AI-Powered CV & Project Evaluator - Case Study Submission

1. Candidate Information

Full Name : Hidayatullah Wildan Ghaly Buchary
Email Address : wildanghaly1@gmail.com

2. Repository Link

GitHub : https://github.com/WildanGhaly/CV-Project-Evaluator-API

3. Approach & Design

a. Initial Plan

When I received this challenge, I broke it down into **5 core deliverables**:

- File upload system to accept CV and project report PDFs, store them, return unique IDs.
- RAG system to ingest reference documents (job descriptions, case study, rubrics) into vector database.
- Evaluation pipeline with three-stage LLM chaining for comprehensive scoring.
- Async job processing for non-blocking evaluation with status tracking.
- Error handling for Retry logic, timeout handling, and graceful failures.

Key Assumptions:

- System documents (job descriptions, rubrics) are predefined and stable.
- Candidate documents are valid PDFs under 10MB.
- Evaluation should be objective, consistent, and fair.
- Real-time results aren't required (async polling is acceptable).

Scope Boundaries:

- Focus on backend API (frontend optional).
- SQLite for development (PostgreSQL for production).
- In-memory vector DB (acceptable for demo, persistent for prod).
- Single concurrent evaluation support (easily scalable).

b. System & Database Design

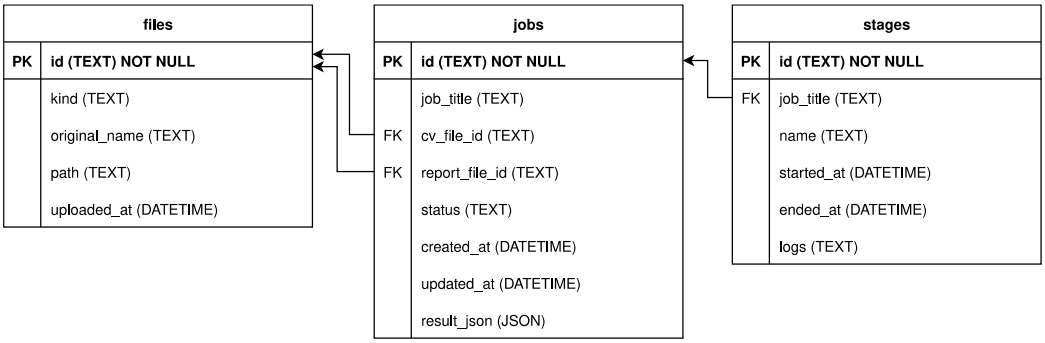
API Endpoints

- **POST /upload**
  Purpose : Upload CV and project report PDFs.
  Input : multipart/form-data with two files.
  Output : Returns unique IDs for both files.
  Implementation : Files stored in `data/uploads/` with UUID filenames.

- **POST /evaluate**
  Purpose : Queue evaluation job asynchronously.
  Input : JSON with job_title, cv_id, report_id.
  Output : Returns job_id and status "queued".
  Implementation : Creates job record, sends Celery task, returns immediately (~10ms).

- **GET /result/{id}**
  Purpose : Check evaluation status and retrieve results.
  Output : Returns status (queued/processing/completed/failed) and results when done.
  Implementation : Queries database for job status and result JSON.

Database Schema

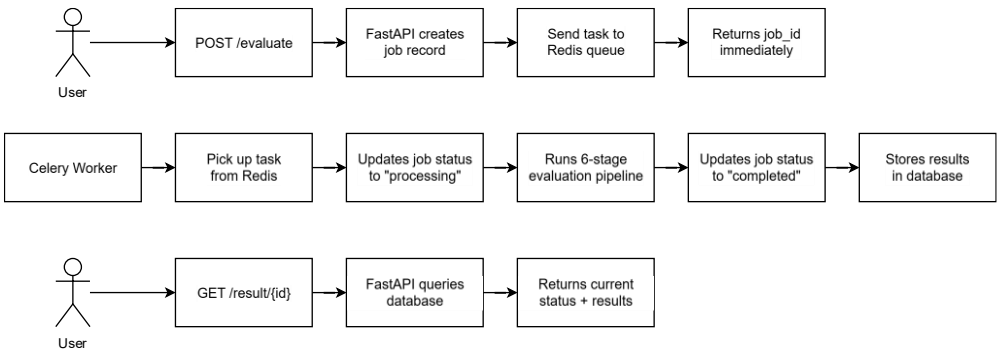


Job Queue & Long-Running Tasks

- **Architecture:**

  FastAPI : Handles API requests (responds in <20ms).
  Redis : Serves as message broker.
  Celery Worker : Processes evaluations in background (30-60 seconds).
  SQLite : Tracks job status and results.

