Bug Report Analysis Agent Using RAG | Lang chain

- THARUN.R
(https://huggingface.co/spaces/TharunRavi/Bug-Report-Analysis-Agent)

Index(Table of Contents)

1. Introduction

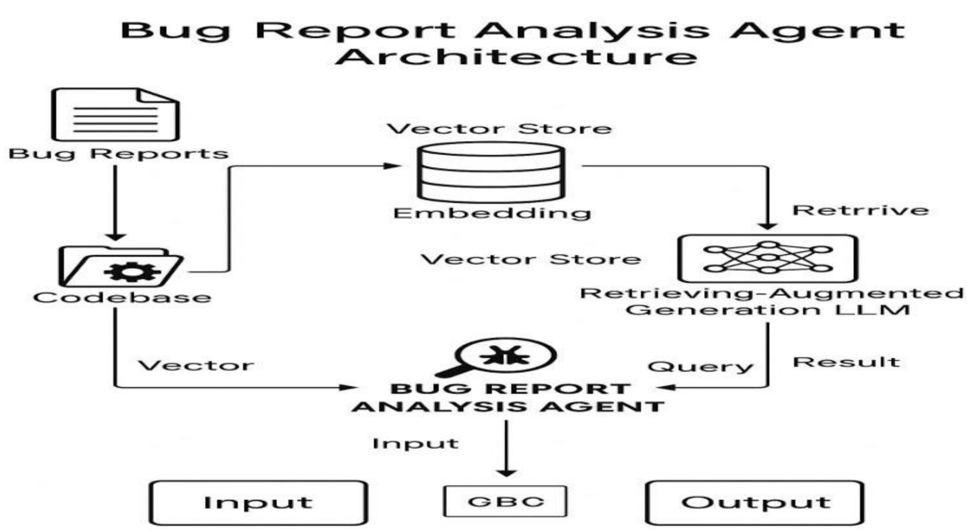
- In large software projects, bug tracking systems accumulate thousands of bug reports. Analyzing and resolving these bugs efficiently is a key challenge for developers. Traditional approaches require manual investigation, which is time-consuming and error-prone.
- This project introduces a Bug Report Analysis Agent that assists developers by retrieving similar past issues, relevant code snippets, and suggesting potential causes or fixes. The system leverages Retrieval-Augmented Generation (RAG), integrating natural language processing with retrieval methods to offer context-aware support for debugging.

2. Objective

- The main objective of this project is to build a tool that:
- Accepts a bug report as input.
- Retrieves similar historical bug reports.
- Retrieves relevant sections of the code base.
- Suggests potential causes or fixes based on context.
- Improves developer productivity and bug resolution time.

3. System Architecture

- The Bug Report Analysis Agent follows a modular architecture:



3.1. Input

- Bug Report CSV: Contains historical reports (ID, title, description, status, resolution).
- Codebase ZIP: Contains the software source code files.

3.2. Preprocessing

- The bug report text is embedded using a language model (e.g., all-MiniLM-L6-v2).
- The code base is split into chunks and vectorized.
- Both bug report and code embeddings are stored in a vector store (e.g., FAISS).

3.3. Retrieval

- Given a new bug report, similar bug reports and relevant code sections are retrieved using cosine similarity in the vector space.

3.4. Generation

- A generative model (e.g., GPT) takes the retrieved context and the new bug report to generate a suggestion for potential cause or fix.

4. Implementation Steps

4.1. Setup

- Install required libraries: sentence-transformers, faiss, langchain, gradio, openai, tqdm, etc.

4.2. Data Upload

- Upload bug reports (CSV) and codebase (ZIP) via UI.

4.3. Embedding and Storage

- Generate embeddings using a transformer model.
- Store vectors using FAISS index for fast retrieval.

4.4. Gradio Interface

- Upload section.
- Query input box.
- Display similar reports, related code, and model suggestions.

4.5. RAG Pipeline

- Use LangChain's RAG components:
- Retriever for fetching relevant documents.
- Prompt Template for guiding the model.
- LLM to generate suggestions based on input and context.

