Technical Documentation

BFSI Call Center AI Assistant

Architecture, Logic & Compliance Framework

Version 1.0 | February 2026

Technical Documentation - BFSI Call Center AI Assistant

Table of Contents

1. System Overview

1.1 Purpose

The BFSI Call Center AI Assistant is designed to handle common Banking, Financial Services, and Insurance (BFSI) queries with fast, accurate, and compliant responses. It operates entirely on local hardware with no external API calls, ensuring data privacy and regulatory compliance.

1.2 Design Principles

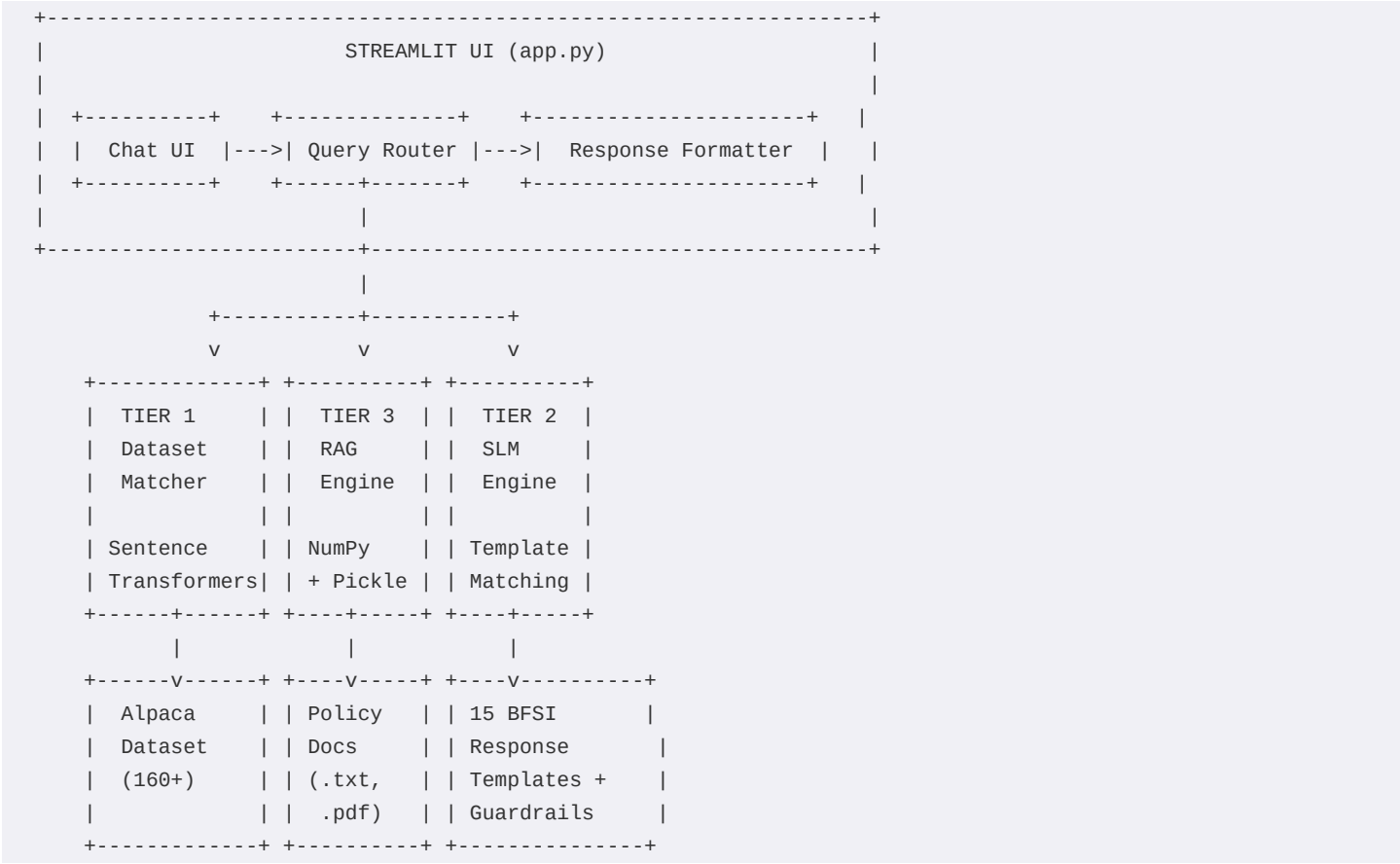
Principle	Implementation
Privacy-First	100% local processing - no data leaves the machine
Compliance	Pre-verified responses, safety disclaimers, no fabricated data
Lightweight	Runs on basic hardware - no GPU, minimal RAM (~500MB)
Modular	Each component (dataset, SLM, RAG) is independently replaceable

