TAPL: Técnicas Avanzadas de Procesamiento de Lenguaje

Interview Generator

Sistema Inteligente de Evaluación y Preparación de Entrevistas Técnicas

Autores:

- Pablo Chantada Saborido (Portavoz)
  Login: pablo.chantada *Email:* pablo.chantada@udc.es
- Guillermo García Engelmo
  Login: g.garcia2 *Email:* g.garcia2@udc.es
- Héctor Aldao Amoedo
  Login: hector.aldao *Email:* hector.aldao@udc.es

19 de diciembre de 2025

Índice

1. Introducción y Visión General

1.1. Resumen

Interview Generator consiste en un sistema de evaluación inteligente que, mediante el uso de Large Language Models (LLMs) [**?**, **?**, **?**, **?**], examina al usuario siguiendo métricas específicas de procesamiento de lenguaje natural. El objetivo principal es crear un sistema que permita al usuario prepararse eficazmente para entrevistas técnicas, exámenes, o cualquier tipo de evaluación compleja; ofreciendo un entorno de simulación realista y retroalimentación instantánea.

1.2. Implementación del Sistema

Para facilitar el aprendizaje y diferenciarse de un sistema de preguntas y respuestas (QA) básico, *Interview Generator* implementa características avanzadas de evaluación y adaptación:

- **Dificultad Adaptativa**: Al comienzo de la entrevista, el usuario selecciona un nivel de dificultad inicial. A medida que avanza la sesión, el sistema ajusta dinámicamente (Figura **??**) la complejidad de las preguntas de acuerdo con el rendimiento del usuario en tiempo real (promoción ante el éxito y refuerzo ante el fallo).

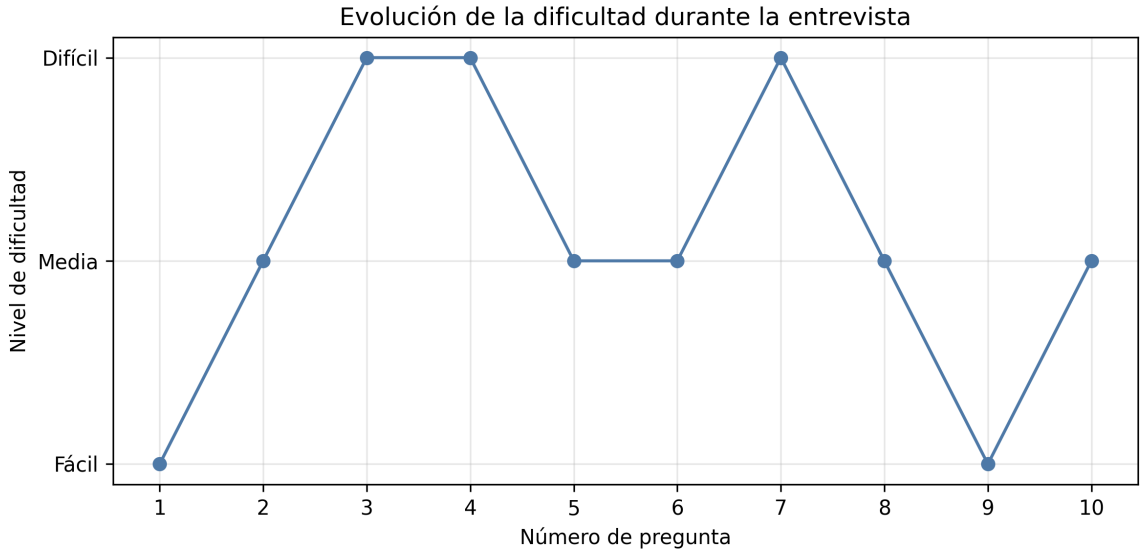

Figura 1: Cambio de la dificultad a lo largo de la entrevista

- **Sistema de Evaluación Híbrido**: El motor de evaluación descompone la respuesta del usuario en cuatro dimensiones (Figura **??**) fundamentales para calcular un puntaje de precisión integral:
  - **Similitud Semántica**: Utiliza modelos de *embeddings* (como `all-mpnet-base-v2` [**?**, **?**]) para validar que el significado global y el contexto de la respuesta

coincidan con la solución esperada, penalizando desviaciones semánticas independientemente de la redacción exacta.

- **Validación Numérica y Simbólica**: Emplea procesamiento simbólico (mediante `SymPy` [**?**]) y expresiones regulares para verificar la exactitud matemática, aplicando tolerancias al redondeo y validando la lógica cuantitativa.
- **Cobertura Conceptual**: Realiza un análisis de densidad terminológica mediante técnicas como *KeyBERT* [**?**] y *spaCy* [**?**] para asegurar la presencia de palabras clave, tecnicismos y conceptos esenciales en la respuesta.
- **Estructura de Razonamiento**: Evalúa la coherencia lógica y la calidad de la exposición a través de la detección de conectores, pasos procedimentales secuenciales e indicadores de formalidad técnica.

La validación numérica supone la mayor parte del rendimiento del sistema. Esto se debe a la dificultad de analizar mediante sistemas puramente sintácticos cuál es una respuesta correcta frente a otra. Por ello, identificamos que si el usuario consigue la respuesta bien, debería por lo menos *aprobar* la evaluación. **??**

- **Módulo de Teoría con RAG (Retrieval-Augmented Generation)**[**?**]: Vinculación del modelo Google Gemini [**?**] con una base de datos bibliográfica. Esto permite generar explicaciones teóricas fundamentadas, donde el usuario puede verificar la fuente original de la información, reduciendo además las alucinaciones del modelo. [1]
- **Interfaz Web Interactiva**: Se proporciona una interfaz web intuitiva y optimizada (Véase **??**) para la gestión de sesiones de entrevista, visualización de métricas en tiempo real y revisión de resultados.

