AI-Powered Banking Customer Support Chatbot

An Agentic RAG System for Automated Customer Service

Surafel Sentayehu - GSR/5928/17

November 17, 2025

Abstract

This report presents the design and implementation of an AI-powered banking customer support chatbot system that leverages modern artificial intelligence techniques to provide automated customer service. The system employs an Agentic Retrieval Augmented Generation (RAG) architecture, combining large language models (LLMs), vector embeddings, and tool-calling capabilities to handle banking queries. Built using Next.js 16, Google Gemini 2.5 Flash, and Supabase PostgreSQL with pgvector extension, the chatbot demonstrates the practical application of conversational AI in the financial services sector.

Contents

1 Introduction

1.1 Background and Motivation

The banking industry faces increasing pressure to provide instant, accurate customer support while managing operational costs. Traditional call centers struggle with high call volumes, long wait times, and inconsistent service quality [1].

According to recent industry statistics, customer service inquiries account for a significant portion of operational expenses, with average handle times ranging from 5-10 minutes per interaction [2]. Banking customers require immediate assistance for various needs including account inquiries, card management, transaction disputes, and general banking questions.

Artificial intelligence has emerged as a transformative solution to these challenges. Modern large language models (LLMs) combined with Retrieval Augmented Generation (RAG) techniques enable chatbots to provide human-like conversations while grounding responses in factual information [3]. The integration of agentic capabilities, where AI systems can autonomously decide which tools to use and actions to take, represents the next evolution in conversational AI.

This project implements a comprehensive banking customer support chatbot that demonstrates these advanced AI capabilities in a practical, production-ready system [4].

1.2 Problem Statement

Traditional banking support channels face several limitations:

- Limited availability (typically business hours only)
- Long wait times during peak periods
- Inconsistent information quality depending on agent expertise
- High operational costs for staffing call centers
- Difficulty scaling to handle volume spikes
- Language barriers in diverse customer populations

These limitations result in customer frustration, decreased satisfaction scores, and increased operational expenses for financial institutions.

1.3 Project Objectives

The primary objectives of this project are:

1. Design and implement an AI-powered chatbot capable of handling common banking customer service inquiries

2. Integrate Agentic RAG architecture for accurate, grounded responses
3. Implement tool-calling capabilities for actual banking operations
4. Provide seamless escalation to human agents when necessary
5. Demonstrate scalability for production deployment

1.4 Project Scope

This implementation covers:

In Scope:

- Natural language understanding and conversation management
- Vector-based semantic search for knowledge retrieval
- Multiple banking operation tools (card management, balance checking, location finding)
- Intent detection and entity extraction
- Human handoff system
- Real-time streaming responses

Out of Scope:

- Integration with actual banking core systems
- Production-level authentication and authorization
- Multi-language support (English only in current version)
- Voice interface capabilities
- Mobile application development

2 System Architecture

2.1 High-Level Architecture

The system follows a modern three-tier architecture consisting of presentation layer, application layer, and data layer. The architecture is designed for scalability, performance, maintainability, and extensibility.

The main components include:

- **Client Layer**: Next.js frontend with React components
- **API Layer**: Next.js API routes and business logic
- **AI Services**: LLM, embeddings, and RAG pipeline
- **Data Layer**: PostgreSQL database with vector extensions

2.2 Component Overview

2.2.1 Presentation Layer

The frontend is built using Next.js 16 [5] with React 19.2 and TypeScript 5. Key components include:

- **ChatInterface**: Main conversation component with message history
- **MessageBubble**: Individual message rendering with role-based styling
- **TypingIndicator**: Real-time loading states during AI processing
- **QuickReplies**: Suggested actions for common queries

The UI leverages Tailwind CSS v4 for responsive, modern styling and shadcn/ui for accessible component primitives.

2.2.2 Application Layer

The backend consists of Next.js API routes implementing:

- **Chat Endpoint** (/api/chat): Main conversation handler
- **Authentication Routes** (/api/auth): Security verification
- **Handoff System** (/api/handoff): Human agent escalation
- **Knowledge Management** (/api/knowledge): Document ingestion

Core business logic modules include:

- **Intent Detector**: Pattern-based classification of user queries
- **RAG System**: Knowledge retrieval and context formatting
- **Tool Manager**: Banking operations and action execution
- **Security Handler**: Authentication and authorization logic

2.2.3 Data Layer

Data persistence is managed through Supabase [6], providing:

- **PostgreSQL Database**: Relational data storage
- **pgvector Extension**: Vector similarity search
- **Real-time Subscriptions**: Live data updates
- **Row Level Security**: Fine-grained access control

2.3 Technology Stack

Table 1 summarizes the complete technology stack with justifications.

