# Multi-Agentic Code Correction System – Technical Documentation

## 1. Overview

This system is designed to repair buggy algorithms from the QuixBugs dataset using a collaborative multi-agent approach. Each agent specializes in a distinct task within a structured pipeline. Two main implementation strategies are provided:  
1. A modular, multi-agent system composed of dedicated bug analysis, repair, and validation agents.  
2. A single-agent fast correction module implemented in code\_corrector.py for streamlined and rapid fixes.

## 2. Approach 1: Multi-Agent Pipeline

This approach consists of three distinct agents: Bug Analysis, Code Repair, and Code Validation. Each agent operates independently and sequentially within the pipeline.

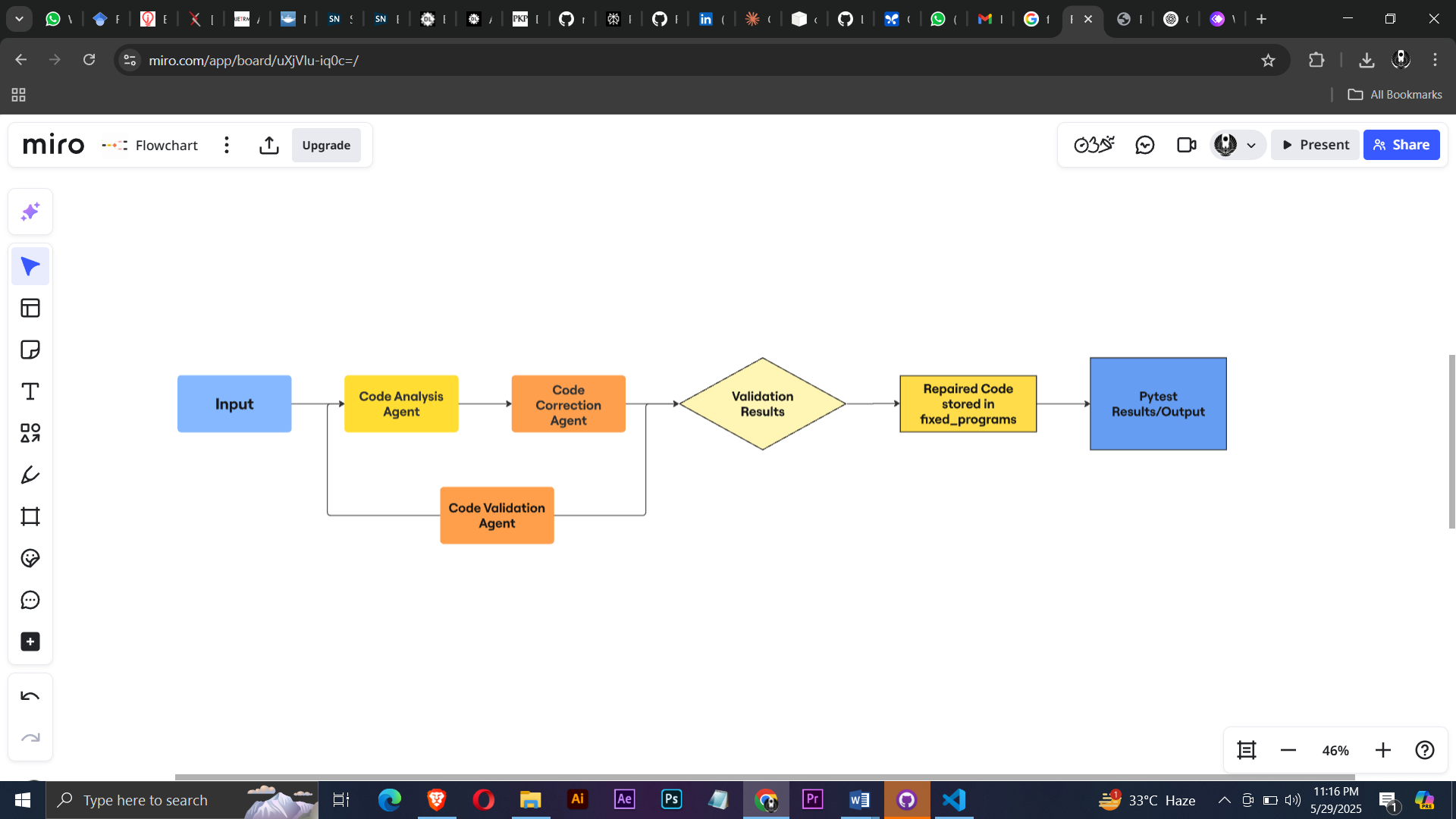


Fig 2.1: Flowchart of workflow

**2.1 Bug Analysis Agent**

File: bug\_analysis\_agent.py  
This agent analyzes the buggy code and outputs a structured explanation of the bug. It provides the bug's type, location, and the expected behavior. Rate limiting is enforced using exponential backoff and jitter strategies. The Gemini API (version 1.5) is used for LLM-based reasoning.

### 2.2 Code Repair Agent

File: code\_repair\_agent.py  
This agent uses the results of the analysis to apply a minimal correction to the code. It maintains structural and naming consistency while correcting the root cause of the error. The repaired output is stored in JSON and Python format.

### 2.3 Code Validation Agent

File: complete\_validation\_agent.py  
This agent performs a final check on the repaired code. It assesses correctness using similarity scoring based on ASTs, token sequences, and syntactic structure. If validation fails, the agent attempts one final correction using the Gemini model.

### 2.4 Pipeline Controller

File: main\_pipeline.py  
Orchestrates the three agents in sequence: analysis, repair, and validation. It also handles user input, progress tracking, and summary reporting.

### 2.5 Evaluation

Results are summarized in a report which includes statistics like success rate, average similarity, and status categories (Exact, High, Medium, Low). A recent run achieved a 97.56% success rate with 66.11% average similarity to ground truth solutions.

