Study CaseSubmissionTemplate

1. **Title** : Backend Engineer

2. **Candidate Information**

- **Full Name : Albert**
- **Email Address : albert.choe73@gmail.com**

3. **Repository Link**
- **https://github.com/AlbertChoe/ai-cv-evaluator**

4. **Approach & Design (Main Section)**
- **Initial Plan**
  Goal:
  Automate the initial screening of applicants by evaluating a CV and a Case Study Project Report against "ground-truth" docs: Job Description (JD), Case Study Brief, and Scoring Rubric.

  Key ideas I implemented:
  - RAG over curated ground-truth PDFs (JD, Brief, Rubrics) using OpenAI embeddings + Qdrant.
  - Strict-JSON evaluators (CV & Project) with final combined evaluation.
  - Async job orchestration via FastAPI background tasks
  - Clean separation between API, Domain (pipelines), and Infra (PDF, embeddings, Qdrant, LLM client).

- **System & Database Design**

  API Endpoints
  ➢ POST /api/v1/upload → Upload CV and Project Report (PDF). Stores files, returns file_ids.
  ➢ POST /api/v1/evaluate → enqueue evaluation job, return job ID.
  ➢ GET /api/v1/result/{job_id} → return status/results.
  ➢ GET vector-db/health (qydrant healthcheck)

  Database Schema
  Using SQLite with 3 tables:
  - files: {id, type(cv|report), path, name, created_at}
  - jobs: {id, status, job_title, cv_file_id, report_file_id, created_at, updated_at}
  - job_results: {job_id, cv_match_rate, cv_feedback, project_score, project_feedback, overall_summary}

  Job Queue / Long-running Tasks
  - Implemented with asyncio.create_task inside FastAPI.
  - Each evaluation runs in the background:
    1. Update job status to processing.
    2. Run pipeline.
    3. Save results or mark as failed.

  LLM Integration

Choice of provider:

1. OpenAI embeddings (text-embedding-3-small) for vector search.
2. OpenRouter for LLM evaluation (flexible model choice).

Data & Vector Store Design (Qdrant)

Collections (as used in code):

1. COLLECTION_CV → holds JD chunks (doc_type="jd_chunk") filtered by job_key.
2. COLLECTION_PROJECT → holds Case Brief chunks (doc_type="case_brief") and Rubric rows (doc_type="rubric"), filtered by job_key.
3. COLLECTION_CATALOG → job title catalog (LLM-generated): { title, aliases[], tags[], job_key }, used to resolve arbitrary titles to a stable job_key (e.g., product-engineer-backend-v1).

Ingestion Pipeline (Ground-Truth Docs)

Implemented in ingest_all.py:

1. Parse PDFs (JD, Case Brief, Rubrics) using pdfplumber.
2. Chunk text into ~1000 chars with overlap to preserve context.
3. Embed with OpenAI text-embedding-3-small (fast & cost-effective).
4. Upsert to Qdrant with metadata (job_key, doc_type, chunk_index).
5. Standardize Job Title → generate_job_catalog_metadata_from_pdf normalizes {title, aliases, tags, job_key} and saves entries in COLLECTION_CATALOG.

Prompt design decisions:

- Separate prompts for CV evaluation, Project evaluation, and Final summary.

Chaining Logic

1. Parse CV → Retrieve JD chunks → LLM → CV score + feedback.
2. Parse Project Report → Retrieve Case Brief + Rubric → LLM → Project score + feedback.
3. Combine both → LLM → Final summary.

RAG Strategy

- Vector DB: Qdrant (via Docker Compose).
- Collections:
    - job_descriptions → ground truth for CV scoring.
    - case_and_rubrics → ground truth for Project scoring and rubric scoring.
- Ingestion flow:
    1. Parse PDFs (JD, Case Brief, Rubrics) using pdfplumber.
    2. Standardize Job Title → generate_job_catalog_metadata_from_pdf normalizes {title, aliases, tags, job_key} and saves entries in COLLECTION_CATALOG.
    3. Chunk into ~1000 characters with overlap to preserve context.
    4. Embed with OpenAI text-embedding-3-small (fast & cost-effective).
    5. Upsert to Qdrant with metadata (job_key, doc_type, chunk_index).
- Retrieval flow:
    1. If client sends a raw job_title, resolve it via COLLECTION_CATALOG → job_key.

2. For CV: query COLLECTION_CV with filters { job_key, doc_type="jd_chunk" }.
3. For Project: query COLLECTION_PROJECT for { job_key, doc_type in ("case_brief","rubric") }.
4. Neighbor stitching groups adjacent chunk indices to deliver coherent blocks to the LLM (less fragmentation, better answers).

