and the system’s robustness. Both technical aspects, such as scalability and performance, and usability aspects, such as the incorporation of new game mechanics, are analyzed. The purpose is to define a future vision for the application.

## 5.4. Conclusions

The development of the application has represented a challenging endeavor, successfully achieving the main objective of creating an attractive and functional platform. This project has not only allowed for the recreation of the television show’s mechanics but also for the exploration of the integration and potential of *large language models* (LLMs) and artificial intelligence in content generation and user experience enhancement.

- **Comprehensive development:** The design and implementation of both the *frontend* and *backend* of the application have been completed using Python and **Reflex**, which has enabled unified and efficient development.
- **User experience faithful to the original program:** The application faithfully replicates the dynamics of the quiz show, including the presentation of questions with increasing difficulty, answer options, and lifelines (50:50, phone-a-friend, and ask the audience).
- **AI Integration:** Multiple AI models have been successfully integrated for the dynamic generation of questions, the simulation of lifelines, and as contestants in the “Machine vs. Machine” mode. This aspect has been fundamental in offering a varied gaming experience and in exploring the capabilities and challenges of different AIs.
- **Game Modes:** Several game modes have been implemented to enrich the experience:

- Infinite Mode: Inspired by games like Balatro, it offers a continuous challenge with progressive difficulty.
  - Normal Mode: Replicates the classic quiz show structure with 15 questions.
  - Thematic Mode: Allows the player to influence the theme of the questions, being able to choose between two topics.
  - Machine vs. Machine Mode: Offers a platform to observe and compare the performance of different AIs answering questions generated by Gemini.
- Optimization and challenge management: Significant challenges were addressed, such as question uniqueness, accurate answer validation using *NLP*, *prompt* optimization to improve latency and the quality of AI responses (especially with DeepSeek), and managing the variability in LLM responses.

The project has highlighted the considerable potential of AIs for dynamic content generation and the creation of interactive experiences. The choice of Python and Reflex proved to be sound, facilitating the integration of diverse components and technologies. Interaction with different AI *APIs* (Gemini, DeepSeek, OpenAI, Llama) has provided valuable experience in designing effective *prompts* and in managing the peculiarities of each model, underscoring that, while they are powerful tools, they are still under development and require supervision and adjustments to ensure accuracy and consistency.

Looking towards the continuous evolution and enrichment of the application, various areas and functionalities have been identified that could be incorporated in future project iterations. These proposals aim not only to expand current capabilities but also to improve the user experience and the overall robustness of the system:

- Implementation of a *Timer*: to add excitement and adjust response time according to question difficulty.
- Keyword *History*: to avoid question repetition during a game session.
- Streak *Bonus* System: to incentivize and reward consistency in correct answers.
- Save Game: to offer greater flexibility to users.
- AI Model Selection: increasing the educational value and efficiency analysis of different AIs.

In conclusion, this Final Degree Project has not only culminated in a functional and entertaining application but has also served as a valuable testing ground for the integration and comparison of various AI technologies in the context of a question-and-answer quiz show, laying the groundwork for future explorations and developments in this exciting field.

