

# Case Study Brief

Hello! Thank you for applying with us as a backend developer. This mini project **should be completed within 5 days after you have received this document**. Please spare your time to complete this project with the best results. We are really pleased to answer your questions if there are unclear things.

## Objective

Your mission is to build a backend service that automates the initial screening of a job application. The service will receive a candidate's CV and a project report, evaluate them against a specific job description and a case study brief, and produce a structured, AI-generated evaluation report.

## Core Logic & Data Flow

The system operates with a clear separation of inputs and reference documents:

### Candidate-Provided Inputs (The Data to be Evaluated):

- Candidate CV:** The candidate's resume (PDF).
- Project Report:** The candidate's project report to our take-home case study (PDF)

### System-Internal Documents (The "Ground Truth" for Comparison):

- Job Description:** A document detailing the requirements and responsibilities for the role — You can use the job description you're currently applying. This document will be used as ground truth for **Candidate CV**.
  - To make sure the vector retrieval is accurate enough, you might need to ingest a few job description documents as well.
- Case Study Brief:** This document. Used as ground truth for **Project Report**. (PDF)
- Scoring Rubric:** A predefined set of parameters for evaluating CV and Report, each has it's own documents. (PDF)

We want to see your ability to combine **backend engineering** with **AI workflows** (prompt design, LLM chaining, retrieval, resilience).

## Deliverables

### 1. Backend Service (API endpoints)

Implement a backend service with at least the following RESTful API endpoints:

- `POST /upload`
  - Accepts multipart/form-data containing the Candidate CV and Project Report (PDF).
  - Stores these files, return each with it's own ID for later processing.
- `POST /evaluate`
  - Triggers the asynchronous AI evaluation pipeline. Receives input job title (string), and **both** document ID.
  - Immediately returns a job ID to track the evaluation process.

```
{
  "id": "456",
  "status": "queued"
}
```

- `GET /result/{id}`
  - Retrieves the status and result of an evaluation job. This endpoint should reflect the asynchronous, **multi-stage** nature of the process.
  - Possible responses:
    - While queued or processing

```
{
  "id": "456",
  "status": "queued" | "processing"
}
```

- Once completed

```
{
  "id": "456",
  "status": "completed",
  "result": {
    "cv_match_rate": 0.82,
    "cv_feedback": "Strong in backend and cloud, limited AI integration experience...",
    "project_score": 4.5,
    "project_feedback": "Meets prompt chaining requirements, lacks error handling robustness...",
    "overall_summary": "Good candidate fit, would benefit from deeper RAG knowledge..."
  }
}
```

## 2. Evaluation Pipeline

Design and implement an AI-driven pipeline which will be triggered by **[POST] /evaluate** endpoint. Should consist these key Components:

- **RAG (Context Retrieval)**
  - Ingest all **System-Internal Documents** (Job Description, Case Study Brief, Both Scoring Rubrics) into a vector database.
  - Retrieve relevant sections and inject into prompts (e.g., “for CV scoring” vs “for project scoring”).

- **Prompt Design & LLM Chaining**

The pipeline should consists of

- CV Evaluation
  - Parse the candidate's CV into structured data.
  - Retrieve relevant information from both Job Description and CV Scoring Rubrics.
  - Use an LLM to get these result: `cv_match_rate` & `cv_feedback`
- Project Report Evaluation
  - Parse the candidate's Project Report into structured data.
  - Retrieve relevant information from both Case Study Brief and CV Scoring Rubrics.
  - Use an LLM to get these result: `project_score` & `project_feedback`
- Final Analysis
  - Use a final LLM call to synthesize the outputs from previous steps into a concise `overall_summary`.
- **Long-Running Process Handling**
  - `POST /evaluate` should **not block** until LLM Chaining finishes.
  - Store task, return job ID, allow `GET /result/{id}` to check later periodically.
- **Error Handling & Randomness Control**
  - Simulate any edge cases you can think of and how well your service can handle them.
  - Simulate failures from LLM API (timeouts, rate limit).
  - Implement retries/back-off.
  - Control LLM temperature or add validation layer to keep responses stable.

---

## 3. Standardized Evaluation Parameters

Define at least these scoring parameters:

### CV Evaluation (Match Rate)

- **Technical Skills Match** (backend, databases, APIs, cloud, AI/LLM exposure).
- **Experience Level** (years, project complexity).
- **Relevant Achievements** (impact, scale).
- **Cultural Fit** (communication, learning attitude).

### Project Deliverable Evaluation

- **Correctness** (meets requirements: prompt design, chaining, RAG, handling errors).
- **Code Quality** (clean, modular, testable).
- **Resilience** (handles failures, retries).

- **Documentation** (clear README, explanation of trade-offs).
- **Creativity / Bonus** (optional improvements like authentication, deployment, dashboards).

Each parameter can be scored **1–5**, then aggregated to final score.

## Requirements

- Use any backend framework (Rails, Django, Node.js, etc.).
- Use a proper LLM service (e.g., OpenAI, Gemini, or OpenRouter). There are several free LLM API providers available.
- Use a simple **vector DB** (e.g. ChromaDB, Qdrant, etc) or **RAG-as-a-service** (e.g. Ragie, S3 Vector, etc), any of your own choice.
- Provide README with run instructions + explanation of design choices.
- Provide the documents together with their ingestion scripts in the repository for reproducability purposes.