# Study Case Submission Template

Please use this template to document your solution. Submit it as a **PDF file** along with your project repository.

## 1. AI CV and Project Report Evaluator Backend

Automated screening service using RAG and LLM integration

## 2. Candidate Information


- **Full Name:** Sendi Setiawan
- **Email Address:** sendisetiawan10902@gmail.com


## 3. Repository Link

- https://github.com/sendist/ai-rag-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**
  - How you broke down the requirements.

    The challenge was to build a backend service that evaluates a candidate CV and project report automatically using AI.

    The service should:

    1. Accept uploads of CV and report files.
    2. Evaluate both documents against reference materials (job description, brief, rubrics).
    3. Return structured evaluation results asynchronously.

    Breakdown of Task:

    Step 1: Design core API structure (/upload, /evaluate, /result/:id).

    Step 2: Set up file upload, job queue, and worker process.

    Step 3: Implement Retrieval-Augmented Generation (RAG) for context-based evaluation.

    Step 4: Integrate LLM (Gemini) for scoring and feedback generation.

    Step 5: Add resilience, logging, and documentation (Swagger)

  - Key assumptions or scope boundaries.

    1. Evaluation focuses on textual similarity and prompt-based LLM reasoning.
    2. The system runs locally using free local embeddings and Gemini API for sustainability.
    3. Database persistence is simulated using JSON or in-memory store since full production DB is optional.

- **System & Database Design**

| Component | Technology | Description |
|---|---|---|
| **Backend Framework** | Node.js (TypeScript) | Express API for all endpoints |
| **Queue System** | BullMQ + Redis | Handles asynchronous job processing |
| **Vector Database** | Qdrant | Stores embeddings for retrieval (RAG) |
| **Embedding Model** | Xenova MiniLM-L6-v2 (384 dim) | Local embedding for cost-free vectorization |
| **LLM API** | Gemini 2.5-flash | Generates evaluation output |
| **Storage** | Local uploads/ folder | Stores uploaded PDFs |
| **Documentation** | Swagger UI / Postman | For API testing and demonstration |

  - API endpoints design.

| Endpoint | Method | Description | Example |
|---|---|---|---|
| /upload | POST | Accepts multipart form-data with cv and report | Returns cv_id and report_id |
| /evaluate | POST | Starts asynchronous evaluation | Returns { id, status: "queued" } |
| /result/:id | GET | Retrieves current job status<br>(“queued”, “processing” or completed result | Returns evaluation JSON |
| /docs | GET | Swagger documentation | Interactive UI for testing |

- Job queue / long-running task handling.

  Jobs are enqueued via BullMQ when /evaluate is called. The worker (src/jobs/evaluator.worker.ts) runs asynchronously:

  1. Extracts text from CV & report PDFs.
  2. Retrieves reference from Qdrant using semantic search.
  3. Calls Gemini API with structured prompts.
  4. Updates job state to processing and completed.

- **LLM Integration**
  - Why you chose a specific LLM or provider.
    1. I selected Gemini 2.5-Flash because it offers reliable performance with a free usage tier, fast inference time, and convenient API accessibility for experimentation without billing constraints.
    2. For embeddings, I used Xenova/MiniLM-L6-v2 (384-dim) since it is open-source, runs efficiently on CPU, and provides strong semantic similarity performance despite its small size.

  - Prompt design decisions.

    I designed the prompts to ensure consistent, structured, and automatable responses across different evaluation tasks. Each prompt follows a role-task-output pattern, where the system role defines the AI behavior (recruiter, reviewer, or hiring manager), and the task clearly specifies the expected evaluation or summary. JSON-based outputs were enforced to simplify downstream parsing and scoring.

  - Chaining logic (if any).

    The evaluation pipeline uses a sequential chaining logic where each step feeds its structured output into the next stage. First, the CV evaluation prompt generates cv_match_rate and cv_feedback. Next, the project evaluation prompt produces project_score and project_feedback. Finally, both outputs are merged and passed into the summary generation prompt, which creates an overall assessment summarizing strengths, gaps, and next steps.

  - RAG (retrieval, embeddings, vector DB) strategy.
    1. Each ground-truth PDF (job description, case brief, rubrics) is chunked and embedded.
    2. Embeddings stored in Qdrant under ground_truth_docs collection with payload { text, kind }.
    3. During evaluation, searchRelevant() fetches top k vectors filtered by kind.

