TAPL: Technical Assessment Preparation Lab

Memoria Técnica del Proyecto

Plataforma de Preparación de Entrevistas basada en RAG e IA Generativa

Autores:
Pablo Chantada Saborido
Guillermo García Engelmo

10 de diciembre de 2025

Índice

1. Introducción y Visión General

1.1. Descripción del Proyecto

TAPL (Technical Assessment Preparation Lab) es una plataforma interactiva diseñada para la preparación de entrevistas técnicas y cuantitativas. La aplicación utiliza Inteligencia Artificial Generativa y técnicas de RAG (*Retrieval-Augmented Generation*) para simular sesiones de entrevista, generar preguntas dinámicas, evaluar respuestas mediante métricas de Procesamiento de Lenguaje Natural (NLP) y proporcionar retroalimentación teórica.

El objetivo principal es ofrecer a los usuarios un entorno realista donde puedan practicar preguntas técnicas extraídas de datasets reconocidos (como SQuAD o CoachQuant) y recibir una evaluación objetiva e instantánea.

1.2. Objetivos

- Implementar un sistema de generación dinámica de preguntas técnicas.
- Desarrollar un motor de evaluación automática utilizando métricas como BERTScore, ROUGE y BLEU.
- Integrar capacidades de RAG para proporcionar soporte teórico contextualizado.
- Crear una interfaz web ágil y persistente para la gestión de sesiones.

2. Cronología y Planificación

2.1. Fase 1: Planificación e Inicio (8/10 - 15/10)

El proyecto comenzó con la definición del alcance y la selección de recursos.

- **Selección del tema:** Se decidió enfocar la herramienta en entrevistas técnicas debido a la necesidad de feedback inmediato en este ámbito.
- **Dataset inicial:** Se seleccionó SQuAD como dataset base para probar la infraestructura, aunque posteriormente se evolucionó hacia datasets más específicos como *CoachQuant*.
- **Planificación:** Se estructuró el proyecto separando lógica de negocio (Backend FastAPI) e interfaz de usuario.

[TODO: Explica aquí brevemente por qué elegisteis Python y FastAPI sobre otras opciones (ej. Django o Node.js). ¿Buscabais rapidez de desarrollo o rendimiento asíncrono?]

2.2. Fase 2: Implementación del Pipeline y Modelos (15/10 - 20/10)

Durante esta semana se estableció el núcleo de la IA:

- **Limpieza de datos:** Procesamiento del dataset SQuAD para adaptarlo al formato de preguntas y respuestas.
- **Selección de Modelos:** Se realizaron pruebas con diversos LLMs. La bitácora registra pruebas con DialGPT, Qwen, Bert Multilingual, Llama, ChatGPT y finalmente Gemini.

2.3. Fase 3: Implementación de RAG y Refinamiento (22/10 en adelante)

La característica clave de TAPL es su capacidad de RAG.

- **RAG Básico vs Avanzado:** Se implementó un pipeline inicial con un *retriever* (FAISS/Chroma) y un *reader*. Posteriormente, se mejoró añadiendo ranking de contexto y ajuste de temperatura.
- **Métricas de Evaluación:** Se integraron librerías como `rouge-score`, `bert-score` y `evaluate` para dar una puntuación numérica a las respuestas del usuario.

[TODO: En la bitácora se menciona .ªjuste de temperatura". Explica la decisión: ¿Queríais respuestas más creativas o más deterministas? (Generalmente para entrevistas técnicas se busca baja temperatura).]

3. Arquitectura Técnica

3.1. Stack Tecnológico

El proyecto se basa en un stack moderno de Python (versiones 3.10 - 3.15):

Backend Framework: FastAPI. Elegido por su soporte nativo asíncrono y facilidad para crear APIs RESTful.

Gestión de Dependencias: Poetry. Utilizado para asegurar la reproducibilidad del entorno, frente al clásico `pip freeze`.

Base de Datos / Caché: Redis. Utilizado para la persistencia del estado de la sesión (preguntas actuales, respuestas previas).