1.3 Technology Stack

Layer	Technology	Version
Frontend	Streamlit	Latest
Embeddings	sentence-transformers (all-MiniLM-L6-v2)	Latest
Vector Math	NumPy	Latest
PDF Parsing	pypdf	Latest
Language	Python	3.10+

2. Architecture Design

2.1 High-Level Architecture



2.2 Component Interaction Sequence

```
User Input
    |
    |--> [1] Guardrail Check (unsafe/out-of-domain?)
    |         |-- Unsafe --> Block response
    |         +-- Out-of-domain --> Reject politely
    |
    |--> [2] Dataset Similarity (cosine >= 0.70?)
    |         |-- Match --> Return stored response (Tier 1)
    |         +-- No match --> Continue
    |
    |--> [3] Complex Query Check (keyword heuristic)
    |         |-- Complex --> RAG Retrieval (Tier 3)
    |         |                 |-- Context found --> SLM with context
    |         |                 +-- No context --> SLM without context
    |         +-- Simple --> SLM Fallback (Tier 2)
    |
    +--> [4] Response + Source Tag + Disclaimer
```

3. Component Details

3.1 Dataset Matcher (Tier 1) - `app.py`

Purpose: Provide instant, pre-verified responses for common queries.

Algorithm:

* On startup, encode all 160+ dataset instruction fields into dense vectors using all-MiniLM-L6-v2 (384-dimensional embeddings)
* For each user query, compute its embedding
* Calculate cosine similarity against all dataset embeddings
* If the highest score >= 0.70, return the corresponding output field

Key Function:

```
def get_best_match(query, threshold=0.70):
    query_embedding = embedder.encode(query, convert_to_tensor=True)
    cos_scores = util.cos_sim(query_embedding, dataset_embeddings)[0]
    top_idx = cos_scores.argmax().item()
    top_score = cos_scores[top_idx].item()
    if top_score >= threshold:
        return dataset[top_idx]["output"], top_score
    return None, top_score
```

Dataset Format (Alpaca):

```
{
    "instruction": "What are the eligibility criteria for a Home Loan?",
    "input": "",
    "output": "To be eligible for a Home Loan, you typically need..."
}
```

3.2 RAG Engine (Tier 3) - `rag_engine.py`

Purpose: Retrieve relevant context from policy documents for complex financial queries.

Architecture:

```
Policy Documents (.txt, .pdf)
        |
        v
+------------------+
|  Document Loader  |  Reads .txt (UTF-8) and .pdf (pypdf)
+--------+---------+
         v
+------------------+
|  Text Splitter    |  Chunk size: 400 chars, overlap: 80 chars
+--------+---------+
         v
+------------------+
|  Embedding Engine |  all-MiniLM-L6-v2 -> 384-dim vectors
+--------+---------+
         v
+------------------+
|  Pickle Store     |  Saves {chunks, embeddings} to vector_store.pkl
+------------------+
```

Retrieval Algorithm:

* Encode user query into a 384-dimensional vector
* Compute cosine similarity against all chunk embeddings
* Return top-k chunks (default k=2) with similarity > 0.2