Prompting Strategy

1. Example (CV evaluation):

```
CV_EVAL_PROMPT = """
You are an impartial evaluator assessing how well a
candidate's CV aligns with the provided References.

The References may contain mixed documents (Job
Description, CV Scoring Rubric, Project Scoring Rubric,
Case Brief). For THIS TASK:
- USE ONLY: Job Description (JD) and sections of the
Rubric that clearly pertain to CV evaluation / candidate
skills & experience (CV-related rubric sections).
- IGNORE ANYTHING about Project deliverables, project
scoring, code quality, chaining, RAG, or case-brief
requirements.

Evaluation rules:
- Base every judgment ONLY on the allowed References
above.
- Quote or paraphrase short evidence snippets (max 1–2
lines total) from the References to justify the feedback
- If References are empty or contain no relevant
content, set:
  "cv_match_rate": 0.0
  "cv_feedback": ["No relevant references found to
evaluate this CV."]
- Do NOT infer missing data. Do NOT use prior knowledge.
- Be consistent: high scores require multiple strong,
explicit matches to the allowed References.

Return ONLY strict JSON:
{
  "cv_match_rate": <float between 0 and 1>,
  "cv_feedback": "<2–4 short bullet points summarizing
supported findings>"
}
"""
```

2. Example (Project evaluation):

```
PROJECT_EVAL_PROMPT = """
You are an impartial evaluator assessing a candidate's
Project Report using the provided References.

The References may contain mixed documents (Job
Description, CV Scoring Rubric, Project Scoring Rubric,
Case Brief). For THIS TASK:
- USE ONLY: Case Study Brief and sections of the Rubric
that clearly pertain to Project evaluation /
deliverables (Project-related rubric sections).
- IGNORE ANYTHING about CV match, candidate background,
```

```
or generic hiring criteria unrelated to project
deliverables.

Evaluation rules:
- Base every judgment ONLY on the allowed References
above.
- Quote or paraphrase tiny evidence snippets (max 1-2
lines total) from the allowed References.
- If the allowed References are empty or irrelevant,
set:
  "project_score": 1.0
  "project_feedback": ["No valid case brief or rubric
information found to support evaluation."]
- Do NOT invent criteria. Only evaluate parameters
explicitly mentioned in the allowed References.
- Assign scores only when evidence clearly supports
them.

Return ONLY strict JSON:
{
  "project_score": <float between 1 and 5>,
  "project_feedback": "<2-4 short bullet points
summarizing supported findings>"
}
"""
```

3. Example (Final summary):

```
FINAL_SUMMARY_PROMPT = """
You are producing the final candidate evaluation
summary. Use ONLY the CV evaluation JSON and Project
evaluation JSON supplied in the prompt.

Requirements:
- Write 3-5 full sentences.
- Mention the CV match rate exactly as the decimal you
receive (0-1 scale).
- Mention the project score exactly as the 1-5 score you
receive.
- Cover the candidate's key strengths, salient gaps, and
conclude with a clear recommendation.
- Do not invent numbers or criteria beyond what is
provided.

Return strict JSON:
{
  "overall_summary": "<text>"
}
"""
```

Resilience & Error Handling

- Failures handled:
    1. Empty PDFs → empty text → evaluator returns defaults, job still completes.
    2. LLM JSON parsing errors → re-parse or fallback stubs (safe defaults) to maintain contract.
    3. Retries with exponential backoff for LLM.
- Fallback stubs: If API key missing or invalid → return safe defaults.

    - Stable outputs: low temperature; strict JSON; validation with safe fallbacks.
    - Async background jobs: Keeps API responsive; failures don't block user requests.

  Edge Cases Considered
    - Candidate uploads only CV (no report) → API still works, returns only CV result.
    - Very large PDF → chunking ensures retrieval still works.

5. **Results & Reflection**

   Outcome:
   - End-to-end flow works: upload → evaluate → get results.
   - Ground truth ingestion into Qdrant is tested and functional.
   - Evaluation pipeline generates structured scores and feedback.

   What worked well
   - FastAPI + SQLite = quick development
   - Qdrant via Docker Compose was lightweight and easy to run.
   - Hybrid OpenAI/OpenRouter approach allowed stable embeddings and flexible evaluation.
   - The modular project structure kept everything tidy and extensible.

   What didn't work as expected
   - Rubric table parsing (PDF) required careful header detection to serialize rows cleanly.
   - Occasional LLM JSON drift still happens

   Evaluation of Results
   - Retrieval pulls the right context (tested with JD & rubric).
   - The system accurately resolves the job title to its corresponding job_key using semantic search in the job catalog, ensuring evaluations always reference the correct ground-truth documents.
   - Scores are consistent across runs with low temperature.
   - Summaries are coherent and aligned with rubric criteria.

   Future Improvements
   - Deterministic chunk IDs (hash of normalized text) to prevent duplicate upserts.
   - Add authentication (API keys) for multi-user use.
   - Use Postgres instead of SQLite for scale.
   - Add tests for ingestion & retrieval quality.