# Bibliografía

- BIURRUN, A. Un ingeniero de google cree que su inteligencia artificial es consciente y publica sus conversaciones con ella. *La Razón*, 2022.
- BROWN, T. B., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J., DHARIWAL, P. y OTHERS. Language models are few-shot learners. En *Advances in Neural Information Processing Systems* (editado por H. Larochelle, V. Dumoulin, A. Mishra, A. Courville, J. Pineau y Y. Bengio). Curran Associates, Inc., 2020.
- COMBINATOR, Y. Reflex - yc w23. *YCombinator*, 2023.
- DEVLIN, J., CHANG, M.-W., LEE, K. y TOUTANOVA, K. BERT: Pre-training of deep bidirectional transformers for language understanding. *ACL Anthology*, 2018.
- DOLAN, E. W. AI models struggle with expert-level global history knowledge. *psypost*, 2025.
- FERRUCCI, D., BROWN, E., CHU-CARROLL, J., FAN, J., GONDEK, D., KALYANPUR, A., LALLY, A., MURDOCK, J. W., NYBERG, E., PRAGER, J. y OTHERS. Building watson: An overview of the deepqa project. *AI Magazine*, vol. 31(3), páginas 59–79, 2010.
- FIRESHIP. Protocolo de contexto de modelo explicado. *Fireship*, 2025.
- GRIGGS, M. B. Ibm watson can help find water wasters in drought-stricken california. *popsci*, 2021.
- JOSHI, M., CHOI, E., WELD, D. S. y ZETTLEMOYER, L. A large scale distantly supervised challenge dataset for reading comprehension. *University of Texas at Austin*, 2017.
- KSENIASE. What is MCP, and why is everyone – suddenly!– talking about it? *Huggingface*, 2025.
- LEO, I. Understanding Model Context Protocol (MCP). *python.useinstructor*, 2025.
- LIU, Y., OTT, M., GOYAL, N., DU, J., JOSHI, M., CHEN, D., LEVY, O., LEWIS, M., ZETTLEMOYER, L. y STOYANOV, V. RoBERTa: A robustly optimized bert pretraining approach. *dblp*, 2019.

- NOVITAAI. Presentación del modelo base RoBERTa: una descripción general completa. *blogs.novita.ai*, 2024.
- RAO, N. Pynecone is now reflex. <https://reflex.dev/blog/2023-06-28-rebrand-to-reflex>, 2023.
- REFLEX. Custom components. *Reflex*, 2023a.
- REFLEX. How reflex works. *Reflex*, 2023b.
- SCHMID, P. Model context protocol (MCP) an overview. *philschmid*, 2025.
- THOPPILAN, R. y OTHERS. LaMDA: Language Models for Dialog Applications. <https://s10251.pcdn.co/pdf/2022-thoppilan-lamda.pdf>, 2022.
- VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L. y POLOSUKHIN, I. Attention is all you need. En *31st Conf. on Neural Information Processing Systems, NIPS'17*, páginas 1–11. Curran Associates, Inc., 2017.
- WIKIPEDIA (Jeopardy!). Entrada: “Jeopardy!”.
- WIKIPEDIA (LaMDA!). Entrada: “LaMDA!”.
- WIKIPEDIA (PostgreSQL). Entrada: “PostgreSQL”.
- WIKIPEDIA (Protocolo de Contexto de Modelo). Entrada: “Protocolo de Contexto de Modelo”.
- WIKIPEDIA (Quiz bowl). Entrada: “Quiz bowl”.
- WIKIPEDIA (Transformador (modelo de aprendizaje automático)). Entrada: “Transformador (modelo de aprendizaje automático)”.
- WIKIPEDIA. Bert (modelo de lenguaje). *Wikipedia*, 2019.
- ZHOU, L., SCHELLAERT, W., MARTÍNEZ-PLUMED, F., MOROS-DAVAL, Y., FERRI, C. y HERNÁNDEZ-ORALLO, J. Larger and more instructable language models become less reliable. *Nature*, 2024.
- ZILBER, A. AI chatbots still can't accurately answer high-level history questions: study. *New York Post*, 2025. Available at <https://nypost.com/2025/01/20/business/ai-still-cant-answer-questions-about-history-study/>.

## Imágenes adjuntas

### Resultados del comienzo sin optimizar

- **Número de datos (n):** 60
  - **Media (promedio):**  $\approx 9,83$
  - **Mediana:**  $\approx 9,66$
  - **Mínimo:** 8.19
  - **Máximo:** 13.69
  - **Rango:** 1.42
  - **Desviación estándar:** 1.08
- Interpretación:**
- **Distribución:** Asimétrica positiva (cola larga hacia la derecha).
  - **Outliers:** 13.69 (marcado como punto)
  - **Moda:** 9.45 (valor más frecuente)