Table 1: Technology Stack Components

Component	Technology	Justification
Frontend Framework	Next.js 16	Server-side rendering
UI Library	React 19.2	Component reusability
Type Safety	TypeScript 5	Error prevention
Styling	Tailwind CSS v4	Responsive design
AI Framework	Vercel AI SDK v5	Streaming support
LLM	Google Gemini 2.5 Flash	Fast and accurate
Embeddings	text-embedding-004	Semantic search
Database	Supabase PostgreSQL	Scalable
Vector Store	pgvector	Similarity search
Deployment	Vercel	Auto-scaling

2.4 Data Flow

The complete data flow for a typical user interaction follows these steps:

1. User sends message through chat interface
2. Frontend transmits message to /api/chat endpoint
3. Backend receives and validates request
4. Intent detector analyzes user message
5. LLM processes message with conversation context
6. If needed, RAG system retrieves relevant documents
7. If needed, appropriate tools are called
8. LLM generates response using retrieved context and tool results
9. Response streams back to frontend word-by-word
10. Frontend renders response in chat interface
11. Conversation logged to database for analytics

3 Agentic RAG System Design

3.1 RAG Fundamentals

Retrieval Augmented Generation addresses the fundamental limitation of LLMs: they can only use information from their training data. RAG enables models to access external knowledge bases, providing:

- **Factual Grounding**: Responses based on actual documents
- **Up-to-date Information**: Knowledge base can be updated without retraining
- **Source Attribution**: Ability to cite where information came from
- **Reduced Hallucination**: Less likely to generate false information

3.2 What Makes It Agentic

Traditional RAG systems retrieve information for every query. Our agentic implementation empowers the AI to make autonomous decisions about when and how to retrieve information. The model acts as an agent that can:

- Decide whether knowledge retrieval is necessary
- Choose which category of documents to search
- Determine optimal search parameters
- Combine multiple information sources
- Select and sequence tool calls

This autonomy enables more efficient and contextually appropriate responses.

3.3 Vector Embeddings

Vector embeddings transform text into numerical representations that capture semantic meaning. Our implementation uses Google's text-embedding-004 model [7], which generates 768-dimensional vectors.

The embedding process:

Listing 1: Embedding Generation Process

```
Input Text: "What are your branch hours?"
         |
         v
Embedding Model (text-embedding-004)
         |
         v
Output Vector: [0.21, 0.42, 0.15, ..., 0.88]
                 (768 numbers total)
```

Similar concepts produce similar vectors, enabling semantic search. For example:

- "What are your hours?" produces [0.21, 0.42, ..., 0.88]
- "When are you open?" produces [0.22, 0.41, ..., 0.87]

Despite different wording, these vectors are mathematically close, allowing the system to recognize them as semantically equivalent queries.

3.4 Knowledge Base Structure

The knowledge base is organized into four categories:

1. **Policies**: Account terms, security policies, compliance rules
2. **Products**: Credit cards, loans, savings accounts, investment options
3. **FAQs**: Common questions and answers
4. **Procedures**: Step-by-step guides for banking operations

Each document is stored with metadata including category, source, tags, and last update timestamp.

3.5 Document Ingestion Pipeline

The knowledge ingestion process follows these steps:

1. Read markdown files from data/knowledge/ directory
2. Parse and validate document structure
3. Generate embedding vector using text-embedding-004
4. Store in Supabase with metadata
5. Create vector index for efficient similarity search

3.6 Retrieval Process

When a user query requires knowledge retrieval, the system executes:

1. **Query Embedding**: Generate vector for user question
2. **Similarity Search**: Find documents with cosine similarity greater than 0.7
3. **Re-ranking**: Apply additional scoring based on keyword matching
4. **Context Formatting**: Prepare retrieved documents for LLM
5. **Response Generation**: LLM generates answer using context

3.7 Vector Similarity Search

The system uses cosine similarity to measure vector closeness:

$$\text{similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{1}$$

Where A is the query embedding and B is a document embedding. Results with similarity greater than 0.7 are considered relevant.