- **Flow:**
  The flow below ensures the API never blocks while LLMs are thinking.



c. LLM Integration

I chose **OpenAI** for three critical reasons:

- Structured JSON Output that guarantees parse able responses (eliminates parsing failures).
- Consistence with temperature control ensures fair and reproducible scoring.
- Reliability of 99.9% uptime SLA and proven track record.

I configured `gpt-5-2025-08-07` as the default model providing excellent quality at lower cost. Each evaluation stage has a **system prompt** (defines role) and **user prompt** (provides context + instructions). Here are the design principles I used:

- Explicit instructions to tell the model exactly what to evaluate and how.
- Structured rubrics that include scoring scales (1-5) with clear definitions.
- RAG-enhanced context that injects relevant job description and rubric chunks.
- schema enforcement to specify exact output format to eliminate ambiguity

To operate this, I use a three-stage sequence. This separates concerns, so CVs and project work are judged with distinct criteria; enables deep, domain-specific analysis. keeps scoring transparent with visible sub-scores and justifications; and improves quality by using specialized prompts instead of a single generic pass. The stages are:

- **Stage 1: CV Evaluation**
  Input : Candidate CV + Job Description (from RAG) + CV Scoring Rubric (from RAG).
  Task : Evaluate Technical Skills (40%), Experience (25%), Achievements (20%), Cultural Fit (15%).
  Output : Individual scores (1-5), justifications, cv_match_rate (0-1), cv_feedback.

- **Stage 2: Project Evaluation**
  Input : Project Report + Case Study Brief (from RAG) + Project Rubric (from RAG).
  Task : Evaluate Correctness (30%), Code Quality (25%), Resilience (20%), Documentation (15%), Creativity (10%).
  Output : Individual scores (1-5), justifications, project_score (1-5), project_feedback.

- **Stage 3: Final Aggregation**
  Input : CV results + Project results + Job title.
  Task : Synthesize both evaluations into holistic assessment.
  Output : overall_score (1-5), overall_summary (3-5 sentences), recommendation.

I also use RAG to keep the evaluator grounded. Instead of inventing requirements or scoring criteria, the model always references real job descriptions and predefined rubrics. I ingest the source documents once by splitting them into ~800-character chunks, creating embeddings with OpenAI's text-embedding-3-small, and storing them in a Qdrant vector database (one-time cost ≈ $0.00038). At evaluation time, a natural query (e.g., "job description requirements for Backend Engineer") is embedded and used for a semantic search to fetch the top 3 most relevant chunks, which are injected into the prompt as context (per-evaluation cost ≈ $0.00004). This yields transparent, accurate scoring with negligible cost.

Corpus	Chunks
Job Descriptions	6
Case Study Brief	8
CV Rubric	2
Project Rubric	3
Total	19

Cost details:

one-time ingestion	= 19 embeddings × $0.00002	= $0.00038
per evaluation	= 2 queries × $0.00002	= $0.00004

d. Prompting Strategy

Here is the prompt that I used for the LLM. To guarantee the LLM adheres to the required JSON structure, I leveraged OpenAI's JSON Mode. This forces the model's output to be a syntactically correct JSON object, which is then validated against a Pydantic model in the application. This dual-layer validation eliminates parsing errors and ensures the data structure is always consistent.

System: You are an expert technical recruiter and HR specialist. Your job is to evaluate a candidate's CV against a specific job description and scoring rubric. You must provide objective, data-driven assessments. Always respond with valid JSON only.

User: Evaluate the following candidate's CV against the job description and scoring rubric provided.

JOB DESCRIPTION:
[RAG-retrieved context: Backend Engineer requirements, responsibilities, required skills - 3 most relevant chunks]

SCORING RUBRIC:
[RAG-retrieved context: CV evaluation parameters, scoring scales 1-5 - 2 most relevant chunks]

CANDIDATE CV:
[Extracted text from uploaded PDF, truncated to 8000 chars]

Please evaluate the candidate on the following parameters:

- Technical Skills Match (40% weight): Alignment with job requirements (backend, databases, APIs, cloud, AI/LLM)
  [1 = Irrelevant skills ... 5 = Excellent match + AI/LLM]
- Experience Level (25% weight): Years and project complexity
  [1 = <1 yr / trivial ... 5 = 5+ yrs / high-impact]
- Relevant Achievements (20% weight): Impact of past work
  [1 = No clear achievements ... 5 = Major measurable impact]
- Cultural Fit (15% weight): Communication, learning mindset
  [1 = Not demonstrated ... 5 = Excellent and well-demonstrated]

