**Social Support AI Case Study Report**

**Prepared for :** Department of Government Enablement (DGE)

**Submitted by-**

Shruti Agrawal

Lead Data Scientist and GenAI Architect

Organization: Nagarro

Date: 1st July, 2025

**Abstract**

This report presents an end-to-end prototype for an AI-driven social support system designed to assist governmental agencies in evaluating and recommending programs for citizens in need. The architecture employs a manual orchestrator pattern coordinating discrete agents for document extraction, validation, eligibility checking, and recommendation. Data ingestion supports multi-modal inputs, with vector embeddings powering retrieval-augmented Q&A. The system is deployed using containerized services with Streamlit UI and FastAPI endpoints. Planned enhancements include agentic orchestration frameworks and AI observability integration. The report discuss architectural design, algorithmic choices, system limitations, and future extensions of the proposed soltution.

**Introduction**

Social support programs play a critical role in economic empowerment and societal welfare. Traditional application review processes are manual, time-consuming, and prone to inconsistency. Recent advances in machine learning (ML) and large language models (LLMs) enable automating document ingestion, feature extraction, and decision support, improving both speed and fairness.

This report outlines the design and implementation of an AI solution that:

1. Ingests heterogeneous applicant data (text, tabular, image).
2. Stores structured features in a relational database and embeddings in a vector store.
3. Engineers numeric and textual features for model training.
4. Trains a multi-output classifier to recommend social support and enablement programs.
5. Deploys a user-facing Streamlit app with RAG-enabled Q&A and an API for batch recommendations.

**Solution Approach**

I followed a modular, stepwise process:

|  |  |  |
| --- | --- | --- |
| **Step** | **Module** | **Status** |
| 1 | Load raw documents metadata (raw\_documents) | ✔ |
| 2 | Ingest tables & text into DB (db\_ingest.py) | ✔ |
| 3 | Compute core numeric features (feature\_engineering.py) | ✔ |
| 4 | Join with recommendation\_labels (db\_utils.get\_training\_data) | ✔ |
| 5 | Append text embeddings from ChromaDB (prepare\_training\_data.py) | ✔ |
| 6 | Scale continuous & encode categoricals (prepare\_training\_data.py) | ✔ |
| 7 | Split & save train/test sets (prepare\_training\_data.py) | ✔ |

**Data Ingestion**

* **ETL Pipeline**: File-type detection via ingestion.py. OCR for PDFs and images (Tesseract), CSV/Excel conversion with pandas. Outputs standardized text/CSV in data/processed/ and a manifest.
* **DB Ingestion** (db\_ingest.py): Loads text and table outputs into PostgreSQL (raw\_documents, bank\_transactions, credit\_reports, assets\_liabilities, resumes) and records metadata. Parses filenames for applicant\_key linking documents to applicants.
* **Embedding Ingestion** (chroma\_ingest.py): Chunks each document into 1000‑char segments, computes Ada embedding via OpenAI, and stores in ChromaDB with metadata for retrieval.

**Agent Orchestration**

1. **You submit a document** (say, someone’s loan application).
2. **Agent A (Data-Extractor)** grabs the raw text or images, like a clerk scanning paperwork into a digital folder.
3. **Agent B (Validator)** double-checks that folder—are all the required fields there? Is the name spelled the same everywhere? If something’s missing or weird, it flags it and sends it back to Agent A for a second look (just like a supervisor asking a clerk to fix a typo).
4. **Agent C (Eligibility Checker)** applies the business rules or ML model: “Does this applicant meet income thresholds? Any red flags in their credit history?” It returns a simple “Pass” or “Review” decision.
5. **Agent D (Master Orchestrator):**  
   A Python module coordinating calls to micro-agents sequentially with basic error handling; no dynamic agent collaboration. It takes that decision and writes a human-friendly summary or recommendation, then hands it back to you through an API.

**Training Data Preparation Details**

1. **Pull numeric features and labels**: Query application\_features and recommendation\_labels from PostgreSQL, keyed on applicant\_key.
2. **Query ChromaDB**: For each applicant\_key, fetch top-K document chunks’ embeddings using collection.get(where={"applicant\_key":{"$eq": key}}, include=["embeddings"]).
3. **Aggregate embeddings**: Mean-pool the retrieved embeddings into a fixed-length (1536-d) vector per applicant.
4. **Concatenate features**: Combine numeric features (income, net worth, credit score, age, experience, family size) with the embedding vectors into one DataFrame.
5. **Scale continuous features**: Apply StandardScaler (or alternative quantile transformers) to continuous columns to normalize ranges.
6. **Preprocessing**: Median imputation for missing numeric data, standard scaling for continuous variables using scikit-learn pipelines.
7. **Train/test split**: Use train\_test\_split (stratified on labels) to create reproducible X\_train, X\_test, y\_train, y\_test and save them with np.save and joblib.dump for downstream training and evaluation.