[1] Esta funcionalidad requiere la configuración de la API de GEMINI.

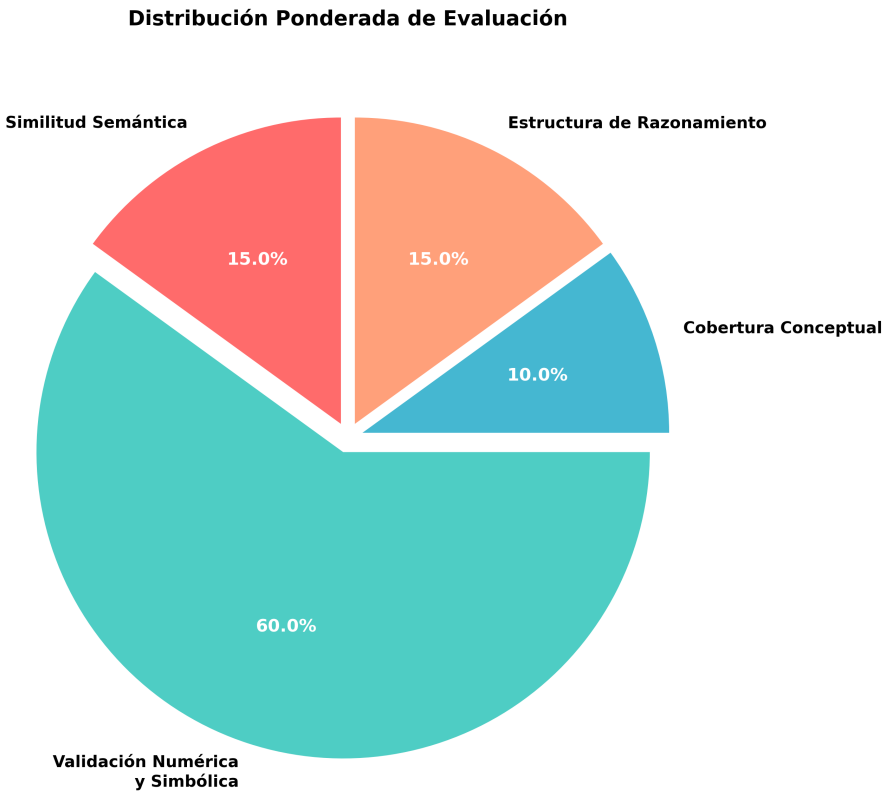

Figura 2: Distribución ponderada de las dimensiones de Evaluación

2. Arquitectura del Sistema

El sistema *Interview Generator* sigue una arquitectura modular (Figura **??**), orquestada mediante el framework **FastAPI** [**?**]. El diseño prioriza la eficiencia y la experiencia de usuario moviendo las operaciones bloqueantes a tareas en segundo plano (*Background Tasks*). A continuación, se detallan los módulos principales, sus responsabilidades y los flujos de datos.

2.1. Diagrama de Componentes

La arquitectura se divide lógicamente en tres capas: *Frontend*, *Backend* y *Manejo de Datos*.

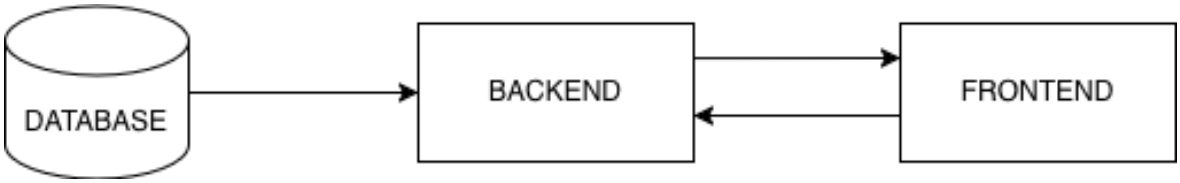

Figura 3: Diagrama de arquitectura del sistema

2.2. Descripción de Módulos

2.2.1. Backend (Controlador Principal)

Implementado en `src/project/app.py`, actúa como el núcleo del sistema, centralizando la lógica de enrutamiento y la orquestación de servicios.

- **Función:** Gestionar el ciclo de vida de la sesión de entrevista (inicio, progreso, finalización), validar las peticiones HTTP y delegar el procesamiento intensivo a hilos secundarios para evitar latencia en la interfaz.
- **Entrada:** Peticiones REST (JSON) provenientes del cliente web.
- **Salida:** Respuestas estructuradas en JSON y renderizado de vistas mediante el motor de plantillas Jinja2 [**?**].
- **Interacción:** Se comunica directamente con **Redis** [**?**] para la gestión de estado y coordina los servicios de RAG y Evaluación.

El sistema Datasets predefinidos (como SQuAD [**?**] o CoachQuant[**?**]), los cuales son pre-procesados y vectorizados en una base de datos ChromaDB [**?**] para optimizar los tiempos de recuperación (véase la Sección **??**).

2.2.2. Motor de Generación (RAG Service)

Ubicado en el paquete `src/project/rag/`, este módulo implementa la lógica de Recuperación Aumentada (RAG) [**?**]. Aunque no es un sistema RAG *clásico* basado en búsqueda semántica, sí comparte la misma filosofía de recuperar contexto desde una base de datos vectorial para guiar al LLM; en este caso, la recuperación se hace mediante muestreo aleatorio, por lo que lo consideramos dentro del mismo abanico de enfoques tipo RAG.

