SHL Assessment Recommendation System - Approach Summary

1. Overview

The SHL Assessment Recommendation System is a full-stack Generative AI application that leverages semantic search and LLM-powered reranking to recommend relevant SHL assessments based on job descriptions or natural language queries. The system is built using modern technologies: FastAPI (backend), React + TailwindCSS (frontend), FAISS (vector store), and OpenAI/Gemini APIs (embeddings and reranking).

2. Architecture & Design

2.1 System Architecture

The system follows a three-tier architecture:

Frontend Layer (React + TailwindCSS) - User interface for entering job descriptions - Displays recommended assessments in a table format - Responsive design with modern UI/UX

API Layer (FastAPI) - RESTful API with `/health` and `/recommend` endpoints - Handles request validation and response formatting - CORS enabled for frontend integration

AI Layer (Embeddings + Vector Search + Reranking) - Embedding generation using OpenAI or Gemini APIs - FAISS vector store for fast similarity search - Optional LLM reranking for improved accuracy

2.2 Data Flow

1. **Data Collection**: Web scraper extracts SHL assessment data (name, description, URL, type) from the product catalog
2. **Embedding Generation**: Assessment descriptions are converted to vector embeddings
3. **Vector Store**: Embeddings are indexed in FAISS for fast similarity search
4. **Query Processing**: User query is embedded and searched against the vector store
5. **Reranking**: Top results are re-ranked using LLM for improved relevance
6. **Response**: Recommendations are returned to the frontend

3. Implementation Details

3.1 Data Collection

The `crawl_catalog.py` script uses BeautifulSoup to scrape the SHL product catalog. It: - Extracts assessment names, descriptions, URLs, and types - Handles pagination and multiple pages - Includes fallback sample data for common SHL assessments - Saves data in both JSON and CSV formats

Key Features: - Robust error handling for network issues - Multiple selector strategies for different page layouts - Deduplication of assessments - Sample data generation for testing

3.2 Embedding Generation

The system supports multiple embedding providers with automatic fallback:

1. **OpenAI Embeddings** (`text-embedding-3-small`)
   - High-quality embeddings
   - 1536 dimensions
   - Fast and reliable
2. **Gemini Embeddings** (`text-embedding-004`)
   - Alternative to OpenAI
   - Comparable quality
   - 768 dimensions
3. **Sentence Transformers** (Fallback)
   - Local model (`all-MiniLM-L6-v2`)
   - No API key required
   - 384 dimensions
   - Useful for development/testing

The embedding generator handles batch processing for efficiency and includes error handling for API failures.

3.3 Vector Store (FAISS)

FAISS (Facebook AI Similarity Search) is used for efficient similarity search:

- **Index Type**: `IndexFlatL2` (exact L2 distance search)
- **Storage**: Persistent storage of index and metadata
- **Search**: Fast retrieval of top-k similar assessments
- **Scalability**: Can handle thousands of assessments efficiently

The vector store automatically loads existing indices or builds new ones from the catalog data.

3.4 LLM Reranking

Optional LLM-powered reranking improves recommendation accuracy:

Process: 1. Retrieve top 2k candidates from vector search (where k is desired results) 2. Create a prompt with the query and candidate assessments 3. Ask LLM to rank assessments by relevance 4. Parse LLM response to get ranked order 5. Return top-k reranked results

Supported LLMs: - Gemini 1.5 Pro (preferred for better instruction following) - GPT-4 (fallback)

Benefits: - Better understanding of job requirements - Context-aware ranking - Improved relevance for complex queries

3.5 Recommendation Engine

The recommendation engine orchestrates the entire process:

1. **Initialization**: Loads or builds vector store from catalog data
2. **Query Embedding**: Converts user query to embedding vector
3. **Vector Search**: Finds similar assessments using FAISS
4. **Reranking**: Optionally reranks results using LLM
5. **Formatting**: Formats results with assessment details

Key Features: - Lazy initialization for faster startup - Error handling and fallbacks - Configurable top-k results - Support for disabling reranking

3.6 API Endpoints

GET `/health` - Simple health check endpoint - Returns `{"status": "ok"}`

POST `/recommend` - Accepts: `{"query": "job description", "top_k": 10}` - Returns: List of recommended assessments with names, URLs, types, and descriptions - Includes error handling and validation

3.7 Frontend

The React frontend provides a clean, modern interface:

- **Input**: Large textarea for job descriptions
- **Results Table**: Displays recommendations with:
  - Rank
  - Assessment name
  - Test type (with color coding)
  - URL link
  - Expandable descriptions
- **Loading States**: Visual feedback during API calls
- **Error Handling**: User-friendly error messages

Styling: TailwindCSS for responsive, modern design

3.8 Evaluation

The evaluation script (`evaluate.py`) provides:

- Batch processing of test queries
- CSV output generation
- Simplified format (Query, Assessment_url) as per requirements

- Detailed format with additional metadata
- API health checking

4. Technical Decisions

4.1 Why FAISS?

- **Fast**: Optimized C++ implementation with Python bindings
- **Scalable**: Handles large vector databases efficiently
- **Persistent**: Can save/load indices for faster startup
- **Flexible**: Supports multiple index types for different use cases

4.2 Why Multiple Embedding Providers?

- **Flexibility**: Users can choose based on API availability
- **Cost**: Different providers have different pricing
- **Reliability**: Fallback options if one provider fails
- **Development**: Local fallback for testing without API keys

4.3 Why LLM Reranking?

- **Accuracy**: Semantic search alone may not capture all nuances
- **Context**: LLMs understand job requirements better
- **Relevance**: Better matching of assessments to job needs
- **Optional**: Can be disabled for faster responses or cost savings

4.4 Why FastAPI?

- **Performance**: High performance, comparable to Node.js
- **Async**: Native async/await support
- **Validation**: Automatic request/response validation with Pydantic
- **Documentation**: Auto-generated API documentation
- **Type Safety**: Python type hints for better code quality

4.5 Why React + Vite?

- **Modern**: Latest React features and best practices
- **Fast**: Vite for fast development and builds
- **Simple**: Easy to set up and deploy
- **Responsive**: TailwindCSS for mobile-friendly design

5. Deployment Considerations

5.1 Backend (Render/Hugging Face)

- Environment variables for API keys
- Persistent storage for vector indices
- Health checks for monitoring

- CORS configuration for frontend access

5.2 Frontend (Vercel)

- Environment variable for API URL
- Build optimization with Vite
- Static asset hosting
- CDN distribution for fast loading

5.3 Scalability

- Vector store can be scaled horizontally
- Embedding generation can be cached
- API responses can be cached for common queries
- Database integration for production (optional)

6. Future Enhancements

1. **Caching**: Redis for caching common queries and embeddings
2. **Multi-domain Balance**: Ensure mix of technical and behavioral assessments
3. **Trace Logging**: Show LLM reasoning for transparency
4. **URL Scraping**: Automatically scrape job descriptions from URLs
5. **User Feedback**: Collect user feedback to improve recommendations
6. **A/B Testing**: Test different reranking strategies
7. **Analytics**: Track recommendation performance and usage

7. Conclusion

The SHL Assessment Recommendation System successfully combines semantic search with LLM-powered reranking to provide accurate, relevant assessment recommendations. The modular architecture allows for easy extension and maintenance, while the use of modern technologies ensures good performance and user experience. The system is production-ready with proper error handling, validation, and deployment configurations.