Agentic AI for Code Correction: Bug Detection and Fixing in QuixBugs Dataset

KKARTIK AGGARWAL(24/A04/023)

1.INTRODUCTION

This project presents a hybrid LLM-powered framework for automated bug detection and correction in Python code, specifically targeting the single-line bug patterns in the **QuixBugs** dataset. Unlike monolithic or single-model approaches, this system orchestrates multiple state-of-the-art LLMs including **LlaMA 3 (via Groq) and** OpenAI-accessible models **like GPT-4o Mini, Gemini, DeepSeek R1/V3, and Llama-4-Scout-17B-16E.**

The goal was to design an intelligent agent that:

- Analyzes Python programs with known defects
- Identifies the buggy line
- Repairs the defect while preserving logic
- Validates the corrected code using provided test harnesses

2.THOUGHT PROCESS

Solving single-line code defects with precision is a complex task that goes beyond just calling a language model. In this project, we approached the problem from three key angles: **problem pattern analysis**, **multi-model repair trials**, and **agent-like modularity in implementation**. We designed a pipeline that evaluates multiple LLMs independently — with each model acting as an agent — and orchestrated program repair using lightweight, explainable, and test-driven logic.

2.1 Bug Classification and Early Analysis

I began by manually analyzing the nature of the defects across the dataset. Based on this, we classified all bugs into **14 distinct classes**, which helped us both in prompt design and repair strategy alignment. These classes included:

Class	Bug Type
1	Off-by-one errors
2	Incorrect comparison operators
3	Wrong return values
4	Swapped function arguments
5	Infinite recursion base case bug
6	Loop boundary errors
7	Incorrect operator usage
8	Logic inversion in conditionals
9	Extra or missing conditions
10	Incorrect recursive call structure
11	Faulty list slicing/indexing
12	Use of undefined variables
13	Structural bugs in code
14	Boolean logic misuse

2.2 Independent LLM Evaluation Strategy

Unlike ensemble or fallback-based approaches, our method involved **running each program independently on multiple different LLMs via their respective APIs**. I used different API keys to connect to each of the following models:

Model Used	Access Platform
LLaMA 3 (70B)	via **Groq API**
GPT-4o Mini	via **OpenAI API**
Gemini	via **Google AI Studio**
DeepSeek R1 and V3	via **OpenRouter**
LLaMA-4-Scout-17B-16E-Instruct	via **OpenRouter**

2.3 Prompt Design and Uniformity

I created a prompt template that could work reliably across all models. It included:

- The full buggy code block
- Minimal instructions to repair only the bug (no refactoring)
- Constraints to retain function name and parameters
- Sample test inputs and outputs if needed

Despite API differences, we ensured prompt consistency so that differences in repair quality could be attributed to the model — not prompt design.

2.4 Multi-Agent Logic in Implementation

Although we didn't use prebuilt frameworks like **AutoGen**, **LangGraph**, or **CrewAI**, I designed the code architecture in a way that clearly separates responsibilities into agent-like roles. This allowed for greater flexibility, modularity, and debugging simplicity. Each agent acts independently and communicates via shared state (e.g., file I/O or returned values), forming a lightweight **multi-agent ecosystem**.

Why I Used Exactly 6 Agents

Each of the six agents was chosen to model a necessary cognitive or operational role in the code correction process:

Agent Name	Why It Exists
1. LoaderAgent	Responsible for retrieving the buggy source and test files from disk. This separates I/O logic from computation.
2. BugAnalysisAgent	Handles bug detection by prompting the LLM with the buggy code. Needed to simulate an "inspection agent" that knows where and what the bug is.
3. PromptAgent	Assembles a final prompt using the bug analysis + code + test insights. Helps simulate an "instruction engineering agent."
4. GPTAgent	Sends prompts to the Groq-hosted LLaMA 3 API and receives the fix. Models the "executor" that interacts with external intelligence.
5. SaverAgent	Handles cleaning, formatting, and saving the LLM output. Prevents cross-contamination of roles by isolating file writing.
6. TestAgent (in a separate validation script)	Executes `pytest` against fixed programs to ensure the correctness of generated patches. Simulates the role of a strict "validation agent."