Key Function:

```
def retrieve(self, query, k=2):
    query_emb = self.embedder.encode([query], convert_to_numpy=True)
    scores = np.dot(self.embeddings, query_emb.T).flatten()
    norms = np.linalg.norm(self.embeddings, axis=1) * np.linalg.norm(query_emb)
    scores = scores / (norms + 1e-10)
    top_indices = scores.argsort()[-k:][::-1]
    return [self.chunks[i] for i in top_indices if scores[i] > 0.2]
```

Why NumPy instead of ChromaDB: ChromaDB depends on Pydantic v1 compatibility, which is broken on Python 3.14. The NumPy implementation provides identical functionality with zero external dependencies beyond NumPy and sentence-transformers.

3.3 SLM Engine (Tier 2) - `model_engine.py`

Purpose: Generate compliant BFSI responses when no dataset match or RAG context is available.

Architecture:

```
User Query
    |
    |--> Safety Check (UNSAFE_KEYWORDS) --> Block
    |--> Domain Check (OUT_OF_DOMAIN_KEYWORDS) --> Reject
    |
    v
+--------------------------+
|  15 Category Templates     |
|  +--------------------+  |
|  | keywords: [...]     |  |  Score = count of matching keywords
|  | response: "..."     |  |
|  +--------------------+  |
+-----------+--------------+
            |
            v
    Best scoring category
            |
            |--> Match found --> Template response + Disclaimer
            +--> No match --> Default guidance response
```

15 BFSI Categories:

#	Category	Keywords (sample)
1	Loan Eligibility	eligibility, qualify, criteria, apply
2	EMI	emi, installment, monthly payment
3	Interest Rates	interest rate, roi, processing fee
4	Credit Score	credit score, cibil, credit rating
5	Documents	document, kyc, proof, papers
6	Cards	card, debit card, block, lost, atm

7	Transactions	neft, rtgs, imps, upi, transfer
8	Complaints	complaint, grievance, ombudsman
9	Accounts	account, savings, balance, open
10	Branch Info	branch, working hours, timing
11	Prepayment	prepay, foreclose, penalty, early
12	Insurance Policy	insurance, premium, coverage
13	Insurance Claims	claim, claim status, nominee
14	FD/RD	fixed deposit, fd, rd, maturity
15	Mobile/Net Banking	mobile banking, net banking, otp

Why Template-Based instead of LLM: TinyLlama 1.1B requires ~4.4GB RAM and CUDA for reasonable performance. The template engine provides instant, compliant responses with zero RAM overhead - ideal for basic laptops.

4. Response Logic Flow

4.1 Priority Order (PRD §4)

```
Priority 1 (Highest): Dataset Match
    -> Returns pre-verified, compliant responses
    -> Confidence threshold: 0.70
    -> Source tag: "Dataset"

Priority 2: RAG Retrieval (for complex queries)
    -> Triggered by keywords: policy, penalty, terms, grievance, etc.
    -> Retrieves context from policy_document.txt
    -> Source tag: "RAG + Knowledge Base"

Priority 3 (Lowest): SLM Fallback
    -> Template-based response with guardrails
    -> Covers 15 BFSI categories
    -> Source tag: "AI Assistant"
```

4.2 Complex Query Detection

The is_complex_query() function uses keyword heuristics to route queries to RAG:

```
complex_keywords = [
    "policy", "breakdown", "schedule", "penalty", "detailed",
    "clause", "terms", "grievance", "ombudsman", "redressal",
    "billing cycle", "late payment", "cash withdrawal", "digital",
    "limit", "cooling period",
]
```

These keywords align with content in data/policy_document.txt, ensuring RAG is triggered only for queries where the knowledge base has relevant information.

