Mini Project – Backend Service with LLM Integration

Candidate: Zulfazriawan

Email: [your email]

Objective:

Build a backend service that can evaluate a candidate’s CV and project report against a given job description using real LLM integration.

Background:

With my background in full-stack development and backend engineering, I approached this challenge as a practical extension of my experience in:

- Designing and implementing APIs with Django and DRF.
- Building modular systems with role-based access.
- Working with Python automation and data handling.
- Integrating external services and handling real-world constraints such as timeouts and retries.

System Design:

- Framework: Django Rest Framework
- Database: SQLite (lightweight for this demo, but easily swappable to PostgreSQL or MySQL)
- Virtual Environment: Managed with `virtualenv` to keep dependencies isolated

Endpoints:

- POST /upload → Upload CV and project report (creates a Job entry).
- POST /evaluate → Trigger evaluation, returns job_id + queued status.
- GET /result/{id} → Retrieve evaluation result (scores + feedback).

Database Schema:

- Job(id, cv_file, report_file, status, result, created_at)

Job Queue Handling:

- Implemented lightweight async using Python threads.
- Jobs move from queued → processing → completed.
- Approach is simple but works for small scale; could scale with Celery + Redis.

LLM Integration:

- Provider: OpenRouter (free tier).
- Model: x-ai/grok-4-fast:free (chosen for JSON reliability and cost).
- Prompt: Designed with strict JSON schema to reduce hallucinations.
- Response: Parsed, validated, and stored as JSON.

Evaluation Pipeline:

1. Upload CV and report (files supported: TXT, PDF, DOCX).
2. Extract text with helper parsers (PyPDF2, python-docx).
3. Build structured prompt with job description + rubric.
4. Send to LLM → get structured evaluation.
5. Validate JSON result with schema rules.
6. Save in DB, fetchable by recruiter through API.

Resilience & Error Handling:

- Implemented retries with exponential backoff for LLM calls.
- Added explicit handling for 429 (Too Many Requests) responses:
- If rate-limited, the system marks the job status as 'rate_limited' instead of 'completed'.
- A structured error JSON is returned: { "error": "Rate limited by provider. Please retry later.", "code": 429 }
- Fallback JSON parsing logic (extract substring if response had extra text).
- Validation layer ensures required fields exist.
- temperature=0.0 used to reduce randomness.

Edge Cases Considered:

- Missing CV or report (optional fields handled).
- Unsupported file type (rejected).
- API timeout or non-JSON response (retry + validation fallback).

Outcome:

- Full pipeline works on real CV and project report.
- Async thread processing prevents request blocking.
- Produced structured results recruiters can use for scoring candidates.

Future Improvements:

- Replace threads with Celery for production-ready job queue.
- Store job description and rubric in a vector DB for flexible RAG.
- Add authentication (JWT) for recruiter/candidate access control.
- Extend system with web dashboard for real-time monitoring.

Reflection:

This project felt like a natural extension of my journey as a backend engineer. In previous work, I’ve often built APIs with Django and DRF, and I’ve dealt with the practical challenges of making them reliable — from handling edge cases in user input to ensuring database queries stay efficient.

Working on this case study gave me the chance to apply those same skills in a new context: integrating an LLM. The process reminded me of past projects where I had to integrate external APIs under tight constraints, making sure errors were caught and responses were structured for downstream use.

One of the biggest lessons reinforced here is that resilience matters more than flashy features. That’s why I focused on retry logic, validation, and clear error handling. These are the same principles I’ve carried through my past work, and I believe they’re what make backend systems dependable.

Overall, this project not only shows I can adapt quickly to new tools like LLMs, but also that I bring with me a consistent approach to designing clean, stable, and extendable backend services.

During testing, I also encountered rate limiting (HTTP 429). The system handled this gracefully by marking the job as 'rate_limited' and returning a structured error JSON, instead of crashing.