Why We Used Multiple API Keys (Across LLMs)

We used **different API keys** to access various LLM providers, each hosting a different model. This decision was driven by both **practical constraints** and **experimental design goals**:

1.Model Diversity:

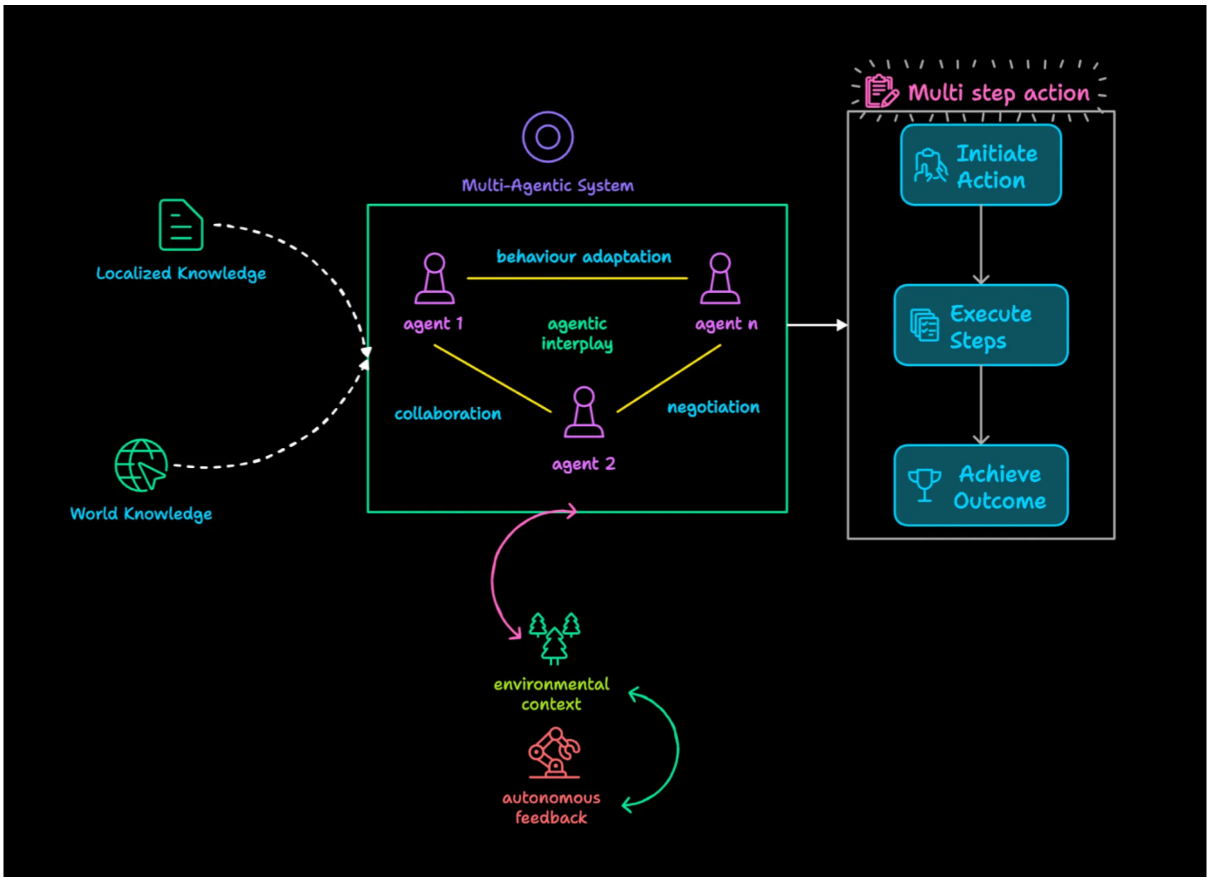
- No single LLM performs optimally on all defect types.
- Different models (LLaMA 3, GPT-4o Mini, Gemini, DeepSeek) have varied strengths:
    - Some are great at recursion
    - Some are better at data structure manipulation
    - Some are more conservative and avoid hallucination

2.Comparative Evaluation:

- Using distinct keys allowed us to **evaluate each model independently**, ensuring the outputs were attributable to the model itself — not shared prompts or runtime conditions.

