Multimodal Code Debug Assistant

Project Report

Author: Karthik Jonnalagadda
Degree: B.Tech
Domain: AI, Full-Stack Development, Generative AI

---

1. Introduction

Modern software development involves complex systems where debugging errors efficiently is a major challenge. Traditional debugging tools rely heavily on textual logs and manual inspection, which becomes difficult when errors are presented visually through screenshots or UI-based error messages.

The **Multimodal Code Debug Assistant** is a practical AI-powered system that combines **image understanding and source code analysis** to automatically identify errors, explain their root causes, and suggest fixes. The project demonstrates how **multimodal AI pipelines** can be integrated into a full-stack application using modern web and AI technologies.

This project serves as:

- A **learning-focused demo** of multimodal AI systems
- A **reference architecture** for AI-driven debugging tools
- A **foundation** for building advanced image-aware debugging assistants

---

2. Objectives

The main objectives of this project are:

- To build a system capable of analyzing **error screenshots and source code together**
- To extract error information using **OCR techniques**

- To apply **LLM-based reasoning** for root cause analysis
- To generate **human-readable explanations and fixes**
- To design a **scalable frontend-backend architecture**
- To optionally persist debugging results using a database

---

3. System Overview

The system follows a multimodal pipeline where both **visual input (screenshots)** and **textual input (code)** are processed together.

High-Level Workflow

1. User uploads an error screenshot and source code via the frontend.
2. The backend extracts text from the image using OCR.
3. A debugging prompt is dynamically constructed.
4. A multimodal LLM analyzes the prompt.
5. The result is returned to the UI and optionally stored in a database.

---

4. Architecture Design

Architecture Flow

Frontend (React)
↓ multipart/form-data
Backend (FastAPI)
↓
OCR Engine (Tesseract)
↓
Prompt Builder
↓

LLM Inference (Ollama / LLaVA)
↓
JSON Analysis Response
↓
User Interface

---

5. Technology Stack

Frontend

- React
- Vite
- TypeScript

Backend

- FastAPI
- Python
- Uvicorn

AI / Machine Learning

- OCR: Tesseract
- LLM: LLaVA via Ollama
- Prompt Engineering

Database

- MongoDB

---

6. Key Features

- Upload error screenshots and source code
- OCR-based extraction of error messages from images
- LLM-powered root cause analysis
- Suggested fixes and debugging explanations
- JSON-based structured response
- Optional persistence of analysis data in MongoDB

---

7. Backend Implementation

The backend is built using FastAPI and handles the complete AI processing pipeline.

Major Responsibilities

- Accept multipart/form-data requests
- Store uploaded images in the filesystem
- Extract text using `pytesseract`
- Build contextual prompts
- Invoke LLaVA via Ollama CLI
- Return analysis results as JSON
- Store metadata in MongoDB (optional)

API Endpoint

- `POST /analyze`
  Returns:

    - Extracted OCR text
    - AI-generated analysis
    - Database record ID (if persistence is enabled)

---

8. Frontend Implementation

The frontend provides a simple and intuitive UI for interacting with the system.

Key UI Components

- Drag-and-drop upload zone for screenshots
- Code editor for source code input
- Language selector
- Result panel displaying:
    - Root cause explanation
    - Suggested fix
    - Debugging insights

The frontend communicates with the backend using REST APIs and displays responses dynamically.

---

9. Database Design (Optional Persistence)

When MongoDB is enabled, the system stores:

- Filename
- File path

- OCR extracted text
- LLM analysis output
- Timestamp

This allows future:

- Debug history tracking
- Model improvement
- Analytics on error patterns

---

10. How the System Works (Request Flow)

1. Frontend sends a POST request to `/analyze`
2. Backend saves the image to `uploads/`
3. OCR extracts error text from the image
4. Prompt builder constructs a debugging prompt
5. LLaVA processes the prompt using Ollama
6. The analysis is returned as JSON
7. Results are displayed in the UI

---

11. Limitations

- Image is not yet directly passed to the multimodal model
- Model output is currently unstructured text

- Performance depends on local LLM availability
- No confidence scoring in current implementation

---

12. Future Enhancements

- Direct image input to multimodal LLMs
- Structured JSON output (root cause, fix, confidence)
- Rule-based heuristics for common errors
- GridFS support for image storage
- Authentication and user history
- Cloud-based LLM API integration

---

13. Learning Outcomes

Through this project, the following skills were gained:

- Multimodal AI pipeline design
- OCR integration in real-world applications
- Prompt engineering for debugging tasks
- Full-stack development using React and FastAPI
- Local LLM deployment and inference
- System architecture and scalability concepts

---

14. Conclusion

The **Multimodal Code Debug Assistant** demonstrates a practical application of generative AI in software engineering. By combining image understanding, OCR, and large language models, the system showcases how debugging can be made more intelligent, automated, and developer-friendly. This project provides a strong foundation for future AI-powered developer tools and highlights the real-world potential of multimodal AI systems.