Project Report: Advanced AI Troubleshooting Chatbot

Project: Intelligent AI Assistant for IC Engine Diagnostics
Author: Rahul G
Date: July 2, 2025
Status: Completed Prototype

1. Objectives

The primary objective of this project was to develop a sophisticated, interactive AI assistant to streamline the diagnostic process for engineers working with Internal Combustion (IC) engine test beds. The specific goals were to:

- **Accelerate Diagnosis:** Significantly reduce the time required to identify potential root causes of engine failures by providing immediate, data-driven analysis and recommendations.
- **Standardize Data Collection:** Implement an automated, conversational process to capture structured and consistent information about each issue, replacing inconsistent manual data entry.
- **Leverage Historical Knowledge:** Build a system capable of analyzing a database of past issues to identify trends and provide accurate, context-aware solutions for new problems.
- **Enhance Usability:** Create a user-friendly, command-line interface that is more intuitive and efficient for engineers than navigating complex databases or manuals.
- **Ensure System Robustness:** Engineer a reliable application with comprehensive logging, state management, and resilient error handling to ensure consistent performance.

2. Design and Architecture

The chatbot was designed with a modular architecture to ensure scalability, maintainability, and a clear separation of concerns.

Core Architectural Components:

1. **Conversational AI Core (OpenAI GPT-4 Turbo):** At the heart of the system, this component manages natural language understanding (NLU) and generation (NLG). It drives the conversational flow, asks targeted clarifying questions, and performs the initial extraction of the issue profile into a structured JSON format.
2. **State Management (ChatbotState Class):** A dedicated class was designed to manage the context and memory of each user session. It encapsulates the conversation history, the extracted user profile, a unique session ID, and the confirmation status, thereby avoiding global variables and enabling clean state

transitions.

3. **Problem Mapping Engine (TF-IDF & Weighted Scoring):** To provide accurate recommendations, a sophisticated mapping engine was developed. It uses Scikit-learn's TfidfVectorizer to perform a semantic similarity search between the user's current issue and the historical database. This is enhanced with a weighted scoring system that boosts the relevance of critical matching data points, such as error codes.
4. **Historical Data Store (CSV Database):** A local CSV file (historical_issues.csv) serves as the system's knowledge base. A synthetic data generation module was created to produce a large, realistic dataset for robust testing and demonstration purposes.
5. **Logging and Auditing System:** A comprehensive logging framework using Python's standard logging module was integrated. It records all system activities and errors to chatbot_activity.log for debugging, while also saving full conversation transcripts in JSONL format to conversations.jsonl for auditing and potential model fine-tuning.
6. **User Command Interface:** A simple but powerful command-line interface (e.g., !help, !new, !review, !export) provides users with direct control over the application's state and functionality.

Architectural Flow:
The application follows a structured, logical flow for each user interaction:
User Input ⟶ Moderation Check ⟶ Command Handling ⟶ AI Conversation (GPT) ⟶ Profile Extraction ⟶ AI Self-Validation ⟶ TF-IDF Mapping ⟶ Recommendation Generation ⟶ Display/Export Report

3. Implementation

The project was implemented in Python 3, leveraging several key libraries to achieve its functionality:

- **openai:** For all interactions with the GPT and Moderation APIs.
- **pandas:** For creating, managing, and querying the historical issue database from the CSV file.
- **scikit-learn:** Used for TfidfVectorizer and cosine_similarity to power the advanced problem mapping engine.
- **tenacity:** Integrated to provide robust, automatic retry logic with exponential backoff for all external API calls, making the system resilient to transient network failures.
- **Standard Libraries:** os, json, re, ast, logging, and uuid were used for file operations, data parsing, regular expressions, logging, and generating unique

identifiers.

Key Implementation Highlights:

- **Advanced Prompt Engineering:** A highly detailed system prompt was engineered to meticulously guide the LLM's behavior. It defines the AI's persona, its primary objective, its conversational strategy, and the critical instruction to output a clean JSON object upon completion.
- **AI Self-Validation:** The intent_confirmation_layer function implements a "checker" pattern, where a second, specialized LLM call is made to validate the completeness and correctness of the JSON profile generated by the primary "worker" LLM.
- **TF-IDF for Semantic Matching:** The map_issue_to_solution_tfidf function combines multiple text fields into a single document for both the user's issue and historical issues. It then uses TF-IDF to convert these documents into numerical vectors and calculates their cosine similarity to find the closest semantic match, which is far more powerful than simple keyword searching.

4. Challenges and Lessons Learned

- **Challenge 1: Handling Ambiguous User Input.**
  - **Problem:** Engineers describe technical issues in varied and often imprecise ways.
  - **Solution:** A multi-turn conversational flow was designed. The AI was programmed via its system prompt to ask targeted, clarifying questions, systematically guiding the user toward providing all the necessary information for a complete diagnostic profile.
- **Challenge 2: Ensuring Reliable Structured Data Extraction.**
  - **Problem:** LLMs can occasionally fail to adhere strictly to formatting instructions, leading to parsing errors.
  - **Solution:** A multi-layered validation strategy was implemented. This included using OpenAI's dedicated JSON mode, employing the AI-based intent_confirmation_layer for self-checking, and including a robust local parsing function (dictionary_present) as a fallback.
- **Challenge 3: Achieving Meaningful Solution Mapping.**
  - **Problem:** An early prototype using simple keyword matching was brittle and often missed relevant historical issues if the user's terminology differed slightly.
  - **Solution:** Integrating a TF-IDF vectorization strategy allowed the system to match issues based on their underlying semantic meaning rather than exact keywords. This dramatically improved the accuracy and relevance of the

recommendations provided.

Key Lessons Learned:

- **The Critical Role of Prompt Engineering:** The quality, detail, and clarity of the system prompt are the most significant factors in determining the reliability and success of the LLM's behavior.
- **The Power of Hybrid AI Systems:** The project's success demonstrates the value of combining different AI techniques. The generative, conversational power of an LLM, when paired with the analytical, deterministic power of TF-IDF on a structured database, creates a solution that is more robust and reliable than either technique could achieve alone.
- **The Necessity of State Management:** For any non-trivial conversational application, a dedicated state management class is essential for writing clean, debuggable, and scalable code.
- **Design for Resilience:** Building applications that depend on external services requires designing for failure. Integrating resilience patterns like automatic retries and comprehensive logging from the outset is crucial for creating a production-ready system.