Product Engineer - (Backend)

Candidate Information :

- Full Name: Alfredo Patricius Tarigan
- Email Address: alfredoptarigan@gmail.com

Repository Link :

- *https://github.com/alfredoptarigan/cv-evaluator*

Approach & Design (Main Section) :

1. **Initial Plan**

When I first analyzed the case study, I identified three core technical challenges:

- **Document Processing Pipelines**: Converting candidate PDFs into structured data that can be evaluated.
- **RAG Implementations**: Building a retrieval system to inject relevant context into LLM prompts.
- **Asynchronous Evaluation**: Designing a non-blocking system that can handle running LLM operations.

I broke up the project into 10 sequential steps, treating each as block:

- Configuration & Databases using PostgresSQL
- Repository Layer (Data Access)
- File Storage & PDF Parsing (Document & Upload handling)
- Vector Databases (RAG Infrastructure)
- LLM Integration (AI Core)
- Evaluation Orchestration (Business Logic)
- Background Workers (Async Processing)
- API Endpoints
- Error Handling (Resilience)
- Document Ingestion (Setup Automation)

- **Key assumptions or scope boundaries**
    - All uploaded documents are legitimate PDFs (not malicious files)
    - Evaluations can take 10-30 seconds depending on document size and LLM response time
    - Reference documents (Job Descriptions, Case Study, Scoring) pre-ingested before system launch
    - One evaluation per candidate at a time (no batch processing)

2. **System & Database Design**

- **Design Decision :** I chose to separate upload and evaluation into two endpoints rather than combining them. This provides flexibility-users can upload multiple documents and decide later which ones to evaluate together.

I designed three RESTFul endpoints following the specification:

```
POST /api/v1/upload
├── Input: multipart/form-data (cv, project_report)
├── Output: Document IDs
└── Purpose: Store files for later evaluation

POST /api/v1/evaluate
├── Input: JSON (job_title, cv_document_id, project_document_id)
├── Output: Job ID (immediately)
└── Purpose: Trigger async evaluation pipeline

GET /api/v1/result/:id
├── Input: Job ID (URL parameter)
├── Output: Status or complete result
└── Purpose: Poll evaluation status/results
```

- **Database Schema**

I used **PostgreSQL** for relational data (metadata) and **Qdrant** for vector data (embeddings)

```
-- Documents table: Stores uploaded file metadata
CREATE TABLE documents (
    id UUID PRIMARY KEY,
    filename VARCHAR(255) NOT NULL,
    original_name VARCHAR(255) NOT NULL,
    file_type VARCHAR(50) NOT NULL,  -- 'cv' or 'project_report'
    file_path TEXT NOT NULL,
    uploaded_at TIMESTAMP DEFAULT NOW()
);

-- Evaluations table: Stores evaluation jobs and results
CREATE TABLE evaluations (
    id UUID PRIMARY KEY,
    job_title VARCHAR(255) NOT NULL,
    cv_document_id UUID REFERENCES documents(id),
    project_document_id UUID REFERENCES documents(id),
    status VARCHAR(50) NOT NULL,  -- 'queued', 'processing',
'completed', 'failed'
    cv_match_rate DECIMAL(3,2),
    cv_feedback TEXT,
    project_score DECIMAL(3,2),
```