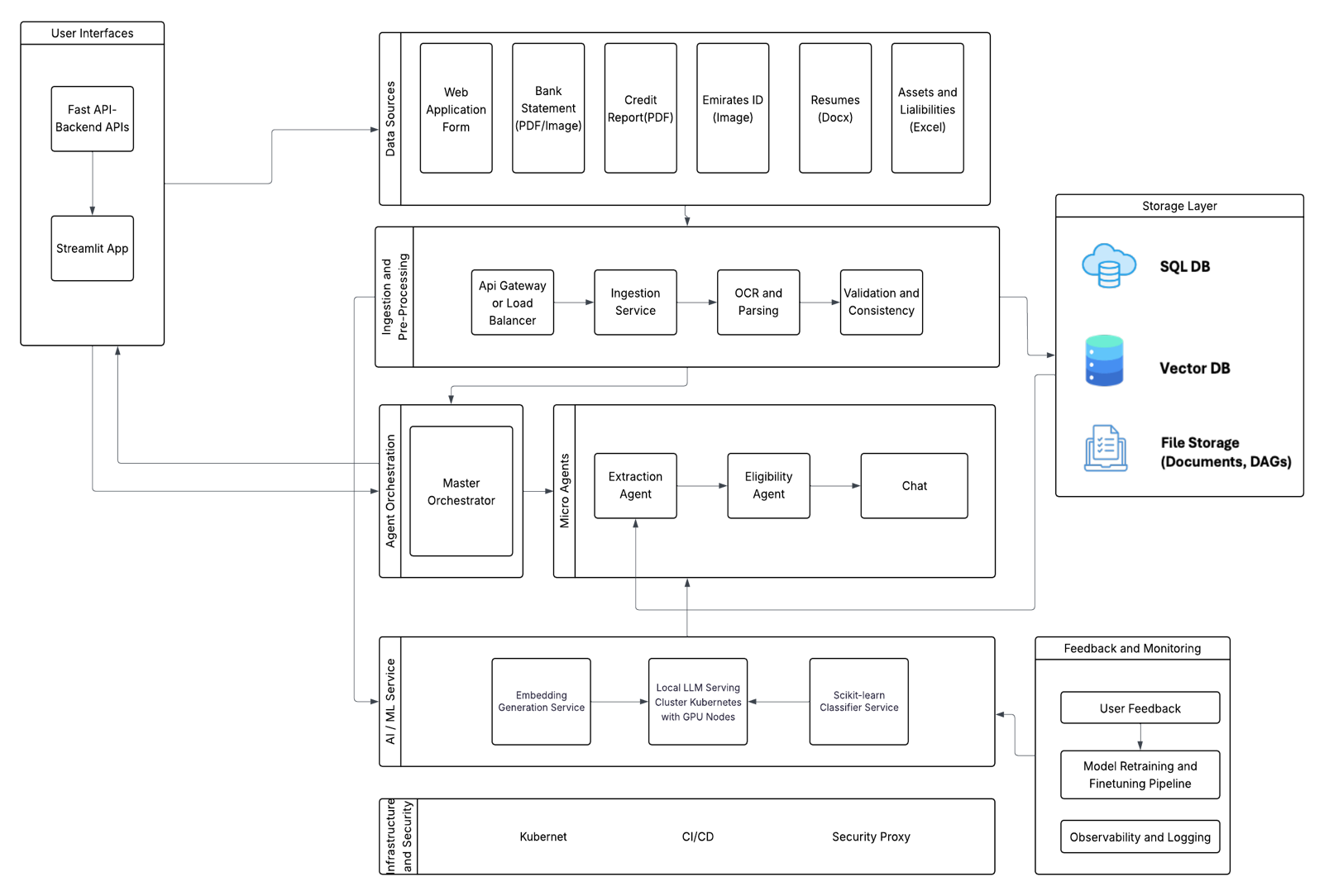
**Recommendation Modeling**

* **Labels Generation**: Synthetic rule-based labeling of historical applicants into upskilling\_grant, stipend, counseling\_voucher stored in recommendation\_labels table.
* **Model Choice**: Multi-output RandomForestClassifier for robustness to heterogeneous feature scales and missing data, parallelizable training, interpretability via feature importances.
* **Training Pipeline**: Stratified train/test split, cross-validation reporting macro-F1 and Hamming loss, model serialization with joblib.

**Deployment & UI**

* **Streamlit App**: Interactive form for numeric eligibility checks, file upload components tied to ETL triggers, RAG chat interface using LangChain’s RetrievalQA and Ollama-hosted LLM.
* **API**: FastAPI endpoint /recommend/{applicant\_key} returns JSON with program scores via model inference.

