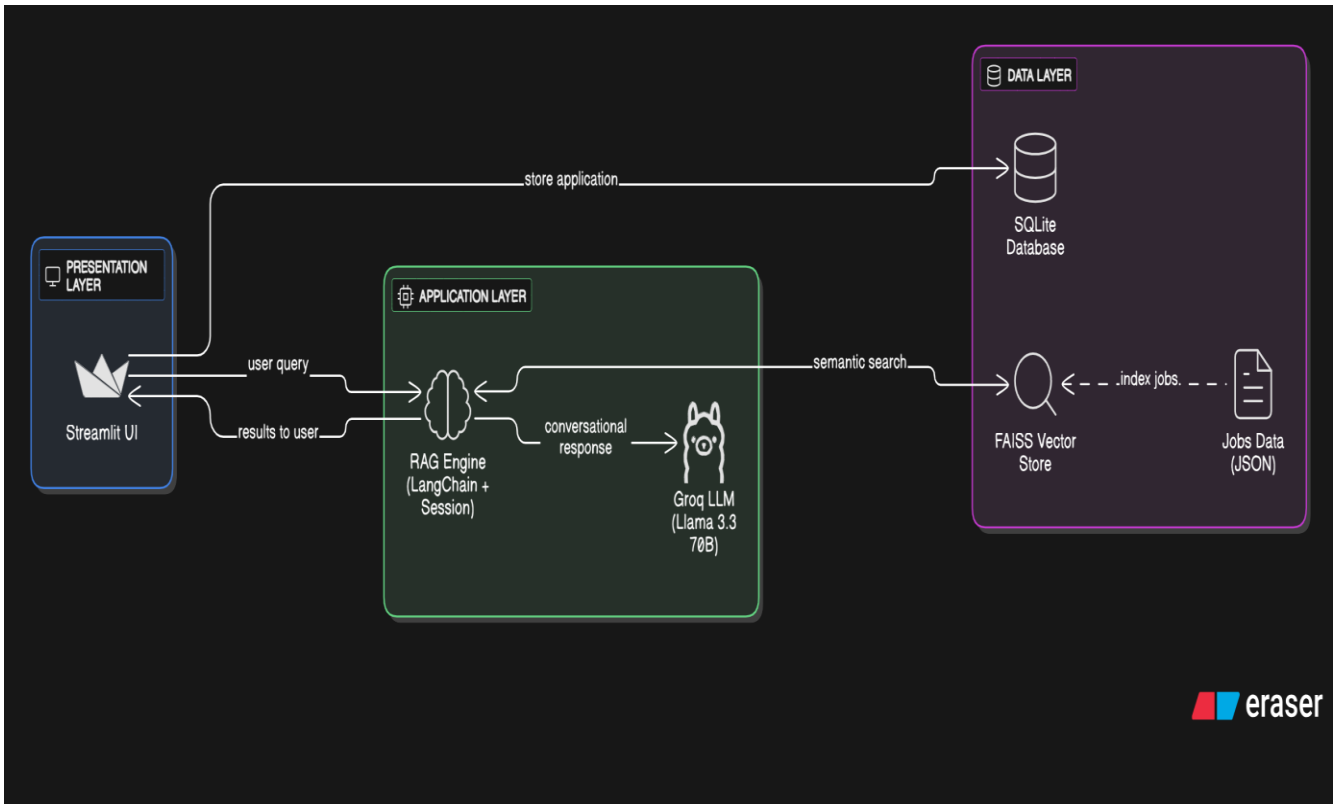
System Design Document – SmartApply RAG Agent

1. Executive Summary

SmartApply is an AI-powered job application system that leverages Retrieval-Augmented Generation (RAG) to provide intelligent job matching and streamlined application processing. The system uses semantic search to match candidates with relevant positions and guides them through a conversational application process.