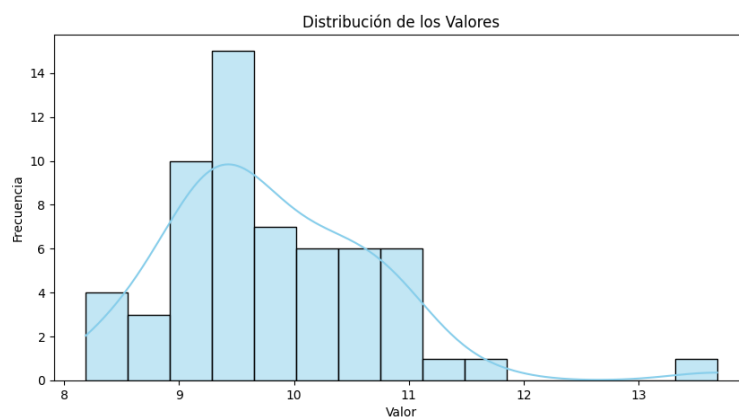


Figura A.1: Gráfica de los resultados del comienzo sin optimizar

### Resultados optimizados

- **Número de datos (n):** 60
- **Media (promedio):**  $\approx 5,51$
- **Mediana:**  $\approx 5,35$
- **Mínimo:** 4.53
- **Máximo:** 8.24
- **Rango:** 3.71
- **Desviación estándar:** 0.74

### Interpretación:

- **Distribución:** Los datos están ligeramente sesgados hacia la derecha (cola larga en valores altos), con una concentración entre 4.5 y 6.0.
- **Outliers:** Hay algunos valores atípicos (ej. 8.24, 7.46, 6.91) que elevan la media.
- **Moda:** La mayoría de los valores se agrupan alrededor de 5.0–5.5.

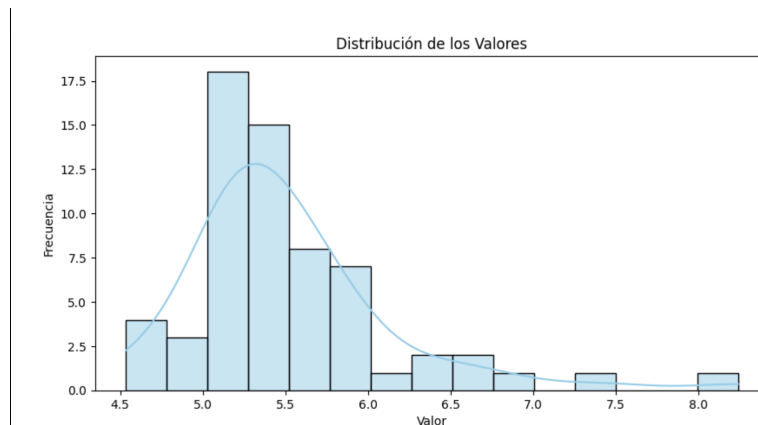


Figura A.2: Gráfica optimizada

✎ conoces quien quiere ser millonario?

◆ Sí, conozco "¿Quién quiere ser millonario?". Es un famoso concurso de televisión internacional en el que los concursantes responden a preguntas de dificultad creciente para ganar grandes sumas de dinero.

En España, el programa ha tenido varias etapas y se ha emitido en diferentes cadenas con distintos presentadores. Actualmente, se emite en laSexta y está presentado por Juanra Bonet.

¿Hay algo específico que te gustaría saber sobre el programa?