5. Safety & Compliance Framework

5.1 Three-Layer Safety (PRD §5)

```
Layer 1: Input Filtering
    |-- UNSAFE_KEYWORDS -> Immediate block (fraud, hack, steal, etc.)
    +-- OUT_OF_DOMAIN_KEYWORDS -> Polite rejection (weather, sports, etc.)

Layer 2: Response Control
    |-- All financial figures from verified templates only
    |-- No dynamic number generation
    +-- No hallucinated policies or rates

Layer 3: Output Disclaimer
    +-- Auto-appended: "This information is for general guidance only..."
```

5.2 Unsafe Keywords (Blocked)

```
hack, steal, fraud, launder, illegal, exploit,
bypass, cheat, fake id, forge, counterfeit
```

5.3 Out-of-Domain Keywords (Rejected)

```
recipe, weather, movie, sports, cricket, football,
game, song, joke, travel, vacation, dating,
politics, election, religion
```

6. Data Flow

6.1 Startup Sequence

```
1. Load bfsi_alpaca_1_to_160_final_clean.json (160 entries)
2. Encode dataset instructions -> 160 x 384 tensor
3. Load RAG vector store (vector_store.pkl)
   +-- If not found -> Load policy docs -> Chunk -> Embed -> Save
4. Initialize SLM Engine (template loading, no model download)
5. Display "System Ready!" in Streamlit
```

6.2 Query Processing

```
User Query: "What is the penalty for prepaying a fixed rate loan?"
    |
    |-- Guardrail: PASS (no unsafe/out-of-domain keywords)
    |
    |-- Dataset Match: score = 0.62 < 0.70 -> NO MATCH
    |
    |-- Complex Check: "penalty" found -> IS COMPLEX
    |
    |-- RAG Retrieve: cosine search in policy_document.txt chunks
    |   +-- Top chunk: "Pre-payment: No penalty for floating rate loans.
    |       2% penalty for fixed rate loans if paid within 3 years."
    |
    |-- SLM Generate: match "prepayment" category -> Template response
    |
    +-- Final Response + Source tag + Disclaimer
```

7. File Reference

File	Lines	Purpose	Key Classes/Functions
`src/app.py`	~140	Main Streamlit application	`load_resources()`, `get_best_match()`,
`src/rag_engine.py`	~143	RAG pipeline	`RAGEngine`: `create_vector_store()`, `
`src/model_engine.py`	~280	SLM with guardrails	`SLMEngine`: `generate_response()`, `_
`src/create_dataset.py`	~130	Dataset generation	`generate_bfsi_dataset()`
`src/train_slm.py`	~95	Optional fine-tuning	`train()` with LoRA/PEFT
`data/policy_document.txt`	31	RAG knowledge base	Home Loan, Credit Card, Grievance, Di
`requirements.txt`	5	Dependencies	sentence-transformers, numpy, streamli

8. Configuration & Thresholds

Parameter	Value	Location	Purpose
Similarity threshold	0.70	`app.py`	Min cosine score for dataset match
RAG chunk size	400 chars	`rag_engine.py`	Document splitting unit
RAG chunk overlap	80 chars	`rag_engine.py`	Overlap between chunks
RAG top-k	2	`rag_engine.py`	Number of chunks retrieved
RAG min score	0.20	`rag_engine.py`	Min cosine score for chunk inclusion
Embedding model	all-MiniLM-L6-v2	Multiple	384-dim, ~80MB
Embedding dimensions	384	Automatic	Vector size

9. Scalability & Maintenance

9.1 Adding New Query Categories

* Add a new entry to RESPONSE_TEMPLATES in model_engine.py
* Define keywords list and response string
* No retraining or redeployment needed

9.2 Expanding the Dataset

* Add entries to create_dataset.py or directly edit the JSON
* Run python src/create_dataset.py to regenerate
* Restart the Streamlit app to reload

9.3 Adding Policy Documents

* Place .txt or .pdf files in data/
* Run python src/rag_engine.py to rebuild the vector store

* Restart the app

9.4 Upgrading to a Real LLM (Future)

When hardware permits:

* Install torch, transformers, peft
* Fine-tune using python src/train_slm.py
* Update model_engine.py to load the fine-tuned model
* The SLMEngine API (load_model(), generate_response()) remains the same

Document Version: 1.0 | Last Updated: February 2026