2. System Architecture

2.1 High-Level Architecture



2.2 Component Details

Presentation Layer (app.py)

- **Technology**: Streamlit
- **Responsibilities**:
    - User interface for job search
    - Application form handling

- Resume upload management
- Admin dashboard
- Session state management

Application Layer (RAG Engine (rag_engine.py))

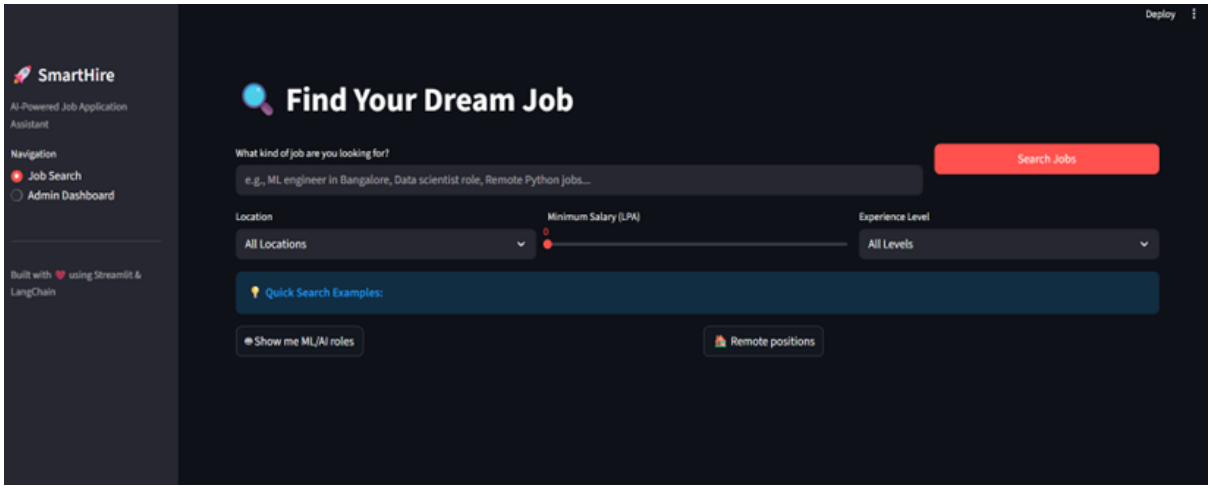
- **Core Components**:
  - **Embeddings**: HuggingFace sentence-transformers/all-MiniLM-L6-v2
  - **Vector Store**: FAISS with persistent indexing
  - **LLM**: Groq API (Llama 3.3 70B)
  - **Memory**: ConversationBufferMemory for context retention

Database Layer (database.py)

- **Technology**: SQLite with SQLAlchemy ORM
- **Tables**:
  - applications: Core application data
  - screening_responses: Q&A storage
  - resumes: Binary file storage