Figura A.3: Desarrollo de la optimización

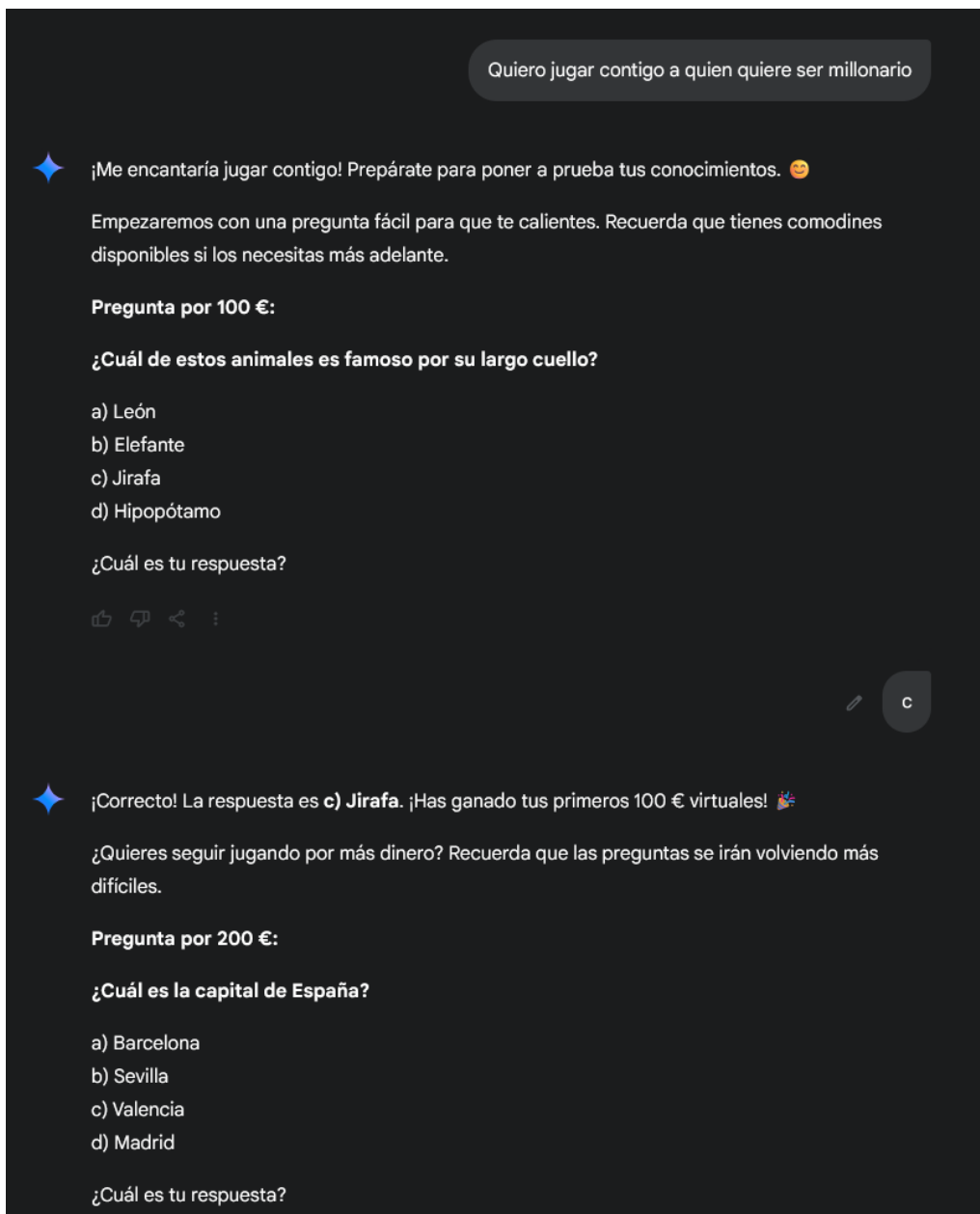


Figura A.4: Optimización del prompt

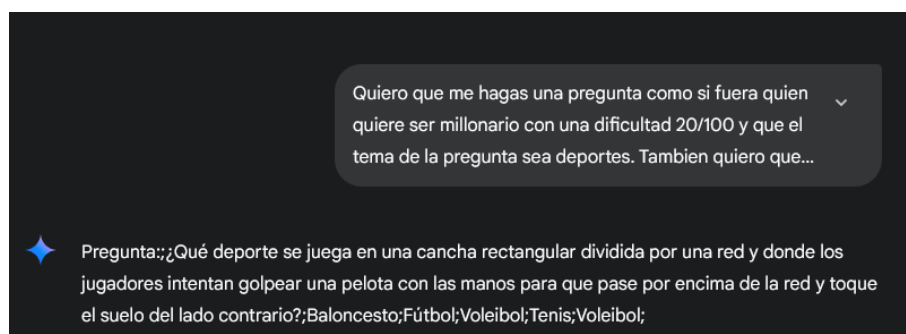


Figura A.5: Optimización con prompt mejorado

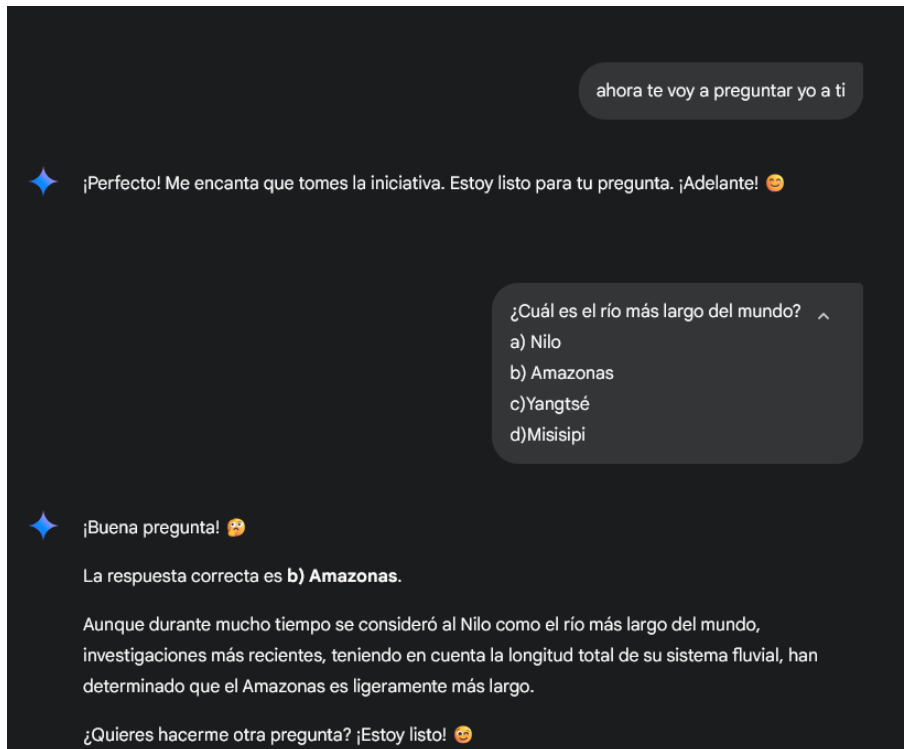


Figura A.6: Pregunta inversa para la optimización, IA responde correctamente

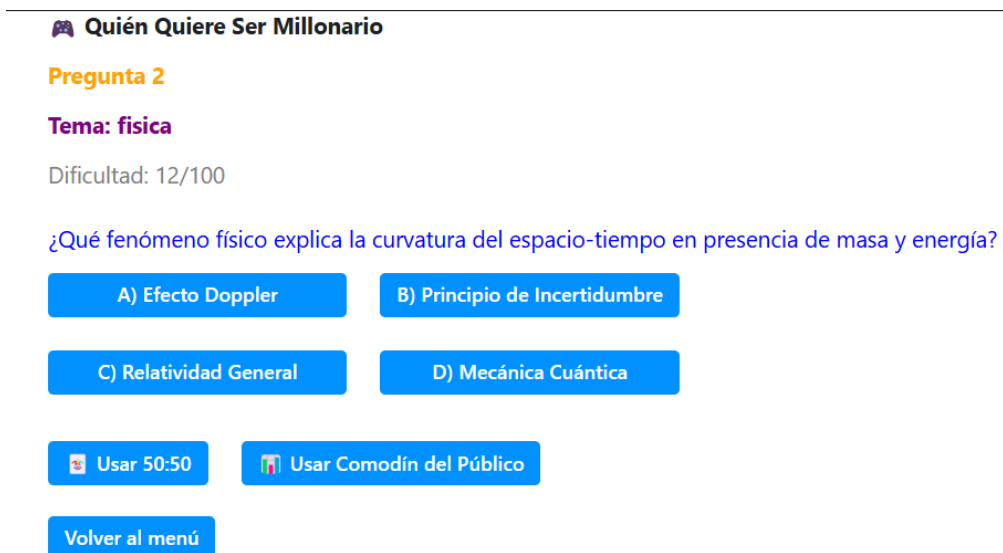


Figura A.7: Front al principio del desarrollo



Figura A.8: Página principal de la aplicación



## Implementaciones

### B.1. Implementación básica para Temporizador

Aquí se detalla una posible implementación básica de un temporizador creada con ayuda de la inteligencia artificial de la página oficial de Reflex.

El tiempo inicial es 20 segundos, pero se reduce según la dificultad (mínimo 5 segundos) usando una fórmula que resta 1.5 segundos por cada nivel de dificultad. Cuando el usuario inicia una pregunta, un *temporizador asíncrono* comienza a contar hacia atrás, actualizando el tiempo cada segundo hasta que se agota o el usuario responde.

El comportamiento del temporizador incluye:

1. Si el tiempo se acaba, muestra un mensaje de “*¡Se acabó el tiempo!*”.
2. Si el usuario responde antes, detiene el conteo y muestra el *tiempo restante*.

La dificultad se puede cambiar mediante un campo de entrada, ajustando el tiempo disponible para la siguiente pregunta. La interfaz, creada con componentes de Reflex, incluye:

1. Un *título*.
2. El *tiempo restante*.
3. Un *mensaje*.
4. Un campo para la *dificultad*.
5. Botones para *iniciar* y *responder*.

