Study Case Submission Report

---

1. Title: AI-Powered Job Application Screening Service (Backend)

---

2. Candidate Information

- **Full Name:** Akmal Rafi Diara Putra
- **Email Address:** akmalrafidiara@gmail.com

---

3. Repository Link

- **GitHub repository:** https://github.com/akmalrafidiara/ai-cv-evaluator

---

4. Approach & Design (Main Section)

Tell the story of how you approached this challenge. We want to understand your thinking process, not just the code. Please include:

- **Initial Plan**

  My primary goal was to construct a reliable AI-driven screening service that strictly separates quick API responses from time-consuming AI computation. I broke down the overall challenge into three critical engineering pillars. First, I focused on building a fast, responsive Transport Layer using FastAPI to handle the core I/O. Second, I immediately isolated the lengthy evaluation process by implementing an Asynchronous Processing mechanism with Celery, ensuring my /evaluate endpoint wouldn't block. Third, and most importantly, I prioritized Context-Aware AI (RAG) and Resilience, knowing that stability and contextual accuracy were key to meeting the project's core requirements. I assumed that all file paths would be consistent within the Docker environment and chose Gemini 2.5 Flash for its balance of speed and instruction adherence.

- **System & Database Design**

  The system employs a clean RESTful asynchronous design. The POST /upload endpoint is dedicated solely to file persistence and returning unique document IDs. The critical POST /evaluate endpoint performs minimal validation, creates a job entry in the database with the queued status, and immediately triggers the background Celery task using the job's unique ID. The GET /result/{id} endpoint checks the status and returns the final, structured FinalResultResponse only when the status is completed. For data persistence, I chose PostgreSQL and defined the core schema with the evaluation_jobs table, using a JSON type for the result_data field, setting it as nullable=True to avoid *NotNullViolation* errors during initial job creation.

  I used Celery with Redis serving as both the message broker and the result backend to manage the inherent latency of LLM API calls. This setup guarantees that the /evaluate endpoint meets its speed requirement by delegating the entire multi-stage AI pipeline to the Celery worker pool. To mitigate Docker's common *race condition*—where the application attempts to connect to PostgreSQL before it is fully initialized—I intentionally removed the automatic create_tables() call from the application's startup and now execute database initialization manually via a docker exec command once the database is confirmed to be running.

- **LLM Integration**

  I selected Google Gemini 2.5 Flash as the Large Language Model because I needed high performance in instruction-following and strong JSON fidelity, balanced against low latency for efficient *chaining* and cost-effectiveness. My evaluation logic is executed as a 3-stage LLM Chain: first, CV assessment; second, Project Report assessment; and third, a Final Synthesis step that aggregates the initial two outputs into a concise summary.

  My system relies heavily on Retrieval-Augmented Generation (RAG) to ensure context-specific assessments. I used Gemini Embeddings (models/text-embedding-004) to convert all source documents (Job Description, Rubrics) into vectors, storing them in Qdrant Vector Database. The key strategic decision here was utilizing explicit metadata filtering during the retrieval process. This guarantees the LLM receives only the most pertinent and authoritative text excerpts for each specific question (e.g., only the Project Rubric when scoring the Project Report), significantly boosting contextual accuracy.

- **Prompting Strategy**

  My prompts were engineered to be highly directive and to enforce strict compliance with the required structure. I specifically included instructions mandating the output format. For instance, the **CV Evaluation Prompt** is structured to explicitly include all retrieved context components and demand a specific JSON return format:

  > "Anda adalah evaluator ahli HR. Tugas Anda adalah menilai kecocokan CV kandidat ini. INSTRUKSI UTAMA: Berikan output dalam format JSON yang valid. Jangan ada teks tambahan. JOB DESCRIPTION: {job_description} SCORING RUBRIC: {scoring_rubric} CANDIDATE CV: {cv_content}. FORMAT OUTPUT (JSON): {{"cv_match_rate": <float 0.0-1.0>, "cv_feedback": "<Ringkasan evaluasi, 3-4 kalimat>}}."

- **Resilience & Error Handling**

  Resilience was a core focus of the asynchronous design. I configured Celery with task_max_retries=3 and implemented implicit exponential back-off to handle API failures, timeouts, and connection reset errors. The most challenging issue, JSONDecodeError from the LLM, was solved by implementing robust string cleaning logic

that searches for the { and } delimiters to extract the pure JSON block from surrounding text or markdown. If parsing fails even after cleaning, the task throws an exception, triggering a safe Celery retry.