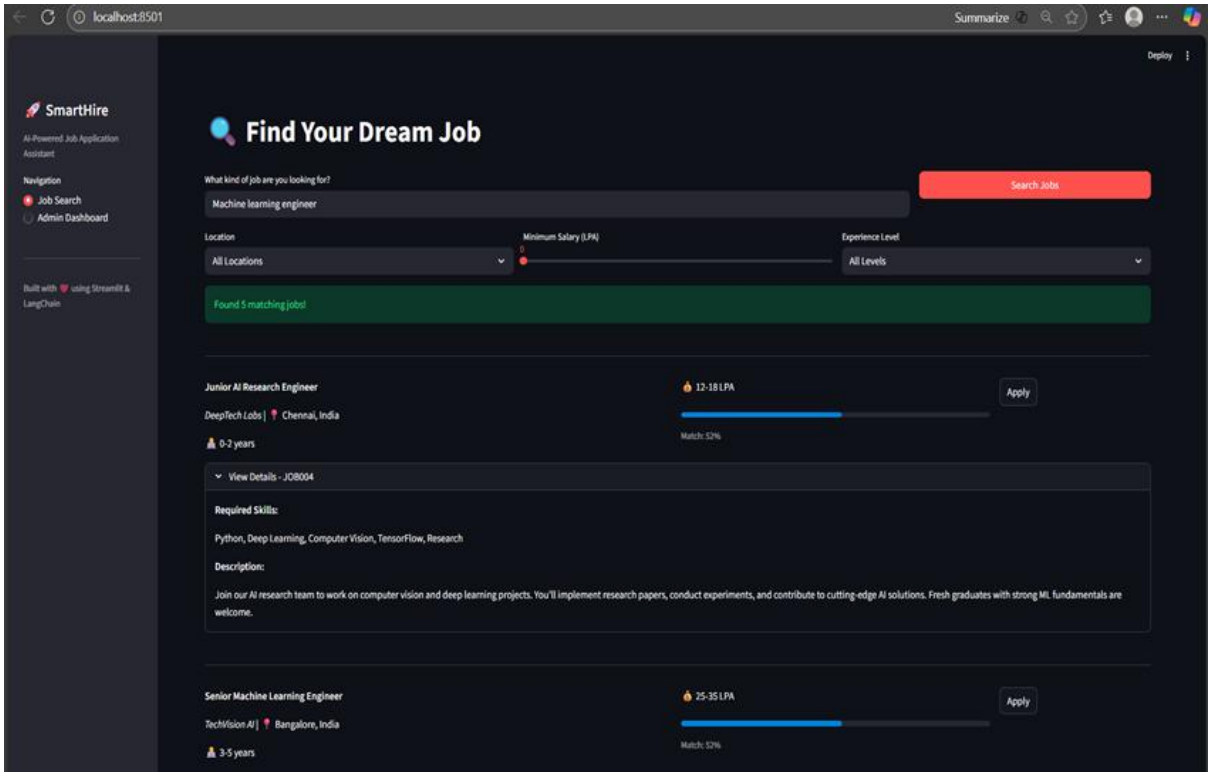
2.3 User Interface Examples

The system architecture translates to these key user interfaces that demonstrate the end-to-end workflow:



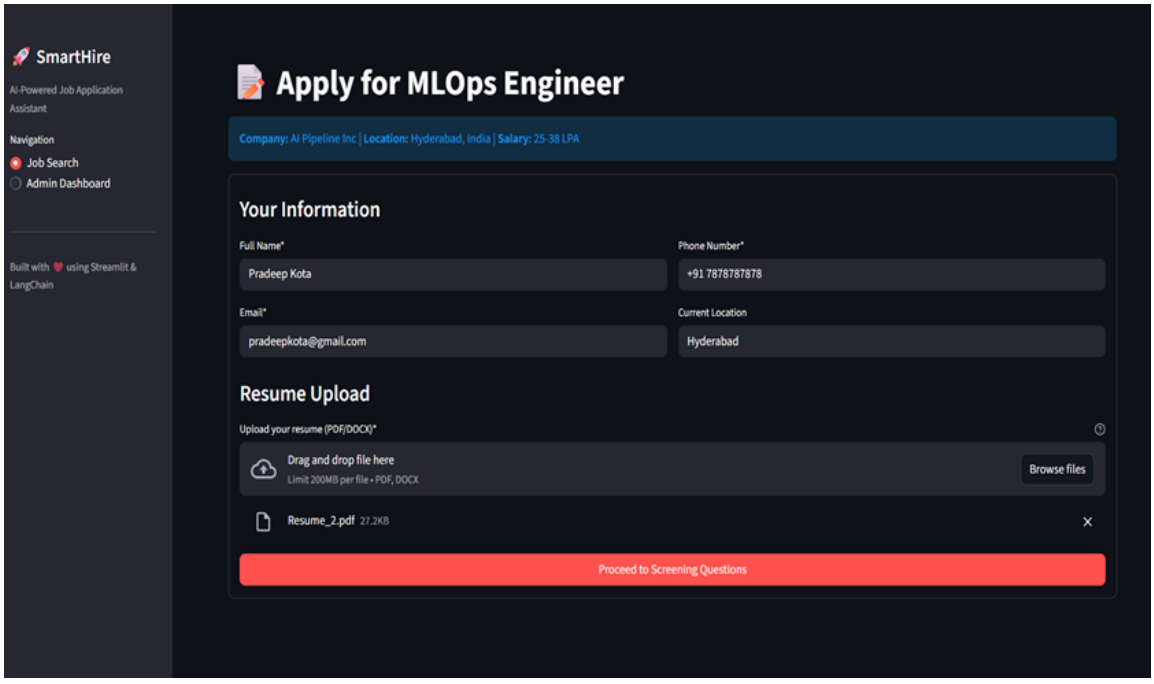
Job Search Interface

Implements semantic search with RAG-powered matching and relevance scoring



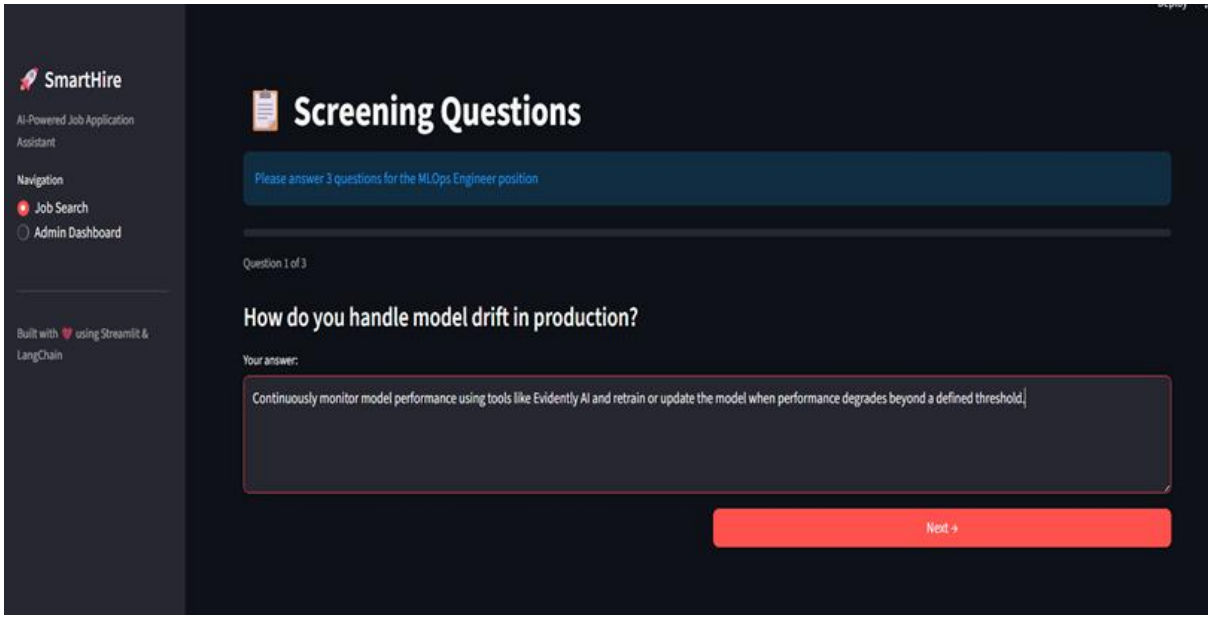
Application Form

Shows multi-step application process with resume upload and candidate information collection



Screening questions

Dynamic question-answering interface with job-specific screening and progress tracking