Los botones se activan o desactivan según si hay una pregunta en curso. Todo esto se logra con un enfoque *reactivo*, donde los cambios en el estado actualizan automáticamente la interfaz, y el uso de `asyncio` asegura que el temporizador no bloquee la aplicación. Es un ejemplo funcional que simula la *dinámica de tiempo* del juego.

Ejemplo de implementación:

```
1 import reflex as rx
2 import asyncio
3 import time
4
5 # Estado para manejar el juego
6 class JuegoState(rx.State):
7     tiempo_restante: int = 20 # Tiempo inicial
8     dificultad: int = 1 # Dificultad inicial (1-10)
9     pregunta_activa: bool = False
10    mensaje: str = ""
11
12    def ajustar_temporizador(self):
13        "Ajusta el tiempo inicial según la dificultad."
14        # Mínimo 5 segundos
15        self.tiempo_restante = max(5, 20 - int(self.dificultad * 1.5))
16
17    async def iniciar_pregunta(self):
18        """Inicia el temporizador para la pregunta."""
19        if not self.pregunta_activa:
20            self.pregunta_activa = True
21            self.mensaje = ""
22            self.ajustar_temporizador()
23            while self.tiempo_restante > 0 and self.pregunta_activa:
24                await asyncio.sleep(1) # Espera 1 segundo
25                self.tiempo_restante -= 1
26                if self.pregunta_activa: # Si no se detuvo manualmente
27                    self.pregunta_activa = False
28                    self.mensaje = "¡Se acabó el tiempo!"
29
30    def responder(self):
