Backend HR Screening Service - Project Report

Candidate: Mike Chen **Project Date:** October 2024 **Duration:** 5 days

1. Project Introduction

This project implements a backend service for screening job applications using AI. The system evaluates candidate CVs and project reports against job requirements using LLM integration.

2. Implementation Overview

2.1 Technology Stack Used

- **Backend Framework:** Flask (Python)
- **Database:** SQLite3
- **AI Service:** Google Gemini API
- **Vector Database:** ChromaDB (basic implementation)
- **Background Processing:** Celery with Redis
- **File Handling:** PyPDF2 for PDF processing

2.2 Project Structure

```
hr-service/
├── main.py              # Flask application and API endpoints
├── database.py          # Database setup and initialization
├── models.py            # Database models and operations
├── llm.py              # LLM integration functions
├── rag.py              # Vector database operations
├── celery_app.py       # Celery configuration
├── tasks.py            # Background task definitions
├── workers.py          # Worker functions
├── uploads/            # File storage directory
├── docs/               # Documentation
├── requirements.txt    # Python dependencies
└── README.md          # Project documentation
```

3. API Implementation

3.1 Upload Endpoint

Route: `POST /upload`

Functionality: - Accepts CV and report files via multipart form data - Saves files with secure names and timestamps - Creates database records for uploaded files - Processes files in background for text extraction

Basic Implementation:

```
@app.route('/upload', methods=['POST'])
def upload_documents():
    try:
        cv_file = request.files['cv']
        report_file = request.files['report']

        # Save CV file
        cv_name = secure_filename(cv_file.filename)
        cv_path = os.path.join(UPLOAD_DIR,
        f"cv_{datetime.now().strftime('%Y%m%d%H%M%S')}_{cv_name}")
        cv_file.save(cv_path)
        cv_id = Document.create('cv', cv_name, cv_path)

        # Save Report file
        report_name = secure_filename(report_file.filename)
        report_path = os.path.join(UPLOAD_DIR,
        f"report_{datetime.now().strftime('%Y%m%d%H%M%S')}_{report_name}")
        report_file.save(report_path)
        report_id = Document.create('report', report_name,
        report_path)

        return jsonify({'cv_id': cv_id, 'report_id': report_id}), 201
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

3.2 Evaluate Endpoint

Route: `POST /evaluate`

Functionality: - Accepts job title, CV ID, and report ID - Creates evaluation job in database - Starts background processing - Returns job ID for tracking

3.3 Result Endpoint

Route: `GET /result/{id}`

Functionality: - Returns job status and results - Handles different states: queued, processing, completed, failed

4. Database Design

4.1 Tables Created

Users Table:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Documents Table:

```
CREATE TABLE documents (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    doc_type TEXT NOT NULL,
    filename TEXT NOT NULL,
    path TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Jobs Table:

```
CREATE TABLE jobs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    job_title TEXT NOT NULL,
    cv_id INTEGER,
    report_id INTEGER,
    status TEXT NOT NULL DEFAULT 'queued',
    result_json TEXT,
    error_message TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (cv_id) REFERENCES documents(id),
    FOREIGN KEY (report_id) REFERENCES documents(id)
);
```

5. LLM Integration

5.1 Basic LLM Wrapper

```python
import google.generativeai as genai

class LLMWrapper:
    def __init__(self):
        genai.configure(api_key=os.getenv('GEMINI_API_KEY'))
        self.model = genai.GenerativeModel('gemini-pro')

    def evaluate_text(self, prompt):
        try:
            response = self.model.generate_content(prompt)
            return response.text
        except Exception as e:
            return None

    def available(self):
        try:
            # Simple test to check if API is available
            self.model.generate_content("test")
            return True
        except:
            return False
```

5.2 Basic Prompt Engineering

CV Evaluation Prompt:

```
Evaluate this CV for the position: {job_title}

CV Content: {cv_text}

Provide:
1. Match rate (0-1 scale)
2. Brief feedback about strengths and areas for improvement
```

Project Evaluation Prompt:

```
Evaluate this project report based on technical implementation.

Project Content: {project_text}

Provide:
1. Score (1-5 scale)
2. Feedback about code quality and technical approach
```

6. RAG Implementation (Basic)

6.1 Vector Database Setup

```
import chromadb
from chromadb.utils import embedding_functions

# Initialize ChromaDB
RAG_DIR = os.path.join(os.path.dirname(__file__), 'uploads', 'chroma')
_client = chromadb.PersistentClient(path=RAG_DIR)
_embedder = embedding_functions.DefaultEmbeddingFunction()
_collection = _client.get_or_create_collection(name="system_docs")

def ingest_text(doc_id, text, metadata=None):
    if text:
        _collection.add(documents=[text], metadatas=[metadata or {}],
        ids=[doc_id])

def query_text(query, n_results=3):
    if query:
        results = _collection.query(query_texts=[query],
        n_results=n_results)
        return results['documents'][0] if results['documents'] else []
    return []
```

6.2 Document Processing

```
def process_document(doc_id, file_path, doc_type):
    try:
        # Extract text from PDF
        text = extract_pdf_text(file_path)

        # Store text sidecar file
        sidecar_path = f"{file_path}.txt"
        with open(sidecar_path, 'w', encoding='utf-8') as f:
            f.write(text)

        # Ingest into vector database
        ingest_text(f"doc:{doc_id}", text, {'doc_type': doc_type})

    except Exception as e:
        print(f"Error processing document {doc_id}: {e}")
```