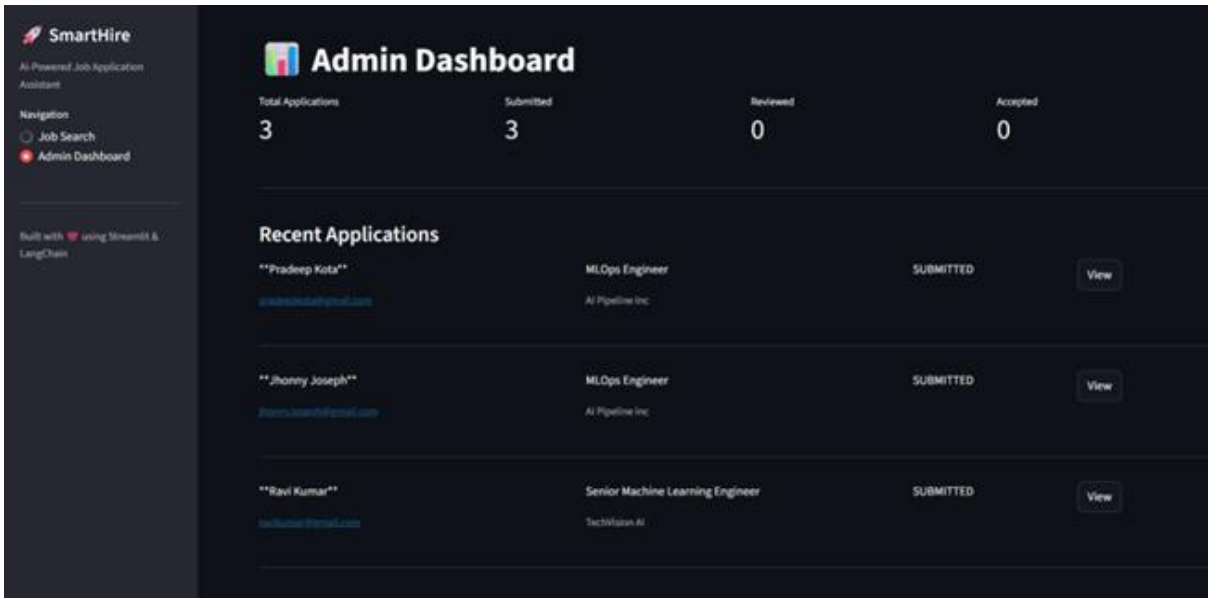


Application Submission screen

Success confirmation interface displaying application ID and submission status

Admin Dashboard

Provides application management, analytics, and candidate tracking capabilities



3. Data Flow

3.1 Job Search Flow

1. User Query → "ML engineer in Bangalore"

2. Query Embedding → Convert to vector representation

3. Semantic Search → FAISS similarity search

4. Retrieve Jobs → Top-k matching jobs

5. Display Results → Ranked by relevance score

3.2 Application Flow

1. Job Selection → User clicks "Apply"

2. Information Collection → Name, email, phone, location

3. Resume Upload → PDF/DOCX file processing

4. Screening Questions → Dynamic Q&A based on job

5. Data Persistence → Store in SQLite

6. Confirmation → Application ID generation

4. Key Features

4.1 Semantic Job Search

- Vector similarity matching using FAISS
- Relevance scoring (0-100% match)
- Filter support (location, salary, experience)

4.2 Intelligent Caching

- MD5 hash-based change detection for jobs.json
- Persistent FAISS index to avoid re-computation
- Automatic reindexing on data changes

4.3 Conversational Interface

- Natural language job queries
- Context-aware responses using conversation memory
- Multi-turn dialogue support

4.4 Application Management

- Complete application lifecycle tracking
- Status management (submitted/reviewed/accepted/rejected)
- Admin dashboard with statistics

5. Technical Specifications

5.1 Dependencies

- **LangChain**: Orchestration framework
- **FAISS**: Vector similarity search
- **Groq**: LLM inference
- **Streamlit**: Web interface
- **SQLAlchemy**: Database ORM
- **Sentence-Transformers**: Free embeddings

5.2 Data Models

Job Schema

{

```
 "job_id": "string",
 "title": "string",
 "company": "string",
 "location": "string",
 "experience_required": "string",
 "salary_range": "string",
 "skills_required": ["array"],
 "screening_questions": ["array"]
}
```

Application Schema

```
applications (
 id: INTEGER PRIMARY KEY,
 job_id: VARCHAR,
 candidate_name: VARCHAR,
 candidate_email: VARCHAR,
 status: VARCHAR,
 submitted_at: DATETIME
)
```

6. Performance Optimizations

6.1 Embedding Caching

- First run: ~30 seconds to create embeddings
- Subsequent runs: <2 seconds (cached loading)
- Hash-based invalidation for data updates

6.2 Resource Usage

- Memory: ~200MB for vector store (20 jobs)
- Storage: ~10MB for FAISS index
- API Calls: Minimal (only for LLM responses)

7. Security Considerations

- API keys stored in environment variables
- SQL injection prevention via ORM
- File upload validation (type and size checks)
- No PII exposed in logs

8. Scalability

Current Limitations

- Single-user session management
- Local file storage for resumes
- In-memory conversation history

Scaling Strategy

- Move to PostgreSQL for production
- Implement Redis for session management
- Use cloud storage (S3) for resumes
- Deploy on cloud platforms (AWS/GCP)

9. Testing Approach

Functional Testing

- End-to-end application flow
- Job search accuracy validation
- Database persistence verification

Performance Testing

- Embedding generation time
- Search response latency
- Concurrent user handling

10. Deployment

Local Development

pip install -r requirements.txt

streamlit run app.py

Production Deployment

- Containerize with Docker
- Environment variable management
- HTTPS enforcement
- Rate limiting implementation

11. Future Enhancements

- Multi-language support
- Resume parsing and skill extraction
- Email notifications
- Advanced analytics dashboard
- Integration with ATS systems
- Real-time job updates

12. Conclusion

The SmartApply RAG Agent successfully demonstrates an end-to-end AI-powered recruitment system with semantic search, conversational interaction, and complete application management. The architecture is modular, scalable, and optimized for performance while maintaining simplicity.