31        "Detiene el temporizador cuando el usuario responde."
32        if self.pregunta_activa:
33            self.pregunta_activa = False
34            self.mensaje = f"Respuesta enviada. Tiempo restante:
35                {self.tiempo_restante} segundos."
36
37    def cambiar_dificultad(self, nueva_dificultad: str):
38        "Actualiza la dificultad basada en la entrada del usuario."
39        try:
40            # Entre 1 y 10
41            self.dificultad = min(max(1, int(nueva_dificultad)), 10)
42            self.ajustar_temporizador()
43        except ValueError:
44            self.dificultad = 1
45
46    # Componente de la interfaz
47    def interfaz_juego():
48        return rx.vstack(
```

```

49     rx.heading("¿Quién Quiere Ser Millonario?"),
50     rx.text(f"Tiempo restante: {JuegoState.tiempo_restante} segundos"),
51     rx.text(JuegoState.mensaje),
52     rx.hstack(
53         rx.text("Dificultad (1-10):"),
54         rx.input(
55             placeholder="Ingresa dificultad",
56             on_change=JuegoState.cambiar_dificultad,
57             default_value="1"
58         ),
59     ),
60     rx.button(
61         "Iniciar Pregunta",
62         on_click=JuegoState.iniciar_pregunta,
63         is_disabled=JuegoState.pregunta_activa,
64     ),
65     rx.button(
66         "Responder",
67         on_click=JuegoState.responder,
68         is_disabled=~JuegoState.pregunta_activa,
69     ),
70     spacing="20px",
71     align_items="center",
72 )
73
74 # Crear la aplicación
75 app = rx.App()
76 app.add_page(interfaz_juego)