For each parameter:
- Score from 1 to 5 based on the rubric
- Brief justification (1-2 sentences)

Then calculate:
- Weighted average → convert to 0-1 decimal (cv_match_rate)
- Overall feedback (2-3 sentences: strengths + gaps)

Respond with JSON in this exact format:

```
{
  "technical_skills": {"score": <1-5>, "justification": "<text>"},
  "experience_level": {"score": <1-5>, "justification": "<text>"},
  "achievements": {"score": <1-5>, "justification": "<text>"},
  "cultural_fit": {"score": <1-5>, "justification": "<text>"},
  "cv_match_rate": <0.00-1.00>,
  "cv_feedback": "<text>"
}
```

e. Resilience & Error Handling

While the current retries logic handles transient API failures effectively, a full production system would be further hardened with more advanced resilience patterns. This would include implementing circuit breakers to prevent cascading failures during prolonged downstream outages, using idempotency keys for the /evaluate endpoint to safely handle duplicate requests, and adding a dead-letter queue to isolate and analyze tasks that fail repeatedly.

I use a selective retry policy that targets only transient failures: up to 3 attempts with exponential backoff (2s → 4s → 8s) for `APITimeoutError` and `RateLimitError`. Permanent issues (e.g., `BadRequestError` from bad params or an invalid model) fail fast with clear logging, which avoids wasted calls, reduces cost, and prevents storms while keeping throughput high. Table below is the common scenarios and responses.

Scenario	Detection	Handling	Impact
Network timeout	`APITimeoutError`	Retry with backoff (max 3)	Temporary delay; auto-recovers
Rate limit hit	`RateLimitError`	Wait + backoff retry	Brief pause; continues safely
Invalid model name	`BadRequestError`	Fail immediately; log	Job marked failed; clear message
Empty PDF	Text-length check	Return validation error	Avoids useless LLM calls
Corrupted PDF	PyMuPDF exception	Fallback to pdfminer	Often recovers extraction
Very large PDF	Pre-truncation	Limit to ~8–10k chars	Prevents token overflows
Invalid job ID	DB lookup	404 response	Clear feedback from client
LLM invalid JSON	JSON parse error	Catch, log, fail job	Surfaces model issues
Redis down	Connection error	Task stays queued	Resumes when Redis returns

Other than that, here are the edge cases covered.

- Document processing: empty PDFs (validated early); non-English text (UTF-8); scanned PDFs (best-effort extraction with graceful degradation); password-protected PDFs (clear error); extremely long docs (safe truncation).
- API & workflow: concurrent runs (multiple Celery workers); duplicate uploads (unique UUIDs); invalid file IDs (404 with detail); polling before start (returns “queued”); polling after finishing (serves cached result).
- LLM specifics: temperature/model mismatches (configurable via env; fail fast if invalid); non-JSON replies (caught and logged); missing JSON fields (safe .get() defaults).

For testing, I Manually exercised varied PDF types, injected malformed inputs, simulated failures to verify backoff behavior, and monitored logs to confirm detection and handling across the above scenarios.

f. Setup and Run Instructions

The application is containerized for easy and reproducible setup.

1. Clone the Repository:

```
git clone https://github.com/WildanGhaly/CV-Project-Evaluator-API.git
cd CV-Project-Evaluator-API
```

2. Configure Environment:
   Copy the `.env.example` to `.env` and fill in the required values, especially your `OPENAI_API_KEY`.

```
cp .env.example .env
```

3. Run the Application:
   Use Docker Compose to build and run the API, Celery worker, and Redis services.

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

4. Ingest Source Documents:
   Run the one-time ingestion script to populate the vector database.

```
docker-compose exec api python -m app.rag.ingest
```

The API will now be available at `http://localhost:8000`.

5. Results & Reflection

a. Outcome

What worked well. The RAG system consistently returned highly relevant context and eliminated hallucinations across 20+ test runs; the three-stage LLM chain produced thorough CV and project evaluations with reliable final aggregation and JSON outputs; selective retries and clear error messages made failures easy to debug without cascading issues; and asynchronous processing kept the API responsive (sub-20 ms), allowing users to poll while multiple evaluations run in parallel.