LLM Provider: Soporte multi-proveedor (Gemini, DeepSeek, OpenAI), configurado mediante variables de entorno.

3.2. Estructura del Proyecto

El código se organiza siguiendo principios de modularidad en `src/project`:

- `rag/`: Contiene la lógica de generación de preguntas (`question_generator.py`) y conexión con LLMs.
- `metrics/`: Módulos aislados para evaluación (`evaluator.py`, `metrics.py`) y generación de feedback.
- `app.py`: Punto de entrada, inyección de dependencias y definición de endpoints.

3.3. Decisiones de Diseño Clave

3.3.1. Persistencia: Redis vs Memoria

El sistema implementa un patrón de *fallback* interesante en `app.py`. Se intenta conectar a una instancia de Redis definida en las variables de entorno. Si la conexión falla, el sistema conmuta automáticamente a una clase `MockRedis` en memoria.

Justificación Técnica: **[TODO: Aquí explica por qué hiciste esto. Ejemplo: ."Esto permite a otros desarrolladores probar el proyecto localmente sin necesidad de levantar un contenedor Docker de Redis, facilitando el onboarding, mientras que en producción asegura escalabilidad para múltiples workers".]**

3.3.2. Evaluación en Segundo Plano (Background Tasks)

El endpoint `/api/interview/answer` no devuelve la evaluación inmediatamente. En su lugar, utiliza `BackgroundTasks` de FastAPI.

Justificación Técnica: El cálculo de métricas como BERTScore requiere inferencia de modelos pesados (Transformers), lo que puede tardar varios segundos. Bloquear la respuesta HTTP afectaría negativamente la experiencia de usuario (UX). Al moverlo a segundo plano, la interfaz responde rápido y los resultados se actualizan posteriormente.

3.3.3. Estrategia de RAG (Retrieval-Augmented Generation)

Se utiliza RAG no solo para responder preguntas, sino para el módulo de teoría. Los documentos se cargan y se vectorizan para que, cuando un usuario pide ayuda teórica, el sistema recupere el contexto relevante de libros subidos previamente.

[TODO: Explica la decisión de usar Gemini para la parte de RAG. ¿Fue por la ventana de contexto, el costo o la calidad de la API?]

4. Implementación Funcional

4.1. Gestión de Datasets

El sistema soporta la carga dinámica de datasets. Originalmente basado en SQuAD, se extendió para soportar `CoachQuant`, un dataset específico para entrevistas cuantitativas, demostrando la extensibilidad de la arquitectura de `dataset_readers.py`.

4.2. Interfaz de Usuario

Se optó por una solución de *Server-Side Rendering* ligera utilizando **Jinja2** integrado en FastAPI, servido junto con archivos estáticos (CSS/JS). Esto evita la complejidad de mantener un frontend separado (como React) para este prototipo, permitiendo una iteración más rápida.

4.3. Métricas Implementadas

El sistema proporciona una evaluación multidimensional:

- **BLEU:** Para medir la superposición de n-gramas.
- **ROUGE:** Para medir la calidad del resumen/recuperación.
- **BERTScore:** Para medir la similitud semántica, crucial en respuestas técnicas donde el fraseo puede variar pero el significado debe ser exacto.

5. Líneas Futuras y Conclusiones

Basado en el análisis de opciones finales (`final_iter_options.md`), los siguientes pasos para TAPL incluyen:

1. **Historial Persistente:** Migrar de Redis (volátil) a una base de datos SQL (PostgreSQL) para guardar históricos de usuarios a largo plazo.
2. **Agentificación:** Crear agentes especializados por dominio (Matemáticas, Historia, Coding) con contextos propios.
3. **Mejora de UX:** Implementar un selector de número de preguntas y dificultad en el frontend.

TAPL demuestra la viabilidad de utilizar LLMs y RAG para la educación técnica personalizada. La arquitectura híbrida (FastAPI + Redis + LLM API) permite un despliegue flexible, y el sistema de evaluación automática ofrece un valor añadido significativo frente a los simuladores de entrevista tradicionales que carecen de feedback semántico.