- **Submódulos:**
  - `QuestionGenerator`: Responsable de seleccionar contextos desde la base de datos vectorial y utilizar el LLM para formular preguntas. En esta fase, el sistema utiliza una estrategia de *muestreo aleatorio* sobre los vectores disponibles en lugar de búsqueda semántica, garantizando así la variabilidad y no repetición de los temas en cada entrevista.
  - `GeminiTheoryService`: Módulo especializado que consulta documentos bibliográficos (PDFs) cargados en memoria para generar explicaciones teóricas fundamentadas bajo demanda.
- **Entrada:** Nivel de dificultad objetivo (Fácil/Medio/Difícil) y tipo de dataset.
- **Salida:** Objeto de pregunta normalizado conteniendo el enunciado, la respuesta canónica y metadatos de dificultad.

2.2.3. Motor de Evaluación Híbrido (Evaluator)

El núcleo analítico del sistema, definido en `src/project/metrics/evaluator.py`. A diferencia de los evaluadores tradicionales de NLP (como ROUGE o BLEU), este módulo ejecuta un pipeline secuencial de cuatro etapas para cada respuesta:

- **Entrada:** Respuesta del usuario (texto libre) y Respuesta correcta (referencia).
- **Proceso de Análisis:**
  1. Cálculo de **Similitud Semántica** mediante modelos Transformer (`Sentence-BERT` [**?**]).
  2. **Validación Numérica** exacta y tolerante utilizando cálculo simbólico (`SymPy` [**?**]).
  3. Análisis de **Cobertura Conceptual** extrayendo entidades y palabras clave (`KeyBERT` [**?**], `spaCy` [**?**]).
  4. Evaluación heurística de la **Estructura Lógica** y formalidad.
- **Salida:** Objeto JSON con las puntuaciones parciales y el Score Global (0-1).

Adicionalmente a las métricas cuantitativas, el sistema genera cuatro secciones de retroalimentación accesibles desde el frontend:

- **Comparativa Directa**: Visualización de la respuesta del usuario frente a la solución almacenada.
- **Feedback de IA**: Análisis crítico generado por el LLM que explica las discrepancias semánticas o errores de concepto (ej. explicar por qué una respuesta es incompleta aunque contenga las palabras clave correctas).
- **Solución Paso a Paso**: Generación de una guía detallada de resolución del problema (Chain-of-Thought) [**?**], permitiendo al usuario identificar en qué etapa del razonamiento falló.
- **Fundamentación Teórica**: Explicación académica obtenida mediante el sistema RAG [**?**] sobre la bibliografía cargada, ofreciendo una fuente externa al modelo generativo.

2.2.4. Gestor de Estado y Persistencia (Session Manager)

Se emplea **Redis** [**?**] como almacén de datos en memoria (key-value store) de baja latencia para mantener el contexto de la entrevista.

- **Función:** Almacenar temporalmente el historial de la sesión, incluyendo las preguntas generadas, las respuestas del usuario y las métricas calculadas, permitiendo la recuperación de estado entre peticiones HTTP.
- **Datos:** Serialización JSON de los objetos de sesión identificados por UUID.

2.2.5. Cliente Web (Frontend)

Interfaz ligera construida con HTML, CSS, JavaScript, renderizada desde el servidor mediante **Jinja2** [**?**].

- **Función:** Captura de datos, gestión de la interacción de usuario y visualización de métricas en tiempo real.
- **Comunicación:** Ejecución de llamadas asíncronas a los endpoints del backend para una experiencia de usuario fluida sin recargas de página.

2.3. Flujo de Datos e Interacción

El ciclo de vida (Figura **??**) de una pregunta dentro de una sesión sigue el siguiente esquema secuencial:

1. **Solicitud:** El Usuario requiere una nueva pregunta. El *Backend* invoca al *Motor RAG* solicitando contenido acorde al nivel de dificultad actual registrado en *Redis*.
2. **Procesamiento:** El *QuestionGenerator* recupera el contexto, normaliza el enunciado con el LLM y lo devuelve al frontend.
3. **Respuesta:** El Usuario envía su solución. El *Backend* almacena la respuesta en *Redis* inmediatamente y delega el análisis al *Background Task*.
4. **Evaluación Asíncrona:** El *Evaluator* procesa la respuesta (análisis numérico, semántico y conceptual) en segundo plano, actualizando el registro en *Redis* con las métricas resultantes una vez finalizado el cálculo.
5. **Adaptación:** Basándose en el *Global Score* calculado, el sistema actualiza el estado de la dificultad (promoción/democión) para la siguiente iteración.
6. **Resultados:** Al finalizar el conjunto de preguntas, el sistema consolida los datos de *Redis* y presenta un informe detallado de rendimiento por pregunta.