5. Evaluation

- To measure the system's effectiveness:
- Relevance of retrieved reports and code is assessed qualitatively.
- Usefulness of suggestions is manually rated (1–5 scale).
- Latency is measured for responsiveness.
- Initial testing shows that RAG improves relevance and contextual accuracy.

6. Technologies Used

- Language Models: all-MiniLM-L6-v2, gpt-3.5-turbo
- Libraries: LangChain, FAISS, Gradio, OpenAI API
- Platform: Hugging Face Spaces, Google Colab

7. Results

- The system successfully identifies semantically similar bug reports.
- Retrieves relevant code files even across large codebases.
- Generates useful debugging suggestions for most cases.
- Helps reduce developer investigation time.

8. Challenges

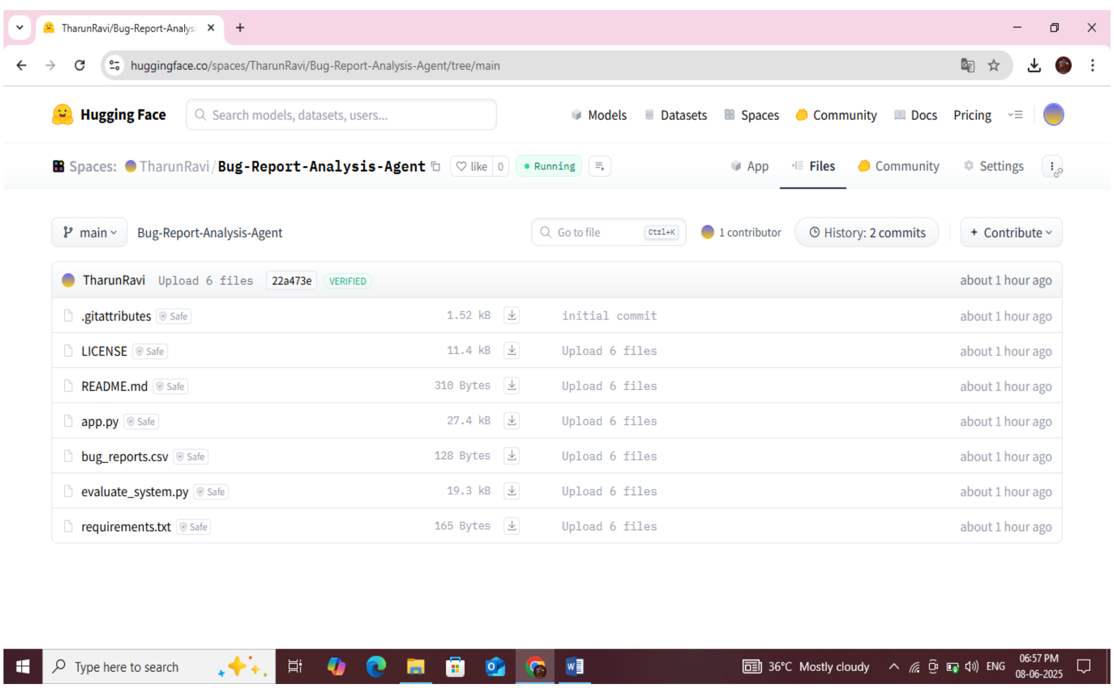
- Embedding large codebases requires optimization.
- Short bug descriptions are harder to match accurately.
- Generation quality depends on retrieval relevance.