6. Screenshots of Real Responses
   - POST /api/evaluate response showing

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8008/evaluate' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "job_title": "Backend Engineer",
  "cv_id": "file_d9d9b9d57266474980182d37b4510833",
  "report_id": "file_da90c702ad6745b380e20459b88aaea6"
}'
```

Request URL

```
http://127.0.0.1:8008/evaluate
```

Server response

Code	Details
200	Response body

```
{
  "id": "job_239a39c623964a72b160b7bf94851e54",
  "status": "queued",
  "result": null,
  "error": null
}
```

Download

Response headers

```
content-length: 90
content-type: application/json
date: Wed,08 Oct 2025 09:49:31 GMT
server: uvicorn
```

- GET /api/result/{job_id}

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8008/result/job_239a39c623964a72b160b7bf94851e54' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8008/result/job_239a39c623964a72b160b7bf94851e54
```

Server response

Code	Details
200	Response body

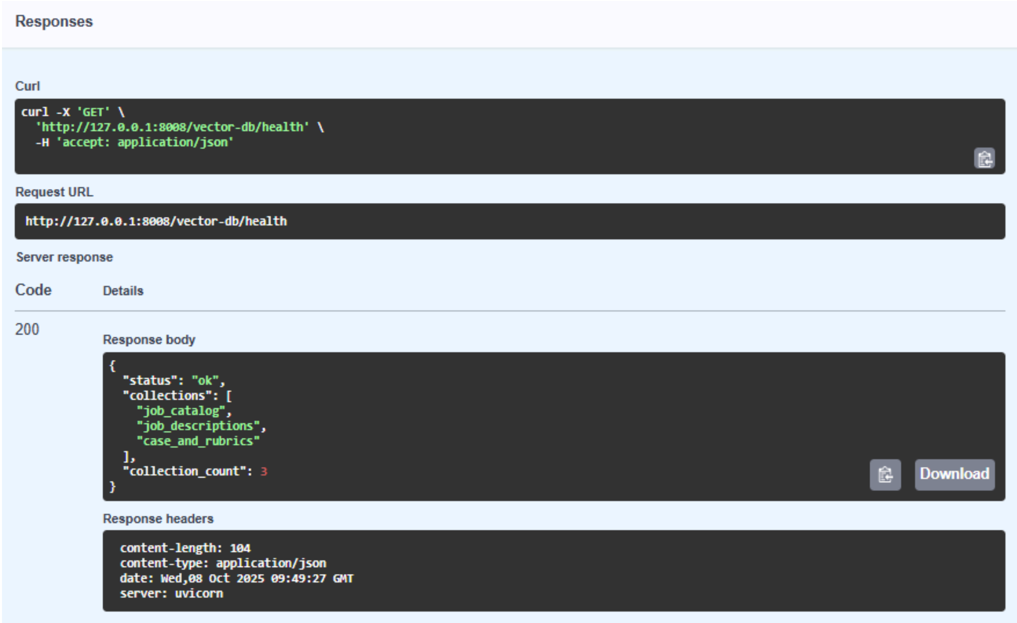
```
{
  "id": "job_239a39c623964a72b160b7bf94851e54",
  "status": "completed",
  "result": {
    "cv_match_rate": 0.6,
    "cv_feedback": "['Experience with backend technologies is evident, as the candidate has worked on backend features and API
s.', 'The candidate has utilized cloud technologies like AWS and has experience with database management, aligning with job requi
rements.', 'Skills in programming languages such as Python and JavaScript match the server-side languages mentioned in the JD.',
\"The candidate's teamwork experience in developing applications aligns with the collaborative nature of the role.\"]",
    "project_score": 4.5,
    "project_feedback": "['Meets prompt chaining requirements, lacks error handling robustness.', 'Demonstrates a solid understan
ding of API design and database schema.', 'Includes thoughtful resilience strategies for API failures.', 'Documentation is clear
and provides adequate setup instructions.']",
    "overall_summary": "The candidate has a CV match rate of 0.6, indicating a moderate alignment with the job requirements. Thei
r project score is 4.5, reflecting strong capabilities in API design and thoughtful resilience strategies. Key strengths include
experience with backend technologies, cloud services, and relevant programming languages, while a notable gap is the lack of robu
st error handling in their project. Overall, I recommend considering this candidate for the role, as they demonstrate solid techn
ical skills and a collaborative mindset."
  },
  "error": null
}
```

Download

Response headers

```
content-length: 1479
content-type: application/json
date: Wed,08 Oct 2025 09:49:55 GMT
server: uvicorn
```

- GET /api/vector-db/health



7. Bonus Work Implemented / Planned
    - Implemented a semantic job catalog that maps different title variations (e.g., “Backend Developer”, “Backend Engineer”) to a unified job_key with shared aliases and tags. This ensures consistent retrieval and evaluation even when users input alternate job titles.
    - Added context stitching around retrieved chunks to provide the LLM with richer, more complete information, improving evaluation accuracy.
    - Implemented automatic retry logic with exponential backoff to handle rate limits and transient API errors more gracefully.