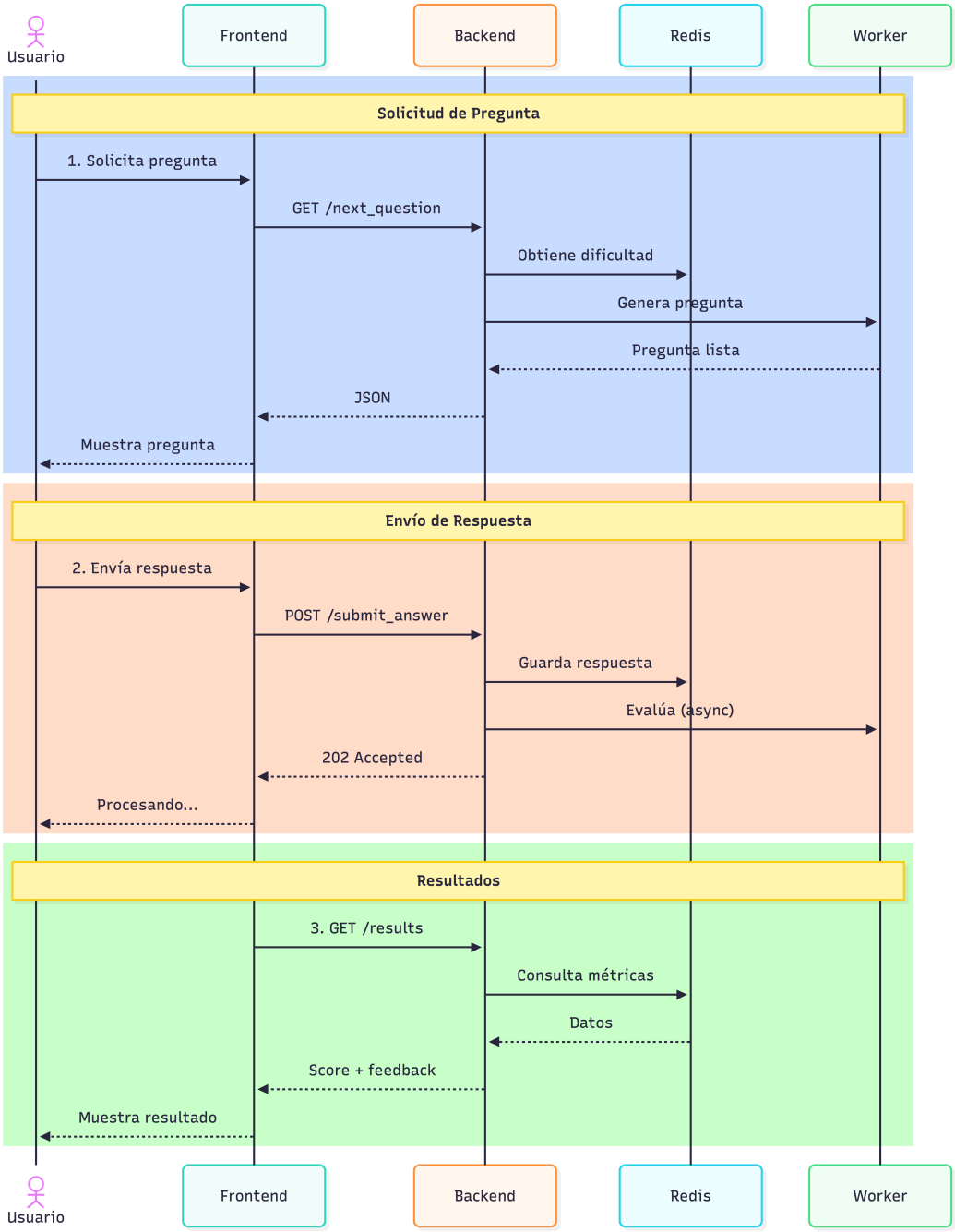

Figura 4: Ciclo de vida de sesión

3. Diario de Trabajo

Esta sección documenta cronológicamente las decisiones técnicas, las pruebas fallidas y las evoluciones del sistema.

3.1. Fase 1: Definición y Estructura

El proyecto inició con la definición del alcanze. Dentro de este bloque encontramos elementos como: selección de temática del proyecto, herramientas iniciales, objetivo final; y otros elementos correspondientes al **Análisis de Proyecto**.

- Se realizó la planificación de módulos. Separación lógica del sistema de backend (véase Sección **??**). Esto nos permitió enfocar el desarrollo desde un principio y evitar fallos por conflictos de módulos.
- **15/10:** Búsqueda de datasets. Inicialmente se consideró crear un dataset propio, pero por limitaciones de tiempo y calidad de datos se pivotó hacia **SQuAD** (Stanford Question Answering Dataset) [**?**] como base. Posteriormente, se implementó un Crawler mediante Scrapy [**?**], con el que se obtuvo el dataset **CoachQuant**[**?**], lo que obligó a refactorizar nuestra carga de datasets para soportar múltiples formatos.

3.2. Fase 2: Selección de Modelos e IA

Durante esta semana se probaron intensivamente diversos LLMs para el motor de generación.

- **Modelos Probados:** DialGPT [**?**] (observamos respuestas incoherentes en contexto técnico), Qwen [**?**] y BERT Multilingual [**?**] (buenos para clasificación, pero poco adecuados para generación), y Llama [**?**] (requería demasiada VRAM local para el tamaño de modelo deseado).
- **Decisión Final:** Se seleccionó **Gemini** [**?**] vía API. Ofrecía la mejor ventana de contexto para el RAG y tiempos de respuesta moderados, vitales para la UX.

3.3. Fase 3: Implementación del RAG y Evaluación

La implementación del RAG [**?**] presentó un reto importante. Si el RAG devolvía erróneamente documentos o el LLM no entendía bien el formato, las preguntas generadas resultaban en texto sin sentido. Para solucionar este problema, se mejoraron los prompts generados, así como la comprobación del contenido que devolvía el RAG.

- **Prueba 1 (Fallida):** Uso de TF-IDF simple. Los resultados carecían de comprensión semántica.
- **Prueba 2 (Exitosa):** Implementación de RAG avanzado con *Context Ranking*. Se añadió un paso intermedio donde el LLM evalúa la relevancia de los fragmentos recuperados antes de formular la respuesta.