3.8 Re-ranking Strategy

After initial vector search, documents undergo re-ranking to improve relevance:

$$\text{final\_score} = \text{similarity\_score} + (\text{keyword\_matches} \times 0.01) \tag{2}$$

This hybrid approach combines semantic similarity with keyword matching for optimal results.

3.9 Database Schema for RAG

The documents table schema:

Listing 2: Documents Table Schema

```sql
CREATE TABLE documents (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  content TEXT NOT NULL,
  embedding vector(768),
  category VARCHAR(50),
  metadata JSONB,
  source VARCHAR(255),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_documents_embedding
  ON documents USING ivfflat (embedding vector_cosine_ops);
```

The ivfflat index enables efficient approximate nearest neighbor search for large vector datasets.

4 Tool-Based Agentic System

4.1 Tool Calling Mechanism

Tool calling enables the LLM to execute functions rather than just generating text. The Vercel AI SDK [8] provides a structured interface for defining tools with parameters and execution logic.

Each tool consists of:

- **Description**: Explains when the AI should use this tool
- **Parameters**: Defines required and optional inputs with validation
- **Execute Function**: Contains the actual implementation logic

4.2 Implemented Banking Tools

4.2.1 searchKnowledgeBase Tool

Searches the vector database for relevant banking information.

Parameters:

- query (string, required): Search terms or user question
- category (enum, optional): Filter by policy/product/faq/procedure

Returns:

- success (boolean): Whether search succeeded
- context (string): Formatted relevant documents
- sourceCount (number): How many sources were used

4.2.2 cardManagement Tool

Performs operations on credit/debit cards.

Parameters:

- action (enum, required): freeze/unfreeze/report_lost/order_replacement
- cardLast4 (string, optional): Last 4 digits of card
- reason (string, optional): Reason for action

Returns:

- success (boolean): Operation result
- message (string): Confirmation message
- timestamp (string): When operation occurred

4.2.3 checkBalance Tool

Retrieves account balance information.

Parameters:

- accountType (enum, required): checking/savings/credit

Returns:

- currentBalance (number): Total balance
- availableBalance (number): Spendable amount
- pending (number): Transactions not yet cleared
- currency (string): Currency code (USD)

4.2.4 findLocation Tool

Locates nearby bank branches or ATMs.

Parameters:

- locationType (enum, required): branch/atm/both
- location (string, optional): City or zip code
- limit (number, optional): Maximum results to return

Returns:

- locations (array): List of nearby locations with details
- Each location includes: name, address, distance, hours

4.2.5 requestHumanAgent Tool

Escalates conversation to a human agent.

Parameters:

- reason (string, required): Why escalation is needed
- priority (enum, required): low/medium/high/urgent

Returns:

- queuePosition (number): Position in support queue
- estimatedWait (string): Expected wait time
- ticketId (string): Support ticket identifier

4.3 Tool Orchestration

The AI model can call multiple tools in sequence to complete complex tasks. For example, when a user reports a lost card:

1. searchKnowledgeBase(query: ”lost card policy”)
2. cardManagement(action: ”freeze”, reason: ”lost”)
3. Generate response combining policy info and freeze confirmation

This orchestration happens autonomously based on the model’s understanding of the situation.

4.4 Tool Implementation Pattern

Tools follow a consistent implementation pattern using Zod for schema validation:

Listing 3: Tool Implementation Pattern

```
import { tool } from 'ai';
import { z } from 'zod';

export const exampleTool = tool({
  description: 'Clear description for AI',

  parameters: z.object({
    param1: z.string().describe('Parameter description'),
    param2: z.number().optional()
  }),

  execute: async ({ param1, param2 }) => {
    // Implementation logic
    const result = await performAction(param1, param2);

    return {
      success: true,
      data: result
    };
  }
});
```

This pattern ensures type safety, parameter validation, and consistent error handling across all tools.