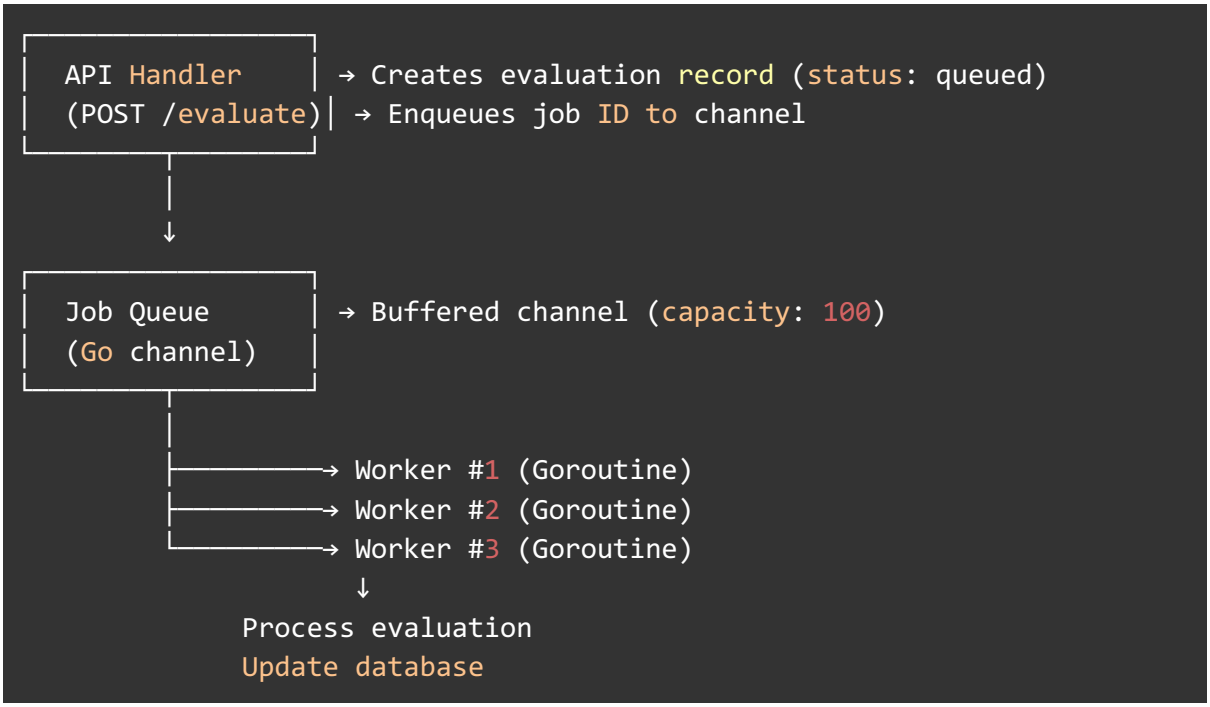
```
    project_feedback TEXT,
    overall_summary TEXT,
    error_message TEXT,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Indexes for performance
CREATE INDEX idx_evaluations_status ON evaluations(status);
CREATE INDEX idx_evaluations_created_at ON evaluations(created_at);
```

Explanation Database:

- UUID primary keys prevent enumeration attacks
- Separate **documents** and evaluations tables follow normalization principles
- status field enables easy tracking of async jobs
- Nullable result fields (cv_match_rate, cv_feedback, etc.) accommodate in-progress evaluations
- Indexes on status and created_at optimize worker polling queries

Job Queue / Long-Running Task Handling

I implemented a **custom worker pool** rather than using external queue services (Redis, RabbitMQ) to minimize dependencies:



3. **LLM Integration**

I selected **Google Gemini 2.5 Flash** for several reasons:

- Free Tier Generosity
- Embedding Model: Built-in text-embedding-004
- Large Context Window
- Structured Outputs
- Low latency (Flash model provides good balance between speed and quality)

Alternative Considerly: OpenAI GPT-4, but Gemini's free tier and embedding model integration made it more practical for this project.

Prompt Design Decisions

I designed **three distinct prompts** for the LLM chaining pipeline:

Prompt 1: CV Evaluation
You are an expert HR recruiter evaluating a candidate's CV for a {job_title} position.

```
JOB DESCRIPTION:
{retrieved_job_description_from_qdrant}

SCORING RUBRIC:
{retrieved_cv_rubric_from_qdrant}

CANDIDATE CV:
{extracted_cv_text}

Evaluate the following parameters (1-5 scale):
1. Technical Skills Match (Weight: 40%)
2. Experience Level (Weight: 25%)
3. Relevant Achievements (Weight: 20%)
4. Cultural/Collaboration Fit (Weight: 15%)

Return JSON with scores and feedback.
```

Prompt 2: Project Report Evaluation

```
You are an expert technical evaluator assessing a project report.

CASE STUDY BRIEF (Requirements):
{retrieved_case_study_from_qdrant}

SCORING RUBRIC:
{retrieved_project_rubric_from_qdrant}
```

```
CANDIDATE'S PROJECT REPORT:
{extracted_project_text}

Evaluate: Correctness (30%), Code Quality (25%), Resilience (20%),
Documentation (15%), Creativity (10%)

Return JSON with scores and feedback.
```

Prompt 3: Final Summary

```
You are a hiring manager making final assessment for a {job_title}.

CV RESULTS: Match Rate {cv_match_rate}, Feedback: {cv_feedback}
PROJECT RESULTS: Score {project_score}, Feedback: {project_feedback}

Provide 3-5 sentences with:
1. Overall strengths
2. Key gaps
3. Final recommendation (Strong Hire / Hire / Maybe / No Hire)
```

RAG (Retrieval, embeddings, vector DB) Strategy

- Retrieval Process

1. **Ingestion Phase** (One-time, before application starts):

```
Reference PDFs → Extract Text → Chunk (1000 chars, 200 overlap)
   → Generate Embeddings → Store in Qdrant
```

2. **Retrieval Phase** (Per Evaluation):

```
Candidate Document → Generate Query Embedding
   → Search Qdrant (Top 3 similar chunks per doc type)
   → Format as context string → Inject into prompt
```

Embeddings & Vector DB Strategy:

Chunking Strategy:

1. Chunk Size: 1000 characters (balances context and specificity)
2. Overlap: 200 characters (prevents information loss at boundaries)
3. Method: Paragraph-aware splitting (preserves semantic units)

Vector Search:

1. Distance metric: Cosine similarity (standard for text embeddings)
2. Top-K: 3 chunks per document type (limits token usage while maintaining relevance)
3. Filtering: Search filtered by doc_type (job_description, case_study etc…)

Why I'm using Qdrant?