- **Evaluación Semántica:** Se descubrió que comparar cadenas de texto exactas fallaba si el usuario usaba sinónimos. Se integró **BERTScore** [**?**] y **Sentence-Transformers** [**?**]. Esto introdujo un problema de latencia (5-10 segundos por respuesta).

3.4. Fase 4: Optimización y Adaptabilidad

Para solucionar la latencia descubierta en la fase anterior y mejorar la experiencia:

1. **Background Tasks:** Se movió el cálculo pesado (`metrics/evaluator.py`) a segundo plano usando `BackgroundTasks` de FastAPI [**?**]. El usuario recibe confirmación inmediata y el frontend consulta el resultado.
2. **Algoritmo Adaptativo:** Se observó que con una dificultad estática las preguntas podían cambiar de dificultad abruptamente. Como solución se programó una lógica de *rachas*: si el usuario acierta 2 veces con score $> 0{,}85$, sube de nivel; si falla con $< 0{,}45$, baja.

3.5. Fase 5: Desarrollo constante y actualidad

Durante todo el desarrollo, a la vez que se profundizaba en el Backend se mejoraba el Frontend. Esto nos permitió una exploración de ideas realmente curiosa. Cuando implementábamos una feature en el backend, al ponerla en el frontend, se nos ocurrían otras mejoras para el sistema.

Un ejemplo de este proceso fue con las pistas. Mientras implementábamos el feedback a la respuesta para que el usuario supiese que se estaba procesando, se nos ocurrió la posibilidad de ayudar al entrevistado mediante pistas para facilitar las preguntas más complejas. Esto además nos llevó a la implementación de la dificultad dinámica, generando así un ciclo de desarrollo realmente vivo y experimental.

Por último, implementamos otros modelos (GROQ [**?**]) gratuitos por problemas de tokens con la API de Gemini, permitiendo al usuario usar otros LLM si lo ve preciso.

Actualmente el sistema implementa todos los objetivos que habíamos planteado. Sin embargo, existen ciertos conceptos o mejoras que dejamos para desarrollar en la Sección **??**.

4. Trabajo Futuro

Si bien el sistema presenta un buen rendimiento, su escalabilidad puede mejorar bastante, llegando a un desarrollo completo y real. [2] A continuación mostramos diferentes apartados que podrían incluirse sobre nuestra base:

[2] Este *Desarrollo completo y real* supone una profundidad que se escapa al enfoque de la asignatura. Las mejoras que presentamos aquí deben tomarse como un objetivo de despliegue real o un trabajo superior al de una práctica.

- **Modelos**: los modelos actuales son todos versiones de prueba o gratuitos, la implementación de modelos más complejos (ChatGPT 4o [**?**], Claude Opus [**?**] u otros) seguramente resulten en un mejor rendimiento del presentado. Además, una posible solución es el uso de modelos específicos para categorías, p. ej.: modelo de biología, modelo de historia, etc. Permitiendo el uso de modelos más pequeños y eficientes, con el coste de desarrollar todos estos.
- **Despliegue no local:** actualmente es necesario descargar el repositorio, dependencias, etc. Una implementación web completa, en la que el usuario únicamente tiene que acceder a la web sin ningún proceso intermedio. Mejoraría considerablemente el alcanze del programa.
- **Datasets personalizados:** la capacidad de que el usuario suba sus propios datasets, o que se generen dado un PDF, URL u otro medio. Expandiría enormemente la profundidad que puede alcanzar la aplicación, permitiéndole ser un experto en cualquier situación que desee el usuario.
- **Niveles de dificultad:** el sistema de dificultad contiene ciertos fallos a la hora de seleccionar la dificultad de las preguntas. Este problema puede ser inherente del dataset, o del sistema en sí. Como solución a esto se plantea un etiquetado manual para las preguntas o separación de los dataset por dificultad.
- **Evaluación:** la evaluación planteada esta principalmente enfocada a los dataset que utilizamos. Esto puede afectar al rendimiento de la evaluación en otros sistemas. Como solución se podría implementar un sistema adaptativo que reconozca la naturaleza del dataset (matemático, literario, etc.) y adapte la distribución de la evaluación de acuerdo con esto.
- **Análisis de resultados:** el formato de las respuestas del usuario puede generar fallos en el análisis final. Por ello, añadir algún tipo de estandarización u metodología mejoraría el rendimiento del análisis.

A. Manual de Instalación y Uso

A.1. Requisitos del Sistema

Para el correcto despliegue de *Interview Generator*, se deben verificar los siguientes prerrequisitos en el entorno de host:

- **Python:** Versión compatible entre 3.10 y 3.14. Se excluye Python 3.15+ debido a restricciones actuales en las dependencias de `torch` y `transformers`.
- **Redis Server** [**?**]: Componente recomendado para la gestión eficiente de sesiones y colas de tareas. El sistema cuenta con un mecanismo de detección automática; en ausencia de un servidor Redis activo, la aplicación operará en modo degradado (memoria volátil) con capacidad limitada a un único proceso worker.
- **Sistema Operativo:** Compatible con Linux, macOS y Windows (se recomienda WSL2 para este último).