5 Intent Detection System

5.1 Intent Categories

The system recognizes the following intent categories:

- BALANCE_INQUIRY: Account balance checks
- BRANCH_LOCATOR: Finding physical locations
- CARD_LOST: Reporting lost or stolen cards
- CARD_MANAGEMENT: Card operations (freeze, unfreeze)
- TRANSACTION_INQUIRY: Transaction history and details
- FAQ: General banking questions
- HUMAN_HANDOFF: Requests for human assistance
- UNKNOWN: Unrecognized intents

5.2 Pattern-Based Detection

Intent detection uses regular expression pattern matching for speed and reliability. Each intent has associated patterns with priority levels:

Listing 4: Intent Pattern Example

```
{
  intent: 'CARD_LOST',
  patterns: [
    /lost.*card/i,
    /card.*stolen/i,
    /missing.*card/i,
    /can't find.*card/i
  ],
  priority: 1
}
```

Patterns are checked in priority order, with the first match winning. Case-insensitive matching handles variations in capitalization.

5.3 Entity Extraction

Beyond intent classification, the system extracts entities from user messages:

- Card types (credit, debit, prepaid)
- Account types (checking, savings)
- Amounts ($500, 500 dollars)
- Dates and times (yesterday, last week)
- Locations (New York, 10001)

Entities provide context for tool parameter population.

5.4 Confidence Scoring

Each intent detection includes a confidence score:

- Pattern match: 0.9 confidence
- Multiple pattern matches: 0.95 confidence
- Weak signals: 0.5 to 0.7 confidence
- No match: 0.0 confidence (UNKNOWN intent)

Low confidence scores may trigger clarification questions.

6 Implementation Details

6.1 Project Structure

The codebase follows Next.js App Router conventions:

6.2 Main Chat Handler

The core chat endpoint implementation:

Listing 5: Chat API Handler

```
export async function POST(req: Request) {
  const { messages } = await req.json();

  // Intent detection
  const intent = await intentDetector.detect(
    lastUserMessage
  );

  // Stream response with tools
  const result = streamText({
    model: geminiModel,
    messages: modelMessages,
    system: BANKING_SYSTEM_PROMPT,
    tools: bankingTools,
    temperature: 0.3,
    maxRetries: 2,
    stopWhen: stepCountIs(5),

    onFinish: async ({ text, toolCalls }) => {
      // Log conversation
      await logMessage({
        intent: intent.type,
        toolsUsed: toolCalls
```

```
    });
  }
});

return result.toUIMessageStreamResponse();
}
```

6.3 RAG Retrieval Implementation

The knowledge retrieval system:

Listing 6: RAG Retriever

```
class KnowledgeRetriever {
  async retrieve(query, options) {
    // Generate query embedding
    const embedding = await generateEmbedding(query);

    // Search vector store
    const documents = await searchSimilarDocuments(
      embedding,
      options.threshold,
      options.topK
    );

    // Re-rank results
    const reranked = this.rerank(documents, query);

    return reranked;
  }

  formatContext(documents) {
    return documents.map((doc, index) =>
      `[Document ${index + 1}]
       Source: ${doc.metadata.source}
       ${doc.content}
       ---`
    ).join('\n\n');
  }
}
```

6.4 Vector Store Operations

Database interaction for similarity search:

Listing 7: Vector Search

```
async function searchSimilarDocuments(query, options) {
  // Generate embedding
  const queryEmbedding = await generateEmbedding(query);
```

```

  // Call database function
  const { data, error } = await supabase.rpc(
    'match_documents',
    {
      query_embedding: queryEmbedding,
      match_threshold: options.threshold,
      match_count: options.topK,
      category_filter: options.category
    }
  );

  return data || [];
}
```

The match_documents PostgreSQL function:

Listing 8: Vector Similarity Function

```sql
CREATE FUNCTION match_documents(
  query_embedding vector(768),
  match_threshold float,
  match_count int,
  category_filter varchar DEFAULT NULL
)
RETURNS TABLE (
  id UUID,
  content TEXT,
  metadata JSONB,
  similarity float
)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY
  SELECT
    d.id,
    d.content,
    d.metadata,
    1 - (d.embedding <=> query_embedding) as similarity
  FROM documents d
  WHERE
    (1 - (d.embedding <=> query_embedding)) > match_threshold
    AND (category_filter IS NULL
         OR d.category = category_filter)
  ORDER BY d.embedding <=> query_embedding
  LIMIT match_count;
END;
$$;
```

The cosine distance operator enables efficient similarity calculation.