- **Prompting Strategy** (examples of your actual prompts)

1. **CV Evaluation**

| |
|---|
| **System:** You are a precise recruiter AI. Score CV vs job requirements using the rubric. Return strict JSON.<br>**User:**<br>CV:{cvText}<br>JOB_DESCRIPTION_SNIPPETS:{jobDesc}<br>RUBRIC_SNIPPETS:{cvRubric}<br>Task: 1) compute cv_match_rate (0..1) and 2) write cv_feedback (3-5 sentences). Return JSON with keys: cv_match_rate, cv_feedback. |

2. **Project Evaluation**

| |
|---|
| **System:** You are a strict code reviewer AI. Evaluate project report vs brief and rubric. Return strict JSON.<br>**User:**<br>REPORT:{reportText}<br>CASE_BRIEF_SNIPPETS:{brief}<br>RUBRIC_SNIPPETS:{reportRubric}<br>Task: 1) score project_score (1..5) and 2) write project_feedback (3-5 sentences). Return JSON keys: project_score, project_feedback. |

3. **Summary Generation**

| |
|---|
| **System:** You are a hiring manager. Summarize strengths, gaps, and next steps.<br>**User:** Make a concise overall summary (3–5 sentences). Data: {JSON.stringify(inputs) [the cv_match_rate, cv_feedback, project_score, and project_feedback data]} |

- **Resilience & Error Handling**
  1. Retry & Backoff: BullMQ worker retries failed jobs automatically.
  2. Safe JSON Parsing: safeJson() ensures malformed LLM outputs don't crash the pipeline.
  3. Fallback LLM Handling: If Gemini fails, the system logs the error and returns a default low-confidence score.
  4. Timeout Protection: Long-running tasks handled asynchronously via Redis queue.
  5. File Validation: Skips missing files during ingestion and logs missing references.

- **Edge Cases Considered**
  - What unusual inputs or scenarios you thought about.
  - How you tested them.

| Case | Handling |
| --- | --- |
| Invalid or empty PDF | Returns error via file validation |
| Missing reference files | Skips file gracefully with log |
| LLM timeout or quota | Falls back to placeholder result |
| Repeated evaluations | Creates new job ID each time |
| Large documents | Text chunking with size 1200 to avoid token overflow |

> ✍ This is your chance to be a storyteller. Imagine you're presenting to a CTO, clarity and reasoning matter more than buzzwords.

---

## 5. Results & Reflection

- **Outcome**
  - What worked well in your implementation?
    1. RAG retrieval and context injection improved evaluation accuracy.
    2. Local embeddings worked offline and avoided API cost limits.
    3. Job queue made the system scalable and asynchronous.
  - What didn't work as expected?
    1. Embeddings with Gemini or OpenAI cannot be done because it is not free.
    2. There is a problem when using pdf-parse library, so I change it to pdfjs-dist.
    3. At first I want to ingest the data from one pdf file, but the separation is not correct, so I need to split it manually and ingest from multiple pdf files.
    4. I cannot use larger embedding model due to hardware limitations.
- **Evaluation of Results**
  - If they were good, explain what made them stable.
    1. Fixed text chunking strategy, each reference document (job description, brief, rubrics) was split into consistent 1200 character chunks. This ensures that the same semantic sections are retrieved during each RAG search, avoiding randomness in which parts of the text are passed to the LLM.
    2. Deterministic prompt templates, which minimize generative variability and keeps the LLM output forma consistent across requests.
- **Future Improvements**
  - What would you do differently with more time?
    1. Build web dashboard to visualize job progress and evaluation results
    2. Secure endpoints with JWT Authentication
    3. Deploy to a cloud service with persistent Redis and Qdrant volumes
    4. Using larger embedding model for better accuracy
    5. Add unit test for evaluation pipeline
  - What constraints (time, tools, API limits) affected your solution?
    1. As a beginner I need time to learning new concepts and implementing the system efficiently.
    2. I was limited to open-source and locally runnable models since paid APIs are not accessible under my current budget.
    3. My main laptop was under repair, so I had to use my sister laptop, which has lower performance. Because of this, I avoided using heavier components like PostgreSQL and instead relied on lightweight storage solutions to keep the system responsive.

---

## 6. Screenshots of Real Responses

- Show **real JSON response** from your API using your own **CV** + **Project Report**.
- Minimum:
  - `/evaluate` → returns job_id + status
  - `/result/:id` → returns final evaluation (scores + feedback)
- Paste screenshots or Postman/terminal logs.

---

## 7. (Optional) Bonus Work

If you added extra features, describe them here

1. API Key Authentication, adds simple security for evaluation endpoints.
2. Swagger Documentation, Interactive docs at /docs for easier testing.
3. Docker Compose is used to deploy the API, Redis, and Qdrant in a portable and reproducible environment.