A.2. Proceso de Instalación

Para ver el código fuente ver: Source Code

A.2.1. Obtención del Código Fuente

Descargue el repositorio oficial y acceda al directorio del proyecto:

```
git clone https://github.com/pabloChantada/TAPL.git
cd TAPL
```

A.2.2. Configuración del Entorno Virtual y Dependencias

El sistema utiliza el gestor de paquetes estándar de Python (`pip`). Se recomienda el uso de un entorno virtual para aislar las librerías del sistema:

```
# 1. Crear entorno virtual
python -m venv .venv

# 2. Activar el entorno
# En Linux/macOS:
source .venv/bin/activate
# En Windows:
.venv\Scripts\activate

# 3. Instalar las dependencias exactas
pip install -r requirements.txt
```

A.2.3. Configuración de Variables de Entorno

Para que el sistema se conecte a los servicios externos, es necesario crear un archivo `.env` en la raíz del proyecto. Puede basarse en el archivo de ejemplo proporcionado o crear uno nuevo con las siguientes variables críticas:

```
# Proveedor de Inteligencia Artificial
LLM_PROVIDER=GEMINI

# Credenciales de API (Obligatorio para RAG y Evaluaci n)
GEMINI_API_KEY="tu_clave_api_google_aqui"
DEEPSEEK_API_KEY="tu_api_key_de_deepseek_aqui"
GROQ_API_KEY="tu_api_key_de_groq_aqui"

# Referencias a los libros cargados (IDs de archivo o rutas, separados
    por comas)
THEORY_BOOKS="files/id_libro_ejemplo"
```

A.3. Ejecución del Sistema

Para facilitar el despliegue, se incluye un script de arranque (`scripts/run_app.sh`) que configura automáticamente el servidor de aplicaciones *Uvicorn*:

```
chmod +x scripts/run_app.sh
./scripts/run_app.sh
```

El script detectará si Redis está disponible para iniciar múltiples workers (modo producción) o uno solo (modo desarrollo). Una vez iniciado, acceda a la interfaz web en:

`http://localhost:8000`

A.4. Guía de Uso

1. **Inicio de Sesión:** En la página de bienvenida, seleccione el número de preguntas y su dificultad inicial; y después pulse en *Comenzar Entrevista*. Esto inicializará una nueva sesión única y cargará los contextos vectoriales.
2. **Interacción:** Responda a la pregunta planteada. Puede usar la función de *Pista* si necesita ayuda contextual sin revelar la solución.
3. **Evaluación:** Tras enviar su respuesta, el sistema procesará su entrada en segundo plano. Si su desempeño es alto, la siguiente pregunta aumentará de dificultad automáticamente, y viceversa.
4. **Resultados:** Al finalizar las preguntas, será redirigido al panel de métricas donde podrá ver el análisis semántico y matemático detallado así como explicaciones de la IA.