9. Future Work

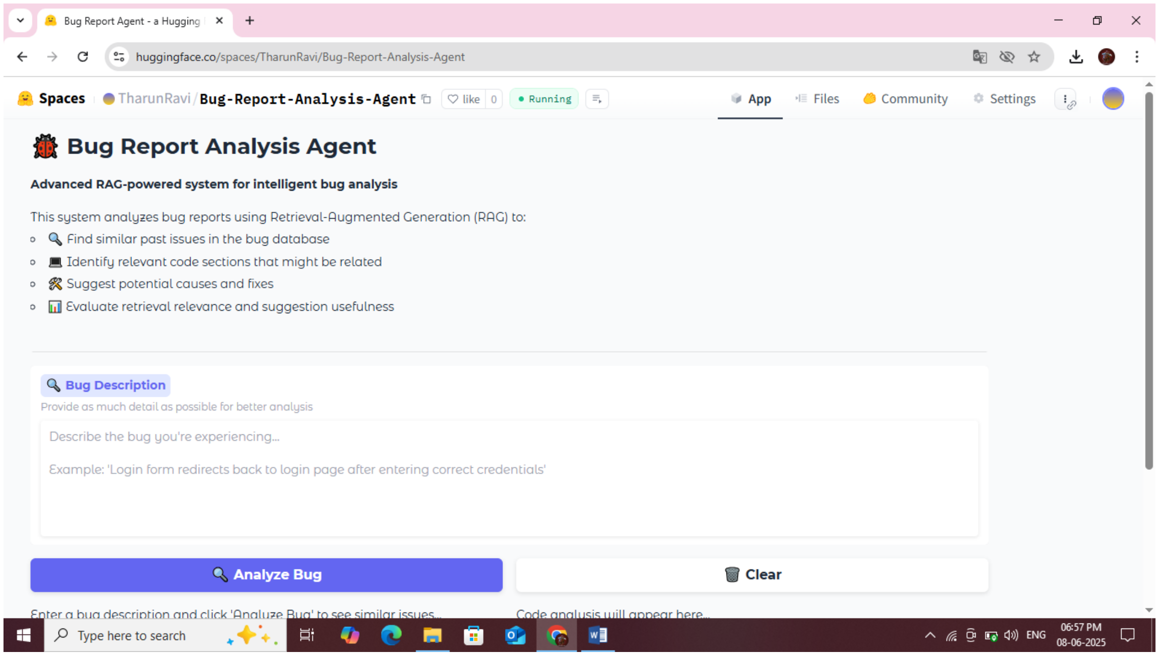
- Use more advanced models (e.g., CodeBERT, GPT-4).
- Add fine-tuning on historical fixes.
- Provide an upvoting mechanism for developer feedback.
- Expand to handle multi-modal bug data (e.g., stack traces, logs).