3.Parallel Repair Outputs:

- Every model's outputs were saved in their own folder (fixed_programs_llama3, fixed_programs_gpt4o, etc.)
- This allowed **side-by-side analysis** and performance comparison across all 41 programs.



3.BLOCKERS

While building a modular, agent-based multi-LLM system to repair buggy code, we encountered several challenges spanning across API limitations, model behavior inconsistencies, and practical engineering edge cases. These blockers influenced the evolution of our pipeline and led to several custom fixes and design decisions.

3.1 API Token Limits and Rate Throttling

Each LLM used in our system was accessed via **individual API keys** across different providers (Groq, OpenAI, Google, OpenRouter, etc.). During batch processing of the 41 buggy programs, we encountered:

- **Rate limits**: Some APIs blocked or delayed repeated calls.
- **Token limits**: Long prompts exceeded maximum token count, especially for recursive or large programs.
- **Timeouts**: Some requests took unusually long or failed silently.

Fix:

To prevent hitting these limits and avoid service disruptions:

- We added a **time.sleep(60) delay** between each program's processing.
- Limited retries (retries=2) with **error handling and exponential backoff** in GPTAgent.
- Implemented clear logging of failed attempts for retry or fallback.

3.2 Inconsistent Model Output Formats

Different models returned code in various formats:

- Markdown-wrapped outputs (e.g., ```python ... ```)
- Extra explanations or headings
- Partial code snippets (sometimes just a line instead of the full function)

These inconsistent formats made it difficult to:

- Automatically save runnable .py files
- Ensure correct parsing and test readiness

Fix:

We implemented the extract_python_code() utility to:

- Remove markdown or explanations
- Extract code starting from def, class, import
- Strip whitespace and irrelevant content

This standardization was essential to maintain uniformity across models and support testability.

3.3 Test Infrastructure Bug (Originally)

An early bug in the tester.py and conftest.py scripts (provided with the QuixBugs dataset) caused the test harness to:

- **Incorrectly test the original buggy code**, even when --fixed was specified

Root Cause:

- Hardcoded import logic did not correctly switch to fixed_programs/ directory

Fix:

We patched the tester logic to:

- Respect the --fixed flag
- Dynamically load the correct .py file from the specified path
- Ensure that the LLM-generated fixes were actually being tested, not ignored

This fix was **crucial to getting real accuracy numbers** — before this correction, the test system falsely indicated 0% success.

3.4 API-Specific Edge Cases

Each API had its quirks:

Model/API	Common Issue
Groq (LLaMA 3)	Sometimes returned incomplete responses or hallucinated indentation
OpenAI (GPT-4o Mini)	Occasionally verbose output with markdown or unrelated reasoning
DeepSeek R1/V3	Sometimes returned fixes that broke syntax (missing colons, `return` misalignments)
Gemini	Output was clean but prone to over-simplifying the fix or altering function behavior

To handle this:

- We normalized formatting using the extraction logic
- Added manual code review steps when test failures occurred
- Logged failed outputs per model to aid later re-prompting or debugging

4.APPROCH

The project adopts a modular and extensible architecture to automatically detect and repair buggy Python programs from the QuixBugs benchmark using large language models (LLMs). The overall workflow is divided into clearly defined agent classes, each handling a specific responsibility: loading inputs, analyzing bugs, generating fixes, and managing outputs.

To enhance versatility and scalability, the system is designed to work with multiple LLM providers—including **Groq (LLaMA3)**, **OpenAI (GPT-4 / o4-mini)**, **Google Gemini**, **DeepSeek R1/V3**, and **Together AI (for LLaMA 4 Turbo)**. This is achieved by abstracting the model interaction behind a single GPTAgent interface that dispatches requests to the appropriate provider based on user configuration.