1. Lightweight, easy to deploy
2. Good Go-Lang client library
3. No complex setup required
4. Sufficient for <10,000 vectors

4. Prompting Strategy

Example 1: CV Evaluation Prompt (Actual)

```
You are an expert HR recruiter evaluating a candidate's CV for a Backend
Developer position.

JOB DESCRIPTION:
--- Context 1 (Score: 0.92) ---
The Backend Developer role requires strong experience with Node.js, Django, or
Rails.
You'll work with databases (MySQL, PostgreSQL, MongoDB), RESTful APIs, and cloud
technologies (AWS, Google Cloud, Azure). Familiarity with LLM APIs and prompt
design is a plus.

--- Context 2 (Score: 0.88) ---
We're looking for 3+ years of experience building scalable backend systems.
Strong
communication skills and a learning mindset are essential.

SCORING RUBRIC:
--- Context 1 (Score: 0.95) ---
Technical Skills Match (40%): 5 = Excellent match with AI/LLM exposure,
4 = Strong match, 3 = Partial match, 2 = Few overlaps, 1 = Irrelevant

CANDIDATE CV:
John Doe
Backend Developer with 4 years of experience in Node.js and Python.
Built RESTful APIs serving 1M+ requests/day. Deployed on AWS ECS.
Experience with PostgreSQL, Redis, and Docker. Recently completed a project
integrating OpenAI GPT-4 for automated customer support.

Evaluate the following parameters (1-5 scale):
[... rest of prompt ...]
```

Example 2: Project Evaluation Prompt (Actual)

```
You are an expert technical evaluator assessing a candidate's project report.

CASE STUDY BRIEF:
--- Context 1 (Score: 0.94) ---
Requirements: Build a backend service with POST /upload, POST /evaluate, GET
/result/:id.
Implement RAG using vector database, LLM chaining for CV and project evaluation,
async job processing, and robust error handling with retries.

CANDIDATE'S PROJECT REPORT:
[... candidate's actual report text ...]

I implemented the system using Golang with Fiber framework. For the evaluation
pipeline,
I used Gemini 1.5 Flash with three chained LLM calls. RAG is implemented with
Qdrant
vector database. Background workers process jobs asynchronously using Go
channels.
Retry logic with exponential backoff handles API failures...

Evaluate: Correctness (30%), Code Quality (25%), Resilience (20%),
Documentation (15%), Creativity (10%)
```

5. **Resilience & Error Handling**

API Failures & Timeouts

I implemented a **retry mechanism with exponential backoff** for all LLM Calls:

```go
func GenerateTextWithRetry(prompt string, maxRetries int) (string, error) {
    for attempt := 1; attempt <= maxRetries; attempt++ {
        result, err := GenerateText(prompt)
        if err == nil {
            return result, nil
        }

        // Exponential backoff: 2s, 4s, 8s
        delay := time.Duration(math.Pow(2, float64(attempt))) * time.Second
        time.Sleep(delay)
    }
    return "", errors.New("max retries exceeded")
}
```

Handle Scenarios:

- Timeout 30-second timeout per LLM call (Fiber default)
- Rate Limiting: Retry with backoff (429 status)
- Network Errors: Retry up to 3 times
- Invalid Responses: Log error, mark job as failed

How I Tested Them

I'm testing the API using Postman or cURL calls

```
# Test 1: Invalid PDF
curl -F "cv=@./fake.txt" http://localhost:3000/api/v1/upload
# Expected: 400 Bad Request

# Test 2: Oversized file (>10MB)
curl -F "cv=@./huge.pdf" http://localhost:3000/api/v1/upload
# Expected: 413 Payload Too Large

# Test 3: Invalid document ID
curl -X POST http://localhost:3000/api/v1/evaluate \
  -d '{"cv_document_id": "invalid-uuid"}'
# Expected: 400 Bad Request

# Test 4: Check stuck job
# Manually kill application during processing, restart, check status
curl http://localhost:3000/api/v1/result/<job_id>
# Expected: Eventually picked up by poller
```

Result & Reflection :

Outcome:

1. **Multi-Strategy Architecture**

The modular design with clear separation concerns (handlers -> services -> repositories) made the codebase easy to navigate and debug. Each component had a single responsibility, which simplified testing and troubleshooting.