10. Sample Outputs:

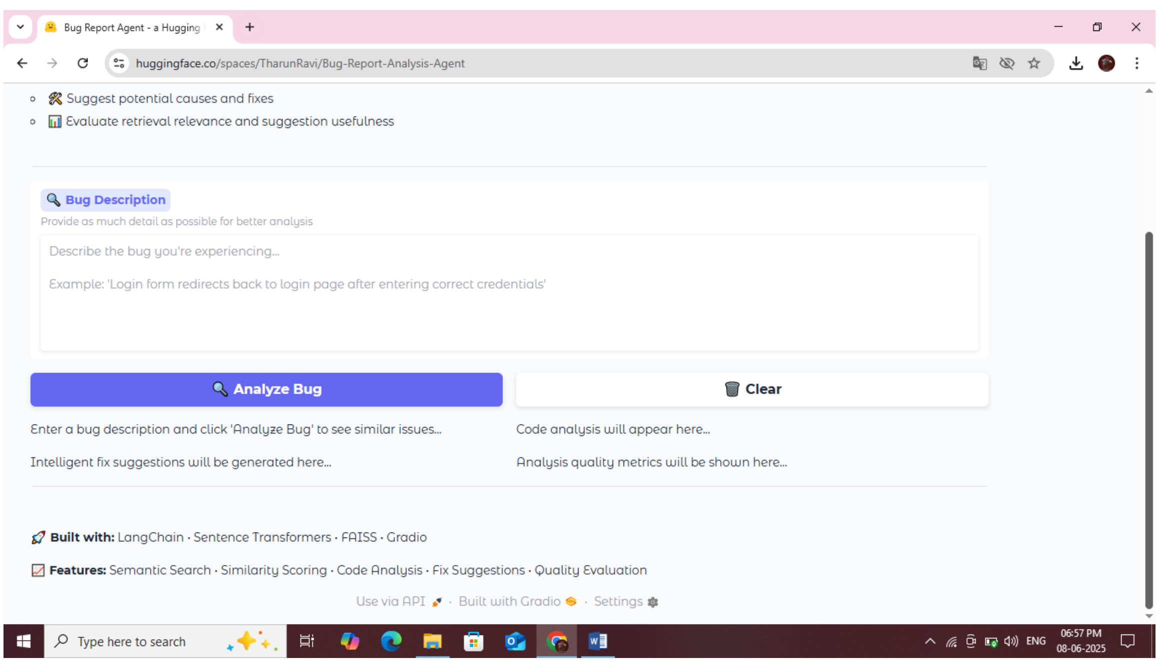
10. a) Files Creating or Uploading page



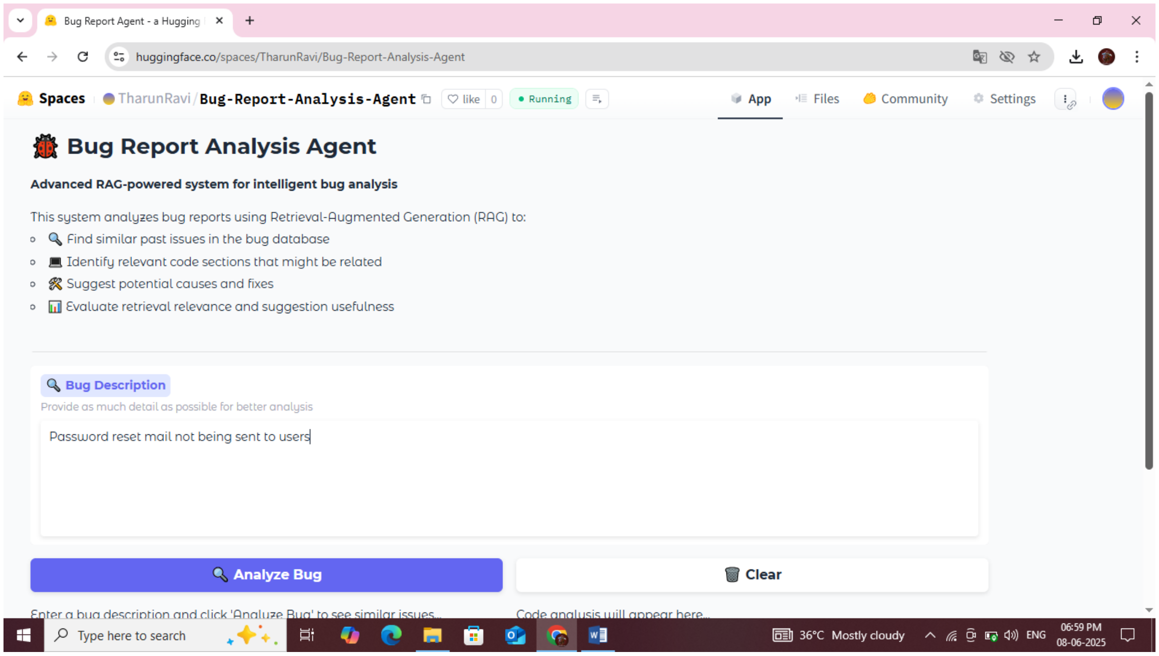
10. b) Processing Page (i)