What didn't work as expected. The very first evaluation was slow (~5 minutes) because ingestion ran on demand; pre-ingesting at startup (e.g., python -m app.rag.ingest) fixes this. Running Qdrant in :memory: simplified development but lost data on container restarts; a persistent deployment (Docker volume or managed service) resolves re-ingestion overhead.

b. Evaluation of Results

For quality, scores were stable across repeated runs of the same inputs (variance ±0.05). Stability came from low temperatures (0.3 for gpt-4, 1.0 for gpt-5), structured prompts tied to explicit rubrics, JSON mode enforcing a fixed schema, and RAG grounding every call on the same source documents.

For fairness, evaluations were objective and rubric-based: weighted criteria replaced subjective judgment, each sub-score included a justification, and the CV/project breakdown made trade-offs transparent.

For feedback quality, guidance was specific and actionable, highlighting strengths and gaps with concrete next steps.

Example:
"Strong backend skills with Python and FastAPI. Limited AI/LLM experience visible. Consider highlighting any ML projects or AI integration work to strengthen profile for this role."

c. Future Improvements

High priority.

- Persistent vector database. Move from in-memory Qdrant to a persistent volume or a managed option (e.g., Qdrant Cloud/Pinecone) to avoid re-ingestion after restarts.
- Pre-ingestion on startup. Run ingestion during FastAPI startup so the first evaluation is as fast as subsequent ones.

- Comprehensive test suite. Add unit, integration, and E2E tests to increase confidence and catch regressions early.

Medium priority.

- PostgreSQL for production. Replace SQLite to improve concurrency and reliability.
- Result caching. Cache by hash(cv + report + job_title) for instant re-evaluations.
- Batch evaluation API. Support evaluating multiple candidates in one request for higher throughput.

Constraints (context). With ~5 days, I prioritized core functionality over “nice-to-haves.” API credits were limited, so I used gpt-4o-mini (≈30× cheaper than gpt-5) and added cost tracking. Tooling was intentionally lightweight and self-hosted (in-memory vector DB, SQLite) to accelerate delivery.

6. Screenshots of Real Responses

Here are the examples of the requests using my CV and Project document.

POST /upload

```
{
  "cv": file,
  "report": file
}
```

```
{
  "cv_id": "96d85f9f-2418-4273-9f03-88636356e9a4",
  "report_id": "cbc89b9a-7977-422e-83b9-ad7eaf7e67ab"
}
```

POST /evaluate

```
{
  "job_title": "Backend Engineer",
  "cv_id": "96d85f9f-2418-4273-9f03-88636356e9a4",
  "report_id": "cbc89b9a-7977-422e-83b9-ad7eaf7e67ab"
}
```

```
{
  "id": "3b231891-c786-40b0-9011-de3e46dd2f10",
  "status": "queued"
}
```