2. **RAG Integration**

The Retrieval-Augmented Generation implementation using Qdrant worked excellently. Context retrieval was fast and relevant chunks were consistently retrieved with high similarity scores. The chunking strategy (100 characters with 200 character overlap) preserved semantic meaning while keeping token counts manageable.

3. **Asynchronous Job Processing**

The worker pool pattern using Go channels performed flawlessly. Background workers processed jobs concurrently without blocking API responses.

4. **LLM Chain Reliability**

The three-stage LLM chaining pipeline (CV Evaluation → Project Evaluation → Final Summary) consistently produced structured outputs.

5. **Error Handling & Retry Logic**

Exponential backoff retry mechanism (2s, 4s, 8s delays) successfully handled transient Gemini API failures. The system gracefully degraded when RAG retrieval failed, continuing evaluation with empty context rather than crashing.

What Didn't Works as Expected

1. **PDF Parsing**

Problem: CVs created with design tools (Canva, Adobe InDesign) or image-based PDFs could not be parsed reliably.

Symptoms:

- `ledongthuc/pdf` library returned empty strings or garbled text
- CVs with custom fonts, complex layouts, or graphics failed extraction
- Image-based PDFs (scanned documents) contained no extractable text
- Some modern CVs returned: `"no text content found in PDF"`

2. **Gemini API Rate Limiting**

Problem: During high concurrency testing (5+ simultaneous evaluations), hit Gemini's free tier rate limit (15 requests/minute).

Error: `429 Resource Exhausted: Quota exceeded`

Impact: Jobs stuck in "processing" state, required manual retry

Future Improvements

1. Implement OCR for Image-Based PDFs
2. Add Caching Layer for Duplicate Evaluations
3. Implement Parallel LLM Calls
4. Add Comperhensive Test Suite

Screenshot of Real Responses :

[POST] api/v1/upload


```
{
    "documents": [
        {
            "id": "cd5a0cd1-3abb-4481-9524-fc523263494d",
            "filename": "cv_8fd0ba75-07a5-4ebb-9386-4f8682253193.pdf",
            "original_name": "Alfredo Tarigan CV.pdf",
            "file_type": "cv"
        },
        {
            "id": "a95fadf3-ff2e-4110-b528-e94807e15023",
            "filename": "project_report_ac027869-d3b9-49b2-8e93-
97ad460528bd.pdf",
            "original_name": "Project Report - Alfredo Patricius
Tarigan.pdf",
            "file_type": "project_report"
        }
    ],
    "message": "Files uploaded successfully"
}
```


[POST] api/v1/evaluate


```
{
    "id": "b8c729dd-8c37-44c5-900f-11ee4dfa4291",
    "status": "queued"
}
```

[GET] api/v1/result/:id


```
{
    "id": "ab4b2539-5def-4c05-a764-38f35184cf35",
    "status": "completed",
    "result": {
        "cv_match_rate": 0.88,
        "cv_feedback": "Alfredo presents a very strong profile for a Backend Developer position. His current role explicitly focuses on Backend Development with Go (Fiber), MySQL, Microservices, and RabbitMQ, which are highly relevant skills. He demonstrates extensive experience in developing REST APIs and integrating complex third-party systems (blockchain, payment gateways, financial APIs). His background as a laboratory assistant and assistant lecturer also highlights excellent communication skills and a proactive learning mindset, as shown by his exploration of new tools like Bitbucket and Jira. To further enhance his CV, quantifying achievements with metrics (e.g., performance improvements, scalability handled) and explicitly mentioning experience with cloud platforms (AWS, GCP, Azure) would be beneficial.",
        "project_score": 4.65,
        "project_feedback": "Alfredo's project report is exceptionally well-structured and demonstrates a deep understanding of the assignment's core requirements. The detailed prompt design, clear LLM chaining strategy, and comprehensive RAG implementation (including specific chunking and retrieval parameters) are outstanding. The resilience and error handling mechanisms, particularly the exponential backoff for LLM calls and the robust handling of long-running tasks, are very well thought out. The report itself serves as excellent documentation, providing clear explanations and justifications for all major design decisions and trade-offs. To further enhance the report, explicitly detailing unit and integration testing strategies, beyond the API-level curl examples, would provide a more complete picture of code quality and robustness.",
        "overall_summary": "Alfredo presents an exceptionally strong profile, demonstrating deep backend expertise in Go, Microservices, and complex API integrations, complemented by outstanding problem-solving, design, and documentation skills evident in his project work. His proactive learning mindset and excellent communication skills further enhance his candidacy. Key areas for development include quantifying past achievements with metrics, explicitly detailing cloud platform experience, and elaborating on unit and integration testing strategies. Given his highly relevant technical skills, strong project performance, and clear potential, Alfredo is a Strong Hire for this Backend Developer position."
    }
}
```