B. Funcionamiento de la Página Web

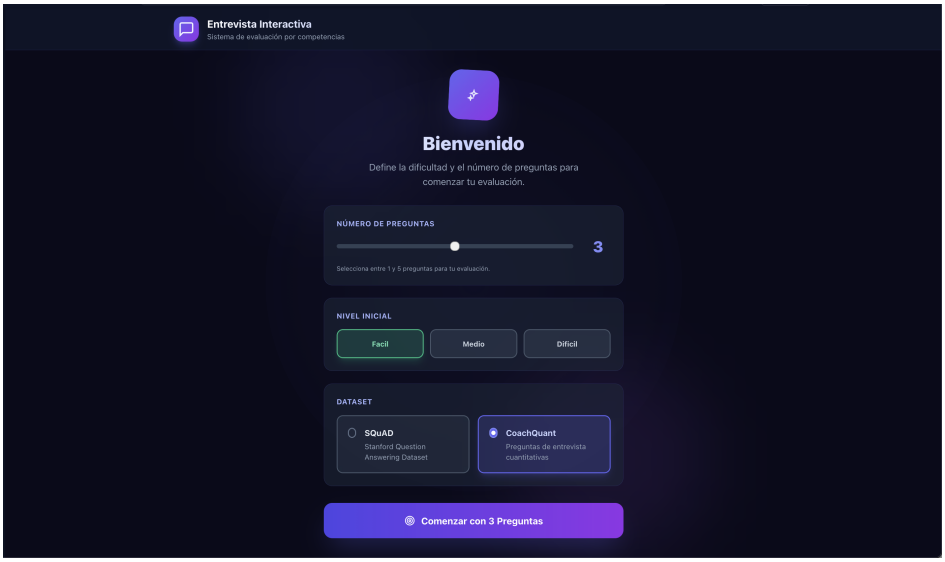

Figura 5: Página Inicial

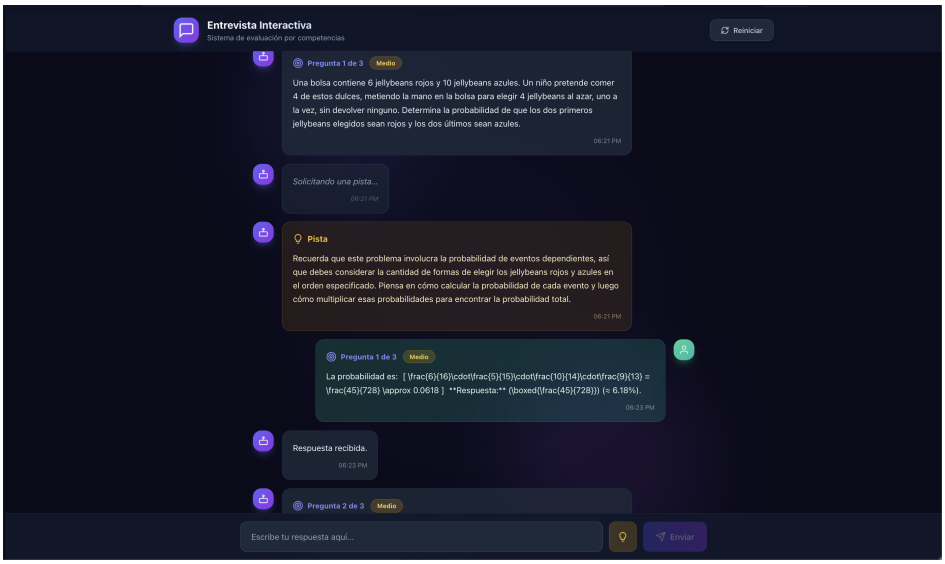

Figura 6: Entrevista en Proceso

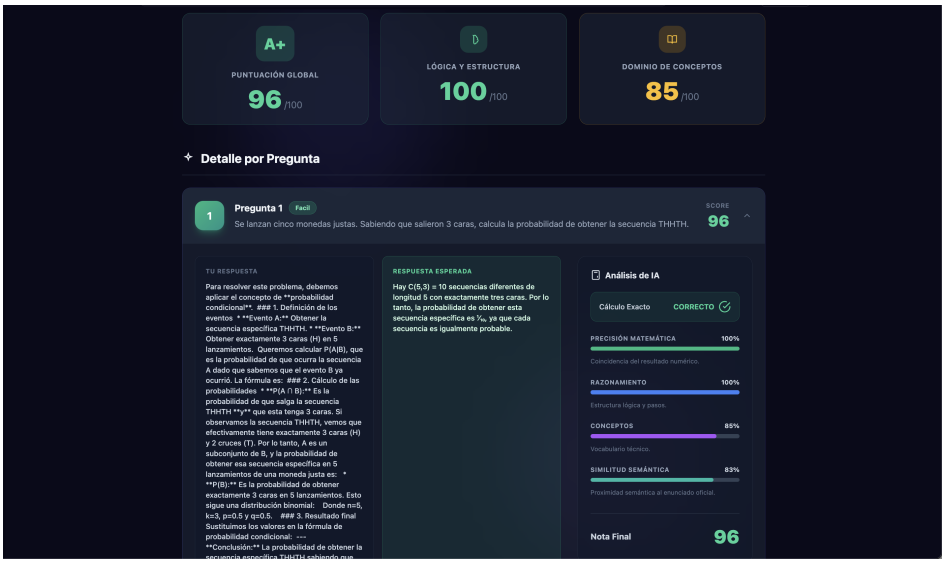

Figura 7: Resultados de Evaluación (Númerico)

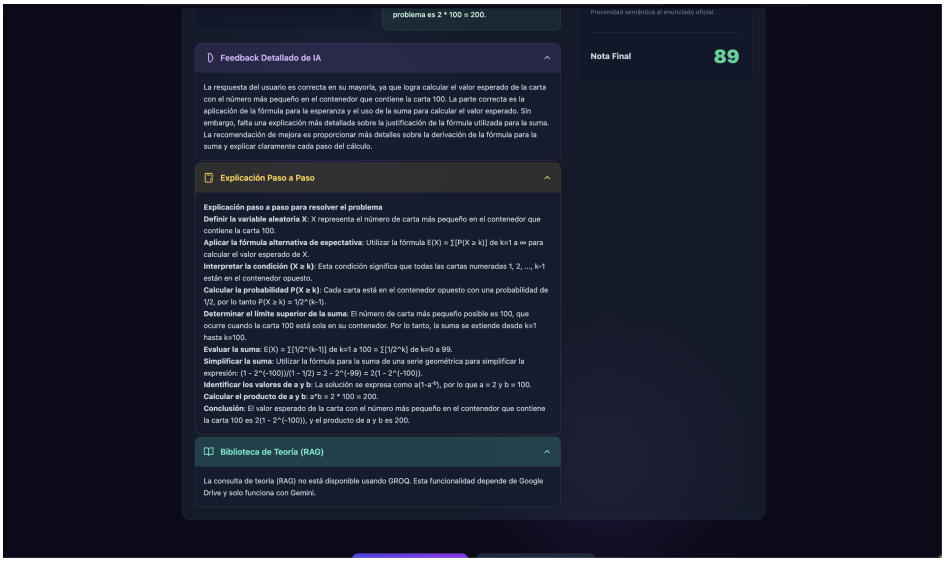

Figura 8: Resultados de Evaluación (Análisis de IA)

C. Herramientas de Terceros

La arquitectura de *Interview Generator* se fundamenta en componentes de software libre. A continuación, se detallan las principales librerías empleadas en el diseño e implementación:

Herramienta	Descripción	Licencia	URL
FastAPI	Framework web moderno y de alto rendimiento para la construcción de APIs con Python. Se utiliza como el núcleo del backend para gestionar rutas y peticiones asíncronas.	MIT	`https://fastapi.tiangolo.com`
LangChain	Framework diseñado para simplificar la creación de aplicaciones que usen modelos de lenguaje. En este proyecto, controla el flujo de Recuperación Aumentada (RAG) y la abstracción de las interacciones con la base de datos vectorial.	MIT	`https://www.langchain.com`
Redis	Almacén de estructura de datos en memoria de código abierto, utilizado como base de datos, caché y broker de mensajes. Su función principal aquí es mantener el estado de las sesiones de usuario y gestionar las colas de tareas en segundo plano.	BSD-3	`https://redis.io`
ChromaDB	Base de datos vectorial nativa para IA, diseñada para facilitar la construcción de aplicaciones con LLMs. Se emplea para almacenar y consultar eficientemente los embeddings generados a partir de los datasets de preguntas.	Apache 2.0	`https://www.trychroma.com`
Sentence Transformers	Framework de Python para embeddings de oraciones, textos e imágenes de última generación. Permite calcular representaciones vectoriales densas para realizar comparaciones de similitud semántica precisas.	Apache 2.0	`https://www.sbert.net`
SymPy	Biblioteca de Python para matemáticas simbólicas. Se utiliza aquí para la validación exacta y tolerante de respuestas numéricas complejas.	BSD	`https://www.sympy.org`
Google GenAI SDK	Kit de desarrollo de software oficial de Google para interactuar con los modelos Gemini. Facilita la generación de contenido, chat y razonamiento multimodal dentro de la aplicación.	Apache 2.0	`https://ai.google.dev`
KeyBERT	Librería minimalista para la extracción de palabras clave utilizando embeddings BERT. Se utiliza para analizar la cobertura conceptual de las respuestas del usuario frente a las respuestas canónicas.	MIT	`https://github.com/MaartenGr/KeyBERT`

spaCy	Biblioteca de software libre para Procesamiento de Lenguaje Natural (NLP) avanzado en Python. Proporciona capacidades de lematización y reconocimiento de entidades nombradas necesarias para el análisis lingüístico del evaluador.	MIT	`https://spacy.io`
Pydantic	Librería de validación de datos y gestión de configuraciones mediante anotaciones de tipo de Python. Garantiza que los datos que fluyen entre el frontend y el backend cumplan con los esquemas definidos.	MIT	`https://docs.pydantic.dev`
Jinja2	Motor de plantillas rápido y expresivo para Python. Se utiliza para renderizar las vistas del frontend con datos dinámicos del backend.	BSD-3	`https://jinja.palletsprojects.com`
Scrapy	Framework de código abierto para web scraping y rastreo web. Se utilizó para construir el crawler que recopiló el dataset CoachQuant.	BSD-3	`https://scrapy.org`

Referencias

[1] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., ... McGrew, B. (2023). Gpt-4 technical report. arXiv preprint arXiv:2303.08774.

[2] Team, G., Anil, R., Borgeaud, S., Alayrac, J. B., Yu, J., Soricut, R., ... Blanco, L. (2023). Gemini: a family of highly capable multimodal models. arXiv preprint arXiv:2312.11805.

[3] Google. Gemini API: Using files and File Search for Retrieval-Augmented Generation. `https://ai.google.dev/gemini-api/docs/file-search`.

[4] Martínez, A. L., Cano, A., Ruiz-Martínez, A. (2025). Generative artificial intelligence-supported pentesting: a comparison between claude opus, gpt-4, and copilot. arXiv preprint arXiv:2501.06963.

[5] Groq Inc. Groq: Language Processing Unit for Fast AI Inference, 2024. `https://groq.com`.

[6] Zhang, Y., Sun, S., Galley, M., Chen, Y.-C., Brockett, C., Gao, X., Gao, J., Liu, J., & Dolan, B. (2020). DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

[7] Bai, J., Bai, S., Li, Y., Dong, L., Zhang, Z., Huang, S., et al. (2023). Qwen Technical Report. arXiv preprint.

[8] Pires, T., Schlinger, E., & Garrette, D. (2019). How Multilingual is Multilingual BERT? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

[9] Meta AI. Llama model family and hardware requirements. `https://llamaimodel.com/requirements/`.

[10] Emmons, S., Jenner, E., Elson, D. K., Saurous, R. A., Rajamanoharan, S., Chen, H., ... Shah, R. (2025). When chain of thought is necessary, language models struggle to evade monitors. arXiv preprint arXiv:2507.05246.

[11] Es, S., James, J., Anke, L. E., Schockaert, S. (2024, March). Ragas: Automated evaluation of retrieval augmented generation. In Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations (pp. 150-158).

[12] Rajpurkar, P., Zhang, J., Lopyrev, K., Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. arXiv preprint arXiv:1606.05250.

[13] CoachQuant. Quant Interview Questions. `https://www.coachquant.com`.

[14] Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., Artzi, Y. (2019). Bertscore: Evaluating text generation with bert. arXiv preprint arXiv:1904.09675.

[15] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., ... Scopatz, A. (2017). SymPy: symbolic computing in Python. PeerJ Computer Science, 3, e103.

[16] Reimers, N., Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084.

[17] Nils Reimers and Iryna Gurevych. Sentence-Transformers: Multilingual Sentence, Paragraph, and Image Embeddings. `https://www.sbert.net`.

[18] Grootendorst, M. (2020, July). KeyBERT: Minimal keyword extraction with BERT.

[19] Honnibal, M. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. (No Title).

[20] Redis Ltd. Redis: The Real-time Data Platform. `https://redis.io`.

[21] Chroma Team. Chroma: The AI-native open-source embedding database. `https://www.trychroma.com`.

[22] Harrison Chase. LangChain: Building applications with LLMs through composability. `https://www.langchain.com`.

[23] Sebastián Ramírez. FastAPI: High performance, easy to learn, fast to code, ready for production. `https://fastapi.tiangolo.com`.

[24] Samuel Colvin. Pydantic: Data validation and settings management using Python type hints. `https://docs.pydantic.dev`.

[25] Armin Ronacher. Jinja2: A modern and designer-friendly templating language for Python. `https://jinja.palletsprojects.com`.

[26] Scrapy Developers. Scrapy: A Fast and Powerful Scraping and Web Crowling Framework. `https://scrapy.org`.