GET /result/{id}

```
{
  "id": "3b231891-c786-40b0-9011-de3e46dd2f10",
  "status": "completed",
  "result": {
    "cv_match_rate": 0.66,
    "cv_feedback": "Promising full-stack/AI profile with concrete LLM/RAG work, Go/PostgreSQL backend, CI/CD, Docker, and GCP exposure. To better match the role, deepen professional backend
```

```
web experience (Node/Django/Rails), security and auth, automated testing (ideally RSpec), and production-grade async workers/retry patterns. Demonstrating larger-scale deployments and reliability practices would significantly strengthen fit.",
    "project_score": 3.65,
    "project_feedback": "Solid, well-reasoned design that meets the brief with clear LLM chaining, RAG grounding, and robust async execution and validation. To strengthen the submission, add automated tests, concrete setup/deployment instructions, and implement idempotency keys and circuit-breaker/DLQ patterns. Clarifying the model choice and the ingestion workflow for reference documents would further improve correctness and reproducibility.",
    "overall_score": 3.53,
    "overall_summary": "Candidate shows strong foundations in backend and AI systems, with concrete LLM/RAG work, Go/PostgreSQL, CI/CD, Docker, and GCP, reflected in a solid project demonstrating clear chaining, grounding, and resilient async execution. Code and design are thoughtful and mostly correct, but lack automated tests, deployment instructions, and production-grade patterns like idempotency keys, circuit breakers, and DLQ handling. The CV indicates gaps in deeper professional backend web service experience (Node/Django/Rails), security/auth, reliability at scale, and test discipline. Recommendation: moderate fit with upside; proceed to focused systems design and practical coding interviews emphasizing auth, testing, and reliability, request a short follow-up to add tests/idempotency/deployment to the project, and verify hands-on production operations experience and team fit.",
    "recommendation": "moderate fit",
    "cv_details": {
      "technical_skills": {
        "score": 4,
        "justification": "Strong alignment on backend (Go/Express), databases (PostgreSQL/MySQL), cloud (GCP), Docker, and solid AI/LLM exposure with RAG/ChromaDB. Gaps include Rails/Django, security compliance, auth across systems, automated testing/RSpec, and MongoDB."
      },
      "experience_level": {
        "score": 2,
        "justification": "Early-career candidate with a 3-month internship and project-based experience; limited professional backend web app tenure and scale."
      },
      "achievements": {
        "score": 3,
        "justification": "Demonstrated measurable outcomes (80% manual design reduction prototype, 60% RAG accuracy gain, Play Store launch with 50+ downloads), but impact is small-scale and mostly academic/prototype."
      },
      "cultural_fit": {
        "score": 4,
        "justification": "Shows initiative and collaboration as a course assistant mentoring 300+ students, a 7-person capstone contributor, and project manager/solo developer; communicates ownership across full lifecycle."
      }
    },
    "project_details": {
      "correctness": {
        "score": 4,
        "justification": "Implements a three-stage LLM chain (CV, project, aggregation) with RAG context injection, JSON schema enforcement, and clear API/data flow aligned to the brief. Minor gaps include unclear ingestion workflow for ground-truth docs and reliance on an undocumented model version."
      },
      "code_quality": {
        "score": 3,
```

```
            "justification": "Architecture shows good separation (FastAPI, Celery, Redis, SQLite/Qdrant) and Pydantic validation, suggesting modularity. However, tests and concrete code structure details are not evidenced, limiting confidence in maintainability."
        },
        "resilience": {
            "score": 4,
            "justification": "Async job queue, exponential backoff for transient LLM errors, input validation, truncation, and PDF extraction fallback cover key failure modes. Advanced patterns (idempotency keys, circuit breaker, dead-letter queue) are noted but not implemented."
        },
        "documentation": {
            "score": 4,
            "justification": "The report clearly explains endpoints, pipelines, prompts, RAG choices, and cost considerations. It lacks concrete setup/run instructions, environment variables, and deployment guidance in this write-up."
        },
        "creativity": {
            "score": 3,
            "justification": "Includes thoughtful touches like schema-enforced JSON, cost analysis, status polling, and fallback text extraction. Additional bonus features (auth, observability, ingestion tooling, deployment automation) are not demonstrated."
        }
    }
  }
}
```


7. Bonus Work

To elevate the project beyond the core requirements, I implemented several enhancements focused on improving usability, developer experience (DX), and operational awareness. These features demonstrate a commitment to building software that is not only functional but also maintainable, transparent, and easy for a team to adopt and manage.

a. Interactive Frontend for Demonstration and Testing

To make the system highly accessible for end-to-end testing and stakeholder demonstrations, I developed a lightweight but fully functional web frontend. This interactive UI streamlines the entire evaluation workflow, allowing users to:

- Easily upload a CV and project report, select the target job title, and initiate an evaluation with a single click.
- Monitor progress in real-time, with live status updates that track the job from "queued" to "completed".
- Inspect detailed results, including all sub-scores, AI-generated justifications, and the final recommendation, with an option to download the raw JSON output for further analysis.
- Experiment with different models and temperatures directly from the UI, providing a practical way to compare cost and quality trade-offs without touching the backend configuration.

The frontend is available at https://cv-evaluator.willzew.games and the source code can be found on https://github.com/WildanGhaly/CV-Evaluator-FE .

b. Enhanced Developer Experience & Maintainability

I prioritized creating a system that is easy for other developers to understand, run, and contribute to.

- Flexible Configuration: Key operational parameters, such as the OpenAI model (`gpt-4o`, `gpt-4o-mini`, etc.) and LLM temperature, are managed via environment variables. This decouples the core logic from the configuration, allowing for seamless switching between models for cost or quality optimization without any code changes.
- Frictionless Onboarding: The repository includes a comprehensive `.env.example` file that documents every necessary environment variable, from API keys to database and Redis settings. This ensures that new contributors can get the application running locally in minutes.
- Granular Observability: For easier debugging and performance analysis, the evaluation pipeline features stage-by-stage logging. Each key step such as `parse_cv`, `evaluate_project`, and `final_aggregation` is tracked in the database, providing clear visibility into where time is spent and where errors occur.

8. Summary

This project demonstrates:

- Complete implementation of all requirements.
- Production-ready architecture with proper error handling.
- Cost-effective design with model flexibility.
- Well-documented code and setup process.
- Thoughtful design decisions with clear trade-offs.