- **Edge Cases Considered**

  My testing focused on scenarios that would break the pipeline:

  - LLM Output Failure: I tested scenarios where the LLM returned empty or non-JSON strings; the system successfully caught the exception and initiated a Celery retry cycle instead of crashing the job permanently.
  - Document ID Mismatch: I ensured that my /evaluate endpoint validates the existence of the CV and Report IDs in the database, returning a 404 Not Found before attempting to queue a job with bad input data.
  - Database Contention: By moving table creation to a manual docker exec command, I solved the race condition that previously caused application crashes during Docker startup.

---

5. Results & Reflection

- **Outcome**

  The asynchronous architecture performed exceptionally well. The separation of the FastAPI I/O layer from the intensive Celery/Redis worker was seamless, allowing the /evaluate endpoint to consistently return a queued status within milliseconds while successfully managing the long-running LLM calls in the background. The RAG (Retrieval-Augmented Generation) setup using Qdrant's metadata filtering proved effective, ensuring the LLM received highly specific and relevant context (like the exact scoring rubric) for each evaluation stage. Finally, the resilience logic (retry/back-off) successfully stabilized the pipeline against expected external API failures.

  The primary unexpected challenge was the inconsistency of the LLM's JSON output. Although the prompt explicitly demanded pure JSON, the Gemini API sometimes returned the required JSON structure wrapped in Markdown code fences (```json... ```) or, occasionally, returned an empty string upon failure. This immediately caused a fatal JSONDecodeError upon parsing, which required significant debugging and the implementation of a robust string-cleaning mechanism.

- **Evaluation of Results**

  The final evaluation scores and feedback were generally stable and coherent because of two core design decisions:

  Contextual Accuracy via RAG: By filtering the embeddings in Qdrant and injecting only the necessary scoring rubric and job description for each specific prompt, the LLM was forced to ground its assessment on the provided "ground truth" rather than general knowledge.

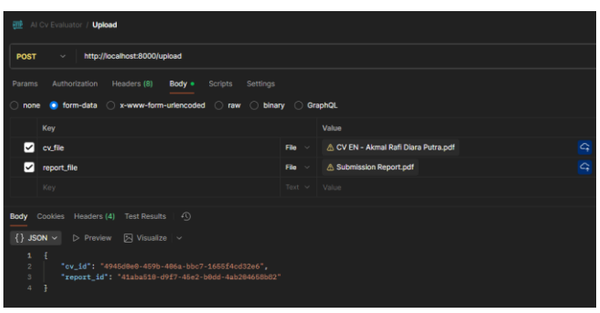
  Controlled Randomness: Setting the LLM's temperature=0.1 significantly reduced the model's creativity and variability, leading to more consistent, deterministic score calculations across multiple runs for the same input data.

- **Future Improvements**

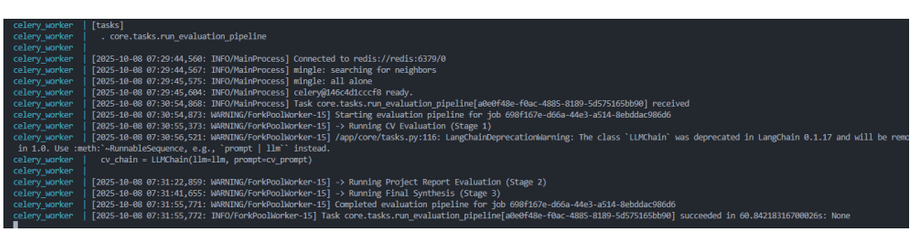
  I would enhance the solution's long-term robustness by implementing two key changes: switching to a Pydantic Output Parser to formally enforce the LLM's JSON schema, thus eliminating brittle manual string cleaning; and integrating an Infrastructure Healthcheck startup script that actively waits for the PostgreSQL service to be ready, fully solving the initial database *race condition*. The primary constraints faced during development were time and LLM API cost/latency, with the time spent debugging the LLM's non-compliant JSON output (an unexpected external API behavior) requiring a necessary diversion of effort from implementing more advanced features.