```

## B.2. Implementación básica para Historial

Se detalla a continuación un ejemplo de código realizado con la ayuda de la inteligencia artificial que está en la página de Reflex para implementar un historial.

El código propuesto simula un sistema para evitar preguntas repetidas en el juego. La función `generar_pregunta_simulada()` selecciona aleatoriamente una pregunta de una lista predefinida, como simulacro de un generador de preguntas. Luego, `extraer_palabras_clave()` procesa una pregunta eliminando signos de puntuación y palabras comunes (artículos, preposiciones), devolviendo un conjunto de palabras relevantes (ej. `capital`, `Francia`). La clase `HistorialPreguntas` gestiona el historial: `self.historial` almacena diccionarios con cada pregunta y sus palabras clave. El método `agregar_pregunta()` añade una nueva entrada al historial tras extraer sus palabras clave. El método `es_repetida()` compara una nueva pregunta con el historial, calculando la intersección de sus palabras clave con las almacenadas; si hay más de una coincidencia (umbral ajustable), la marca como repetida. Esto permite descartar preguntas similares (ej. `"¿Cuál es la capital de Francia?"` `"¿Qué capital tiene Francia?"`). El uso de conjuntos (set) optimiza las comparaciones. Aunque es un ejemplo básico, podría integrarse en Reflex actualizando el estado reactivo para

reflejar el historial y generar nuevas preguntas en tiempo real, asegurando variedad en cada partida.

---

```

1 import random
2
3 # Simulamos una función que genera preguntas (podría ser tu IA)
4 def generar_pregunta_simulada():
5     preguntas_ejemplo = [
6         "¿Cuál es la capital de Francia?",
7         "¿Qué planeta es el más grande del sistema solar?",
8         "¿En qué año llegó el hombre a la luna?",
9         "¿Cuál es el río más largo del mundo?",
10        "¿Qué país tiene más habitantes?"
11    ]
12    return random.choice(preguntas_ejemplo)
13
14 # Función para extraer palabras clave de una pregunta
15 def extraer_palabras_clave(pregunta):
16     # Eliminamos signos de puntuación y convertimos a minúsculas
17     pregunta = pregunta.lower().replace("¿", "")
18     .replace("?", "").replace(", ", "").strip()
19     # Dividimos en palabras y filtramos las más relevantes
20     palabras_comunes = {"es", "la", "el", "de", "en", "qué", (...)}
21     palabras = set(pregunta.split()) - palabras_comunes
22     return palabras
23
24 # Clase para manejar el historial y verificar repeticiones
25 class HistorialPreguntas:
26     def __init__(self):
27         self.historial = []
28     # Lista de diccionarios con pregunta y palabras clave
29
30     def agregar_pregunta(self, pregunta):
31         palabras_clave = extraer_palabras_clave(pregunta)
32         self.historial.append({"pregunta": pregunta,
33                                "palabras_clave": palabras_clave})
34
35     def es_repetida(self, nueva_pregunta):
36         nuevas_palabras = extraer_palabras_clave(nueva_pregunta)
37         for entrada in self.historial:
38             # Calculamos la intersección de palabras clave
39             coincidencias = nuevas_palabras.intersection
40             (entrada["palabras_clave"])
41     # Si hay más de un cierto número de coincidencias, consideramos que es repetida
42         if len(coincidencias) > 1: # Puedes ajustar este umbral
43             return True
44         return False
45