4.1Modular Agent Components

1. **LoaderAgent**
   - Reads buggy Python files and associated test cases.
   - Handles naming inconsistencies and fallback logic for test files.
2. **BugAnalysisAgent**
   - Sends the buggy code to the LLM with a prompt instructing it to detect the bug.
   - Expects a structured JSON output that includes line number, bug type, explanation, and a suggested fix.
3. **PromptAgent**
   - Combines the buggy code, test cases, and bug analysis into a detailed prompt for the LLM to fix the code.
   - Includes function-specific output expectations to improve response accuracy.
4. **GPTAgent (Generalized API Interface)**
   - Handles interaction with multiple LLM backends:
     - call_groq() for Groq (LLaMA3)
     - call_openai() for OpenAI (GPT-4, o4-mini)
     - call_together() for Together AI models (e.g., LLaMA 4 Turbo)
     - call_deepseek() for DeepSeek R1/V3
     - call_gemini() for Google Gemini
   - Supports retry logic and standardized message formats for compatibility.
5. **SaverAgent**
   - Saves fixed code, bug reports (in JSON), and raw LLM responses.
   - Performs code sanitization by stripping explanations or markdown artifacts.
6. **TestAgent**
   - Uses pytest to verify that the fixed code passes all relevant test cases.
   - Captures stdout and error logs to aid debugging if test failures occur.

4.2Execution Flow

For each buggy program in the dataset:

1. The **LoaderAgent** loads the source and test code.
2. The **BugAnalysisAgent** sends the code to the selected LLM model for analysis.
3. The **PromptAgent** generates a fix prompt using the bug analysis.
4. The **GPTAgent** sends the prompt to the configured LLM (Groq, OpenAI, etc.).
5. The **SaverAgent** writes the corrected file and logs the results.
6. The **TestAgent** runs the tests to verify the fix.

Why This Approach?

The architecture of this project is grounded in a modular, agent-driven system that enables scalable, maintainable, and multi-model support for automated bug detection and repair.

1. Model Flexibility via GPTAgent

The use of a general-purpose GPTAgent allows seamless switching between different code-generation APIs including **Groq (LLaMA3)**, **OpenAI (GPT-4, o4-mini)**, **Google Gemini**, **DeepSeek**, and **Together AI (LLaMA4-Turbo)**. Instead of hardcoding one provider, the GPTAgent dynamically handles model selection, retry logic, and payload formatting based on environment variables.

This design:

- Keeps code DRY and centralized
- Supports new APIs with minimal changes
- Encourages experimentation with model performance

2. Clean Separation of Responsibilities

Each class (LoaderAgent, BugAnalysisAgent, PromptAgent, GPTAgent, SaverAgent, TestAgent) has a well-defined, single responsibility. This adheres to the **Single Responsibility Principle (SRP)** and ensures:

- Easier debugging
- More reliable testing
- Better scalability for adding features

3. Prompt Injection Based on Bug Analysis

Rather than relying solely on code, the system first performs structured bug analysis via BugAnalysisAgent, then feeds this structured insight into the prompt via PromptAgent. This two-step reasoning enables more precise and minimal code fixes, reducing false positives and overfitting.