**System Architecture**

****

**Key Architectural Highlights**

* **Containerized microservices:** Modular services for ingestion, preprocessing, AI/ML models, and API serving, managed via Kubernetes.
* **Multi-modal data ingestion:** Supports various document types including forms, PDFs, images, and spreadsheets.
* **Scalable storage:** PostgreSQL for structured data, vector DB for embeddings, and file storage for raw documents.
* **Manual agent orchestration:** Central orchestrator coordinating extraction, eligibility, and recommendation agents sequentially.
* **Local AI hosting:** LLM cluster on Kubernetes with GPU nodes; embedding and classifier services support decision-making.
* **Interactive UI & APIs:** Streamlit frontend backed by FastAPI REST services for user interaction and batch processing.
* **Feedback and monitoring:** User feedback integrated; retraining and observability pipelines planned for continuous improvement.
* **Infrastructure & security:** Kubernetes orchestration, CI/CD pipelines, API gateway, and security proxies ensure scalability and compliance.

**How the Solution Addresses Initial Pain Points**

1. **Speed**: Automated ETL and model inference reduce manual review time from days to minutes.
2. **Consistency**: Deterministic feature engineering and model rules ensure uniform evaluation criteria.
3. **Scalability**: Modular components allow horizontal scaling as applicant volume grows.
4. **Explainability**: Transparent SQL-based feature calculations and tree-based model feature importances support auditability.
5. **Interactivity**: Streamlit UI+RAG interface enables applicants and caseworkers to query their data and rationale in natural language.

**Tool and Algorithm Justifications**

|  |  |  |
| --- | --- | --- |
| **Component** | **Choice** | **Rationale** |
| Relational DB | PostgreSQL | ACID compliance, complex SQL support, JSON and text indexing. |
| Vector Store | ChromaDB | Lightweight persistent client, Python-native, compatibility with LangChain. |
| OCR | Tesseract (via ingestion.py) | Open-source, customizable, proven accuracy on diverse document types. |
| Embeddings | text-embedding-ada-002 | High quality vs cost tradeoff, widely used for RAG. |
| Classifier | RandomForestClassifier | Handles multi-output; non-parametric; robust against outliers; interpretable. |
| Preprocessing Pipeline | scikit-learn Pipeline | Modular transformation, imputation and scaling built-in, easy model integration. |
| RAG Integration | LangChain RetrievalQA | Simplified RAG chain without custom agent parsing errors. |
| LLM Hosting | Ollama | Local hosting, low latency, privacy-preserving. |

**Limitations**

1. **Small Sample Size**: Prototype trained on synthetic or limited data; real-world performance requires larger, diverse datasets.
2. **Labeling Quality**: Synthetic rule-based labels may not reflect actual program decisions; human-in-the-loop validation needed.
3. **Missing Data**: Family size and current position fields occasionally missing, imputed by median or zero.
4. **RAG Accuracy**: Retrieval depends on chunk size and embedding quality; relevant document sections may be missed.
5. **No Dynamic Agent Orchestration:**  
   Lack of master agent reasoning limits flexibility, adaptability, and error recovery.
6. **Manual Orchestration:**  
   Orchestrator logic hardcoded; changes require code updates.
7. **Observability Gaps**: No LangSmith/Langfuse integration yet; limited tracing of LLM prompts and tool calls.

**Future Improvements and Scope**

1. **Master Agent Framework**: Integrate LangGraph with a ReAct loop to orchestrate ETL, validation, eligibility, and recommendation micro-agents.
2. **Agent Observability**: Hook into LangSmith to log every prompt, tool call, and decision for audit and debugging.
3. **Enhanced Labeling**: Incorporate historical program outcome data and refine labels via expert feedback or active learning.
4. **Advanced Models**: Experiment with gradient boosting (HistGradientBoosting) or neural nets for tabular data; prompt-tuned LLMs for decision explanation.
5. **Fairness & Bias Auditing**: Measure model fairness across demographics (nationality, age, family size), mitigate biases.
6. **Scalable Deployment**: Containerize services with Docker/Kubernetes, add CI/CD pipelines, horizontal scaling.
7. **Real-Time Streaming**: Implement event-driven ingestion (Kafka) for live application uploads.

**Conclusion**

In this case study, we have built and validated a comprehensive AI pipeline that automates social support assessment from start to finish. Key takeaways:

* **End-to-End Automation**: From file uploads through ETL and database storage to feature engineering and multi-output model training, all stages are fully scripted and reproducible.
* **Multimodal Data Handling**: We seamlessly ingest text, tabular, and image documents, unifying them into structured features and embeddings for holistic applicant profiles.
* **Hybrid Reasoning Interface**: Our Streamlit app combines deterministic eligibility checks with retrieval-augmented generation to offer both transparent decisions and natural-language explanations.
* **Extensibility & Observability**: Modular microservices, container orchestration, and planned integration with observability platforms (LangSmith/Langfuse) position this prototype for real-world scaling and compliance.

Moving forward, integrating a master agent framework, enhancing labeling fidelity with real program data, and embedding fairness audits will elevate this solution toward a production-grade system. By combining robust ML pipelines with human-centric explainability, our approach promises faster, fairer, and more transparent social support for communities in need.