## 3. Approach 2: Fast Code Corrector

Implemented as a single-agent system in code\_corrector.py, this approach is optimized for speed and minimal overhead.

### 3.1 Core Agent

Class: FastCodeCorrectionAgent  
This agent fixes the code directly using the Gemini API. It includes multiple accuracy modes (fast, standard, high, maximum) to control retries, reasoning style, and delay.

### 3.2 Features

- Direct repair prompt without relying on prior analysis  
- Optional ReACT-style reasoning during the second attempt  
- Syntax checking and AST-based structural similarity scoring  
- Optional unit testing using json\_testcases (skipped in fast mode)

### 3.3 Reporting

The agent provides comprehensive logs including similarity scores, syntax validity, and correction status. Supports both sequential and concurrent processing using thread pools.

## 4. Comparison of Approaches

The table below summarizes the differences between the two approaches:

|  |  |  |
| --- | --- | --- |
| Feature | Multi-Agentic Pipeline | Fast Code Corrector |
| Bug Analysis | Yes (Explicit) | No (Implicit via prompt) |
| Minimal Fix Strategy | Yes | Yes |
| Multi-Attempt Fixing | No | Yes |
| Validation Phase | Yes (AST, token, testing) | Optional/Skipped |
| Accuracy Control | No | Yes (4 modes) |
| Concurrency Support | No | Yes (Threaded) |
| Use Case | Research, Quality-focused | Speed-focused, Benchmarking |

## 5. Recommendation

Use the multi-agent pipeline when high correctness and traceability are essential, such as in research, academic analysis, or automated grading environments. The fast code corrector is suitable for hackathons, large-scale batch correction, or scenarios where runtime and throughput are more important than perfect accuracy.

**6. Results**

In a recent run (standard mode, gemini-1.5-flash), the agent processed 41 buggy programs, successfully fixing 40.

- Success Rate: 97.56%

- Average AST Similarity: 66.07%

- Programs per Minute: ~22

- Total test Cases passed after pytest: 22/31 = 70.96% (excluded those 10 cases where json testcases is not present in folder)