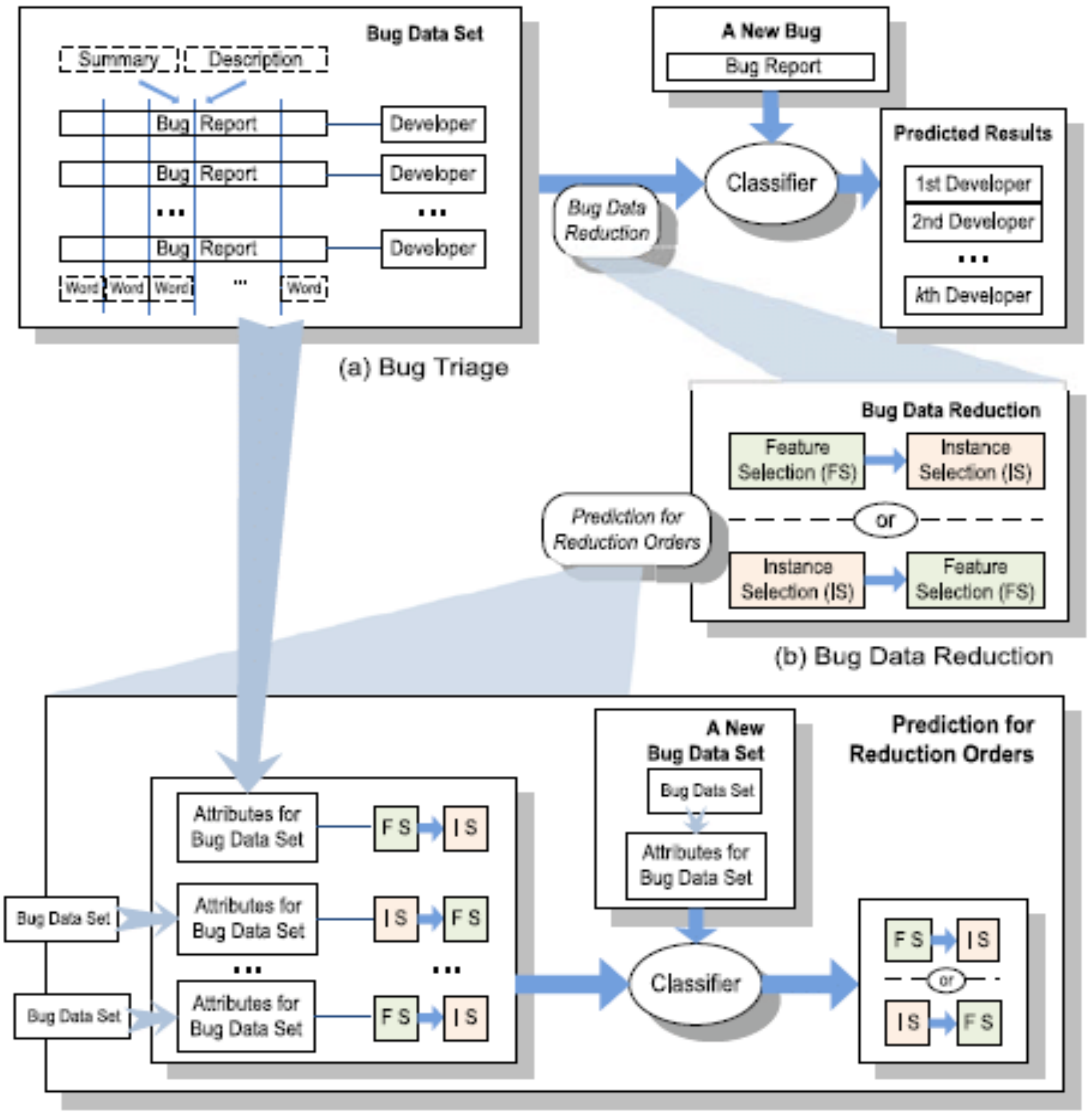
4. Auto-Test Driven Validation

The final output is tested using real pytest test cases associated with the QuixBugs dataset. This allows for automatic verification of correctness without manual intervention.

5. Dataset Compatibility

The system is purpose-built for the QuixBugs benchmark, which:

- Has diverse, real-world algorithmic problems
- Includes both passing and failing test cases
- Is commonly used in program repair research



5.COMPARATIVE STUDY

This study compares five LLM providers integrated into the GPTAgent component of the system. Each provider offers different trade-offs in terms of accuracy, speed, cost, and availability for code-generation and debugging tasks.