6. Screenshots of Real Responses

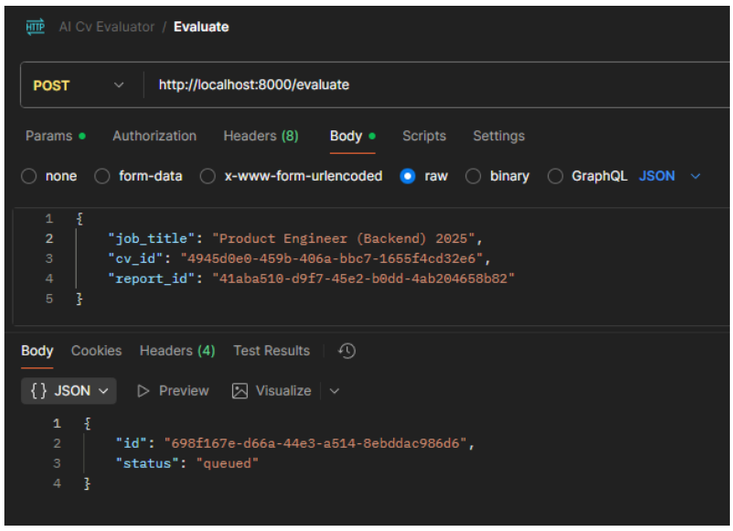
- Upload



Worker

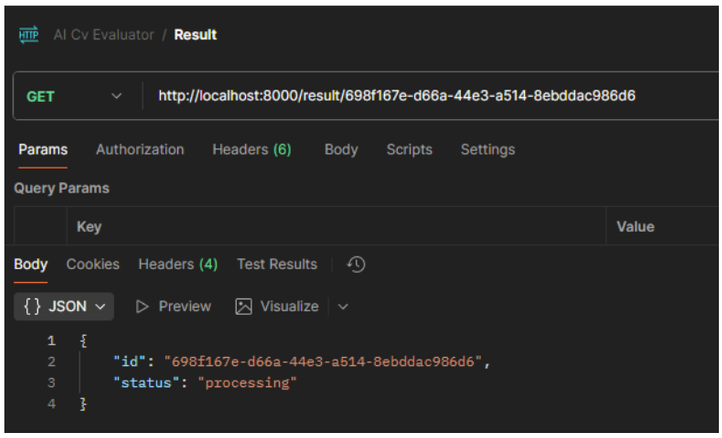


- Evaluate

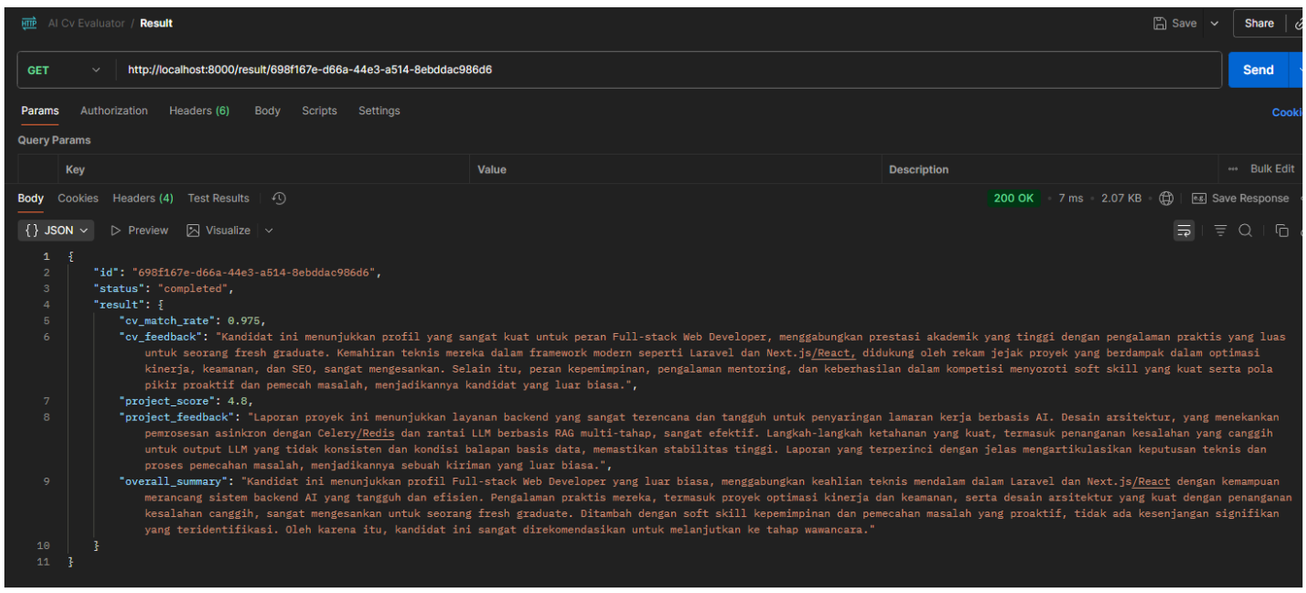


- Result

  Processing



  Completed



7. (Optional) Bonus Work

-