HR Screening Backend Service - Project Implementation

Candidate: Sarah Wilson **Project Duration:** 5 days **Date:** October 2024

1. Project Overview

This project implements a backend service for automated CV screening that evaluates candidate applications against job requirements and case study briefings. The system uses AI integration with RAG (Retrieval-Augmented Generation) and comprehensive error handling.

2. System Design

2.1 Architecture

The application uses Flask framework with the following components:

```
Flask Application (main.py)
├── Database Layer (SQLite3)
├── File Storage System
├── Celery Workers (Background Processing)
├── ChromaDB (Vector Storage)
└── Gemini API (AI Evaluation)
```

2.2 Database Structure

Created three main tables:

Users Table: - id (PRIMARY KEY) - name (TEXT) - email (TEXT, UNIQUE) - created_at (TIMESTAMP)

Documents Table: - id (PRIMARY KEY) - doc_type (TEXT) - ‘cv’ or ‘report’ - filename (TEXT) - path (TEXT) - created_at (TIMESTAMP)

Jobs Table: - id (PRIMARY KEY) - job_title (TEXT) - cv_id (FOREIGN KEY) - report_id (FOREIGN KEY) - status (TEXT) - ‘queued’, ‘processing’, ‘completed’, ‘failed’ - result_json (TEXT) - error_message (TEXT) - created_at (TIMESTAMP)

3. API Implementation

3.1 Upload Endpoint (`POST /upload`)

```python
@app.route('/upload', methods=['POST'])
def upload_documents():
    try:
        if 'cv' not in request.files or 'report' not in request.files:
            return jsonify({'error': 'Form harus berisi file cv dan
        report'}), 400

        cv_file = request.files['cv']
        report_file = request.files['report']

        # Secure filename and save files
        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)

        # Create database records
        cv_id = Document.create('cv', cv_name, cv_path)

        # Queue for processing
        try:
            process_uploaded_file_task.delay(cv_id, cv_path, 'cv')
        except Exception:
            upload_queue.put({'doc_id': cv_id, 'path': cv_path,
        'doc_type': 'cv'})

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

Features: - Multipart form-data handling - Secure filename generation - Async file processing via Celery - Fallback to local queue if Celery unavailable

3.2 Evaluation Endpoint (`POST /evaluate`)

```python
@app.route('/evaluate', methods=['POST'])
def evaluate():
    try:
        data = request.get_json()
        job_title = data.get('job_title')
        cv_id = data.get('cv_id')
        report_id = data.get('report_id')
```

```
        if not job_title or not cv_id or not report_id:
            return jsonify({'error': 'job_title, cv_id, dan report_id
        wajib diisi'}), 400

        job_id = Job.create(job_title, cv_id, report_id)

        # Start async processing
        try:
            run_job_task.delay(job_id)
        except Exception:
            t = threading.Thread(target=_run_job, args=(job_id,),
        daemon=True)
            t.start()

        return jsonify({'id': job_id, 'status': 'queued'}), 202
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

3.3 Result Endpoint (`GET /result/{id}`)

```
@app.route('/result/<int:job_id>', methods=['GET'])
def get_result(job_id):
    try:
        job = Job.get_by_id(job_id)
        if not job:
            return jsonify({'error': 'Job tidak ditemukan'}), 404

        status = job['status']
        if status in ['queued', 'processing']:
            return jsonify({'id': job_id, 'status': status})
        elif status == 'completed':
            result = json.loads(job['result_json']) if
        job['result_json'] else {}
            return jsonify({'id': job_id, 'status': status, 'result':
        result})
        else:
            return jsonify({'id': job_id, 'status': status, 'error':
        job['error_message'] or 'Unknown error'}), 500
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

4. AI Integration

4.1 RAG System Implementation

Using ChromaDB for vector storage and retrieval:

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

def ingest_text(doc_id: str, text: str, metadata: Dict[str, Any] |
        None = None):
    if not text:
        return
    _collection.add(documents=[text], metadatas=[metadata or {}], ids=
        [doc_id])

def query(query_text: str, n_results: int = 5):
    if not query_text:
        return []
    res = _collection.query(query_texts=[query_text],
        n_results=n_results)
    return [{'document': d, 'metadata': m} for d, m in
        zip(res['documents'][0], res['metadatas'][0])]
```