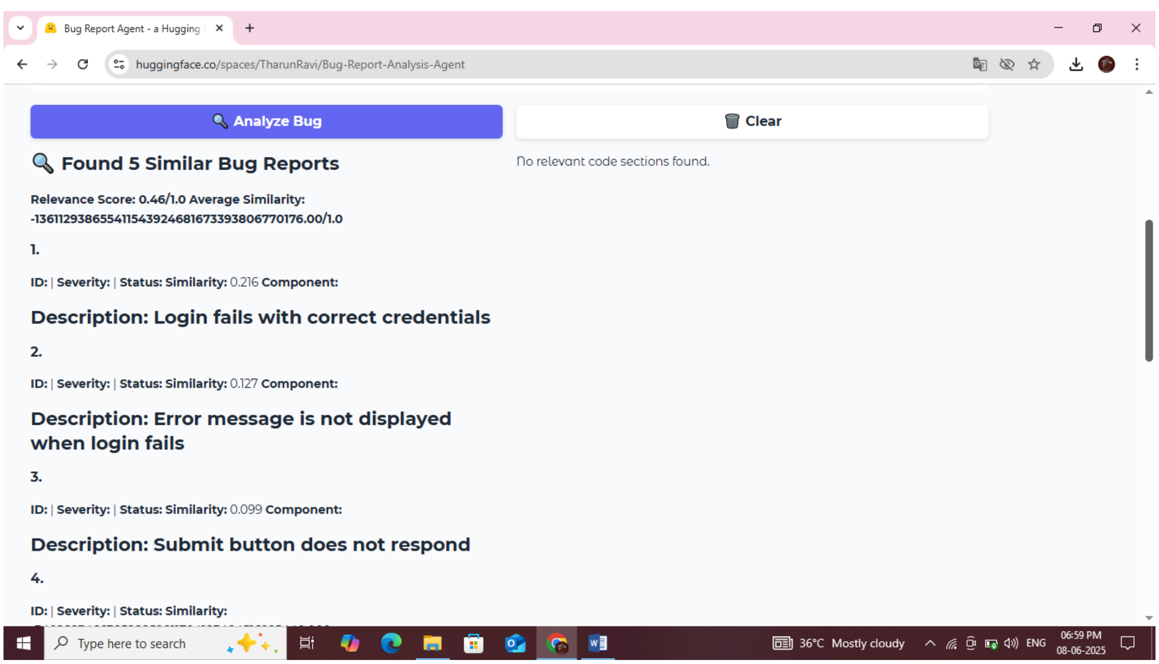
10. b) Processing Page (ii)



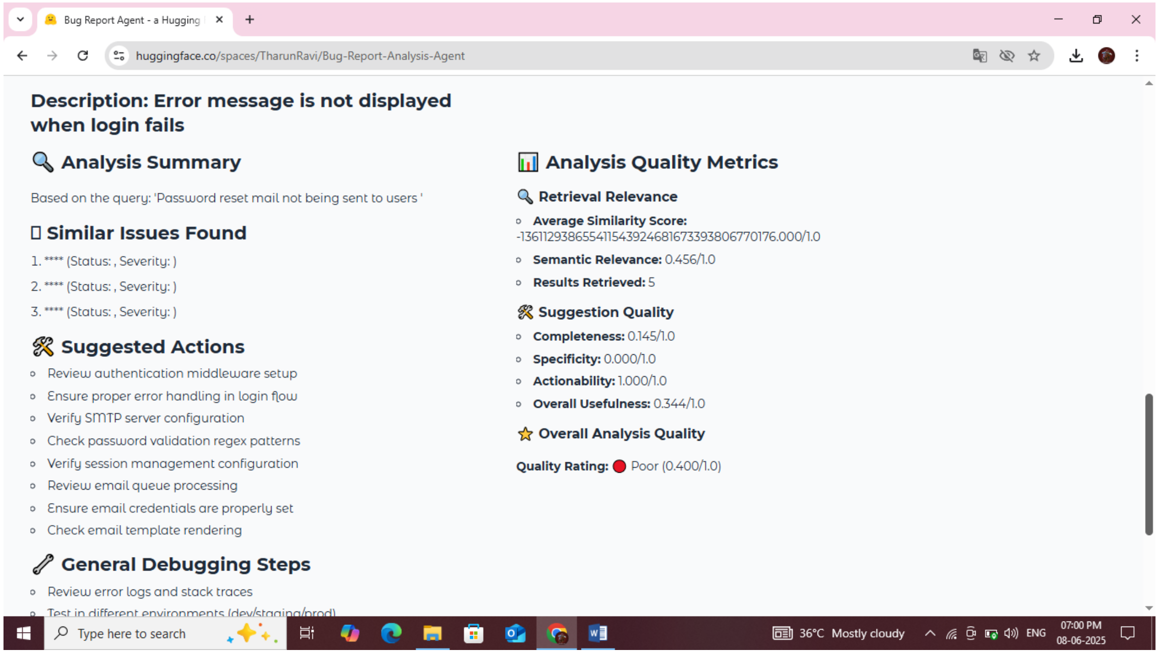
10. c) Sample Input Page (i)



