ZenBot- An AI Chatbot

A Policy-Aware, Tool-Augmented Generative Agent
Senior ML Scientist Challenge Project Report

Overview

This solution implements a **modular and scalable customer support system** using LLM-driven reasoning, tool integration, and retrieval-augmented generation (RAG) with a vector DB.

It balances natural language flexibility with the structure of policy enforcement and real-time tool interaction.

Problem Scoping

In modern customer service, users expect real-time responses, personalized support, and strict adherence to company policies. This project aims to build an LLM-powered agentic chatbot that:

- Handles user queries such as order tracking, cancellation, returns, refunds, and password resets.
- Follows strict business rules (e.g., "Orders can't be cancelled after 10 days").
- Calls backend APIs/tools when needed.
- Ensures grounded, non-hallucinated responses through policy and memory integration.
- Supports multi-user conversational flow.

Key Assumptions

To focus on the agentic orchestration and evaluation, I made the following controlled assumptions:

- The chatbot operates in a simulated environment with mock tools and in-memory orders.
- Users are limited to: Joe, Magda, Shashi, Mahima with pre-defined orders.
- Orders include fields like status, date, user_id, order_id etc.
- Policies are static text files embedded into a vector database.
- Tools are Python functions that validate conditions and return structured outputs.
- No external database or persistent memory — designed as a functional proof of concept.
- LLM responses are grounded in retrieved policy text and tool results.

Design Strategy

Agentic Design:
Adopted an agent loop that lets the LLM decide which action to take (e.g., call tool, retrieve policy, respond directly) and reason about the outcome. This gives the system flexibility and reasoning traceability.

Tool Integration:
Tools were treated as **external functions with access controls** (e.g., order status, cancellation window). This ensures real-world constraints are enforced programmatically and not left to the LLM alone.

Policy Grounding via Vector DB:
To simulate real-world documentation, company policies were embedded and stored in a vector database. The chatbot retrieves the most relevant policies using semantic similarity, ensuring that responses remain grounded in actual rules.

Memory Module:
A simple in-memory store retains user state and previous order context across turns, mimicking session continuity without full user authentication or database dependency.

LLM Reasoning Flow:
Used the LLM to:

- Classify user intent
- Select and call the correct tool
- Retrieve and interpret policies
- Generate grounded, helpful responses from structured data

Evaluation Design:
Created a test suite with diverse user intents. Responses were evaluated on:

- Intent classification accuracy
- Correct tool invocation
- Semantic similarity (via CrossEncoder) to ideal answers
- Policy match confidence
- Overall pass/fail per interaction

Technologies Used

Language Model

- Ollama – Local model runner
- **Mistral 7B** – Lightweight open-source LLM used for generating natural responses
- Sentence Transformers – For semantic similarity and crossencoder-based evaluation

Backend & Agent Logic

- **Python 3.10+**
- **FastAPI** – High-performance API framework for handling chat requests
- **Langchain-inspired agent design** – LLM + tools + memory
- **Custom tools** – CancelOrder, TrackOrder, ReturnOrder, PasswordReset
- **In-memory user state management** – Custom memory store for context tracking

Policy Retrieval

- **FAISS** – Vector database to store and retrieve company policies
- **OpenAI embedding model / all-MiniLM-L6** – Used for converting policies into vectors
- **Policy confidence scoring** – Based on cosine similarity of embeddings

Evaluation Framework

- **JSON-based test cases** – Structured inputs, expected outputs
- **CrossEncoder (semantic scoring)** – Fine-grained LLM evaluation with thresholding
- **Matplotlib / pandas** – For plots and metrics analysis
- **Jupyter Notebook** – For reporting and visualization

Frontend UI

- **Gradio** – Interactive chatbot UI with user selection
- **REST API** – Communication between UI and backend agent

Others

- **Requests** – To handle API calls from frontend
- **Pydantic** – For request validation in FastAPI
- **Tabulate** – Pretty-printed metrics in CLI/notebook

Why this approach?

- It balances **structured rule enforcement** (via tools/policies) with **LLM flexibility and language understanding**.
- The architecture is **modular, testable, and extensible**, enabling deeper experiments or productionization.
- The evaluation framework provides **quantifiable insights** into reasoning quality and response reliability — essential for ML-focused roles.

✅ **Visualizations included bar plots, semantic histograms, and pie charts for clarity.**
📈 **Full evaluation in experiment_analysis.ipynb.**

Author Notes

The core idea behind this chatbot is to showcase how reasoning over retrieval and APIs can be elegantly orchestrated using LLMs. I kept the UI simple to allow maximum focus on architecture and

evaluation.
— Kallya Mahima Rao