4.2 LLM Integration with Gemini

Created a simple wrapper for Gemini API:

```
import google.generativeai as genai

def evaluate_cv(cv_text, job_title, context_snippets):
    prompt = f"""
    Evaluate this CV for the position: {job_title}

    CV: {cv_text}
    Context: {context_snippets}

    Provide match rate (0-1) and detailed feedback.
    """

    model = genai.GenerativeModel('gemini-pro')
    response = model.generate_content(prompt)
    return parse_llm_response(response.text)
```

4.3 Prompt Chaining

Implemented multi-stage evaluation:

1. **CV Evaluation:** Analyze CV against job requirements
2. **Project Evaluation:** Compare project report with case study
3. **Synthesis:** Combine results into overall assessment

5. Error Handling

5.1 API Error Handling

```
@app.errorhandler(404)
def not_found(error):
    return jsonify({'error': 'Endpoint tidak ditemukan'}), 404

@app.errorhandler(500)
def internal_error(error):
    return jsonify({'error': 'Terjadi kesalahan internal'}), 500
```

5.2 Background Processing Error Handling

```
@celery.task(name="evaluate.run_job", bind=True, autoretry_for=
        (Exception,), retry_backoff=10, retry_kwargs={"max_retries":
        5})
def run_job_task(self, job_id: int):
    try:
        run_job(job_id=job_id)
    except Exception as e:
        Job.update_status(job_id, 'failed', error_message=str(e))
        raise self.retry(exc=e, countdown=60 * (2 **
        self.request.retries))
```

5.3 Fallback Mechanisms

- Mock evaluation when LLM API is unavailable
- Local threading queue when Celery broker is down
- Graceful degradation with partial functionality

6. File Processing

6.1 PDF Text Extraction

```
def _read_pdf_text(path):
    try:
        from PyPDF2 import PdfReader
        reader = PdfReader(path)
        return "\n\n".join((p.extract_text() or '') for p in
        reader.pages)
    except Exception:
        return ''
```

6.2 Async File Ingestion

```
def _process_uploaded_file(doc_id, path, doc_type):
    try:
        text = _read_pdf_text(path)
        sidecar = f"{path}.txt"
        with open(sidecar, 'w', encoding='utf-8') as f:
            f.write(text or '')
        # Ingest to Chroma for RAG
        rag.ingest_text(f"doc:{doc_id}", text or '', metadata={'path':
        path, 'doc_type': doc_type})
    except Exception:
        pass
```

7. Testing

7.1 Manual Testing

Tested the complete workflow:

1. **Upload Documents:**

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

2. **Start Evaluation:**

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

3. **Check Results:**

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

7.2 Error Scenarios Tested

- Invalid file formats
- Missing required fields
- LLM API failures
- Database connection issues
- File upload size limits

8. Deployment Setup

8.1 Environment Configuration

```
# .env file
GEMINI_API_KEY=your_api_key_here
REDIS_URL=redis://localhost:6379/0
REDIS_BACKEND=redis://localhost:6379/1
```

8.2 Running the Application

```
# Install dependencies
pip install -r requirements.txt

# Initialize database
python database.py

# Start Redis (optional)
docker run -p 6379:6379 redis:7-alpine

# Start Celery worker (optional)
celery -A celery_app.celery worker -l info --concurrency 4

# Start Flask app
python main.py
```

9. Performance Considerations

9.1 Async Processing

- Celery workers for long-running tasks
- Queue management with proper prioritization
- Background processing for file ingestion

9.2 Database Optimization

- Proper indexing on frequently queried columns
- Connection pooling for better performance
- Query optimization for large datasets

10. Security Features

10.1 File Upload Security

- Filename sanitization with `secure_filename()`
- File type validation
- Size limit considerations

10.2 API Security

- Input validation and sanitization
- Error message sanitization
- Environment variable management

11. Limitations and Future Improvements

11.1 Current Limitations

- Limited document format support (PDF, TXT)
- Single language support (Indonesian/English)
- Basic UI/UX design
- Limited scalability without proper load balancing

11.2 Future Enhancements

- Support for more document formats (DOCX, PPTX)
- Multi-language capability
- Advanced analytics dashboard
- Real-time WebSocket updates
- Enhanced security with authentication
- Microservices decomposition
- Advanced prompt engineering

12. Conclusion

The HR Screening Backend Service successfully implements the core requirements:

✅ **RESTful API endpoints** for upload, evaluate, and result retrieval ✅ **AI-driven pipeline** with RAG and prompt chaining ✅ **Long-running process handling** with Celery workers ✅ **Comprehensive error handling** with fallback mechanisms ✅ **Vector database integration** for context retrieval ✅ **Production-ready code** with proper structure and documentation

The system provides a solid foundation for automated candidate screening with room for future enhancements and improvements.