7. Background Processing

7.1 Celery Configuration

```python
from celery import Celery

def make_celery():
    broker = os.getenv("REDIS_URL", "redis://localhost:6379/0")
    backend = os.getenv("REDIS_BACKEND", "redis://localhost:6379/1")

    celery = Celery("hr_service", broker=broker, backend=backend)
    celery.conf.update(
        task_serializer="json",
        result_serializer="json",
        accept_content=["json"],
        timezone="UTC",
        enable_utc=True,
    )
    return celery

celery = make_celery()
```

7.2 Background Tasks

```python
@celery.task(name="process.upload")
def process_uploaded_file_task(doc_id, path, doc_type):
    process_uploaded_file(doc_id, path, doc_type)

@celery.task(name="evaluate.job")
def run_job_task(job_id):
    run_job(job_id)
```

8. Error Handling

8.1 Basic Error Handling

```python
@app.errorhandler(404)
def not_found(error):
    return jsonify({'error': 'Endpoint not found'}), 404

@app.errorhandler(500)
def internal_error(error):
    return jsonify({'error': 'Internal server error'}), 500
```

8.2 Try-Catch in API Functions

```python
@app.route('/upload', methods=['POST'])
def upload_documents():
    try:
        # Main logic here
        pass
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

8.3 Fallback Mechanism

```python
def evaluate_with_llm(text, job_title):
    if llm.available():
        return llm.evaluate_text(text, job_title)
    else:
        # Fallback to basic evaluation
        return basic_evaluation(text, job_title)
```

9. Testing Results

9.1 Manual Testing

Upload Test:

```bash
curl -X POST http://localhost:5000/upload \
  -F "cv=@test_cv.pdf" \
  -F "report=@test_report.pdf"
```

Evaluate Test:

```bash
curl -X POST http://localhost:5000/evaluate \
  -H "Content-Type: application/json" \
  -d '{"job_title":"Backend Engineer","cv_id":1,"report_id":2}'
```

Result Test:

```bash
curl http://localhost:5000/result/1
```

9.2 Response Examples

Upload Response:

```json
{
  "cv_id": 1,
  "report_id": 2
}
```

Evaluate Response:

```
{
  "id": 1,
  "status": "queued"
}
```

Result Response:

```
{
  "id": 1,
  "status": "completed",
  "result": {
    "cv_match_rate": 0.75,
    "cv_feedback": "Good technical skills, needs more experience",
    "project_score": 4.0,
    "project_feedback": "Well implemented basic features",
    "overall_summary": "Candidate shows potential for growth"
  }
}
```

10. Challenges Faced

10.1 Technical Challenges

1. **LLM API Integration:** Initial difficulties with API key setup and rate limiting
2. **PDF Text Extraction:** Some PDFs had formatting issues affecting text extraction
3. **Vector Database:** Learning curve for ChromaDB implementation
4. **Background Processing:** Celery configuration and Redis setup

10.2 Solutions Implemented

1. **Error Handling:** Added comprehensive try-catch blocks and fallback mechanisms
2. **File Processing:** Multiple text extraction methods with fallbacks
3. **Simplified RAG:** Basic implementation with room for improvement
4. **Local Testing:** Implemented fallback to local threading when Celery unavailable

11. Current Limitations

11.1 Technical Limitations

1. **Basic Prompt Engineering:** Prompts could be more sophisticated
2. **Limited Error Scenarios:** Haven’t tested all edge cases
3. **Simple RAG Implementation:** Basic vector search without advanced techniques
4. **Limited Testing:** No automated tests implemented

11.2 Functional Limitations

1. **Document Formats:** Only supports PDF and text files
2. **Single Language:** No multi-language support
3. **Basic Scoring:** Scoring algorithm could be more sophisticated
4. **No Authentication:** API endpoints are not secured

12. Future Improvements

12.1 Technical Enhancements

1. **Advanced Prompt Engineering:** Implement more sophisticated prompt chains
2. **Better RAG System:** Improve context retrieval and relevance
3. **Comprehensive Testing:** Add unit tests and integration tests
4. **Monitoring:** Add logging and monitoring capabilities

12.2 Feature Enhancements

1. **Multi-format Support:** Support for DOCX, PPTX, and other formats
2. **Authentication:** Add API key or JWT authentication
3. **Dashboard:** Web interface for monitoring and management
4. **Analytics:** Advanced reporting and analytics features

13. Conclusion

This project successfully implements a basic HR screening service with the following features:

✅ **File Upload System:** Handles CV and report uploads ✅ **Basic LLM Integration:** Uses Gemini API for evaluation ✅ **Vector Database:** Simple RAG implementation with ChromaDB ✅ **Background Processing:** Celery for async task processing ✅ **API Endpoints:** RESTful API for upload, evaluate, and results ✅ **Error Handling:** Basic error handling with fallback mechanisms

The system provides a solid foundation for an AI-powered HR screening tool with room for significant improvements and enhancements. The implementation demonstrates understanding of backend development, AI integration, and modern web application architecture.