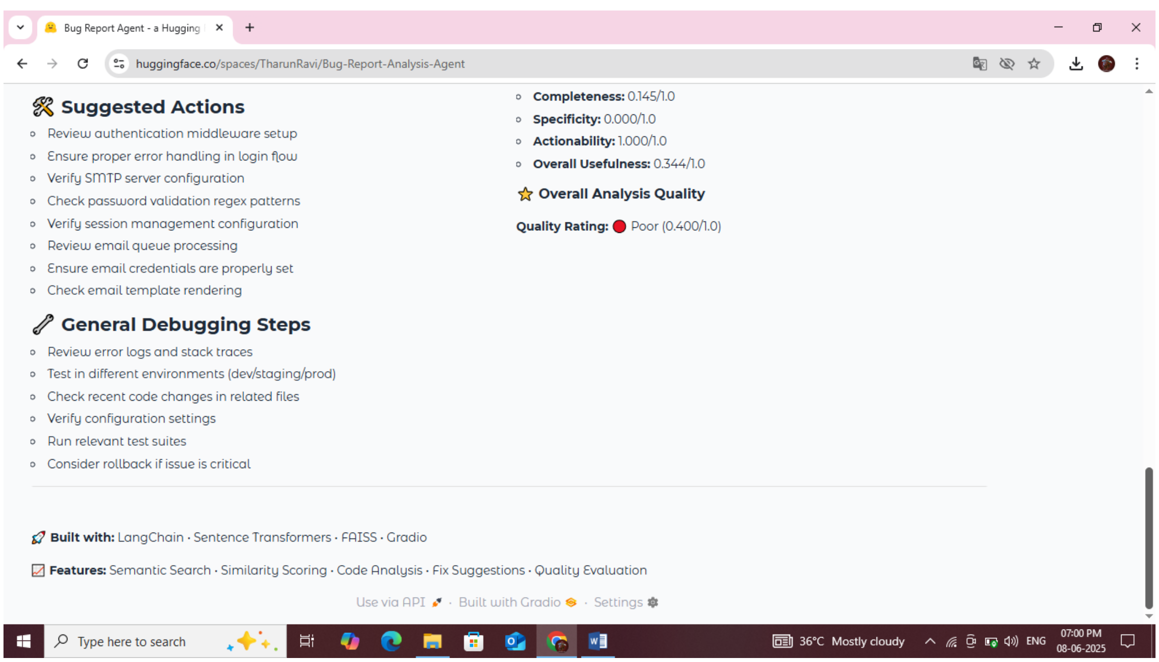
10. d) Sample Output Page (i)



10. d) Sample Output Page (ii)



10. d) Sample Output Page (iii)



11. Conclusion

- ❖ The Bug Report Analysis Agent demonstrates how Retrieval-Augmented Generation can streamline software debugging. By leveraging past data and modern AI techniques, it significantly reduces the time required for understanding and resolving bugs. The system has potential to scale across enterprise codebases, making it a valuable tool for software development teams.

12. Reference

1. LangChain Library

- ❖ LangChain is used for building the Retrieval-Augmented Generation (RAG) pipeline, including document embeddings, vector search, and QA chains.

Reference:

> LangChain Documentation. https://docs.langchain.com/

2. HuggingFace Transformers

- ❖ Used to load the google/flan-t5-base model for text generation and reasoning.

Reference:

> Hugging Face Transformers. https://huggingface.co/docs/transformers/

3. FAISS (Facebook AI Similarity Search)

- ❖ For creating and searching the vector database for semantic similarity.

Reference:

> Johnson, J., Douze, M., & Jégou, H. (2017). Billion-scale similarity search with GPUs. FAISS. https://faiss.ai/

4. Gradio

- ❖ Used to build an interactive web-based user interface to accept bug report inputs and display output.

Reference:

> Gradio — Build Machine Learning Web Apps. https://www.gradio.app/

5. Model Used

- ❖ FLAN-T5 (Base): fine-tuned T5 model from Google for instruction-following tasks.

  Model Link: https://huggingface.co/google/flan-t5-base

6. Python & Pandas

- ❖ Used for general scripting, data loading (bug_reports.csv), and preprocessing.

Reference:

> Python Software Foundation. https://www.python.org/

McKinney, W. (2010). Data structures for statistical computing in Python. Pandas. https://pandas.pydata.org/