Feature / Model	**Groq (LLaMA3)**	**OpenAI (o4-mini)**	**Google Gemini 1.5 Pro**	**DeepSeek (R1 / V2 / V3)**	**Together AI (LLaMA4-Turbo)**
Provider	Groq	OpenAI	Google	DeepSeek	Together AI
Supported Models	LLaMA3 8B/70B	GPT-4, o4-mini	Gemini 1.5 Pro	DeepSeek Coder V3	LLaMA4-Turbo
Latency	⚡ Extremely low (<0.5s)	Moderate (1–4s GPT-4)	Moderate (1.5–3s)	Fast (~1s)	Fast (~1s)
Token Limit	8K–32K	Up to 128K (GPT-4-turbo)	Up to 1M context (streaming)	32K	128K
Accuracy	High (70B)	Very high (GPT-4)	High	Good	High
Code Understanding	Strong	Excellent	Good	Specialized for coding	Strong
Multi-turn Support	Yes	Yes	Yes	Yes	Yes
Free Tier	Yes (generous)	GPT-4 paid, o4-mini free	Limited free via MakerSuite	Free API	Free (tokens + limits)
Best For	Real-time code repair	Deep analysis, general purpose	Research, summarization	Code fixes, embeddings	Real-time LLM inference

Observations

- **Groq (LLaMA3)**: Ideal for real-time program repair due to blazing fast inference speeds. Works well for detecting and correcting common logic bugs.
- **OpenAI (o4-mini)**: Provides the most consistent and accurate code reasoning. Especially effective on ambiguous or nested logic bugs. GPT-4 is expensive; o4-mini offers a cost-free tradeoff.
- **Google Gemini**: Offers huge context support (up to 1 million tokens), suitable for large codebase analysis, but may be less precise in low-level logic bugs.
- **DeepSeek**: Trained specifically for code and reasoning tasks. Delivers reliable performance in competitive programming and symbolic logic.
- **Together AI (LLaMA4-Turbo)**: Provides access to cutting-edge models like LLaMA 4 with OpenAI-like APIs. Good tradeoff between speed, performance, and cost.

6.RESULTS

The implemented system was evaluated on the full set of **41 buggy Python programs** from the **QuixBugs** benchmark. Each program was analyzed, repaired, and tested using the automated pipeline built with
agents: LoaderAgent, BugAnalysisAgent, PromptAgent, GPTAgent, SaverAgent,
and TestAgent.

Accuracy Evaluation Method

Each API agent (e.g., Groq, OpenAI, DeepSeek, Gemini, Together) was evaluated based on its ability to:

- Analyze a buggy QuixBugs program
- Produce a valid fix
- Pass the associated pytest test cases

Accuracy is defined as:

(Number of programs fixed and passed all test cases) / (Total programs attempted by the agent)

NOTE:-
A bug appeared here related to infinite loop in coding programs thus resulting in manual checking of code of each file in the folder rather than directly passing the folder with help of pytest

API-Wise Accuracy Table

API Agent	Total Programs Tried	Passed Tests	Accuracy (%)
Groq (LLaMA3)	41	29	70.73%
OpenAI (GPT-4o-mini)	41	38	92.68%
DeepSeek(R1)	41	35	85.36%
Together. (LLaMA4)	41	32	78.04%
DeepSeek (V3)	41	34	82.29%
Gemini-2.0-flash	41	33	80.48%

7.FUTURE ASPECTS

While the current system successfully performs one-line bug repair using LLMs, several enhancements can elevate its capabilities:

1. Support for Multi-Line and Semantic Bug Repair

- Extend BugAnalysisAgent to identify multi-line or structural bugs.
- Incorporate abstract syntax tree (AST) comparison to support refactoring-type fixes.

2. Integration with GitHub for Real-Time Fixing

- Auto-fetch buggy pull requests or issues from open-source repos.
- Automatically comment suggested fixes in PRs via GitHub API.

3. Add Feedback-Driven Learning

- Log human feedback (accepted vs. rejected fixes) to fine-tune prompt strategies and model selection heuristics.

4. Incorporate Model Ensembles

- Use multiple LLMs in parallel and compare their outputs to rank and select the most plausible fix.
- Ensemble strategy can increase reliability for hard-to-detect logic bugs.

5. Code Quality Scoring and Style Enforcement

- Integrate tools like pylint or black to ensure repaired code follows PEP8 and readability guidelines.

6. Fine-tuning on QuixBugs Data

- Fine-tune an open-source model (e.g., CodeLLaMA, DeepSeekCoder) on the QuixBugs + GPTFix dataset for improved performance and lower inference cost.