A Sample Conversations

A.1 Lost Card Scenario

Listing 9: Lost Card Interaction

```
User: I think I lost my credit card at the mall

Bot: I understand you've lost your credit card at the mall.
     This must be concerning. Let me help you secure your
     account immediately.

     For your protection, I can:
     1. Freeze your card instantly
     2. Help you order a replacement card

     Would you like me to freeze your card now?

User: Yes please

Bot: Thank you for verifying your identity.

     Your credit card ending in 1234 has been frozen.

     Your card is now secure:
     - No transactions can be made
     - You can unfreeze it if you find it
     - Virtual card available for immediate use

     Would you like to order a replacement card?
```

A.2 Balance Inquiry

Listing 10: Balance Check Interaction

```
User: What's my checking account balance?

Bot: I'd be happy to check your account balance.
     For security, I need to verify your identity first.

     [Authentication Required]

Bot: Your checking account balance:
     - Current Balance: $2,547.89
     - Available Balance: $2,147.89
     - Pending: $400.00

     Would you like to see recent transactions?
```

B Database Schema

B.1 Complete Schema

Listing 11: Full Database Schema

```sql
-- Enable extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS vector;

-- Users table
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  email VARCHAR(255) UNIQUE,
  phone VARCHAR(20),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Conversations table
CREATE TABLE conversations (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID REFERENCES users(id),
  session_id VARCHAR(255),
  status VARCHAR(50) DEFAULT 'active',
  started_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  ended_at TIMESTAMP,
  metadata JSONB
);

-- Messages table
CREATE TABLE messages (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  conversation_id UUID REFERENCES conversations(id),
  role VARCHAR(20) CHECK (role IN ('user', 'assistant')),
  content TEXT,
  intent VARCHAR(50),
  confidence FLOAT,
  tools_used JSONB,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Documents table
CREATE TABLE documents (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  content TEXT NOT NULL,
  embedding vector(768),
  category VARCHAR(50),
  metadata JSONB,
  source VARCHAR(255),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```sql

-- Create indexes
CREATE INDEX idx_conversations_user_id
  ON conversations(user_id);
CREATE INDEX idx_messages_conversation_id
  ON messages(conversation_id);
CREATE INDEX idx_documents_category
  ON documents(category);
CREATE INDEX idx_documents_embedding
  ON documents USING ivfflat (embedding vector_cosine_ops);
```

C Environment Configuration

Listing 12: Environment Variables

```
# Google AI API
GOOGLE_GENERATIVE_AI_API_KEY=your_key_here

# Supabase
NEXT_PUBLIC_SUPABASE_URL=your_project_url
NEXT_PUBLIC_SUPABASE_ANON_KEY=your_anon_key
SUPABASE_SERVICE_KEY=your_service_key
```

References

[1] Zericho R. Marak, Subhajit Pahari, Rishabh Shekhar, and Ashish Tiwari. Factors affecting chatbots in banking services: the utaut2 and innovation resistance theory perspective. *Journal of Innovation and Entrepreneurship*, 14(47), 2025.

[2] Shahedul Hasan, Eshatir Radiat Godhuli, Md Shezanur Rahman, and Md Abdullah Al Mamun. The adoption of conversational assistants in the banking industry: is the perceived risk a moderator? *Heliyon*, 9(9):e20220, 2023.

[3] Yinheng Nie, Yue Kong, Xingli Dong, John M. Mulvey, H. Vincent Poor, Qingsong Wen, and Stefan Zohren. A survey of large language models for financial applications: Progress, prospects and challenges. *arXiv preprint arXiv:2406.11903*, 2024.

[4] Ricardo Ramos, José Casaca, and Rita Patrício. Adoption drivers of intelligent virtual assistants in banking: Rethinking the artificial intelligence banker. *Computers*, 14(6):209, 2025.

[5] Vercel. Next.js documentation. https://nextjs.org/docs, 2024. Accessed: 2025-01-17.

[6] Supabase. Supabase documentation. https://supabase.com/docs, 2024. Accessed: 2025-01-17.

[7] Google. Google gemini api documentation. https://ai.google.dev/docs, 2024. Accessed: 2025-01-17.

[8] Vercel. Vercel ai sdk documentation. https://sdk.vercel.ai/docs, 2024. Accessed: 2025-01-17.