```

---

## B.3. Implementación básica para Bonificación

Generación de un código Reflex básico para la posible implementación en un futuro de un sistema de premios en el juego. El código define una aplicación Reflex con un estado que rastrea el número de preguntas correctas y el premio acumulado. La función `responder_correctamente` incrementa el contador de preguntas correctas y actualiza el premio usando la función `calcular_premio`. Si el jugador responde 20 preguntas correctamente, gana el premio mayor. La función `responder_incorrectamente` reinicia el contador y el premio. La función `calcular_premio` determina el premio basado en el número de preguntas correctas, con incrementos exponenciales en los premios a medida que el jugador avanza. La interfaz de usuario muestra el número de preguntas correctas, el premio acumulado y dos botones para simular respuestas correctas e incorrectas.

---

```
1 import reflex as rx
2
3 class State(rx.State):
4     preguntas_correctas: int = 0
5     premio_acumulado: int = 0
6     premio_mayor: int = 1000000
7
8     def responder_correctamente(self):
9         self.preguntas_correctas += 1
10        self.premio_acumulado = self.calcular_premio(self.preguntas_correctas)
11
12        if self.preguntas_correctas == 20:
13            self.premio_acumulado = self.premio_mayor
14
15    def responder_incorrectamente(self):
16        self.preguntas_correctas = 0
17        self.premio_acumulado = 0
18
19    def calcular_premio(self, num_preguntas):
20        if num_preguntas == 0:
21            return 0
22        elif num_preguntas <= 5:
23            return 100 * (2 ** (num_preguntas - 1))
24        elif num_preguntas <= 10:
25            return 3200 * (2 ** (num_preguntas - 6))
26        elif num_preguntas <= 15:
27            return 102400 * (2 ** (num_preguntas - 11))
28        else:
29            return 3276800 * (2 ** (num_preguntas - 16))
30
31 def index():
32     return rx.vstack(
33         rx.text(f"Preguntas correctas: {State.preguntas_correctas}"),
34         rx.text(f"Premio acumulado: ${State.premio_acumulado}"),
```

```

35         rx.button("Respuesta correcta", on_click=State.responder_correctamente),
36         rx.button("Respuesta incorrecta", on_click=State.responder_incorrectamente),
37     )
38
39     app = rx.App()
40     app.add_page(index)
41     app.compile()
42     import reflex as rx
43
44     class State(rx.State):
45         preguntas_correctas: int = 0
46         premio_acumulado: int = 0
47         premio_mayor: int = 1000000
48
49         def responder_correctamente(self):
50             self.preguntas_correctas += 1
51             self.premio_acumulado = self.calcular_premio(self.preguntas_correctas)
52
53             if self.preguntas_correctas == 20:
54                 self.premio_acumulado = self.premio_mayor
55
56         def responder_incorrectamente(self):
57             self.preguntas_correctas = 0
58             self.premio_acumulado = 0
59
60         def calcular_premio(self, num_preguntas):
61             if num_preguntas == 0:
62                 return 0
63             elif num_preguntas <= 5:
64                 return 100 * (2 ** (num_preguntas - 1))
65             elif num_preguntas <= 10:
66                 return 3200 * (2 ** (num_preguntas - 6))
67             elif num_preguntas <= 15:
68                 return 102400 * (2 ** (num_preguntas - 11))
69             else:
70                 return 3276800 * (2 ** (num_preguntas - 16))
71
72     def index():
73         return rx.vstack(
74             rx.text(f"Preguntas correctas: {State.preguntas_correctas}"),
75             rx.text(f"Premio acumulado: ${State.premio_acumulado}"),
76             rx.button("Respuesta correcta", on_click=State.responder_correctamente),
77             rx.button("Respuesta incorrecta", on_click=State.responder_incorrectamente),
78         )
79
80     app = rx.App()
81     app.add_page(index)
82     app.compile()

```

---



